2016

Dependent Types in Haskell: Theory and Practice

Richard A. Eisenberg
Bryn Mawr College, rae@cs.brynmawr.edu

DEPENDENT TYPES IN HASKELL: THEORY AND PRACTICE

Richard A. Eisenberg

A DISSERTATION

in

Computer and Information Sciences

Presented to the Faculties of the University of Pennsylvania
in
Partial Fulfillment of the Requirements for the
Degree of Doctor of Philosophy

2016

Supervisor of Dissertation

Stephanie Weirich, PhD
Professor of CIS

Graduate Group Chairperson

Lyle Ungar, PhD
Professor of CIS

Dissertation Committee
Rajeev Alur, PhD (Professor of CIS)
Simon Peyton Jones (Principal Researcher, Microsoft Research)
Benjamin Pierce, PhD (Professor of CIS; Committee Chair)
Steve Zdancewic, PhD (Professor of CIS)

DEPENDENT TYPES IN HASKELL: THEORY AND PRACTICE

The complete source code for this document is available from

http://github.com/goldfirere/thesis

To Amanda,

who has given more of herself toward this doctorate than I could ever ask.

Acknowledgments

I have so many people to thank.

First and foremost, I thank my sponsors. This material is based upon work supported by the National Science Foundation under Grant No. 1116620. I also gratefully acknowledge my Microsoft Research Graduate Student Fellowship, which has supported me in my final two years.

Thanks to the Haskell community, who have welcomed this relative newcomer with open arms and minds. My first line of Haskell was written only in 2011! There are far too many to name, but I'll call out Ben Gamari and Austin Seipp for their commendable job at shepherding the GHC development process.

I am not one for large displays of school spirit. Nevertheless, I cannot imagine a better place to get my doctorate than Penn. I will remain passionate in my advocacy for this graduate program for many years to come.

The PLClub at Penn has been a constant source of camaraderie, help on various subjects, and great talks. Thanks to Jianzhou, Mike, Chris, Marco, Emilio, Cătălin, Benoît, Benoît, Maxime, Delphine, Vilhelm, Daniel, Bob, Justin, Arthur, Antal, Jennifer, Dmitri, William, Leo, Robert, Antoine, and Pedro.

Thanks to "FC crew" who put up with my last-minute reminders and frequent rescheduling. David Darais, Iavor Diatchki, Kenny Foner, Andres Löh, Pedro Magalhães, Conor McBride: thanks for all the great discussions, and I look forward to many more to come.

I can thank Joachim Breitner for helping me with perhaps the hardest part of this dissertation: *not* working on roles, a topic I have tried to escape for the better part of three years. Joachim spearheaded our papers on the subject, and his excellent organization at writing papers will serve as a template for my future projects.

Jan Stolarek co-authored several papers with me and his probing questions helped me greatly to understand certain aspects of Dependent Haskell better. In particular, the idea of having matchable vs. unmatchable functions is directly due to work done in concert with Jan.

Adam Gundry bulldozed the path for me. His dissertation was something of a road map for mine, and I always learn from his insight.

Sincere thanks to Peter-Michael Osera for leading the way toward a position at a liberal arts college and for much humor, some of it appropriate.

I owe a debt of gratitude to Brent Yorgey. He decided not to continue pursuing

dependent types in Haskell just as I came along. He also (co-)wrote a grant that was approved just in time to free up his advisor to take on another student. Much of my success is due to Brent's paving the way for me.

Dimitrios Vytiniotis was a welcoming co-host at Microsoft Research when I was there. I still have scars from the many hours of battling the proof dragon in his office.

None of this, quite literally, would be possible without the leap of faith taken by Benjamin Pierce, to whom I argued for my acceptance to Penn, over the phone, on a shared line in the middle of a campground in the Caribbean, surrounded by children and families enjoying their vacation. A condition of my acceptance was that I would not work with Stephanie, who had no room for me, despite our matching interests. I trust Benjamin does not regret this decision, even though I violated this condition.

I offer a heartfelt thanks to Steve Zdancewic. From the beginning of my time at Penn, I felt entirely at home knocking on his door at any time to ask for advice or mentorship. I did not often take advantage of this, but it was indeed a comfort knowing I could seek him out.

I cannot express enough gratitude toward my family, new and old, who have supported me in every way possible.

Simon Peyton Jones is a visionary leader for the Haskell community, holding all of us together on the steady stride toward a more perfect language. Simon's mentorship to me, personally, has been invaluable. It is such an honor to work alongside you, Simon, and I look forward to much collaboration to come.

Stephanie Weirich is the best advisor a student could ask for. She is insightful, full of energy and ideas, and simply has an intuitive grasp on how best to nudge me along. And she's brilliant. Stephanie, thanks for pulling me out of your Haskell programming class five years ago—that's what started us on this adventure. Somehow, you made me feel right away that I was having interesting and novel ideas; in retrospect, many of them were really yours, all along. This is surely the sign of excellent academic advising.

I am left to thank my wife Amanda and daughter Emma. Both have been with me every step of the way. Well, Emma missed some steps as she wasn't walking for the first year or so, having been born two months before I started at Penn. But tonight, she accurately summarized to Amanda the difference between Dependent Haskell and Idris (one is a change to an existing language while the other is a brand new one, but both have dependent types). Children grow fast, and I know Emma is eager for the day when I can finally explain to her what it is I do all day.

And for Amanda, these words will have to do, because no words can truly express how I feel: I love you, and thank you.

Richard A. Eisenberg
August 2016

ABSTRACT

DEPENDENT TYPES IN HASKELL: THEORY AND PRACTICE

Richard A. Eisenberg

Stephanie Weirich

Haskell, as implemented in the Glasgow Haskell Compiler (GHC), has been adding new type-level programming features for some time. Many of these features—generalized algebraic datatypes (GADTs), type families, kind polymorphism, and promoted datatypes—have brought Haskell to the doorstep of dependent types. Many dependently typed programs can even currently be encoded, but often the constructions are painful.

In this dissertation, I describe Dependent Haskell, which supports full dependent types via a backward-compatible extension to today's Haskell. An important contribution of this work is an implementation, in GHC, of a portion of Dependent Haskell, with the rest to follow. The features I have implemented are already released, in GHC 8.0. This dissertation contains several practical examples of Dependent Haskell code, a full description of the differences between Dependent Haskell and today's Haskell, a novel dependently typed lambda-calculus (called Pico) suitable for use as an intermediate language for compiling Dependent Haskell, and a type inference and elaboration algorithm, Bake, that translates Dependent Haskell to type-correct Pico. Full proofs of type safety of Pico and the soundness of Bake are included in the appendix.

Contents

List of Figures

Chapter 1

Introduction

Haskell has become a wonderful playground for type system experimentation. Despite its relative longevity—at roughly 25 years old [45]—type theorists still turn to Haskell as a place to build new type system ideas and see how they work in a practical setting [5, 11, 15, 16, 32, 40, 46, 49, 51, 53, 66, 75, 76]. As a result, Haskell's type system has grown ever more expressive over the years. As the power of types in Haskell has increased, Haskellers have started to integrate dependent types into their programs [4, 30, 56, 60], despite the fact that today's Haskell[1] does not internally support dependent types. Indeed, the desire to program in Haskell but with support for dependent types influenced the creation of Cayenne [3], Agda [68], and Idris [9]; all are Haskell-like languages with support for full dependent types.

This dissertation closes the gap, by adding support for dependent types into Haskell. In this work, I detail both the changes to GHC's internal language, previously known as System FC [87] but which I have renamed Pico, and the changes to the surface language necessary to support dependent types. Naturally, I must also describe the elaboration from the surface language to the internal language, including type inference through my novel algorithm Bake. Along with the textual description contained in this dissertation, I have also partially implemented these ideas in GHC directly; indeed, my contributions were one of the key factors in making the current release of GHC a new major version. It is my expectation that I will implement the internal language and type inference algorithm described in this work in GHC in the near future. Much of my work builds upon the critical work of Gundry [37]; one of my chief contributions is adapting his work to work with the GHC implementation and further features of Haskell.

1.1 Contributions

I offer the following contributions:

[1] Throughout this dissertation, a reference to "today's Haskell" refers to the language implemented by the Glasgow Haskell Compiler (GHC), version 8.0, released in 2016.

- Chapter 3 includes a series of examples of dependently typed programming in Haskell. Though a fine line is hard to draw, these examples are divided into two categories: programs where rich types give a programmer more compile-time checks of her algorithms, and programs where rich types allow a programmer to express a more intricate algorithm that may not be well typed under a simpler system.

  Although no new results are presented in Chapter 3, these examples are a true contribution of this dissertation. Dependently typed programs are still something of a rarity, as evidenced by the success at publishing novel dependently typed programs [8, 23, 61, 69]. This chapter extends our knowledge of dependently typed programming by showing how certain programs might look in Haskell. The two most elaborate examples are:

  - a dependently typed database access library based on the design of Oury and Swierstra [69] but with the ability to infer a database schema based on how its fields are used, and
  - a translation of Idris's algebraic effects library [8] into Dependent Haskell that allows for an easy-to-use alternative to monad transformer stacks. With heavy use of singletons, it is possible to encode this library in today's Haskell due to my implementation work.

  Section 3.3 then argues why dependent types in Haskell, in particular, are an interesting and worthwhile subject of study.

- Dependent Haskell (Chapter 4) is the surface language I have designed in this dissertation. This chapter is written to be useful to practitioners, being a user manual of sorts of the new features. In combination with Chapter 3, this chapter could serve to educate Haskellers on how to use the new features.

  In some ways, Dependent Haskell is similar to existing dependently typed languages, drawing no distinction between terms and types and allowing rich specifications in types. However, it differs in several key ways from existing approaches to dependent types:

  1. Dependent Haskell has the **Type** : **Type** axiom, avoiding the need for an infinite hierarchy of sorts [57, 80] used in other languages. (Section 4.4.1)
  2. A key issue when writing dependently typed programs is in figuring out what information is needed at runtime. Dependent Haskell's approach is to require the programmer to choose whether a quantified variable should be retained (making a proper Π-type) or discarded (making a ∀-type) during compilation.
  3. In contrast to many dependently typed languages, Dependent Haskell is agnostic to the issue of termination. There is no termination checker in the

language, and termination is not a prerequisite of type safety. A drawback of this approach is that some proofs of type equivalence must be executed at runtime, as discussed in Section 4.4.5.

4. As elaborated in Chapter 6, Dependent Haskell retains important type inference characteristics that exist in previous versions of Haskell (e.g., those characteristics described by Vytiniotis et al. [99]). In particular, all programs accepted by today's GHC—including those without type signatures—are also valid in Dependent Haskell.

- Pico (pronounced "Π-co", never "peek-o") is a new dependently typed λ-calculus, intended as an internal language suitable as a target for compiling Dependent Haskell. (Chapter 5) Pico allows full dependent types, has the **Type** : **Type** axiom, and yet has no computation in types. Instead of allowing type equality to include, say, $\beta\eta$-equivalence (as in Coq), type equality in Pico is just α-equivalence. A richer notion of type equivalence is permitted through coercions, which witness the equivalence between two types. In this way, Pico is a direct descendent of System FC [11, 32, 87, 105, 107] and of the *evidence* language of Gundry [37].

  Pico supports unsaturated functions in types, while still allowing function application decomposition in its equivalence relation.[2] This is achieved by my novel separation of the function spaces of type constants, which are generative and injective, from the ordinary, unrestricted function space Allowing unsaturated functions in types is a key step forward Pico makes over Gundry's *evidence* language [37]; it means that *all* expressions can be promoted to types, in contrast to Gundry's subset of terms shared with the language of types.

  In Appendix C, I prove the usual preservation and progress theorems for Pico as well as a type erasure theorem that relates the operational semantics of Pico to that of a simple λ-calculus with datatypes and **fix**. In this way, I show that all the fancy types really can be erased at runtime.

- The novel algorithm Bake (Chapter 6) performs type inference on the Dependent Haskell surface language, providing typing rules and an elaboration into Pico. I am unaware of a similarly careful study of type inference in the context of dependent types. These typing rules contain an algorithmic specification of Dependent Haskell, detailing which programs should be accepted and which should be rejected. The type system is bidirectional and contains a novel treatment for inferring types around dependent pattern matches, among a few other, smaller innovations. I prove that the elaborated program is always well typed in Pico.

- A partial implementation of the type system in this dissertation is available in GHC 8.0. Chapter 7 discusses implementation details, including the current state

[2] I am referring to the **left** and **right** coercions of System FC here.

of the implementation. It focuses on the released implementation of the system from Weirich et al. [105]. Considerations about implementing full Dependent Haskell are also included here.

- Chapter 8 puts this work in context by comparing it to several other dependently typed systems, both theories and implementations. This chapter also suggests some future work that can build from the base I lay down here.

Though not a new contribution, Chapter 2 contains a review of features available in today's Haskell that support dependently typed programming. This is included as a primer to these features for readers less experienced in Haskell, and also as a counterpoint to the features discussed as parts of Dependent Haskell.

This dissertation is most closely based upon my prior work with Weirich and Hsu [105]. That paper, focusing solely on the internal language, merges the type and kind languages but does not incorporate dependent types. I wrote the implementation of these ideas as a component of GHC 8, incorporating Peyton Jones's extensive feedback. This dissertation work—particularly Chapter 6—also builds on a more recent paper with Weirich and Ahmed [33], which develops the theory around type inference where some arguments are visible (and must be supplied) and others are invisible (and may be omitted). Despite this background, almost the entirety of this dissertation is new work; none of my previous published work has dealt directly with dependent types.

1.2 Implications beyond Haskell

This dissertation necessarily focuses quite narrowly on discussing dependent types within the context of Haskell. What good is this work to someone uninterested in Haskell? I offer a few answers:

- In my experience, many people both in the academic community and beyond believe that a dependently typed language must be total in order to be type-safe. Though Dependent Haskell is not the first counterexample to this mistaken notion (e.g., [3, 12]), the existence of this type-safe, dependently typed, non-total language may help to dispel this myth.

- This is the first work, to my knowledge, to address type inference with **let**-generalization (of top-level constructs only, see Section 6.2.2) and dependent types. With the caveat that non-top-level **let** declarations are not generalized, I claim that the Bake algorithm I present in Chapter 6 is conservative over today's Haskell and thus over Hindley-Milner. See Section 6.8.2.

- Even disregarding **let**-generalization, Bake is the first (to my knowledge) thorough treatment of type inference for dependent types. My bidirectional type inference algorithm infers whether or not a pattern match should be treated

as a dependent or a traditional match, a feature that could be ported to other languages.

- Once Dependent Haskell becomes available, I believe dependent types will become popular within the Haskell community, given the strong encouragement I have received from the community and the popularity of my singletons library [29, 30]. Perhaps this popularity will inspire other languages to consider adding dependent types, amplifying the impact of this work.

As the features in this dissertation continue to become available, I look forward to seeing how the Haskell community builds on top of my work and discovers more and more applications of dependent types.

Chapter 2

Preliminaries

This chapter is a primer for type-level programming facilities that exist in today's Haskell. It serves both as a way for readers less experienced in Haskell to understand the remainder of the dissertation and as a point of comparison against the Dependent Haskell language I describe in Chapter 4. Those more experienced with Haskell may easily skip this chapter. However, all readers may wish to consult Appendix A to learn the typographical conventions used throughout this dissertation.

I assume that the reader is comfortable with a typed functional programming language, such as Haskell98 or a variant of ML.

2.1 Type classes and dictionaries

Haskell supports type classes [102]. An example is worth a thousand words:

```
class Show a where
  show :: a → String
instance Show Bool where
  show True  = "True"
  show False = "False"
```

This declares the class *Show*, parameterized over a type variable *a*, with one method *show*. The class is then instantiated at the type *Bool*, with a custom implementation of *show* for *Bool*s. Note that, in the *Show Bool* instance, the *show* function can use the fact that *a* is now *Bool*: the one argument to *show* can be pattern-matched against *True* and *False*. This is in stark contrast to the usual parametric polymorphism of a function *show'* :: $a \to$ *String*, where the body of *show'* *cannot* assume any particular instantiation for *a*.

With *Show* declared, we can now use this as a constraint on types. For example:

```
smooshList :: Show a ⇒ [a] → String
smooshList xs = concat (map show xs)
```

The type of *smooshList* says that it can be called at any type *a*, as long as there exists an instance *Show a*. The body of *smooshList* can then make use of the *Show a* constraint by calling the *show* method. If we leave out the *Show a* constraint, then the call to *show* does not type-check. This is a direct result of the fact that the full type of *show* is really $Show\ a \Rightarrow a \to String$. (The *Show a* constraint on *show* is implicit, as the method is declared within the *Show* class declaration.) Thus, we need to know that the instance *Show a* exists before calling *show* at type *a*.

Operationally, type classes work by passing *dictionaries* [39]. A type class dictionary is simply a record containing all of the methods defined in the type class. It is as if we had these definitions:

```
data ShowDict a = MkShowDict {showMethod :: a → String}

showBool :: Bool → String
showBool True  = "True"
showBool False = "False"

showDictBool :: ShowDict Bool
showDictBool = MkShowDict showBool
```

Then, whenever a constraint *Show Bool* must be satisfied, GHC produces the dictionary for *showDictBool*. This dictionary actually becomes a runtime argument to functions with a *Show* constraint. Thus, in a running program, the *smooshList* function actually takes two arguments: the dictionary corresponding to *Show a* and the list $[a]$.

2.2 Families

2.2.1 Type families

A *type family* [15, 16, 32] is simply a function on types. (I sometimes use "type function" and "type family" interchangeably.) Here is an uninteresting example:

```
type family F1 a where
  F1 Int  = Bool
  F1 Char = Double

useF1 :: F1 Int → F1 Char
useF1 True  = 1.0
useF1 False = (−1.0)
```

We see that GHC simplifies $F_1\ Int$ to *Bool* and $F_1\ Char$ to *Double* in order to type-check $useF_1$.

F_1 is a *closed* type family, in that all of its defining equations are given in one place. This most closely corresponds to what functional programmers expect from their functions. Today's Haskell also supports *open* type families, where the set of defining equations can be extended arbitrarily. Open type families interact particularly

well with Haskell's type classes, which can also be extended arbitrarily. Here is a more interesting example than the one above:

```
type family Element c
class Collection c where
  singleton :: Element c → c

type instance Element [a] = a
instance Collection [a] where
  singleton x = [x]

type instance Element (Set a) = a
instance Collection (Set a) where
  singleton = Set.singleton
```

Because the type family *Element* is open, it can be extended whenever a programmer creates a new collection type.

Often, open type families are extended in close correspondence with a type class, as we see here. For this reason, GHC supports *associated* open type families, using this syntax:

```
class Collection' c where
  type Element' c
  singleton' :: Element' c → c

instance Collection' [a] where
  type Element' [a] = a
  singleton' x = [x]

instance Collection' (Set a) where
  type Element' (Set a) = a
  singleton' = Set.singleton
```

Associated type families are essentially syntactic sugar for regular open type families.

Partiality in type families A type family may optionally be *partial*, in that it is not defined over all possible inputs. This poses no problems in the theory or practice of type families. If a type family is used at a type for which it is not defined, the type family application is considered to be *stuck*. For example:

```
type family F2 a
type instance F2 Int = Bool
```

Suppose there are no further instances of F_2. Then, the type F_2 *Char* is stuck. It does not evaluate, and is equal only to itself.

It is impossible for a Haskell program to detect whether or not a type is stuck, as doing so would require pattern-matching on a type family application—this is not

possible. This is a good design because a stuck open type family might become unstuck with the inclusion of more modules, defining more type family instances. Stuckness is therefore fragile and may depend on what modules are in scope; it would be disastrous if a type family could branch on whether or not a type is stuck.

2.2.2 Data families

A *data family* defines a family of datatypes. An example shows best how this works:

```
data family Array a   -- compact storage of elements of type a
data instance Array Bool = MkArrayBool ByteArray
data instance Array Int  = MkArrayInt  (Vector Int)
```

With such a definition, we can have a different runtime representation for *Array Bool* than we do for *Array Int*, something not possible with more traditional parameterized types.

Data families do not play a large role in this dissertation.

2.3 Rich kinds

2.3.1 Kinds in Haskell98

With type families, we can write type-level programs. But are our type-level programs correct? We can gain confidence in the correctness of the type-level programs by ensuring that they are well-kinded. Indeed, GHC does this already. For example, if we try to say *Element Maybe*, we get a type error saying that the argument to *Element* should have kind $\star$, but *Maybe* has kind $\star \to \star$.

Kinds in Haskell are not a new invention; they are precisely defined in the Haskell98 report [71]. Because type constructors in Haskell may appear without their arguments, Haskell needs a kinding system to keep all the types in line. For example, consider the library definition of *Maybe*:

```
data Maybe a = Nothing | Just a
```

The word *Maybe*, all by itself, does not really represent a type. *Maybe Int* and *Maybe Bool* are types, but *Maybe* is not. The type-level constant *Maybe* needs to be given a type to become a type. The kind-level constant $\star$ contains proper types, like *Int* and *Bool*. Thus, *Maybe* has kind $\star \to \star$.

Accordingly, Haskell's kind system accepts *Maybe Int* and *Element* [*Bool*], but rejects *Maybe Maybe* and *Bool Int* as ill-kinded.

2.3.2 Promoted datatypes

The kind system in Haskell98 is rather limited. It is generated by the grammar $\kappa ::= \star \mid \kappa \to \kappa$, and that's it. When we start writing interesting type-level programs, this almost-unityped limitation bites.

For example, previous to recent innovations, Haskellers wishing to work with natural numbers in types would use these declarations:

```
data Zero
data Succ a
```

We can now discuss *Succ* (*Succ Zero*) in a type and treat it as the number 2. However, we could also write nonsense such as *Succ Bool* and *Maybe Zero*. These errors do not imperil type safety, but it is natural for a programmer who values strong typing to also pine for strong kinding.

Accordingly, Yorgey et al. [107] introduce promoted datatypes. The central idea behind promoted datatypes is that when we say

```
data Bool = False | True
```

we declare two entities: a type *Bool* inhabited by terms *False* and *True*; and a kind *Bool* inhabited by types *'False* and *'True*.[3] We can then use the promoted datatypes for more richly kinded type-level programming.

A nice, simple example is type-level addition over promoted unary natural numbers:

```
data Nat = Zero | Succ Nat
type family a + b where
  'Zero   + b = b
  'Succ a + b = 'Succ (a + b)
```

Now, we can say *'Succ 'Zero* + *'Succ* (*'Succ 'Zero*) and GHC will simplify the type to *'Succ* (*'Succ* (*'Succ 'Zero*)). We can also see here that GHC does kind inference on the definition for the type-level +. We could also specify the kinds ourselves like this:

```
type family (a :: Nat) + (b :: Nat) :: Nat where ...
```

Yorgey et al. [107] detail certain restrictions in what datatypes can be promoted. A chief contribution of this dissertation is lifting these restrictions.

2.3.3 Kind polymorphism

A separate contribution of the work of Yorgey et al. [107] is to enable *kind polymorphism*. Kind polymorphism is nothing more than allowing kind variables to be held abstract,

[3] The new kind does not get a tick ' but the new types do. This is to disambiguate a promoted data constructor *'X* from a declared type *X*; Haskell maintains separate type and term namespaces. The ticks are optional if there is no ambiguity, but I will always use them throughout this dissertation.

just like functional programmers frequently do with type variables. For example, here is a type function that calculates the length of a type-level list at any kind:

```
type family Length (list :: [k]) :: Nat where
  Length '[]       = 'Zero
  Length (x ': xs) = 'Succ (Length xs)
```

Kind polymorphism extends naturally to constructs other than type functions. Consider this datatype:

```
data T f a = MkT (f a)
```

With the `PolyKinds` extension enabled, GHC will infer a most-general kind $\forall\ k.\ (k \to \star) \to k \to \star$ for T. In Haskell98, on the other hand, this type would have kind $(\star \to \star) \to \star \to \star$, which is less general.

A kind-polymorphic type has extra, invisible parameters that correspond to kind arguments. When I say *invisible* here, I mean that the arguments do not appear in Haskell source code. With the `-fprint-explicit-kinds` flag, GHC will print kind parameters when they occur. Thus, if a Haskell program contains the type $T\ Maybe\ Bool$ and GHC needs to print this type with `-fprint-explicit-kinds`, it will print $T \star Maybe\ Bool$, making the $\star$ kind parameter visible. Today's Haskell makes an inflexible choice that kind arguments are always invisible, which is relaxed in Dependent Haskell. See Section 4.2.3 for more information on visibility in Dependent Haskell.

2.3.4 Constraint kinds

Bolingbroke introduced *constraint kinds* to GHC.[4] Haskell allows constraints to be given on types. For example, the type $Show\ a \Rightarrow a \to String$ classifies a function that takes one argument, of type a. The $Show\ a \Rightarrow$ constraint means that a is required to be a member of the $Show$ type class. Constraint kinds make constraints fully first-class. We can now write the kind of $Show$ as $\star \to Constraint$. That is, $Show\ Int$ (for example) is of kind $Constraint$. $Constraint$ is a first-class kind, and can be quantified over. A useful construct over $Constraint$s is the $Some$ type:

```
data Some :: (⋆ → Constraint) → ⋆ where
  Some :: c a ⇒ a → Some c
```

If we have a value of $Some\ Show$, stored inside it must be a term of some (existentially quantified) type a such that $Show\ a$. When we pattern-match against the constructor $Some$, we can use this $Show\ a$ constraint. Accordingly, the following function type-checks (where $show :: Show\ a \Rightarrow a \to String$ is a standard library function):

[4] `http://blog.omega-prime.co.uk/?p=127`

```
showSomething :: Some Show → String
showSomething (Some thing) = show thing
```

Note that there is no *Show a* constraint in the function signature—we get the constraint from pattern-matching on *Some*, instead.

The type *Some* is useful if, say, we want a heterogeneous list such that every element of the list satisfies some constraint. That is, each element of [*Some Show*] can be a different type *a*, as long as *Show a* holds:

```
heteroList :: [Some Show]
heteroList = [Some True, Some (5 :: Int), Some (Just ())]
printList :: [Some Show] → String
printList things = "[" ++ intercalate ", " (map showSomething things) ++ "]"


λ> putStrLn $ printList heteroList
[True, 5, Just ()]
```

2.4 Generalized algebraic datatypes

Generalized algebraic datatypes (or GADTs) are a powerful feature that allows term-level pattern matches to refine information about types. They undergird much of the programming we will see in the examples in Chapter 3, and so I defer most of the discussion of GADTs to that chapter.

Here, I introduce one particularly important GADT: propositional equality. The following definition appears now as part of the standard library shipped with GHC, in the *Data.Type.Equality* module:

```
data (a :: k) :~: (b :: k) where
  Refl :: a :~: a
```

The idea here is that a value of type $\tau :\sim: \sigma$ (for some τ and σ) represents evidence that the type τ is in fact equal to the type σ. Here is a use of this type, also from *Data.***Type**.*Equality*:

```
castWith :: (a :~: b) → a → b
castWith Refl x = x
```

Here, the *castWith* function takes a term of type *a* :~: *b*—evidence that *a* equals *b*—and a term of type *a*. It can immediately return this term, *x*, because GHC knows that *a* and *b* are the same type. Thus, *x* also has type *b* and the function is well typed.

Note that *castWith* must pattern-match against *Refl*. The reason this is necessary becomes more apparent if we look at an alternate, entirely equivalent way of defining (:~:):

```
data (a :: k) :~: (b :: k) where
  Refl :: (a ~ b) ⇒ a :~: b
```

In this variant, I define the type using the Haskell98-style syntax for datatypes. This says that the *Refl* constructor takes no arguments, but does require the constraint that $a \sim b$. The constraint $(\sim)$ is GHC's notation for a proper type equality constraint. Accordingly, to use *Refl* at a type $\tau :\sim: \sigma$, GHC must know that $\tau \sim \sigma$—in other words, that τ and σ are the same type. When *Refl* is matched against, this constraint $\tau \sim \sigma$ becomes available for use in the body of the pattern match.

Returning to *castWith*, pattern-matching against *Refl* brings $a \sim b$ into the context, and GHC can apply this equality in the right-hand side of the equation to say that x has type b.

Operationally, the pattern-match against *Refl* is also important. This match is what forces the equality evidence to be reduced to a value. As Haskell is a lazy language, it is possible to pass around equality evidence that is $\bot$. Matching evaluates the argument, making sure that the evidence is real. The fact that type equality evidence must exist and be executed at runtime is somewhat unfortunate. See Section 3.3.3 and Section 4.4.5 for some discussion.

2.5 Higher-rank types

Standard ML and Haskell98 both use, essentially, the Hindley-Milner (HM) type system [20, 43, 63]. The HM type system allows only *prenex quantification*, where a type can quantify over type variables only at the very top. The system is based on *types*, which have no quantification, and *type schemes*, which do:

$$\begin{array}{ll} \tau ::= \alpha \mid H \mid \tau_1\ \tau_2 & \text{types} \\ \sigma ::= \forall \alpha.\sigma \mid \tau & \text{type schemes} \end{array}$$

Here, I use α to stand for any of a countably infinite set of type variables and H to stand for any type constant (including $(\rightarrow)$).

Let-bound definitions in HM are assigned type schemes; lambda-bound definitions are assigned monomorphic types, only. Thus, in HM, it is appropriate to have a function *length* :: ∀ *a*. [*a*] → *Int* but disallowed to have one like *bad* :: (∀ *a*. *a* → *a* → *a*) → *Int*: *bad*'s type has a ∀ somewhere other than at the top of the type. This type is of the second rank, and is forbidden in HM.

On the other hand, today's GHC allows types of arbitrary rank. Though a full example of the usefulness of this ability would take us too far afield, Lämmel and Peyton Jones [53] and Washburn and Weirich [103] (among others) make critical use of this ability. The cost, however, is that higher-rank types cannot be inferred. For this reason, this definition of *higherRank*

```
higherRank f = (f True, f 'x')
```

will not compile without a type signature. Without the signature, GHC tries to unify the types *Char* and *Bool*, failing. However, providing a signature

```
higherRank :: (∀ a. a → a) → (Bool, Char)
```

does the trick nicely.

Type inference in the presence of higher-rank types is well studied, and can be made practical via bidirectional type-checking [24, 74].

2.6 Scoped type variables

A modest, but subtle, extension in GHC is `ScopedTypeVariables`, which allows a programmer to refer back to a declared type variable from within the body of a function. As dealing with scoped type variables can be a point of confusion for Haskell type-level programmers, I include a discussion of it here.

Consider this implementation of the left fold *foldl*:

```
foldl :: (b → a → b) → b → [a] → b
foldl f z0 xs0 = lgo z0 xs0
  where
    lgo z []     = z
    lgo z (x:xs) = lgo (f z x) xs
```

It can be a little hard to see what is going on here, so it would be helpful to add a type signature to the function *lgo*, thus:

```
lgo :: b → [a] → b
```

Yet, doing so leads to type errors. The root cause is that the *a* and *b* in *lgo*'s type signature are considered independent from the *a* and *b* in *foldl*'s type signature. It is as if we've assigned the type *b0* → [*a0*] → *b0* to *lgo*. Note that *lgo* uses *f* in its definition. This *f* is a parameter to the outer *foldl*, and it has type *b* → *a* → *b*. When we call *f z x* in *lgo*, we're passing *z* :: *b0* and *x* :: [*a0*] to *f*, and type errors ensue.

To make the *a* and *b* in *foldl*'s signature be lexically scoped, we simply need to quantify them explicitly. Thus, the following gets accepted:

```
foldl :: ∀ a b. (b → a → b) → b → [a] → b
foldl f z0 xs0 = lgo z0 xs0
  where
    lgo :: b → [a] → b
    lgo z []     = z
    lgo z (x:xs) = lgo (f z x) xs
```

Another particular tricky point around `ScopedTypeVariables` is that GHC will not warn you if you are missing this extension.

2.7 Functional dependencies

Although this dissertation does not dwell much on functional dependencies, I include them here for completeness.

Functional dependencies are GHC's earliest feature introduced to enable rich type-level programming [49, 88]. They are, in many ways, a competitor to type families. With functional dependencies, we can declare that the choice of one parameter to a type class fixes the choice of another parameter. For example:

```
class Pred (a :: Nat) (b :: Nat) | a → b
instance Pred 'Zero     'Zero
instance Pred ('Succ n) n
```

In the declaration for class *Pred* ("predecessor"), we say that the first parameter, *a*, determines the second one, *b*. In other words, *b* has a functional dependency on *a*. The two instance declarations respect the functional dependency, because there are no two instances where the same choice for *a* but differing choices for *b* are made.

Functional dependencies are, in some ways, more powerful than type families. For example, consider this definition of *Plus*:

```
class Plus (a :: Nat) (b :: Nat) (r :: Nat) | a b → r, r a → b
instance                 Plus 'Zero     b b
instance Plus a b r ⇒ Plus ('Succ a) b ('Succ r)
```

The functional dependencies for *Plus* are more expressive than what we can do for type families. (However, see the work of Stolarek et al. [86], which attempts to close this gap.) They say that *a* and *b* determine *r*, just like the arguments to a type family determine the result, but also that *r* and *a* determine *b*. Using this second declared functional dependency, if we know *Plus a b r* and *Plus a b′ r*, we can conclude $b = b'$. Although the functional dependency $r\ b \to a$ also holds, GHC is unable to prove this and thus we cannot declare it.

Functional dependencies have enjoyed a rich history of aiding type-level programming [52, 59, 70]. Yet, they require a different paradigm to much of functional programming. When writing term-level definitions, functional programmers think in terms of functions that take a set of arguments and produce a result. Functional dependencies, however, encode type-level programming through relations, not proper functions. Though both functional dependencies and type families have their place in the Haskell ecosystem, I have followed the path taken by other dependently typed languages and use type-level functions as the main building blocks of Dependent Haskell, as opposed to functional dependencies.

Chapter 3

Motivation

Functional programmers use dependent types in two primary ways, broadly speaking: in order to prevent erroneous programs from being accepted, and in order to write programs that a simply typed language cannot accept. In this chapter, I will motivate the use of dependent types from both of these angles. The chapter concludes with a section motivating why Haskell, in particular, is ripe for dependent types.

As a check for accuracy in these examples and examples throughout this dissertation, all the indented, typeset code is type-checked against my implementation every time the text is typeset.

The code snippets throughout this dissertation are presented on a variety of background colors. A white background indicates code that works in GHC 7.10 and (perhaps) earlier. A light green background highlights code that newly works in GHC 8.0 due to my implementations of previously published papers [33, 105]. A light yellow background indicates code that does not work verbatim in GHC 8.0, but could still be implemented via the use of singletons [30] and similar workarounds. A light red background marks code that does not currently work in due to bugs. To my knowledge, there is nothing more than engineering (and perhaps the use of singletons) to get these examples working.

Beyond the examples presented here, the literature is accumulating a wide variety of examples of dependently typed programming. Particularly applicable are the examples in Oury and Swierstra [69], Lindley and McBride [56], and Gundry [37, Chapter 8].

3.1 Eliminating erroneous programs

3.1.1 Simple example: Length-indexed vectors

We start by examining length-indexed vectors. This well-worn example is still useful, as it is easy to understand and still can show off many of the new features of Dependent Haskell.

3.1.1.1 *Vec* definition

Here is the definition of a length-indexed vector:

```
data Nat = Zero | Succ Nat   -- first, some natural numbers
data Vec :: Type → Nat → Type where
  Nil  :: Vec a 'Zero
  (:>) :: a → Vec a n → Vec a ('Succ n)
infixr 5 :>
```

I will use ordinary numerals as elements of *Nat* in this text.[5] The *Vec* type is parameterized by both the type of the vector elements and the length of the vector. Thus *True* :> *Nil* has type *Vec Bool* 1 and `'x'` :> `'y'` :> `'z'` :> *Nil* has type *Vec Char* 3.

While *Vec* is a fairly ordinary GADT, we already see one feature newly introduced by my work: the use of **Type** in place of ⋆. Using ⋆ to classify ordinary types is troublesome because ⋆ can also be a binary operator. For example, should *F* ⋆ *Int* be a function *F* applied to ⋆ and *Int* or the function ⋆ applied to *F* and *Int*? In order to avoid getting caught on this detail, Dependent Haskell introduces **Type** to classify ordinary types. (Section 7.4 discusses a migration strategy from legacy Haskell code that uses ⋆.)

Another question that may come up right away is about my decision to use *Nat*s in the index. Why not *Integer*s? In Dependent Haskell, *Integer*s are indeed available in types. However, since we lack simple definitions for *Integer* operations (for example, what is the body of *Integer*'s + operation?), it is hard to reason about them in types. This point is addressed more fully in Section 7.5. For now, it is best to stick to the simpler *Nat* type.

3.1.1.2 *append*

Let's first write an operation that appends two vectors. We already need to think carefully about types, because the types include information about the vectors' lengths. In this case, if we combine a *Vec a n* and a *Vec a m*, we had surely better get a *Vec a* $(n+m)$. Because we are working over our *Nat* type, we must first define addition:

$$
\begin{aligned}
&(+) :: \mathit{Nat} \to \mathit{Nat} \to \mathit{Nat} \\
&\mathit{Zero} \quad + m = m \\
&\mathit{Succ}\ n + m = \mathit{Succ}\ (n + m)
\end{aligned}
$$

Now that we have worked out the hard bit in the type, appending the vectors themselves is easy:

[5] In contrast, numerals used in types in GHC are elements of a built-in type *Nat* that uses a more efficient binary representation. It cannot be pattern-matched against.

```
append :: Vec a n → Vec a m → Vec a (n '+ m)
append Nil       w = w
append (a :> v) w = a :> (append v w)
```

There is a curiosity in the type of *append*: the addition between *n* and *m* is performed by the operation '+. Yet we have defined the addition operation +. What's going on here?

Haskell maintains two separate namespaces: one for types and one for terms. Doing so allows declarations like **data** $X = X$, where the data constructor X has type X. With Dependent Haskell, however, terms may appear in types. (And types may, less frequently, appear in terms; see Section 3.1.3.2.) We thus need a mechanism for telling the compiler which namespace we want. In a construct that is syntactically a type (that is, appearing after a :: marker or in some other grammatical location that is "obviously" a type), the default namespace is the type namespace. If a user wishes to use a term-level definition, the term-level definition is prefixed with a '. Thus, '+ simply uses the term-level + in a type. Note that the ' mark has no semantic content—it is *not* a promotion operator. It is simply a marker in the source code to denote that the following identifier lives in the term-level namespace.

The fact that Dependent Haskell allows us to use our old, trusty, term-level + in a type is one of the two chief qualities that makes it a dependently typed language.

3.1.1.3 *replicate*

Let's now write a function that can create a vector of a given length with all elements equal. Before looking at the function over vectors, we'll start by considering a version of this function over lists:

```
listReplicate :: Nat → a → [a]
listReplicate Zero     _ = []
listReplicate (Succ n) x = x : listReplicate n x
```

With vectors, what will the return type be? It surely will mention the element type *a*, but it also has to mention the desired length of the list. This means that we must give a name to the *Nat* passed in. Here is how it is written in Dependent Haskell:

```
replicate :: ∀ a. Π (n :: Nat) → a → Vec a n
replicate Zero     _ = Nil
replicate (Succ n) x = x :> replicate n x
```

The first argument to *replicate* is bound by Π (*n* :: *Nat*). Such an argument is available for pattern matching at runtime but is also available in the type. We see the value *n*

used in the result *Vec a n*. This is an example of a dependent pattern match, and how this function is well-typed is considered is some depth in Section 4.3.3.

The ability to have an argument available for runtime pattern matching and compile-time type checking is the other chief quality that makes Dependent Haskell dependently typed.

3.1.1.4 Invisibility in *replicate*

The first parameter to *replicate* above is actually redundant, as it can be inferred from the result type. We can thus write a version with this type:

```
replicateInvis :: Π (n :: Nat). ∀ a. a → Vec a n
```

Note that the type begins with Π (*n* :: *Nat*). instead of Π (*n* :: *Nat*) →. The use of the . there recalls the existing Haskell syntax of ∀ *a*. , which denotes an invisible argument *a*. Invisible arguments are omitted at function calls and definitions. On the other hand, the → in Π (*n* :: *Nat*) → means that the argument is visible and must be provided at every function invocation and defining equation. This choice of syntax is due to Gundry [37]. Some readers may prefer the terms *explicit* and *implicit* to describe visibility; however, these terms are sometimes used in the literature (e.g., [64]) when talking about erasure properties. I will stick to *visible* and *invisible* throughout this dissertation.

We can now use type inference to work out the value of *n* that should be used:

```
fourTrues :: Vec Bool 4
fourTrues = replicateInvis True
```

How should we implement *replicateInvis*, however? We need to use an *invisibility override*. The implementation looks like this:

```
replicateInvis @Zero     _ = Nil
replicateInvis @(Succ _) x = x :> replicateInvis x
```

The @ in those patterns means that we are writing an ordinarily invisible argument visibly. This is necessary in the body of *replicateInvis* as we need to pattern match on the choice of *n*. An invisibility override can also be used at call sites: *replicateInvis* @2 `'q'` produces the vector `'q'` :> `'q'` :> *Nil* of type *Vec Char* 2. It is useful when we do not know the result type of a call to *replicateInvis*.[6]

[6]The use of @ here is a generalization of its use in GHC 8 in visible type application [33].

3.1.1.5 Computing the length of a vector

Given a vector, we would like to be able to compute its length. At first, such an idea might seem trivial—the length is right there in the type! However, we must be careful here. While the length is indeed in the type, types are erased in Haskell. That length is thus not automatically available at runtime for computation. We have two choices for our implementation of *length*:

```
lengthRel :: Π n. ∀ a. Vec a n → Nat
lengthRel @n _ = n
```

```
lengthIrrel :: ∀ n a. Vec a n → Nat
lengthIrrel Nil       = 0
lengthIrrel (_ :> v) = 1 + lengthIrrel v
```

The difference between these two functions is whether or not they quantify *n* relevantly. A *relevant* parameter, bound by Π, is one available at runtime.[7] In *lengthRel*, the type declares that the value of *n*, the length of the *Vec a n* is available at runtime. Accordingly, *lengthRel* can simply return this value. The one visible parameter, of type *Vec a n* is needed only so that type inference can infer the value of *n*. This value must be somehow known at runtime in the calling context, possibly because it is statically known (as in *lengthRel fourTrues*) or because *n* is available relevantly in the calling function.

On the other hand, *lengthIrrel* does not need runtime access to *n*; the length is computed by walking down the vector and counting the elements. When *lengthRel* is available to be called, both *lengthRel* and *lengthIrrel* should always return the same value. (In contrast, *lengthIrrel* is always available to be called.)

The choice of relevant vs. irrelevant parameter is denoted by the use of Π or ∀ in the type: *lengthRel* says Π *n* while *lengthIrrel* says ∀ *n*. The programmer must choose between relevant and irrelevant quantification when writing or calling functions. (See Section 8.7 for a discussion of how this choice relates to decisions in other dependently typed languages.)

We see also that *lengthRel* takes *n* before *a*. Both are invisible, but the order is important because we wish to bind the first one in the body of *lengthRel*. If I had written *lengthRel*'s type beginning with ∀ *a*. Π *n*. , then the body would have to be *lengthRel* @_ @*n* _ = *n*.

3.1.1.6 Conclusion

These examples have warmed us up to examine more complex uses of dependent types in Haskell. We have seen the importance of discerning the relevance of a parameter, invisibility overrides, and dependent pattern matching.

[7] This is a slight simplification, as relevance still has meaning in types that are erased. See Section 4.2.2.

3.1.2 A strongly typed simply typed λ-calculus interpreter

It is straightforward to write an interpreter for the simply typed λ-calculus (STLC) in Haskell. However, how can we be sure that our interpreter is written correctly? Using some features of dependent types—notably, generalized algebraic datatypes, or GADTs—we can incorporate the STLC's type discipline into our interpreter.[8] Using the extra features in Dependent Haskell, we can then write both a big-step semantics and a small-step semantics and have GHC check that they correspond.

3.1.2.1 Type definitions

Our first step is to write a type to represent the types in our λ-calculus:

```
data Ty = Unit | Ty :⇝ Ty
infixr 0 :⇝
```

I choose *Unit* as our one and only base type, for simplicity. This calculus is clearly not suitable for computation, but it demonstrates the use of GADTs well. The model described here scales up to a more featureful λ-calculus.[9] The **infixr** declaration declares that the constructor :⇝ is right-associative, as usual.

We are then confronted quickly with the decision of how to encode bound variables. Let's choose de Bruijn indices [21], as these are well known and conceptually simple. However, instead of using natural numbers to represent our variables, we'll use a custom *Elem* type:

```
data Elem :: [a] → a → Type where
  EZ ::                Elem (x ': xs) x
  ES :: Elem xs x → Elem (y ': xs) x
```

A value of type *Elem xs x* is a proof that *x* is in the list *xs*. This proof naturally takes the form of a natural number, naming the place in *xs* where *x* lives. The first constructor *EZ* is a proof that *x* is the first element in *x* ': *xs*. The second constructor *ES* says that, if we know *x* is an element in *xs*, then it is also an element in *y* ': *xs*.

We can now write our expression type:

```
data Expr :: [Ty] → Ty → Type where
  Var  :: Elem ctx ty                                → Expr ctx ty
  Lam  :: Expr (arg ': ctx) res                      → Expr ctx (arg ':⇝ res)
  App  :: Expr ctx (arg ':⇝ res) → Expr ctx arg → Expr ctx res
  TT   ::                                            Expr ctx 'Unit
```

As with *Elem list elt*, a value of type *Expr ctx ty* serves two purposes: it records the structure of our expression, *and* it proves a property, namely that the expression is

[8] The skeleton of this example—using GADTs to verify the implementation of the STLC—is not novel, but I am unaware of a canonical reference for it.

[9] For example, see my work on glambda at `https://github.com/goldfirere/glambda`.

well-typed in context *ctx* with type *ty*. Indeed, with some practice, we can read off the typing rules for the simply typed λ-calculus direct from *Expr*'s definition. In this way, it is impossible to create an ill-typed *Expr*.

3.1.2.2 Big-step evaluator

We now wish to write both small-step and big-step operational semantics for our expressions. First, we'll need a way to denote values in our language:

```
data Val :: Ty → Type where
  LamVal :: Expr '[arg] res → Val (arg ':⇝ res)
  TTVal  ::                   Val 'Unit
```

Our big-step evaluator has a straightforward type:

```
eval :: Expr '[] ty → Val ty
```

This type says that a well-typed, closed expression (that is, the context is empty) can evaluate to a well-typed value of the same type *ty*. Only a type-preserving evaluator will have that type, so GHC can check the type-soundness of our λ-calculus as it compiles our interpreter.

To implement *eval*, we'll need several auxiliary functions, each with an intriguing type:

```
  -- Shift the de Bruijn indices in an expression
shift :: ∀ ctx ty x. Expr ctx ty → Expr (x ': ctx) ty
  -- Substitute one expression into another
subst :: ∀ ctx s ty. Expr ctx s → Expr (s ': ctx) ty → Expr ctx ty
  -- Perform β-reduction
apply :: Val (arg ':⇝ res) → Expr '[] arg → Expr '[] res
```

The type of *shift* is precisely the content of a weakening lemma: that we can add a type to a context without changing the type of a well-typed expression. The type of *subst* is precisely the content of a substitution lemma: that given an expression of type *s* and an expression of type *t* (typed in a context containing a variable bound to *s*), we can substitute and get a new expression of type *t*. The type of *apply* shows that it does β-reduction: it takes an abstraction of type *arg* ':⇝ *res* and an argument of type *arg*, producing a result of type *res*.

The implementations of these functions, unsurprisingly, read much like the proof of the corresponding lemmas. We even have to "strengthen the induction hypothesis" for *shift* and *subst*; we need an internal recursive function with extra arguments. Here are the first few lines of *shift* and *subst*:

```
shift = go []
  where
    go :: ∀ ty. Π ctx₀ → Expr (ctx₀ '++ ctx) ty → Expr (ctx₀ '++ x ': ctx) ty
    go = ...
subst e = go []
  where
    go :: ∀ ty. Π ctx₀ → Expr (ctx₀ '++ s ': ctx) ty → Expr (ctx₀ '++ ctx) ty
    go = ...
```

As many readers will be aware, to prove the weakening and substitution lemmas, it is necessary to consider the possibility that the context change is not happening at the beginning of the list of types, but somewhere in the middle. This generality is needed in the *Lam* case, where we wish to use an induction hypothesis; the typing rule for *Lam* adds the type of the argument to the context, and thus the context change is no longer at the beginning of the context.

Naturally, this issue comes up in our interpreter's implementation, too. The *go* helper functions have types generalized over a possibly non-empty context prefix, ctx_0. This context prefix is appended to the existing context using '++, the promoted form of the existing ++ list-append operator. (Using ' for promoting functions is a natural extension of the existing convention of using ' to promote constructors from terms to types; see also Section 3.1.1.2.) The *go* functions also Π-quantify over ctx_0, meaning that the value of this context prefix is available in types (as we can see) and also at runtime. This is necessary because the functions need the length of ctx_0 at runtime, in order to know how to shift or substitute. Note also the syntax Π ctx_0 →, where the Π-bound variable is followed by an →. The use of an arrow here (as opposed to a .) indicates that the parameter is *visible* in source programs; the empty list is passed in visibly in the invocation of *go*. (See also Section 4.2.3.) The final interesting feature of these types is that they re-quantify *ty*. This is necessary because the recursive invocations of the functions may be at a different type than the outer invocation. The other type variables—which do not change during recursive calls to the *go* helper functions—are lexically bound by the ∀ in the type signature of the outer function.

The implementation of these functions is fiddly and uninteresting, and is omitted from this text. However, writing this implementation is made much easier by the precise types. If I were to make a mistake in the delicate de Bruijn shifting operation, I would learn of my mistake immediately, without any testing. In an algorithm so easy to get wrong, this feedback is wonderful, indeed.

With all of these supporting functions written, the evaluator itself is dead simple:

```
eval (Var v)      = case v of {}   -- no variables in an empty context
eval (Lam body)   = LamVal body
eval (App e1 e2)  = eval (apply (eval e1) e2)
eval TT           = TTVal
```

The only curiosity here is the empty **case** expression in the *Var* case, which eliminates *v* of the uninhabited type *Elem* '[] *ty*.

3.1.2.3 Small-step stepper

We now turn to writing the small-step semantics. We could proceed in a very similar fashion to the big-step semantics, by defining a *step* function that steps an expression either to another expression or to a value. But we want better than this.

Instead, we want to ensure that the small-step semantics respects the big-step semantics. That is, after every step, we want the value—as given by the big-step semantics—to remain the same. We thus want the small-step stepper to return a custom datatype, marrying the result of stepping with evidence that the value of this result agrees with the value of the original expression:[10]

```
data StepResult :: Expr '[] ty → Type where
  Stepped :: Π (e' :: Expr '[] ty) → ('eval e ∼ 'eval e') ⇒ StepResult e
  Value   :: Π (v  :: Val      ty) → ('eval e ∼ v)        ⇒ StepResult e
```

A *StepResult e* is the result of stepping an expression *e*. It either contains a new expression *e'* whose value equals *e*'s value, or it contains the value *v* that is the result of evaluating *e*.

An interesting detail about these constructors is that they feature an equality constraint *after* a runtime argument. Currently, GHC requires that all data constructors take a sequence of type arguments, followed by constraints, followed by regular arguments. Generalizing this form poses no real difficulty, however.

With this in hand, the *step* function is remarkably easy to write:

```
step :: Π (e :: Expr '[] ty) → StepResult e
step (Var v)       = case v of {}   -- no variables in an empty context
step (Lam body)    = Value (LamVal body)
step (App e1 e2)   = case step e1 of
  Stepped e1' → Stepped (App e1' e2)
  Value v     → Stepped (apply v e2)
step TT            = Value TTVal
```

[10] This example fails for two reasons:

- It contains data constructors with constraints occurring after visible parameters, but GHC imposes rigid requirements on the shape of data constructor types.
- Writing a type-level version of *shift* (automatic promotion with ' is not yet implemented) is not yet possible. The problem is that one of the helper function's arguments has a type that mentions the + function, a feature that is not yet implemented.

I do not expect fixing either of these problems to be a significant challenge.

Due to GHC's ability to freely use equality assumptions, *step* requires no explicit manipulation of equality proofs. Let's look at the *App* case above. We first check whether or not *e1* can take a step. If it can, we get the result of the step *e1'* *and* a proof that *'eval e1* $\sim$ *'eval e1'*. This proof enters into the type-checking context and is invisible in the program text. On the right-hand side of the match, we conclude that *App e1 e2* steps to *App e1' e2*. This requires a proof that *'eval* (*App e1 e2*) $\sim$ *'eval* (*App e1' e2*). Reducing *'eval* on both sides of that equality gives us

$$\textit{'eval}\ (\textit{'apply}\ (\textit{'eval}\ \textit{e1})\ \textit{e2}) \sim \textit{'eval}\ (\textit{'apply}\ (\textit{'eval}\ \textit{e1'})\ \textit{e2}).$$

Since we know *'eval e1* $\sim$ *'eval e1'*, however, this equality is easily solvable; GHC does the heavy lifting for us. Similar reasoning proves the equality in the second branch of the **case**, and the other clauses of *step* are straightforward.

The ease with which these equalities are solved is unique to Haskell. I have translated this example to Coq, Agda, and Idris; each has its shortcomings:

- Coq deals quite poorly with indexed types, such as *Expr*. The problem appears to stem from Coq's weak support for dependent pattern matching. For example, if we inspect a *ctx* to discover that it is empty, Coq, by default, forgets the equality $\textit{ctx} = []$. It then, naturally, fails to use the equality to rewrite the types of the right-hand sides of the pattern match. This can be overcome through various tricks, but it is far from easy. Alternatively, Coq's relatively new **Program** construct helps with this burden somewhat but still does not always work as smoothly as GADT pattern matching in Haskell. Furthermore, even once the challenges around indexed types are surmounted, it is necessary to prove that *eval* terminates—a non-trivial task—for Coq to accept the function.

- Agda does a better job with indexed types, but it is not designed around implicit proof search. A key part of Haskell's elegance in this example is that pattern-matching on a *StepResult* reveals an equality proof to the type-checker, and this proof is then used to rewrite types in the body of the pattern match. This all happens without any direction from the programmer. In Agda, the equality proofs must be unpacked and used with Agda's **rewrite** tactic.

  Like Coq, Agda normally requires that functions terminate. However, we can easily disable the termination checker: use `{-# NO_TERMINATION_CHECK #-}`.

- Like Agda, Idris works well with indexed types. The *eval* function is, unsurprisingly, inferred to be partial, but this is easy enough to fix with a well-placed **assert_total**. However, Idris's proof search mechanism is unable to find proofs that *step* is correct in the *App* cases. (Using an **auto** variable, Idris is able to find the proofs automatically in the other *step* clauses.) Idris comes the closest to Haskell's brevity in this example, but it still requires two places where equality proofs must be explicitly manipulated.

3.1.2.4 Conclusion

We have built up a small-step stepper whose behavior is verified against a big-step evaluator. Despite this extra checking, the *step* function will run in an identical manner to one that is unchecked—there is no runtime effect of the extra verification. We can be sure of this because we can audit the types involved and see that only the expression itself is around at runtime; the rest of the arguments (the indices and the equality proofs) are erased. Furthermore, getting this all done is easier and more straightforward in Dependent Haskell than in the other three dependently typed languages I tried. Key to the ease of encoding in Haskell is that Haskell does not worry about termination (see Section 3.3.3) and has an aggressive rewriting engine used to solve equality predicates.

3.1.3 Type-safe database access with an inferred schema

Many applications need to work in the context of some external database. Haskellers naturally want their interface to the database to be well-typed, and there already exist libraries that use (non-dependent) Haskell's fancy types to good effect for database access. (See opaleye[11] for an advanced, actively developed and actively used example of such a library.) Dependent Haskell allows us to go one step further and use type inference to infer a database schema from the database access code.

This example is inspired by the third example by Oury and Swierstra [69]; the full code powering the example is available online.[12]

Instead of starting with the library design, let's start with a concrete use case. Suppose we are writing an information system for a university. The current task is to write a function that, given the name of a professor, prints out the names of students in that professor's classes. There are two database tables of interest, exemplified in Figure 3.1 on the following page. Our program will retrieve a professor's record and then look up the students by their ID number.

Our goal in this example is understanding the broad strokes of how the database library works and what it is capable of, not all the precise details. If you wish to understand more, please check out the full source code online.

3.1.3.1 Accessing the database

The main worker function that retrieves and processes the information of interest from the database is *queryDB*, in Figure 3.2 on the next page. Note that this function is not assigned a type signature; we'll return to this interesting point in Section 3.1.3.2. The *queryDB* function takes in the schemas for the two tables it will retrieve the data from. It loads the tables that correspond to the schemas; the *loadTable* function makes sure that the table (as specified by its filename) does indeed correspond to the schema. An

[11] `https://github.com/tomjaguarpaw/haskell-opaleye`
[12] `https://github.com/goldfirere/dependent-db`

The `students` table:

last	first	id	gradyear
"Matthews"	"Maya"	1	2018
"Morley"	"Aimee"	2	2017
"Barnett"	"William"	3	2020
"Leonard"	"Sienna"	4	2019
"Oliveira"	"Pedro"	5	2017
"Peng"	"Qi"	6	2020
"Chakraborty"	"Sangeeta"	7	2018
"Yang"	"Rebecca"	8	2019

The `classes` table:

name	students	course
"Blank"	[2,3,7,8]	"Robotics"
"Eisenberg"	[1,2,5,8]	"Programming Languages"
"Kumar"	[3,6,7,8]	"Artificial Intelligence"
"Xu"	[1,3,4,5]	"Graphics"

Figure 3.1: Database tables used in Section 3.1.3.

```
type NameSchema = [Col "first" String, Col "last" String]
printName :: Row NameSchema → IO ()
printName (first ::> last ::> _) = putStrLn (first ++ "  " ++ last)
queryDB classes_sch students_sch = do
  classes_tab  ← loadTable "classes.table"  classes_sch
  students_tab ← loadTable "students.table" students_sch
  putStr "Whose students do you want to see? "
  prof ← getLine
  let joined
        = project $
          select (field @"id" @Int `elementOf` field @"students") $
          product (select (field @"prof" === literal prof)
                          (read classes_tab))
                  (read students_tab)
  rows ← query joined
  mapM_ printName rows
```

Figure 3.2: The *queryDB* function

```
data Column = Col String Type
type Schema = [Column]
data Table :: Schema → Type   -- a table according to a schema
data RA    :: Schema → Type   -- a Relational Algebra
data Expr  :: Schema → Type → Type   -- an expression
loadTable  :: String → Π (s :: Schema) → IO (Table s)
project    :: Subset s' s ⇒ RA s → RA s'
select     :: Expr s Bool → RA s → RA s
field      :: ∀ name ty s. In name ty s ⇒ Expr s ty
elementOf  :: Eq ty ⇒ Expr s ty → Expr s [ty] → Expr s Bool
product    :: 'disjoint s s' ~ 'True ⇒ RA s → RA s → RA (s '++ s')
literal    :: ty → Expr s ty
read       :: Table s → RA s
```

Figure 3.3: Types used in the example of Section 3.1.3.

I/O interaction with the user then ensues, resulting in a variable *prof* of type *String* containing the desired professor's name.

The *joined* variable then gets assigned to a large query against the database. This query:

1. reads in the `classes` table,
2. selects out any rows that mention the desired *prof*,
3. computes the Cartesian product of these rows and all the rows in the `students` table,
4. selects out those rows where the `id` field is in the `students` list,
5. and finally projects out the name of the student.

The types of the components of this query are in Figure 3.3. There are a few points of interest in looking at this code:

- The query is well-typed by construction. Note the intricate types appearing in Figure 3.3. For example, *select* takes an expression used to select which rows of a table are preserved. This operation naturally requires an *Expr s Bool*, where *s* is the schema of interest and the *Bool* indicates that we have a Boolean expression (as opposed to one that results in a number, say). The *RA* type does not permit ill-typed queries, such as taking the Cartesian product of two tables with overlapping column names (see the type of *product*), as projections from such a combination would be ambiguous.

- Use of *field* requires the @ invisibility override marker, as we wish to specify the name of the field.
- In the first *select* expression, we must specify the type of the field as well as the name, whereas in the second *select* expression, we can omit the type. In the second case, the type can be inferred by comparison with the literal *prof*. In the first, type inference tells us that `id` is the element type of `students`, but we need to be more concrete than this—hence the @*Int* passed to *field*.
- The use of *project* at the top projects out the first and last name of the student, even though neither `first` nor `last` is mentioned there. Type inference does the work for us, as we pass the result of running the query to *printName*, which has a type signature that states it works over only names.

3.1.3.2 Inferring a schema

Type inference works to infer the type of *queryDB*, assigning it this whopper:

```
λ> :type queryDB
queryDB
  :: Π (s :: Schema) (s' :: Schema)
  → ( 'disjoint s s' ~ 'True, In "students" [Int] (s '++ s'),
      In "prof" String s, In "last" [Char] (s '++ s'),
      In "id" Int (s '++ s'), In "first" [Char] (s '++ s'))
  ⇒ IO ()
```

The cavalcade of constraints are all inferred from the query above quite straightforwardly.[13] But how can we call *queryDB* satisfying all of these constraints?

The call to *queryDB* appears here:

```
main :: IO ()
main = do classes_sch  ← loadSchema "classes.schema"
          students_sch ← loadSchema "students.schema"
          $ (checkSchema 'queryDB [ 'classes_sch, 'students_sch])
```

[13] What may be more surprising to the skeptical reader is that a Π-type is inferred, especially if you have already read Chapter 6. However, I maintain that the Bake algorithm in Chapter 6 infers this type. The two parameters to *queryDB* are clearly *Schema*s, and the body of *queryDB* asserts constraints on these *Schema*s. Note that the type inference algorithm infers only relevant, visible parameters, but these arguments are indeed relevant and visible. The dependency comes in after solving, when the quantification telescope Δ output by the solver has constraints depend on a visible argument.

As further justification for stating that Bake infers this type, GHC infers a type quite like this today, albeit using singletons. The appearance of singletons in the type inferred today is why this snippet is presented on a light yellow background.

The two calls to *loadSchema* are uninteresting. The third line of *main* is a Template Haskell [83] splice. Template Haskell is GHC's metaprogramming facility. The quotes we see before the arguments to *checkSchema* are Template Haskell quotes, not the promotion ' mark we have seen so much.

The function *checkSchema* :: *Name* → [*Name*] → *Q Exp* takes the name of a function (*queryDB*, in our case), names of schemas to be passed to the function (*classes_sch* and *students_sch*) and produces some Haskell code that arranges for an appropriate function call. (*Exp* is the Template Haskell type containing a Haskell expression, and *Q* is the name of the monad Template Haskell operates under.) In order to produce the right function call to *queryDB*, *checkSchema* queries for the inferred type of *queryDB*. It then examines this type and extracts out all of the constraints on the schemas. In the produced Haskell expression, *checkSchema* arranges for calls to several functions that establish the constraints before calling *queryDB*. To be concrete, here is the result of the splice; the following code is spliced into the *main* function in place of the call to *checkSchema*:

```
checkDisjoint classes_sch students_sch                          $
checkIn "students" ^[^Int]    (classes_sch ++ students_sch) $
checkIn "prof"     ^String    classes_sch                   $
checkIn "last"     ^[^Char]   (classes_sch ++ students_sch) $
checkIn "id"       ^Int       (classes_sch ++ students_sch) $
checkIn "first"    ^[^Char]   (classes_sch ++ students_sch) $
queryDB classes_sch students_sch
```

Before discussing *checkDisjoint* and *checkIn*, I must explain a new piece of syntax: just as ' allows us to use a term-level name in a type, the new syntax ^ allows us to use a type-level name in a term. That is all the syntax does. For example ^[^*Int*] is the list type constructor applied to the type *Int*, not a one-element list (as it would otherwise appear).

The *checkDisjoint* and *checkIn* functions establish the constraints necessary to call *queryDB*. Here are their types:

```
checkDisjoint :: Π (sch1 :: Schema) (sch2 :: Schema)
              → (('disjoint sch1 sch2 ~ 'True) ⇒ r)
              → r
checkIn       :: Π (name :: String) (ty :: Type) (schema :: Schema)
              → (In name ty schema ⇒ r)
              → r
```

Both functions take input information[14] to validate and a continuation to call if indeed

[14]Readers might be alarmed to see here a **Type** being passed at runtime. After all, a key feature

the input is valid. In this implementation, both functions simply error (that is, return $\bot$) if the input is not valid, though it would not be hard to report an error in a suitable monad.

3.1.3.3 Checking inclusion in a schema

It is instructive to look at the implementation of *checkIn*:

```
checkIn :: Π (name :: String) (ty :: Type) (schema :: Schema)
        → (In name ty schema ⇒ r)
        → r
checkIn name _ [] _
  = error ("Field " ++ show name ++ " not found.")
checkIn name ty (Col name' ty' : rest) k
  = case (name `eq` name', ty `eq` ty') of
      (Just Refl, Just Refl) → k
      (Just _,    _)         → error ("Found " ++ show name ++
                                      " but it maps to " ++ show ty')
      _                      → checkIn name ty rest k
```

This function searches through the *Schema* (which, recall, is just a [*Column*]) for the desired name and type. If the search fails or the search find the column associated with the wrong type, *checkIn* fails. Otherwise, it will eventually call *k*, the continuation that can now assume *In name ty schema*. The constraint *In* is implemented as a class with instances that prove that the (*name*, *ty*) pair is indeed in *schema* whenever *In name ty schema* holds.

The *checkIn* function makes critical use of a new function *eq*:[15]

```
class Eq a where
  ...
  eq :: Π (x :: a) (y :: a) → Maybe (x :~: y)
```

---

of Dependent Haskell is type erasure! However, passing types at runtime is sometimes necessary, and using the type **Type** to do so is a natural extension of what is done today. Indeed, today's *TypeRep* (explored in detail by Peyton Jones et al. [75]) is essentially a singleton for **Type**. As Dependent Haskell removes other singletons, so too will it remove *TypeRep* in favor of dependent pattern matching on **Type**. As with other aspects of type erasure, users will choose which types to erase by the choice between Π-quantification and a ∀-quantification.

[15] I present *eq* here as a member of the ubiquitous *Eq* class, as a definition for *eq* should be writable whenever a definition for == is. (Indeed, == could be implemented in terms of *eq*.) I do not, however, expect that *eq* will end up living directly in the *Eq* class, as I doubt the Haskell community will permit Dependent Haskell to alter such a fundamental class. Nevertheless, the functionality sported by *eq* will be a common need in Dependent Haskell code, and we will need to find a suitable home for the function.

This is just a more informative version of the standard equality operator ==. When two values are *eq*, we can get a proof of their equality. This is necessary in *checkIn*, where assuming this equality is necessary in order to establish the *In* constraint before calling the constrained continuation *k*.

3.1.3.4 Conclusion

This example has highlighted several aspects of Dependent Haskell:

- Writing a well-typed database access is well within the reach of Dependent Haskell. Indeed, much of the work has already been done in released libraries.
- Inferring the type of *queryDB* is a capability unique to Dependent Haskell among dependently typed languages. Other dependently typed languages require type signatures on all top-level functions; this example makes critical use of Haskell's ability to infer a type in deriving the type for *queryDB*.
- Having dependent types in a large language like Haskell sometimes shows synergies with other aspects of the language. In this example, we used Template Haskell to complement our dependent types to achieve something neither one could do alone: Template Haskell's ability to inspect an inferred type allowed us to synthesize the runtime checks necessary to prove that a call to *queryDB* was indeed safe.

3.1.4 Machine-checked sorting algorithms

Using dependent types to check a sorting algorithm is well explored in the literature (e.g., [1, 61]). These algorithms can also be translated into Haskell, as shown in my prior work [25, 30]. I will thus not go into any detail in the implementation here.

At the bottom of one implementation[16] appears this function definition:

$$\mathit{mergeSort} :: [\mathit{Integer}] \to [\mathit{Integer}].$$

Note that the type of the function is completely ordinary—there is no hint of the rich types that lurk beneath, in its implementation. It is this fact that makes machine-checked algorithms, such as sorting, interesting in the context of Haskell.

A Haskell programming team may make a large application with little use for fancy types. Over time, the team notice bugs frequently appearing in a gnarly section of code (like a sorting algorithm, or more realistically, perhaps, an implementation of a cryptographic primitive), and they decide that they want extra assurances that the algorithm is correct. That one algorithm—and no other part of the large application—might be rewritten to use dependent types. Indeed any of the examples considered in

[16] `https://github.com/goldfirere/nyc-hug-oct2014/blob/master/OrdList.hs`

this chapter can be hidden beneath simply typed interfaces and thus form just one component of a larger, *simply* typed application.

3.2 Encoding hard-to-type programs

3.2.1 Variable-arity *zipWith*

The *Data.List* Haskell standard library comes with the following functions:

$$
\begin{aligned}
\mathit{map} &:: (a \to b) \to [a] \to [b] \\
\mathit{zipWith} &:: (a \to b \to c) \to [a] \to [b] \to [c] \\
\mathit{zipWith3} &:: (a \to b \to c \to d) \to [a] \to [b] \to [c] \to [d] \\
\mathit{zipWith4} &:: (a \to b \to c \to d \to e) \to [a] \to [b] \to [c] \to [d] \to [e] \\
&\ldots
\end{aligned}
$$

Let's pretend to rename *map* to *zipWith1* and *zipWith* to *zipWith2*. This sequence continues up to *zipWith7*. The fact that these are different functions means that the user must choose which one to use, based on the arity of the function to be mapped over the lists. However, forcing the user to choose this is a bit silly: the type system should be able to discern which *zipWith* is correct based on the type of the function. Dependent Haskell gives us the power to write such a variable-arity *zipWith* function.[17]

Let's build up our solution one step at a time. We'll first focus on building a *zipWith* that is told what arity to be; then we'll worry about inferring this arity.

Recall the definition of natural numbers from Section 3.1.1:

$$\textbf{data}\ \mathit{Nat} = \mathit{Zero} \mid \mathit{Succ}\ \mathit{Nat}$$

What will the type of our final *zipWith* be? It will first take a function and then several lists. The types of these lists are determined by the type of the function passed in. For example, suppose our function f has type $\mathit{Int} \to \mathit{Bool} \to \mathit{Double}$, then the type of *zipWith* should be $(\mathit{Int} \to \mathit{Bool} \to \mathit{Double}) \to [\mathit{Int}] \to [\mathit{Bool}] \to [\mathit{Double}]$. Thus, we wish to take the type of the function and apply the list type constructor $[\,]$ to each component of it.

Before we write the code for this operation, we pause to note an ambiguity in this definition. Both of the following are sensible concrete types for a *zipWith* over the function f:

$$
\begin{aligned}
\mathit{zipWith} &:: (\mathit{Int} \to \mathit{Bool} \to \mathit{Double}) \\
&\quad \to [\mathit{Int}] \to [\mathit{Bool} \to \mathit{Double}] \\
\mathit{zipWith} &:: (\mathit{Int} \to \mathit{Bool} \to \mathit{Double}) \\
&\quad \to [\mathit{Int}] \to [\mathit{Bool}] \to [\mathit{Double}]
\end{aligned}
$$

[17] This example is adapted from my prior work [31].

The first of these is essentially *map*; the second is the classic function *zipWith* that expects two lists. Thus, we must pass in the desired number of parameters to apply the list type constructor to. The function to apply these list constructors is named *Listify*:

```
type family Listify (n :: Nat) arrows where
  Listify 'Zero     a       = [a]
  Listify ('Succ n) (a → b) = [a] → Listify n b
```

We now need to create some runtime evidence of our choice for the number of arguments. This will be used to control the runtime operation of *zipWith*—after all, our function must have both the correct behavior and the correct type. We use a GADT *NumArgs* that plays two roles: it controls the runtime behavior as just described, and it also is used as evidence to the type checker that the number argument to *Listify* is appropriate. After all, we do not want to call *Listify* 2 (*Int* → *Bool*), as that would be stuck. By pattern-matching on the *NumArgs* GADT, we get enough information to allow *Listify* to fully reduce.

```
data NumArgs :: Nat → Type → Type where
  NAZero :: ∀ a.                                NumArgs 'Zero     a
  NASucc :: ∀ a b (n :: Nat). NumArgs n b → NumArgs ('Succ n) (a → b)
```

We now write the runtime workhorse *listApply*, with the following type:

```
listApply :: NumArgs n a → [a] → Listify n a
```

The first argument is the encoding of the number of arguments to the function. The second argument is a *list* of functions to apply to corresponding elements of the lists passed in after the second argument. Why do we need a list of functions? Consider evaluating *zipWith* (+) [1, 2] [3, 4], where we recur not only on the elements in the list, but on the number of arguments. After processing the first list, we have to be able to apply different functions to each of the elements of the second list. To wit, we need to apply the functions [(1+), (2+)] to corresponding elements in the list [3, 4]. (Here, we are using Haskell's "section" notation for partially-applied operators.)

Here is the definition of *listApply*:

```
listApply NAZero     fs = fs
listApply (NASucc na) fs =
  λargs → listApply na (apply fs args)
  where apply :: [a → b] → [a] → [b]
        apply (f : fs) (x : xs) = (f x : apply fs xs)
        apply _        _        = []
```

It first pattern-matches on its first argument. In the *NAZero* case, each member of the list of functions passed in has 0 arguments, so we just return the list. In the *NASucc*

case, we process one more argument (*args*), apply the list of functions *fs* respectively to the elements of *args*, and then recur. Note how the GADT pattern matching is essential for this to type-check—the type checker gets just enough information for *Listify* to reduce enough so that the second case can expect one more argument than the first case.

Inferring arity In order to infer the arity, we need to have a function that counts up the number of arrows in a function type:

```
type family CountArgs (f :: Type) :: Nat where
  CountArgs (a → b) = 'Succ (CountArgs b)
  CountArgs result  = 'Zero
```

The ability to write this function is unique to Haskell, where pattern-matching on proper types (of kind **Type**) is allowed.

We need to connect this type-level function with the term-level GADT *NumArgs*. We use Haskell's method for reflecting type-level decisions on the term level: type classes. The following definition essentially repeats the definition of *NumArgs*, but because this is a definition for a class, the instance is inferred rather than given explicitly:

```
class CNumArgs (numArgs :: Nat) (arrows :: Type) where
  getNA :: NumArgs numArgs arrows
instance CNumArgs 'Zero a where
  getNA = NAZero
instance CNumArgs n b ⇒
         CNumArgs ('Succ n) (a → b) where
  getNA = NASucc getNA
```

Note that the instances do *not* overlap; they are distinguished by their first parameter.

It is now straightforward to give the final definition of *zipWith*, using the extension `ScopedTypeVariables` to give the body of *zipWith* access to the type variable *f*:

```
zipWith :: ∀ f. CNumArgs (CountArgs f) f
        ⇒ f → Listify (CountArgs f) f
zipWith fun
  = listApply (getNA :: NumArgs (CountArgs f) f) (repeat fun)
```

The standard Haskell function *repeat* creates an infinite list of its one argument.

The following examples show that *zipWith* indeed infers the arity:

```
example1 = zipWith (&&) [False, True, False] [True, True, False]
example2 = zipWith ((+) :: Int → Int → Int) [1, 2, 3] [4, 5, 6]
```

$$
\begin{aligned}
&\mathit{concat} :: \mathit{Int} \to \mathit{Char} \to \mathit{Double} \to \mathit{String} \\
&\mathit{concat}\ a\ b\ c = (\mathit{show}\ a) +\!\!+ (\mathit{show}\ b) +\!\!+ (\mathit{show}\ c) \\
&\mathit{example}_3 = \mathit{zipWith}\ \mathit{concat}\ [1,2,3]\ \texttt{['a', 'b', 'c']} \\
&\qquad\qquad\qquad\qquad\qquad\qquad [3.14, 2.1728, 1.01001]
\end{aligned}
$$

In $\mathit{example}_2$, we must specify the concrete instantiation of $(+)$. In Haskell, built-in numerical operations are generalized over a type class *Num*. In this case, the operator $(+)$ has the type $\mathit{Num}\ a \Rightarrow a \to a \to a$. Because it is theoretically possible (though deeply strange!) for a to be instantiated with a function type, using $(+)$ without an explicit type will not work—there is no way to infer an unambiguous arity. Specifically, *CountArgs* gets stuck. $\mathit{CountArgs}\ (a \to a \to a)$ simplifies to $\mathit{Succ}\ (\mathit{Succ}\ (\mathit{CountArgs}\ a))$ but can go no further; $\mathit{CountArgs}\ a$ will not simplify to *Zero*, because a is not apart from $b \to c$.

3.2.2 Typed reflection

Reflection is the act of reasoning about a programming language from within programs written in that language.[18] In Haskell, we are naturally concerned with reflecting the rich language of Haskell types. A reflection facility such as the one described here will be immediately applicable in the context of Cloud Haskell. Cloud Haskell [35] is an ongoing project, aiming to support writing a Haskell program that can operate on several machines in parallel, communicating over a network. To achieve this goal, we need a way of communicating data of all types over a wire—in other words, we need dynamic types. On the receiving end, we would like to be able to inspect a dynamically typed datum, figure out its type, and then use it at the encoded type. For more information about how kind equalities fit into Cloud Haskell, please see the GHC wiki at `https://ghc.haskell.org/trac/ghc/wiki/DistributedHaskell`.

Reflection of this sort has been possible for some time using the *Typeable* mechanism [53]. However, the lack of kind equalities—the ability to learn about a type's kind via pattern matching—has hindered some of the usefulness of Haskell's reflection facility. In this section, we explore how this is the case and how the problem is fixed.

3.2.2.1 Heterogeneous propositional equality

Kind equalities allow for the definition of *heterogeneous propositional equality*, a natural extension to the propositional equality described in Section 2.4:

$$
\begin{aligned}
&\textbf{data}\ (a :: k_1) :\approx: (b :: k_2)\ \textbf{where} \\
&\quad \mathit{HRefl} :: a :\approx: a
\end{aligned}
$$

[18] Many passages in this example are expanded upon in my prior work [75].

Pattern-matching on a value of type $a :\approx: b$ to get *HRefl*, where $a :: k_1$ and $b :: k_2$, tells us both that $k_1 \sim k_2$ and that $a \sim b$. As we'll see below, this more powerful form of equality is essential in building the typed reflection facility we want.

3.2.2.2 Type representation

Here is our desired representation:[19]

```
data TyCon (a :: k)
    -- abstract; the type Int is represented by the one value of type TyCon Int
data TypeRep (a :: k) where
  TyCon :: TyCon a → TypeRep a
  TyApp :: TypeRep a → TypeRep b → TypeRep (a b)
```

The intent is that, for every new type declared, the compiler would supply an appropriate value of the *TyCon* datatype. The type representation library would supply also the following function, which computes equality over *TyCon*s, returning the heterogeneous equality witness:

```
eqTyCon :: ∀ (a :: k1) (b :: k2).
           TyCon a → TyCon b → Maybe (a :≈: b)
```

It is critical that this function returns $(:\approx:)$, not $(:\sim:)$. This is because *TyCon*s exist at many different kinds. For example, *Int* is at kind **Type**, and *Maybe* is at kind **Type** → **Type**. Thus, when comparing two *TyCon* representations for equality, we want to learn whether the types *and the kinds* are equal. If we used $(:\sim:)$ here, then the *eqTyCon* could be used only when we know, from some other source, that the kinds are equal.

We can now easily write an equality test over these type representations:

```
eqT :: ∀ (a :: k1) (b :: k2).
       TypeRep a → TypeRep b → Maybe (a :≈: b)
eqT (TyCon t1)     (TyCon t2)     = eqTyCon t1 t2
eqT (TyApp a1 b1) (TyApp a2 b2)
  | Just HRefl ← eqT a1 a2
  , Just HRefl ← eqT b1 b2        = Just HRefl
eqT _              _              = Nothing
```

Note the extra power we get by returning *Maybe* $(a :\approx: b)$ instead of just a *Bool*. When the types are indeed equal, we get evidence that GHC can use to be aware of

[19] This representation works well with an open world assumption, where users may introduce new type constants in any module. See my prior work [75, Section 4] for more discussion on this point.

this type equality during type checking. A simple return type of *Bool* would not give the type-checker any information.

3.2.2.3 Dynamic typing

Now that we have a type representation with computable equality, we can package that representation with a chunk of data, and so form a dynamically typed package:

```
data Dyn where
  Dyn :: ∀ (a :: Type). TypeRep a → a → Dyn
```

The *a* type variable there is an *existential* type variable. We can think of this type as being part of the data payload of the *Dyn* constructor; it is chosen at construction time and unpacked at pattern-match time. Because of the *TypeRep a* argument, we can learn more about *a* after unpacking. (Without the *TypeRep a* or some other type-level information about *a*, the unpacking code must treat *a* as an unknown type and must be parametric in the choice of *a*.)

Using *Dyn*, we can pack up arbitrary data along with its type and push that data across a network. The receiving program can then make use of the data, without needing to subvert Haskell's type system. This type representation library must be trusted to recreate the *TypeRep* on the far end of the wire, but the equality tests above and other functions below can live outside the trusted code base.

Suppose we were to send an object with a function type, say *Bool* → *Int* over the network. Let's ignore here the complexities of actually serializing a function—there is a solution to that problem[20], but here we are concerned only with the types. We would want to apply the received function to some argument. What we want is this:

```
dynApply :: Dyn → Dyn → Maybe Dyn
```

The function *dynApply* applies its first argument to the second, as long as the types line up. The definition of this function is fairly straightforward:

```
dynApply (Dyn (TyApp
                 (TyApp (TyCon tarrow) targ)
                 tres)
               fun)
         (Dyn targ' arg)
  | Just HRefl ← eqTyCon tarrow (tyCon :: TyCon (→))
  , Just HRefl ← eqT targ targ'
  = Just (Dyn tres (fun arg))
dynApply _ _ = Nothing
```

[20] `https://ghc.haskell.org/trac/ghc/wiki/StaticPointers`

We first match against the expected type structure—the first *Dyn* argument must be a function type. We then confirm that the *TyCon tarrow* is indeed the representation for $(\rightarrow)$ (the construct *tyCon* :: *TyCon* $(\rightarrow)$ retrieves the compiler-generated representation for $(\rightarrow)$) and that the actual argument type matches the expected argument type. If everything is good so far, we succeed, applying the function in *fun arg*.

3.2.2.4 Conclusion

Heterogeneous equality is necessary throughout this example. It first is necessary in the definition of *eqT*. In the *TyApp* case, we compare *a1* to *a2*. If we had only homogeneous equality, it would be necessary that the types represented by *a1* and *a2* be of the same kind. Yet, we can't know this here! Even if the types represented by *TyApp a1 b1* and *TyApp a2 b2* have the same kind, it is possible that *a1* and *a2* would not. (For example, maybe the type represented by *a1* has kind **Type** $\rightarrow$ **Type** and the type represented by *a2* has kind *Bool* $\rightarrow$ **Type**.) With only homogeneous equality, we cannot even write an equality function over this form of type representation. The problem repeats itself in the definition of *dynApply*, when calling *eqTyCon tarrow TArrow*. The call to *eqT* in *dynApply*, on the other hand, *could* be homogeneous, as we would know at that point that the types represented by *targ* and *targ'* are both of kind **Type**.

In today's Haskell, the lack of heterogeneous equality means that *dynApply* must rely critically on *unsafeCoerce*. With heterogeneous equality, *dynApply* can remain safely outside the trusted code base.

3.2.3 Algebraic effects

Brady [8] describes an approach to the challenge of embedding side effects into a pure, functional language. His approach is to use composable algebraic effects, implemented as a domain-specific language embedded in Idris [9], a full spectrum dependently typed language. This technique is meant to contrast with Haskell's monad transformers [55]. Brady's library, Effects, is translatable directly into Dependent Haskell. With heavy use of singletons, all of the code described in the original paper is even implementable in GHC 8.[21]

3.2.3.1 Example 1: an simple expression interpreter

To give you an idea of the power and flexibility of the algebraic effects approach, let's look at a function that interprets a simple expression language.[22] Here is the expression AST:

data *Expr* = *Val Nat* | *Add Expr Expr* | *Var String* | *Random Nat*

[21] The code is available at `https://github.com/goldfirere/thesis/tree/master/effects`. It does not compile with GHC 8.0.1 due to a small implementation bug. The fix is in the latest development version of GHC and may be available in GHC 8.0.2.

[22] This example is adapted from Brady [8, Section 2.1.3].

Expressions can contain literal numbers,[23] addition, variable references, and naturals randomly generated up to some specified limit. In the version we will consider, the interpreter is instrumented to print out the value of every random number generated. Thus the interpreter needs four different effectful capabilities: the ability to deal with errors (in case a named variable does not exist), the ability to write output, access to a pseudo-random number generator, and an ambient environment of defined variables. This ambient environment has type *Vars*, an association list mapping variable names to their values:

```
type Vars = [(String, Nat)]
```

With all that in mind, here is the evaluator:

```
eval :: Handler StdIO e
     ⇒ Expr → Eff e '[EXCEPTION String, STDIO, RND, STATE Vars] Nat
eval (Val x)          = return x
eval (Var x)          = do vs ← get
                           case lookup x vs of
                             Nothing → raise ("Unknown var: " ++ x)
                             Just val → return val
eval (Add l r)        = (+) ⟨$⟩ eval l ⟨*⟩ eval r
eval (Random upper) = do num ← rndNat 0 upper
                         putStrLn ("Random value: " ++ show num)
                         return num
```

Let's first look at the type of *eval*, with our goal being a general understanding of what this technique brings us, not working out all the details.

The return type of this function is a specialization of *Eff*, a type defined by the Effects library. *Eff* is not a monad; the use of **do**-notation in the code in this section is enabled by the GHC extension `RebindableSyntax`. With `RebindableSyntax`, GHC uses whatever symbols are in scope to implement various features. In our case, Effects defines ≫ and ≫= operators which work over *Eff*.

Eff takes three parameters: an underlying effect handler *e*, a type-level list of capabilities, and the return type of the computation. The underlying effect handler must be able to handle read and write commands. We would generally expect this to be *IO*, but an environment with an input list of strings and an output list of strings could be used to model I/O in a pure environment. The list of capabilities is better viewed as a set, as the order in this list is immaterial. Fancy footwork done by the types of the operations provided by the capabilities (like *get* or *rndNat*) looks up the capability in the list, regardless of order.

[23] I have restricted this and other examples to work with naturals only. This restriction is in place solely to play nicely with the use of singletons to translate the Idris library into a form compatible with GHC 8. In a full Dependent Haskell implementation, this restriction would not be necessary.

Once we've absorbed the type of *eval*, its body is rather uninteresting—and that's exactly the point! We need not *lift* one capability through another (as must be done with monad transformers) nor give any indication of how our capabilities are structured. It all just works.

With *eval* in hand, it is straightforward to write the function that actually can evaluate an expression:

```
runEval :: Vars → Expr → IO Nat
runEval env expr = run (() :> () :> 123 :> env :> Empty) (eval expr)
```

The first argument to the Effects library function *run* is an environment of resources, where each resource is associated with a capability. While the order of capabilities does not matter in the body of *eval*, its order must match up with the order of resources given when running an *Eff* computation. In this case, the *EXCEPTION String* and *STDIO* capabilities have no associated resource (the entries in the environment are both ()). The *RND* capability uses a random generation seed (123 in our case), and the *STATE Vars* needs the initial state, passed as a parameter to *runEval*.

Having defined all of the above, we can now observe this interaction:

```
λ> runEval [("x", 3)] (Var "x" `Add` Random 12)
Random value: 1
4
```

In this output, the 4 at the end is the result of evaluating the expression, which adds the value of `"x"`, 3, to the pseudo-random number 1.

3.2.3.2 Automatic lifting

In the example above, we can use the *STATE* capability with its *get* accessor, despite the fact that *STATE* is buried at the bottom of the list of capabilities. This is done by *get*'s rather clever type:

```
get :: Π (prf :: SubList '[STATE x] xs).
       prf ~ 'findSubListProof '[STATE x] xs
    ⇒ EffM m xs (UpdateWith '[STATE x] xs prf) x
```

The function *get* takes in a proof that `'[STATE x] xs` is a sublist of *xs*, the list of capabilities in the result type. (*EffM* is a generalization of *Eff* that allows for the capabilities to change during a computation. It lists the "before" capabilities and the "after" capabilities. *Eff* is just a type synonym for *EffM* with both lists the same.) Despite taking the proof in as an argument, *get* requires that the proof be the one found by the *findSubListProof* function. In this way, the calling code does not need to write the proof by hand; it can be discovered automatically. However, note that the

proof is Π-bound—it is needed at runtime because each capability is associated with a resource, stored in a list. The proof acts as an index into that list to find the resource.

In Idris, *get*'s type is considerably simpler: *get* :: *Eff m* '[*STATE x*] *x*. This works in Idris because of Idris's *implicits* feature, whereby a user can install an implicit function to be tried in the case of a type mismatch. In our case here, the list of capabilities in *get*'s type will not match the larger list in *eval*'s type, triggering a type error. The Effects library provides an implicit lifting operation which does the proof search I have encapsulated into *findSubListProof*. While it is conceivable to consider adding such an implicits feature to Haskell, doing so is well beyond this dissertation. In the case of my translation of Effects, the lack of implicits bites, but not in a particularly troublesome way; the types of basic operations like *get* just get a little more involved.

3.2.3.3 Example 2: Working with files

Brady [8, Section 2.2.5] also includes an example of how Effects can help us work with files. We first define a *readLines* function that reads all of the lines in a file. This uses primitive operations *readLine* and *eof*.

```
readLines :: Eff IO '[FILE_IO (OpenFile 'Read)] [String]
readLines = readAcc []
  where
    readAcc acc = do e ← eof
                     if (not e)
                       then do str ← readLine
                               readAcc (str : acc)
                       else return (reverse acc)
```

Once again, let's look at the type. The only capability asserted by *readLines* is the ability to access one file opened for reading. The implementation is straightforward.

The function *readLines* is used by *readFile*:

```
readFile :: String → Eff IO '[FILE_IO (), STDIO, EXCEPTION String] [String]
readFile path
   = catch (do _ ← open path Read
               test Here (raise ("Cannot open file: " ++ path)) $
                 do lines ← lift readLines
                    close @Read
                    return lines)
           (λerr → do putStrLn ("Failed: " ++ err)
                      return [])
```

The type of *readFile* is becoming routine: it describes an effectful computation that can

access files (with none open), do input/output, and raise exceptions. The underlying handler is Haskell's *IO* monad, and the result of running *readFile* is a list of strings.

The body of this function, however, deserves scrutiny, as the type system is working hard on our behalf throughout this function. The first line calls the Effects library function *open*, which uses the *FILE_IO* capability. Here is a simplified version of its type, where the automatic lifting mechanism (Section 3.2.3.2) is left out:

```
open :: String → Π (m :: Mode)
     → EffM e '[FILE_IO ()] '[FILE_IO (Either () (OpenFile m))] Bool
```

The function *open* takes the name of a file and whether to open it for reading or writing. Its return type declares that the *open* operation starts with the capability of file operations with no open file but ends with the capability of file operations either with no open file or with a file opened according to the mode requested. Recall that *EffM* is a generalization of *Eff* that declares two lists of capabilities: one before an action and one after it. The *Either* in *open*'s type reflects the possibility of failure. After all, we cannot be sure that *open* will indeed result in an open file.[24] The return value of type *Bool* indicates success or failure.

After running *open*, *readFile* uses *test*, another Effects function, with the following type:

```
test :: Π (prf :: EffElem e (Either l r) xs)
     → EffM m (UpdateResTyImm xs prf l) xs' t
     → EffM m (UpdateResTyImm xs prf r) xs' t
     → EffM m xs xs' t
```

Without looking too closely at that type, we can surmise this:

- The starting capability set, *xs*, contains an effect with an *Either l r* resource.
- The caller of *test* must provide a proof *prf* of this fact. (*EffElem* is a rather standard datatype that witnesses the inclusion of some element in a list, tailored a bit to work with capabilities.)
- The next two arguments of *test* are continuations to pursue depending on the status of the *Either*. Note that the first works with *l* and the second with *r*. Both continuations must result in the same ending capability set *xs'*.

[24] Readers may be wondering at this point how Effects deals with the possibility of multiple open files. The library can indeed handle this possibility through listing *FILE_IO* multiple times in the list of capabilities. Effects includes a mechanism for labeling capabilities (not described here, but implemented in Haskell and described by Brady [8]) that can differentiate among several *FILE_IO* capabilities.

- The *test* operator itself takes the capability set from *xs* to *xs'*.

In our case, *test* is meant to check the *Either* () (*OpenFile* '*Read*), stored in the first capability. (*Here* is the proof that the capability we seek is first in the list.) If the *Either* is *Left*, *raise* an exception. Otherwise, we know that the *open* succeeded, and the inner **do** block can work with a capability *FILE_IO* (*OpenFile* '*Read*).

The inner **do** block runs *readLines*, using *lift* because the type of *readLines* assumes only the one *FILE_IO* capability, and *readFile* has more than just that. The same automatic proof search facility described earlier works with explicit *lift*s.

The use of *close* here is again interesting, because omitting it would be a type error. Here is *close*'s type (again, eliding the lifting machinery):

```
close :: ∀ m e. EffM e '[FILE_IO (OpenFile m)] '[FILE_IO ()] ()
```

It takes an *OpenFile* and closes it. Forgetting this step would be a type error because *test* requires that both paths result in the same set of capabilities. The failure path from *test* has no open files at the end, and so the success path must also end with no open files. The type of *close* achieves this.

A careful reader will note that we have to specify the *Read* invisible parameter to *close*. This is necessary to support the automatic lifting mechanism. Without knowing that it is searching for *FILE_IO* (*OpenFile Read*), it gets quite confused; looking for *FILE_IO* (*OpenFile m*) is just not specific enough. It is conceivable that this restriction could be lifted with a cleverer automatic lifting mechanism or a type-checker plugin [22, 38].

All of the code described above is wrapped in a *catch* in order to deal with any possible exception; *catch* is not intricately typed and does not deserve further study here.

Having written *readFile*, we can now use it:

```
printFile :: FilePath → IO ()
printFile filepath
  = do ls ← run (() :> () :> () :> Empty) (readFile filepath)
       mapM_ putStrLn ls
```

The return type of *printFile* is just the regular Haskell *IO* monad. Due to the way GHC's `RebindableSyntax` extension works, *printFile* must be written in a separate module from the code above in order to access the usual monadic meaning of **do**.

This example has shown us how the Effects not only makes it easy to mix and match different effects without the quadratic code cost of monad transformers, but it also helps us remember to release resources. Forgetting to release a resource has become a type error.

3.2.3.4 Example 3: an interpreter for a well-typed imperative language

The final example with Effects is also the culminating example by Brady [8, Section 4]: an interpreter for an imperative language with mutable state. The goal of presenting this example is simply to show that Effects scales to ever more intricate types, even in its translation to Haskell. Accordingly, I will be suppressing many details in this presentation. The curious can read the full source code online.[25]

This language, Imp, contains both expressions and statements:

```
data Ty = ...   -- types in Imp
interpTy :: Ty → Type   -- consider a Ty as a real Haskell Type
data Expr :: ∀ n. Vec Ty n → Ty → Type where ...
data Imp  :: ∀ n. Vec Ty n → Ty → Type where ...
```

Following the implementation in Idris, my translation uses a deep embedding for the types, using the datatype *Ty* instead of Haskell's types. This is purely a design choice; using Haskell's types works just as well.[26]

Expressions and statements (the datatype *Imp*) are parameterized over a vector of types given to de Bruijn-indexed variables. Both expressions and statements also produce an output value, included in their types above. Thus, an expression of type *Expr g t* has type *t* in the typing context *g*.

Let's focus on the statement form that introduces a new, mutable variable:

```
data Imp :: ∀ n. Vec Ty n → Ty → Type where
  Let :: ∀ t g u. Expr g t → Imp (t :& g) u → Imp g u
  ...
```

The variable, of type *t*, is given an initial value by evaluating the *Expr g t*. The body of the *Let* is an *Imp* (*t* :& *g*) *u*—that is, a statement of type *u* in a context extended by *t*. (The operator :& is the *cons* operator for *Vec*, here.)

Here is how such a statement is interpreted:

[25] `https://github.com/goldfirere/thesis/blob/master/effects/Sec4.hs`

[26] Interestingly, the use of a deep embedding in my implementation means that I have to label *interpTy* as injective [86]. Otherwise, type inference fails. Idris's type inference algorithm must similarly use injectivity to accept this program.

```
interp :: ∀ g t. Imp g t → Eff IO '[STDIO, RND, STATE (Vars g)] ('interpTy t)
interp (Let @t' e sc)
  = do e' ← lift (eval e)
       vars ← get @(Vars g)
       putM @(Vars g) (e' :^ vars)
       res ← interp sc
       (_ :^ vars') ← get @(Vars (t' :& g))
       putM @(Vars (t' :& g)) vars'
       return res
```

I will skip over most of the details here, making only these points:

- It is necessary to use the @ invisibility override (Section 4.2.3.1) several times so that the automatic lifting mechanism knows what to look for. Alternatives to the approach seen here include using explicit labels on capabilities (see Brady [8, Section 2.1.2]), writing down the index of the capability desired, or implementing a type-checker plugin to help do automatic lifting.
- The *putM* function (an operation on *STATE*) changes the type of the stored state. In this case, the stored state is a vector that is extended with the new variable. We must, however, remember to restore the original state, as otherwise the final list of capabilities would be different than the starting list, a violation of *interp*'s type. (Recall that *Eff*, in *interp*'s type, requires the same final capability set as its initial capability set.)
- The *eval* function (elided from this text) uses a smaller set of capabilities. Its use must be *lift*ed.

Despite the ever fancier types seen in this example, Haskell still holds up. The requirement to specify the many invisible arguments (such as @(*Vars g*)) is indeed regrettable; however, I feel confident that some future work could resolve this pain point.

3.2.3.5 Conclusion

The Effects library is a major achievement in Idris and shows some of the power of dependent types for practical programming. I have shown here that this library can be ported to Dependent Haskell, where it remains just as useful. Perhaps as Dependent Haskell is adopted, more users will prefer to use this approach over monad transformers.

3.3 Why Haskell?

There already exist several dependently typed languages. Why do we need another? This section presents several reasons why I believe the work described in this dissertation will have impact.

3.3.1 Increased reach

Haskell currently has some level of adoption in industry.[27] Haskell is also used as the language of choice in several academic programs used to teach functional programming. There is also the ongoing success of the Haskell Symposium. These facts all indicate that the Haskell community is active and sizeable. If GHC, the primary Haskell compiler, offers dependent types, more users will have immediate access to dependent types than ever before.

The existing dependently typed languages were all created, more or less, as playgrounds for dependently typed programming. For a programmer to choose to write her program in an existing dependently typed language, she would have to be thinking about dependent types (or the possibility of dependent types) from the start. However, Haskell is, first and foremost, a general purpose functional programming language. A programmer might start his work in Haskell without even being aware of dependent types, and then as his experience grows, decide to add rich typing to a portion of his program.

With the increased exposure GHC would offer to dependent types, the academic community will gain more insight into dependent types and their practical use in programs meant to get work done.

3.3.2 Backward-compatible type inference

Working in the context of Haskell gives me a stringent, immovable constraint: my work must be backward compatible. In the new version of GHC that supports dependent types, all current programs must continue to compile. In particular, this means that type inference must remain able to infer all the types it does today, including types for definitions with no top-level annotation. Agda and Idris require a top-level type annotation for every function; Coq uses inference where possible for top-level definitions but is sometimes unpredictable. Furthermore, Haskellers expect the type inference engine to work hard on their behalf; they wish to rarely rely on manual proving techniques.

[27] At the time of writing, `https://wiki.haskell.org/Haskell_in_industry` lists 81 companies who use Haskell to some degree. That page, of course, is world-editable and is not authoritative. However, I am personally aware of Haskell's (growing) use in several industrial settings, and I have seen quite a few job postings looking for Haskell programmers in industry. For example, see `http://functionaljobs.com/jobs/search/?q=haskell`.

The requirement of backward compatibility keeps me honest in my design of type inference—I cannot cheat by asking the user for more information. The technical content of this statement is discussed in Chapter 6 by comparison with the work of Vytiniotis et al. [99] and Eisenberg et al. [33]. See Sections 6.8.2 and 6.8.3. A further advantage of working in Haskell is that the type inference of Haskell is well studied in the literature. This dissertation continues this tradition in Chapter 6.

3.3.3 No termination or totality checking

Many dependently typed languages today strive to be proof systems as well as programming languages. These care deeply about totality: that all pattern matches consider all possibilities and that every function can be proved to terminate. Coq does not accept a function until it is proved to terminate. Agda behaves likewise, although the termination checker can be disabled on a per-function basis. Idris embraces partiality, but then refuses to evaluate partial functions during type-checking. Dependent Haskell, on the other hand, does not care about totality.

Dependent Haskell emphatically does *not* strive to be a proof system. In a proof system, whether or not a type is inhabited is equivalent to whether or not a proposition holds. Yet, in Haskell, *all* types are inhabited, by $\bot$ and other looping terms, at a minimum. Even at the type level, all kinds are inhabited by the following type family, defined in GHC's standard library:

```
type family Any :: k   -- no instances
```

The type family *Any* can be used at any kind, and so inhabits all kinds.

Furthermore, Dependent Haskell has the **Type**:**Type** axiom, meaning that instead of having an infinite hierarchy of universes characteristic of Coq, Agda, and Idris, Dependent Haskell has just one universe, which contains itself. It is well known that self-containment of this form leads to logical inconsistency by enabling the construction of a looping term [36], but I am unbothered by this—Haskell has many other looping terms, too! (See Section 4.4.1 for more discussion on this point.) By allowing ourselves to have **Type**:**Type**, the type system is much simpler than in systems with a hierarchy of universes.

There are two clear downsides of the lack of totality:

- What appears to be a proof might not be. Suppose we need to prove that type τ equals type σ in order to type-check a program. We can always use $\bot :: \tau :\approx: \sigma$ to prove this equality, and then the program will type-check. The problem will be discovered only at runtime. Another way to see this problem is that equality proofs must be run, having an impact on performance. However, note that we cannot use the bogus equality without evaluating it; there is no soundness issue.

  This drawback is indeed serious, and important future work includes designing and implementing a totality checker for Haskell. (See the work of Vazou et al.

[94] for one approach toward this goal. Recent work by Karachalias et al. [51] is another key building block.) Unlike in other languages, though, the totality checker would be chiefly used in order to optimize away proofs, rather than to keep the language safe. Once the checker is working, we could also add compiler flags to give programmers compile-time warnings or errors about partial functions, if requested.

- Lack of termination in functions used at the type level might conceivably cause GHC to loop. This is not a great concern, however, because the loop is directly caused by a user's type-level program. In practice, GHC counts steps it uses in reducing types and reports an error after too many steps are taken. The user can, via a compiler flag, increase the limit or disable the check.

The advantages to the lack of totality checking are that Dependent Haskell is simpler for not worrying about totality. It is also more expressive, treating partial functions as first-class citizens.

3.3.4 GHC is an industrial-strength compiler

Hosting dependent types within GHC is likely to reveal new insights about dependent types due to all of the features that GHC offers. Not only are there many surface language extensions that must be made to work with dependent types, but the back end must also be adapted. A dependently typed intermediate language must, for example, allow for optimizations. Working in the context of an industrial-strength compiler also forces the implementation to be more than just "research quality," but ready for a broad audience.

3.3.5 Manifest type erasure properties

A critical property of Haskell is that it can erase types. Despite all the machinery available in Haskell's type system, all type information can be dropped during compilation. In Dependent Haskell, this does not change. However, dependent types certainly blur the line between term and type, and so what, precisely, gets erased can be difficult to discern. Dependent Haskell, in a way different from other dependently typed languages, makes clear which arguments to functions (and data constructors) get erased. This is through the user's choice of relevant vs. irrelevant quantifiers, as explored in Section 4.2.2. Because erasure properties are manifestly available in types, a performance-conscious user can audit a Dependent Haskell program and see exactly what will be removed at runtime.

It is possible that, with practice, this ability will become burdensome, in that the user has to figure out what to keep and what to discard. Idris's progress toward type erasure analysis [10, 90] may benefit Dependent Haskell as well.

3.3.6 Type-checker plugin support

Recent versions of GHC allow *type-checker plugins*, a feature that allows end users to write a custom solver for some domain of interest. For example, Gundry [38] uses a plugin to solve constraints arising from using Haskell's type system to check that a physical computation respects units of measure. As another example, Diatchki [22] has written a plugin that uses an SMT solver to work out certain numerical constraints that can arise using GHC's type-level numbers feature.

Once Haskell is equipped with dependent types, the need for these plugins will only increase. However, because GHC already has this accessible interface, the work of developing the best solvers for Dependent Haskell can be distributed over the Haskell community. This democratizes the development of dependently typed programs and spurs innovation in a way a centralized development process cannot.

3.3.7 Haskellers want dependent types

The design of Haskell has slowly been marching toward having dependent types. Haskellers have enthusiastically taken advantage of the new features. For example, over 1,000 packages published at `hackage.haskell.org` use type families [86]. Anecdotally, Haskellers are excited about getting full dependent types, instead of just faking them [30, 59, 60]. Furthermore, with all of the type-level programming features that exist in Haskell today, it is a reasonable step to go to full dependency.

Chapter 4

Dependent Haskell

This chapter provides an overview of Dependent Haskell. I will review the new features of the type language (Section 4.1), introduce the small menagerie of quantifiers available in Dependent Haskell (Section 4.2), explain pattern matching in the presence of dependent types (Section 4.3), and conclude the chapter by discussing several further points of interest in the design of the language.

There are many examples throughout this chapter, building on the following definitions:

```
  -- Length-indexed vectors, from Section 3.1.1
data Nat = Zero | Succ Nat
data Vec :: Type → Nat → Type where
  Nil  :: Vec a 'Zero
  (:>) :: a → Vec a n → Vec a ('Succ n)
infixr 5 :>

  -- Propositional equality, from Section 2.4
data a :~: b where
  Refl :: a :~: a

  -- Heterogeneous lists, indexed by the list of types of elements
data HList :: [Type] → Type where
  HNil :: HList '[]
  (:::) :: h → HList t → HList (h ': t)
infixr 5 :::
```

4.1 Dependent Haskell is dependently typed

The most noticeable change when going from Haskell to Dependent Haskell is that the latter is a full-spectrum dependently typed language. Expressions and types intermix. This actually is not too great a shock to the Haskell programmer, as the syntax of

Haskell expressions and Haskell types is so similar. However, by utterly dropping the distinction, Dependent Haskell has many more possibilities in types, as seen in the last chapter.

No distinction between types and kinds The kind system of GHC 7.10 and earlier is described in Section 2.3. It maintained a distinction between types, which classify terms, and kinds, which classify types. Yorgey et al. [107] enriched the language of kinds, allowing for some types to be promoted into kinds, but it did not mix the two levels.

My prior work [105] goes one step further than Yorgey et al. [107] and *does* merge types with kinds by allowing non-trivial equalities to exist among kinds. See my prior work for the details; this feature does not come through saliently in this dissertation, as I never consider any distinction between types and kinds. It is this work that is implemented and released in GHC 8. Removing the distinction between types and kinds has opened up new possibilities to the Haskell programmer. Below are brief examples of these new capabilities:

- *Explicit kind quantification.* Previously, kind variables were all quantified implicitly. GHC 8 allows explicit kind quantification:

```
data Proxy k (a :: k) = Proxy
    -- NB: Proxy takes both kind and type arguments
f :: ∀ k (a :: k). Proxy k a → ()
```

- *Kind-indexed GADTs.* Previously, a GADT could vary the return types of constructors only in its type variables, never its kind variables; this restriction is lifted. Here is a contrived example:

```
data G (a :: k) where
  MkG1 :: G Int
  MkG2 :: G Maybe
```

  Notice that *Int* and *Maybe* have different kinds, and thus that the instantiation of the *G*'s *k* parameter is non-uniform between the constructors. Some recent prior work [75] explores applying a kind-indexed to enabling dynamic types within Haskell.

- *Universal promotion.* As outlined by Yorgey et al. [107, Section 3.3], only some types were promoted to kinds in GHC 7.10 and below. In contrast, GHC 8 allows all types to be used in kinds. This includes type synonyms and type families, allowing computation in kinds for the first time.

- *GADT constructors in types.* A constructor for a GADT packs an equality proof, which is then exposed when the constructor is matched against. Because GHC 7.10 and earlier lacked informative equality proofs among kinds, GADT constructors could not be used in types. (They were simply not promoted.) However, with the rich kind equalities permitted in GHC 8, GADT constructors can be used freely in types, and type families may perform GADT pattern matching.

Expression variables in types Dependent Haskell obviates the need for most closed type families by allowing the use of ordinary functions directly in types. Because Haskell has a separate term-level namespace from its type-level namespace, any term-level definition used in a type must be prefixed with a ' mark. This expands the use of a ' mark to promote constructors as initially introduced by Yorgey et al. [107]. For example:

```
(+) :: Nat → Nat → Nat
Zero   + m =   m
Succ n + m =   Succ (n + m)

append :: Vec a n → Vec a m → Vec a (n '+ m)
append Nil       v = v
append (h :> t) v = h :> (append t v)
```

Note that this ability does not eliminate all closed type families, as term-level function definitions cannot use non-linear patterns, nor can they perform unsaturated matches (see Section 5.1.1.2).

Type names in terms It is sometimes necessary to go the other way and mention a type when writing something that syntactically appears to be a term. For the same reasons we need ' when using a term-level name in a type, we use ^ to use a type-level name in a term. A case in point is the code appearing in Section 3.1.3.2.

Pattern matching in types It is now possible to use **case** directly in a type:

```
tailOrNil :: Vec a n → Vec a (case n of
                                'Zero    → 'Zero
                                'Succ n' → n')
tailOrNil Nil       = Nil
tailOrNil (_ :> t) = t
```

Anonymous functions in types Types may now include λ-expressions:

```
eitherize :: HList types → HList ('map (λty → Either ty String) types)
eitherize HNil    = HNil
eitherize (h ::: t) = Left h ::: eitherize t
```

Other expression-level syntax in types Having merged types and expressions, *all* expression-level syntax is now available in types (for example, **do**-notation, **let** bindings, even arrows [46]). From a compilation standpoint, supporting these features is actually not a great challenge (once we have Chapters 5 and 6 implemented); it requires only interleaving type-checking with desugaring.[28] When a type-level use of elaborate expression-level syntax is encountered, we will need to work with the desugared version, hence the interleaving.

4.2 Quantifiers

Beyond simply allowing old syntax in new places, as demonstrated above, Dependent Haskell also introduces new quantifiers that allow users to write a broader set of functions than was previously possible. Before looking at the new quantifiers of Dependent Haskell, it is helpful to understand the several axes along which quantifiers can vary in the context of today's Haskell.

In Haskell, a *quantifier* is a type-level operator that introduces the type of an abstraction, or function. In Dependent Haskell, there are four essential properties of quantifiers, each of which can vary independently of the others. To understand the range of quantifiers that the language offers, we must go through each of these properties. In the text that follows, I use the term *quantifiee* to refer to the argument quantified over. The *quantifier body* is the type "to the right" of the quantifier. The quantifiers introduced in this section are summarized in Figure 4.1 on page 59.

4.2.1 Dependency

A quantifiee may be either dependent or non-dependent. A dependent quantifiee may be used in the quantifier body; a non-dependent quantifiee may not.

Today's Haskell uses ∀ for dependent quantification, as follows:

```
id :: ∀ a. a → a
```

In this example, `a` is the quantifiee, and `a → a` is the quantifier body. Note that the quantifiee `a` is used in the quantifier body.

[28] GHC currently type-checks the Haskell source directly, allowing it to produce better error messages. Only after type-checking and type inference does it convert Haskell source into its internal language, the process called *desugaring*.

The normal function arrow ($\rightarrow$) is an example of a non-dependent quantifier. Consider the predecessor function:

$$\mathit{pred} :: \mathit{Int} \rightarrow \mathit{Int}$$

The *Int* quantifiee is not named in the type, nor is it mentioned in the quantifier body.

In addition to $\forall$, Dependent Haskell adds a new dependent quantifier, Π. The only difference between Π and $\forall$ is that Π-quantifiee is relevant, as we'll explore next.

4.2.2 Relevance

A quantifiee may be either relevant or irrelevant. A relevant quantifiee may be used anywhere in the function quantified over; an irrelevant quantifiee may be used only in irrelevant positions—that is, as an irrelevant argument to other functions or in type annotations. Note that relevance talks about usage in the function quantified over, not the type quantified over (which is covered by the *dependency* property).

Relevance is very closely tied to type erasure. Relevant arguments in terms are precisely those arguments that are not erased. However, the *relevance* property applies equally to type-level functions, where erasure does not make sense, as all types are erased. For gaining an intuition about relevance, thinking about type erasure is a very good guide.

Today's Haskell uses ($\rightarrow$) for relevant quantification. For example, here is the body of *pred*:

$$\mathit{pred}\ x = x - 1$$

Note that x, a relevant quantifiee, is used in a relevant position on the right-hand side. Relevant positions include all places in a term or type that are not within a type annotation, other type-level context, or irrelevant argument context, as will be demonstrated in the next example.

Today's Haskell uses $\forall$ for irrelevant quantification. For example, here is the body of *id* (as given a type signature above):

$$\mathit{id}\ x = (x :: a)$$

The type variable a is the irrelevant quantifiee. According to Haskell's scoped type variables, it is brought into scope by the $\forall\ a$ in *id*'s type annotation. (It could also be brought into scope by using a in a type annotation on the pattern x to the left of the $=$.) Although a is used in the body of *id*, it is used only in an irrelevant position, in the type annotation for x. It would violate the irrelevance of $\forall$ for a to be used outside of a type annotation or other irrelevant context. As functions can take irrelevant arguments, irrelevant contexts include these irrelevant arguments.

Dependent Haskell adds a new relevant quantifier, Π. The fact that Π is both relevant and dependent is the very reason for Π's existence!

4.2.3 Visibility

A quantifiee may be either visible or invisible. The argument used to instantiate a visible quantifiee appears in the Haskell source; the argument used to instantiate an invisible quantifiee is elided.

Today's Haskell uses (→) for visible quantification. That is, when we pass an ordinary function an argument, the argument is visible in the Haskell source. For example, the 3 in *pred* 3 is visible.

On the other hand, today's ∀ and (⇒) are invisible quantifiers. When we call *id True*, the *a* in the type of *id* is instantiated at *Bool*, but *Bool* is elided in the call *id True*. During type inference, GHC uses unification to discover that the correct argument to use for *a* is *Bool*.

Invisible arguments specified with (⇒) are constraints. Take, for example, *show* :: ∀ *a*. *Show a* ⇒ *a* → *String*. The *show* function properly takes 3 arguments: the ∀-quantified type variable *a*, the (⇒)-quantified dictionary for *Show a* (see Section 2.1 if this statement surprises you), and the (→)-quantified argument of type *a*. However, we use *show* as, say, *show True*, passing only one argument visibly. The ∀ *a* argument is discovered by unification to be *Bool*, but the *Show a* argument is discovered using a different mechanism: instance solving and lookup. (See the work of Vytiniotis et al. [99] for the algorithm used.) We thus must be aware that invisible arguments may use different mechanisms for instantiation.

Dependent Haskell offers both visible and invisible forms of ∀ and Π; the invisible forms instantiate only via unification. Dependent Haskell retains, of course, the invisible quantifier (⇒), which is instantiated via instance lookup and solving. Finally, note that visibility is a quality only of source Haskell. All arguments are always "visible" in Pico.

It may be helpful to compare Dependent Haskell's treatment of visibility to that in other languages; see Section 8.6.

4.2.3.1 Visibility overrides

It is often desirable when using rich types to override a declared visibility specification. That is, when a function is declared to have an invisible parameter *a*, a call site may wish to instantiate *a* visibly. Conversely, a function may declare a visible parameter *b*, but a caller knows that the choice for *b* can be discovered by unification and so wishes to omit it at the call site.

Instantiating invisible parameters visibly Dependent Haskell adopts the @... syntax of Eisenberg et al. [33] to instantiate any invisible parameter visibly, whether it is a type or not. Continuing our example with *id*, we could write *id* @*Bool True* instead of *id True*. This syntax works in patterns, expressions, and types. In patterns, the choice of @ conflicts with as-patterns, such as using the pattern *list*@(*x* : *xs*) to bind *list* to the whole list while pattern matching. However, as-patterns are almost always

written without whitespace. I thus use the presence of whitespace before the @ to signal the choice between an as-pattern and a visibility override.[29] Dictionaries cannot be named in Haskell, so this visibility override skips over any constraint arguments.

Omitting visible parameters The function *replicate* :: Π (*n* :: *Nat*) → *a* → *Vec a n* from Section 3.1.1.3 creates a length-indexed vector of length *n*, where *n* is passed in as the first visible argument. (The true first argument is *a*, which is invisible and elided from the type.) However, the choice for *n* can be inferred from the context. For example:

```
theSimons :: Vec String 2
theSimons = replicate 2 "Simon"
```

In this case, the two uses of 2 are redundant. We know from the type signature that the length of *theSimons* should be 2. So we can omit the visible parameter *n* when calling *replicate*:

```
theSimons' :: Vec String 2
theSimons' = replicate _ "Simon"
```

The underscore tells GHC to infer the missing parameter via unification.

The two overrides can usefully be combined, when we wish to infer the instantiation of some invisible parameters but then specify the value for some later invisible parameter. Consider, for example, *coerce* :: ∀ *a b*. *Coercible a b* ⇒ *a* → *b*. In the call *coerce* (*MkAge* 3) (where we have **newtype** *Age* = *MkAge Int*), we can infer the value for *a*, but the choice for *b* is a mystery. We can thus say *coerce* @_ @*Int* (*MkAge* 3), which will convert *MkAge* 3 to an *Int*.

The choice of syntax for omitting visible parameters conflicts somewhat with the feature of *typed holes*, whereby a programmer can leave out a part of an expression, replacing it with an underscore, and then get an informative error message about the type of expression expected at that point in the program. (This is not unlike Agda's *sheds* feature or Idris's *metavariables* feature.) However, this is not a true conflict, as an uninferrable omitted visible parameter is indeed an error and should be reported; the error report is that of a typed hole. Depending on user feedback, this override of the underscore symbol may be hidden behind a language extension or other compiler flag.

[29] This perhaps-surprising decision based on whitespace is regrettable, but it has company. The symbol $ can mean an ordinary, user-defined operator when it is followed by a space but a Template Haskell splice when there is no space. The symbol . can mean an ordinary, user-defined operator when it is preceded by a space but indicate namespace resolution when it is not. Introducing these oddities seems a good bargain for concision in the final language.

4.2.4 Matchability

Suppose we know that $f\ a$ equals $g\ b$. What relationship can we conclude about the individual pieces? In general, nothing: there is no way to reduce $f\ a \sim g\ b$ for arbitrary f and g. Yet Haskell type inference must simplify such equations frequently. For example:

```
class Monad m where
  return :: a → m a
  ...
just5 :: Maybe Int
just5 = return 5
```

When calling *return* in the body of *just5*, type inference must determine how to instantiate the call to *return*. We can see that $m\ a$ (the return type of *return*) must be *Maybe Int*. We surely want type inference to decide to set m to *Maybe* and a to *Int*! Otherwise, much current Haskell code would no longer compile.

The reason it is sensible to reduce $m\ a \sim$ *Maybe Int* to $m \sim$ *Maybe* and $a \sim$ *Int* is that all type constructors in Haskell are generative and injective, according to these definitions:

Definition (Generativity)**.** *If f and g are* generative*, then $f\ a \sim g\ b$ implies $f \sim g$.*[30]

Definition (Injectivity)**.** *If f is* injective*, then $f\ a \sim f\ b$ implies $a \sim b$.*

Because these two notions go together so often in the context of Haskell, I introduce a new word *matchable*, thus:

Definition (Matchability)**.** *A function f is* matchable *iff it is generative and injective.*

Thus, we say that all type constructors in Haskell are matchable. Note that if f and g are matchable, then $f\ a \sim g\ b$ implies $f \sim g$ and $a \sim b$, as desired.

On the other hand, ordinary Haskell functions are not, in general, matchable. The inability to reduce $f\ a \sim g\ b$ to $f \sim g$ and $a \sim b$ for arbitrary functions is precisely why type families must be saturated in today's Haskell. If they were allowed to appear unsaturated, then the type inference algorithm could no longer assume that higher-kinded types are always matchable,[31] and inference would grind to a halt.

The solution is to separate out matchable functions from unmatchable ones, classifying each by their own quantifier, as described in my prior work [29].

The difference already exists in today's Haskell between a matchable arrow and an unmatchable arrow, though this difference is invisible. When we write an arrow in a

[30] As we see in this definition, *generativity* is really a relation between pairs of types. We can consider the type constructors to be a set such that any pair are generative w.r.t. the other. When I talk about a type being generative, it is in relation to this set.

[31] For example, unifying $a\ b$ with *Maybe Int* would no longer have a unique solution.

Quantifier	Dependency	Relevance	Visibility	Matchability
$\forall\ (a :: \tau).\ ...$	dep.	irrel.	inv. (unification)	unmatchable
$\text{'}\forall\ (a :: \tau).\ ...$	dep.	irrel.	inv. (unification)	matchable
$\forall\ (a :: \tau) \to ...$	dep.	irrel.	vis.	unmatchable
$\text{'}\forall\ (a :: \tau) \to ...$	dep.	irrel.	vis.	matchable
$\Pi\ (a :: \tau).\ ...$	dep.	rel.	inv. (unification)	unmatchable
$\text{'}\Pi\ (a :: \tau).\ ...$	dep.	rel.	inv. (unification)	matchable
$\Pi\ (a :: \tau) \to ...$	dep.	rel.	vis.	unmatchable
$\text{'}\Pi\ (a :: \tau) \to ...$	dep.	rel.	vis.	matchable
$\tau \Rightarrow ...$	non-dep.	rel.	inv. (solving)	unmatchable
$\tau\ \text{'}\Rightarrow ...$	non-dep.	rel.	inv. (solving)	matchable
$\tau \to ...$	non-dep.	rel.	vis.	unmatchable
$\tau\ \text{'}\to ...$	non-dep.	rel.	vis.	matchable

Figure 4.1: The twelve quantifiers of Dependent Haskell

type that classifies an expression, that arrow is unmatchable. But when we write an arrow in a kind that classifies a type, the arrow is matchable. This is why $map :: (a \to b) \to [a] \to [b]$ does *not* cleanly promote to the type $Map :: (a \to b) \to [a] \to [b]$; if you write that type family, it is much more restrictive than the term-level function.

The idea of matchability also helps to explain why, so far, we have been able only to promote data constructors into types: data constructors are matchable—this is why pattern matching on constructors makes any sense at all. When we promote a data constructor to a type constructor, the constructor's matchable nature fits well with the fact that all type constructors are matchable.

Dependent Haskell thus introduces a new arrow, spelled '→, that classifies matchable functions. The idea is that ' is used to promote data constructors, and '→ promotes the arrow used in data constructor types. In order to be backward compatible, types of type constructors (as in **data** $Vec :: \mathbf{Type} \to Nat \to \mathbf{Type}$) and types of data constructors (as in $Just :: a \to Maybe\ a$) can still be written with an ordinary arrow, even though those arrows should properly be '→. Along similar lines, any arrow written in a stretch of Haskell that is lexically a kind (that is, in a type signature in a type) is interpreted as '→ as long as the `DependentTypes` extension is not enabled.

We can now say $'map :: (a \to b) \to [a] \to [b]$, with unmatchable →, and retain the flexibility we have in the expression map.

4.2.5 The twelve quantifiers of Dependent Haskell

Now that we have enumerated the quantifier properties, we are ready to describe the twelve quantifiers that exist in Dependent Haskell. They appear in Figure 4.1. The first one ($\forall\ (a :: t).\ ...$) and two near the bottom (⇒ and →) exist in today's Haskell and are completely unchanged. Dependent Haskell adds a visible ∀, the Π quantifiers,

and matchable versions of everything.[32]

It is expected that the matchable quantifiers will be a rarity in user code. These quantifiers are used to describe type and data constructors, but matchability is assumed in a type or data constructor signature. Beyond those signatures, I don't imagine many users will need to write matchable function types. However, there is no reason to *prevent* users from writing these, so I have included them in the user-facing design.

The visible ∀ is useful in situations where a type parameter might otherwise be ambiguous. For example, suppose F is a non-injective [86] type family and consider this:

$$frob :: \forall\ a.\ F\ a \to F\ [a]$$

This type signature is inherently ambiguous—we cannot know the choice of a even if we know we want a such that $frob :: Int \to Bool$—and GHC reports an error when it is written. Suppose that we know we want a particular use of $frob$ to have type $Int \to Bool$. Even with that knowledge, there is no way to determine how to instantiate a. To fix this problem, we simply make a visible:

$$frob :: \forall\ a \to F\ a \to F\ [a]$$

Now, any call to $frob$ must specify the choice for a, and the type is no longer ambiguous.

A Π-quantified parameter is both dependent (it can be used in types) and relevant (it can be used in terms). Critically, pattern-matching (in a term) on a Π-quantified parameter informs our knowledge about that parameter as it is used in types, a subject we explore in the next section.

Lastly, Dependent Haskell omits the non-dependent, irrelevant quantifiers, as a non-dependent, irrelevant quantifiee would not be able to be used anywhere.

4.3 Pattern matching

We will approach an understanding of pattern matches in stages, working through three examples of increasing complexity. All these examples will work over the somewhat hackneyed length-indexed vectors for simplicity and familiarity.

[32] The choice of syntax here is directly due to the work of Gundry [37].

4.3.1 A simple pattern match

Naturally, Dependent Haskell retains the capability for simple pattern matches:

```
  -- isEmpty :: Vec a n → Bool
isEmpty v = case v of
  Nil → True
  _   → False
```

A simple pattern match looks at a *scrutinee*—in this case, *v*—and chooses a **case** alternative depending on the value of the scrutinee. The bodies of the **case** alternatives need no extra information to be well typed. In this case, every body is clearly a *Bool*, with no dependency on which case has been chosen. Indeed, swapping the bodies would yield a well typed pattern match, too. In a simple pattern match, no type signature is required.[33]

4.3.2 A GADT pattern match

Today's Haskell (and Dependent Haskell) supports GADT pattern-matches, where learning about the constructor that forms a scrutinee's value can affect the types in a **case** alternative body. Here is the example:

```
pred :: Nat → Nat
pred Zero     = error "pred Zero"
pred (Succ n) = n

safeTail :: Vec a n → Either (n :~: 'Zero) (Vec a ('pred n))
safeTail Nil       = Left Refl
safeTail (_ :> t) = Right t
```

In this example, we must use type information learned through the pattern match in order for the body of the pattern match to type-check. (Here, and in the last example, I use the more typical syntax of defining a function via pattern matching. The reasoning is the same as if I had used an explicit **case**.) Let's examine the two pattern match bodies individually:

- For *Left Refl* to be well typed at *Either* $(n :\sim: \text{'}Zero)\ \tau$, we need to know that *n* is indeed *'Zero*. This fact is known only because we have pattern-matched on *Nil*. Note that the type of *Nil* is *Vec a 'Zero*. Because we have discovered that our argument of type *Vec a n* is *Nil* :: *Vec a 'Zero*, it must be that $n \sim \text{'}Zero$, as desired.

[33] Expert readers may be puzzled why this example is accepted without a type signature. After all, pattern-matching against *Nil* indeed *does* introduce a type equality, making the result type of the match hard to infer. In this case, however, the existence of the last pattern, _, which introduces no equalities, allows the return type to be inferred as *Bool*.

- For *Right t* to be well typed at *Either* τ (*Vec a* (*'pred n*)) (where *t* :: *Vec a n'* for some *n'*), we need to know that *n* $\sim$ *'Succ n'*, so that we can simplify *'pred n* to *'pred* (*'Succ n'*) to *n'*. The equality *n* $\sim$ *'Succ n'* is exactly what we get by pattern-matching on :>.

Note that I have provided a type signature for *safeTail*. This is necessary in the event of a GADT pattern match, because there is no way, in general, to infer the return type of a pattern match where each branch has a type equality in scope.[34]

4.3.3 Dependent pattern match

New to Dependent Haskell is the dependent pattern match, shown here:

```
replicate :: Π n → a → Vec a n
replicate Zero      _ = Nil
replicate (Succ n') x = x :> replicate n' x
```

Let's again consider the function bodies one at a time:

- Its type signature tells us *Nil* has type *Vec a 'Zero*. Thus for *Nil* to be well typed in *replicate*, we must know that *n* $\sim$ *'Zero*. We indeed do know this, as we have scrutinized *n* and found that *n* is *'Zero*.

- For the recursive call to be well typed, we need to know that *n* $\sim$ *'Succ n'*, which is, once again, what we know by the pattern match.

Note the difference between this case of dependent pattern match and the previous case of GADT pattern match. In GADT pattern matching, the equality assumption of interest is found by looking at the type of the constructor that we have found. In a dependent pattern match, on the other hand, the equality assumption of interest is between the scrutinee and the constructor. In our case here, the scrutinized value is not even of a GADT; *Nat* is a perfectly ordinary, Haskell98 datatype.

A question naturally comes up in this example: when should we do dependent pattern match and when should we do a traditional (non-dependent) pattern match? A naive answer might be to always do dependent pattern matching, as we can always feel free to ignore the extra, unused equality if we do not need it. However, this would not work in practice—with an equality assumption in scope, we cannot accurately infer the return type of a pattern match. Yet this last problem delivers us the solution: *use dependent pattern matching only when we know a match's result type*, as propagated down via a bidirectional type system. (This is much the same way that today's Haskell allows inference in the presence of higher-rank types [74]. See Section 6.4 for the

[34] If this last statement is a surprise to you, the introduction of Vytiniotis et al. [99] has a nice explanation of why this is a hard problem.

details.) If we know a result type and do not need the dependent pattern match equality, no harm is done. On the other hand, if we do not know the result type, this design decision means that dependent pattern matching does not get in the way of inferring the types of Haskell98 programs.

4.4 Discussion

The larger syntactic changes to Haskell as it becomes Dependent Haskell are sketched above. In addition to these changes, Haskell's typing rules naturally become much more involved. Though a declarative specification remains out of reach, Chapter 6 describes (and Appendix D details) the algorithm BAKE, which is used to detect type-correct Dependent Haskell programs. It is important future work to develop a more declarative specification of Dependent Haskell.

This section comments on several topics that affect the design of Dependent Haskell.

4.4.1 Type : Type

Dependent Haskell includes the **Type** : **Type** axiom, avoiding the infinite hierarchy of sorts [57, 80] that appear in other dependently-typed languages. This choice is made solely to simplify the language. Other languages avoid the **Type** : **Type** axiom in order to remain consistent as a logic. However, to have logical consistency, a language must be total. Haskell already has many sources of partiality, so there is little risk in adding one more.

Despite the questionable reputation of the **Type** : **Type** axiom, languages with this feature have been proved type-safe for some time. Cardelli [12] gives a thorough early history of the axiom and presents a type-safe language with **Type** : **Type**. Given the inherent partiality of Haskell, the inclusion of this axiom has little effect on the theory.

4.4.2 Inferring Π

The discussion of quantifiers in this chapter begs a question: which quantifier is chosen when the user has not written any? The answer: →. Despite all of the advances to the type system that come with Dependent Haskell, the non-dependent, relevant, visible, and unmatchable function type, →, remains the bedrock. In absence of other information, this is the quantifier that will be used.

However, as determined by the type inference process (Chapter 6), an inferred type might still have a Π in it. For example, if I declare

$$\mathit{replicate'} = \mathit{replicate}$$

without giving a type signature to *replicate'*, it should naturally get the same type (which includes a Π) as *replicate*. Indeed this is what is delivered by BAKE, Dependent Haskell's type inference algorithm.

On the other hand, the generalized type of the expression $\lambda f\ g\ x \to f\ (g\ x)$ is $\forall\ a\ b\ c.\ (b \to c) \to (a \to b) \to (a \to c)$, the traditional type for function composition, not the much more elaborate type (see Section 6.1) for a dependently typed composition function. The more exotic types are introduced only when written in by the user.

4.4.3 Roles and dependent types

Integrating dependent types with Haskell's *role* mechanism [11] is a challenge, as explored in some depth in my prior, unpublished work [27]. Instead of addressing this issue head-on, I am deferring the resolution until we can find a better solution than what was proposed in that prior work. That approach, unworthy of being repeated here, is far too ornate and hard to predict. Instead, I make a simplifying assumption that all coercions used in types have a nominal role.[35] This choice restricts the way Haskell **newtype**s can work with dependent types if the *coerce* function has been used. A violation of this restriction (yet to be nailed down, exactly) can be detected after type-checking and does not affect the larger type system. It is my hope that, once the rest of Dependent Haskell is implemented, a solution to this thorny problem will present itself. A leading, unexplored candidate is to have two types of casts: representational and nominal. Currently, all casts are representational; possibly, tracking representational casts separately from nominal casts will allow a smoother integration of roles and dependent types than does the ornate approach in my prior work.

4.4.4 Impredicativity, or lack thereof

Despite a published paper [97] and continued attempts at cracking this nut, GHC lacks support for impredicativity.[36] Here, I use the following definitions in my meaning of impredicativity, which has admittedly drifted somewhat from its philosophical origins:

Definition (Simple types)**.** *A* simple type *has no constraint, quantification, or dependency.*

Definition (Impredicativity)**.** *A program is* impredicative *if it requires a non-simple type to be substituted for a type variable.*

Impredicativity is challenging to implement while retaining predictable type inference, essentially because it is impossible to know where to infer invisible arguments—invisible arguments can be hidden behind a type variable in an impredicative type system.

Dependent Haskell does not change this state of affairs in any way. In Dependent Haskell, just like in today's Haskell, impredicativity is simply not allowed.

[35] If you are not familiar with roles, do not fret. Instead, safely skip the rest of this subsection.

[36] There does exist an extension `ImpredicativeTypes`. However, it is unmaintained, deprecated, and quite broken.

There is a tantalizing future direction here, however: are the restrictions around impredicativity due to invisible binders only? Perhaps. Up until now, it has been impossible to have a dependent or irrelevant binder without that binder also being invisible. (To wit, $\forall$ is the invisible, dependent, irrelevant binder of today's Haskell.) One of the tasks of enhancing Haskell with dependent types is picking apart the relationship among all of the qualities of quantifiers [56]. It is conceivable that the reason impredicativity hinders the predictability of type inference has to do only with visibility, allowing arbitrary instantiations of type variables with complex types, as long as they have no invisible binders. Such an idea requires close study before implementing, but by pursuing this idea, we may be able to relax the impredicativity restriction substantially.

4.4.5 Running proofs

Haskell is a partial language. It has a multitude of ways of introducing a computation that does not reduce to a value: $\bot$/*error*, general recursion, incomplete pattern matches, non-strictly-positive datatypes, baked-in type representations [75], and possibly Girard's paradox [36, 48], among others. This is in sharp contrast to many other dependently typed language, which are total. (An important exception is Cayenne. See Section 8.3.)

In a total language, if you have a function *pf* that results in a proof that $a \sim b$, you never need to run the function. (Here, I'm ignoring the possibility of multiple, different proofs of equality [91].) By the totality of that language, you are assured that *pf* will always terminate, and thus running *pf* yields no information.

On the other hand, in a partial language like Haskell, it is always possible that *pf* diverges or errors. We are thus required to run *pf* to make sure that it terminates. This is disappointing, as the only point of running *pf* is to prove a type equality, and types are supposed to be erased. However, the Haskell function *pf* has two possible outcomes: an uninformative (at runtime) proof of type equality, or divergence. There seems to be no easy, sound way around this restriction, which will unfortunately have a real effect on the runtimes of dependently typed Haskell programs.[37]

Despite not having an easy, sound workaround, GHC already comes with an easy, *un*sound workaround: rewrite rules [73]. A rewrite rule (written with a *RULES* pragma) instructs GHC to exchange one fragment of a program in its intermediate language with another, by pattern matching on the program structure. For example, a user can write a rule to change *map id* to *id*. To the case in point, a user could write a rule that changes *pf* ... to *unsafeCoerce Refl*. Such a rule would eliminate the possibility of a runtime cost to the proof. By writing this rule, the user is effectively asserting that the proof always terminates.

[37] Note that running a term like *pf* is the *only* negative consequence of Haskell's partiality. If, say, Agda always ran its proofs, it could be partial, too! This loses logical consistency—and may surprise users expecting something that looks like a proof to actually be a proof—but the language would remain type safe.

4.4.6 Import and export lists

Recall the *safeTail* example from Section 4.3.2. As discussed in that section, for *safeTail* to compile, it is necessary to reduce *'pred* (*'Succ n'*) to *n'*. This reduction requires knowledge of the details of the implementation of *pred*. However, if we imagine that *pred* is defined in another module, it is conceivable that the author of *pred* wishes to keep the precise implementation of *pred* private—after all, it might change in future versions of the module. Naturally, hiding the implementation of *pred* would prevent an importing module from writing *safeTail*, but that should be the library author's prerogative.

Another way of examining this problem is to recognize that the definition of *pred* encompasses two distinct pieces of information: *pred*'s type and *pred*'s body. A module author should have the option of exporting the type without the body.

This finer control is done by a small generalization of the syntax in import and export lists. If a user includes *pred* in an import/export list, only the name *pred* and its type are involved. On the other hand, writing *pred*(..) (with a literal (..) in the source code) in the import/export list also includes *pred*'s implementation. This echoes the current syntax of using, say, *Bool* to export only the *Bool* symbol while *Bool*(..) exports *Bool* with all of its constructors.

4.4.7 Type-checking is undecidable

In order to type-check a Dependent Haskell program, it is sometimes necesary to evaluate expressions used in types. Of course, these expressions might be non-terminating in Haskell. Accordingly, type-checking Dependent Haskell is undecidable.

This fact, however, is not worrisome. Indeed, GHC's type-checker has had the potential to loop for some time. Assuming that the solver's own algorithm terminates, type-checking will loop only when the user has written a type-level program that loops. Programmers are not surprised when they write an ordinary term-level program that loops at runtime; they should be similarly not surprised when they write a type-level program that loops at compile time. In order to provide a better user experience, GHC counts reduction steps and halts with an error message if the count gets too high; users can disable this check or increase the limit via a compiler flag.

4.5 Conclusion

This chapter has offered a concrete description of Dependent Haskell. Other than around the addition of new quantifiers, most of the changes are loosening of restrictions that exist in today's Haskell. (For example, a ' mark in a type today can promote only a constructor; Dependent Haskell allows any identifier to be so promoted.) Accordingly, and in concert with the conservativity of the type inference algorithm (Sections 6.8.2 and 6.8.3), programs that compile today will continue to do so under Dependent Haskell.

Naturally, what is described here is just my own considered vision for Dependent Haskell. I am looking forward to the process of getting feedback from the Haskell community and evolving this description of the language to fit the community's needs.

Chapter 5

PICO: The intermediate language

This chapter presents PICO, the internal language that Dependent Haskell compiles into. I have proved type safety (via the usual preservation and progress theorems, Theorem C.46 and Theorem C.78) and type erasure (Theorem C.83 and Theorem C.86). I believe PICO would make a strong candidate for the internal language in a future version of GHC.

5.1 Overview

PICO (pronounced "Π-co", never "peek-o") descends directly from the long line of work on System FC [87]. It is most closely related to the version of System FC presented in my prior work [105] and in Gundry's thesis [37].

PICO sits in the λ-cube [6] on the same vertex as the Calculus of Constructions [19], but with a very different notion of equality. A typical dependently typed calculus contains a *conversion* rule, something like this:

$$\frac{\tau : \kappa_1 \qquad \kappa_1 \equiv \kappa_2}{\tau : \kappa_2} \ \textsc{Conv}$$

This rule encapsulates the point of type equivalence: if a type τ is found to have some kind κ_1 and κ_1 is known to be equivalent to some κ_2, then we can say that τ has kind κ_2.[38] This rule is flexible and helps a language to be succinct. It has a major drawback, however: it is not syntax directed. In general, determining whether $\kappa_1 \equiv \kappa_2$ might not be easy. Indeed, type equivalence in PICO is undecidable, so we would have a hard time building a type-checker with a CONV rule such as this one. Other dependently typed languages are forced to restrict expressiveness in order to keep

[38] I tend to use the word "kind" when referring to the classification of a type. However, in the languages considered in this dissertation, kinds and types come from the same grammar; the terms "type" and "kind" are technically equivalent. Nevertheless, I find that discerning between these two words can aid intuition and will continue to do so throughout the dissertation.

type-checking decidable; this need for decidable type equivalence is one motivation to design a dependently typed language to be strongly normalizing.

PICO's approach to type equivalence (and the CONV rule) derives from the *coercions* that provide the "C" in "System FC". Instead of relying on a non-syntax-directed equivalence relation, PICO's type equivalence requires evidence of equality in the form of coercions. Here is a simplified version of PICO's take on the CONV rule:

$$\frac{\tau : \kappa_1 \qquad \gamma : \kappa_1 \sim \kappa_2}{\tau \rhd \gamma : \kappa_2} \quad \text{TY\_CAST}$$

In this rule, the metavariable γ stands for a *coercion*, a proof of the equality between two types. Here, we see that γ proves that kinds κ_1 and κ_2 are equivalent. Thus, we can type $\tau \rhd \gamma$ at κ_2 as long as τ can be typed at κ_1. Note the critical appearance of γ in the conclusion of the rule: this rule is syntax-directed. The type-checker simply needs to check the equality proofs against a set of (also syntax-directed) rules, not to check some more general equivalence relation.

The grammar for coercions (in Figure 5.1 on page 76) allows for a wide variety of coercion forms, giving PICO a powerful notion of type equivalence. However, coercions have no notion of evaluation nor proper λ-abstractions.[39] Thus, the fact that evaluation in PICO might not terminate does not threaten the type safety of the language. Coercions are held separate from types, and proving consistency of the coercion language (Section 5.10)—in other words, that we cannot prove $\mathsf{Int} \sim \mathsf{Bool}$—is the heart of the type safety proof. It does not, naturally, depend on any termination proof, nor any termination checking of the program being checked. The independence of PICO's type safety result from termination means that PICO can avoid many potential traps that have snagged other dependently typed languages that rely on intricate termination checks.[40]

5.1.1 Features of PICO

PICO is a dependently typed λ-calculus with mutually recursive algebraic datatypes and a fixpoint operator. Recursion is modeled only via this fixpoint operator; there is no recursive **let**. Other than the way in which the operational semantics deals with coercions in the form of *push rules*, the small-step semantics is what you might expect for a call-by-name λ-calculus.

The typing relations, however, have a few features worth mentioning up front (other unusual features are best explained after the detailed coverage of PICO; see Section 5.12).

[39] There is a coercion form that starts with λ; it is only a congruence form for λ-abstractions in types, not a λ-abstraction in the coercion language. See Section 5.8.5.1.

[40] For example, see `https://coq.inria.fr/cocorico/CoqTerminationDiscussion`.

5.1.1.1 Relevance annotations and type erasure

A key concern when compiling a dependently typed language is type erasure. Given that terms and types can intermingle, what should be erased during compilation? And what data is necessary to be retained until runtime? Dependent Haskell (and, in turn, Pico) forces the user to specify this detail at each quantifier (Section 5.3). In the formal grammar of Pico, we distinguish between $\Pi a{:}_{\mathsf{Rel}}\kappa.\ldots$ and $\Pi a{:}_{\mathsf{Irrel}}\kappa.\ldots$. The former is the type of an abstraction that is retained at runtime, written with a Π in Haskell; the latter, written with $\forall$, is fully erased. In order to back up this claim of full erasure of irrelevant quantification, evaluation happens under irrelevant abstractions; see Section 5.7.1.

So that we can be sure a variable's relevance is respected at use sites, variable contexts Γ track the relevance of bound variables. Only *relevant* variables may appear in the "level" in which they were bound; when a typing premise refers to a higher "level", the context is altered to mark all variables as relevant. For example, the **case** construct $\mathbf{case}_\kappa\,\tau\,\mathbf{of}\,\overline{alt}$ includes the return kind of the entire **case** expression as its κ subscript. This kind is type-checked in a context where all variables are marked as relevant; because the kind is erased during compilation, the use of an irrelevant variable there is allowed. As they are also erased, coercions are considered fully irrelevant as well.

My treatment of resetting the context is precisely like what is done by Mishra-Linger and Sheard [65].

5.1.1.2 Tracking matchable vs. unmatchable functions

Dependent Haskell supports both matchable—that is, generative and injective—abstractions and unmatchable ones (Section 4.2.4). Though at first it might appear that separating out these two modalities is necessary only to support type inference, Pico maintains this distinction. Every Π-type in Pico is labeled as either matchable or unmatchable: $\pi\Pi$ denotes a matchable Π-type and $\underset{\sim}{\Pi}$ denotes an unmatchable one. An unadorned Π is a metavariable which might be instantiated either to $\pi\Pi$ or $\underset{\sim}{\Pi}$. We do not have to label λ-abstractions, however, because all λ-abstractions are always unmatchable—only partially applied type constants (or functions returning them) are matchable.

Pico maintains the matchable/unmatchable distinction for two reasons:

Decomposing coercions over function applications Since at least the invention of System FC [87], GHC has supported application decomposition. That is, from a proof that $\tau_1\,\sigma_1$ equals $\tau_2\,\sigma_2$, we can derive proofs of $\tau_1 \sim \tau_2$ and $\sigma_1 \sim \sigma_2$. I would like to retain this ability in Pico in order to support the claim that Dependent Haskell is a conservative extension over today's Haskell. However, decomposing an application

as above in the presence of unsaturated λ-abstractions is clearly bogus.[41]

The solution here is to keep matchable applications separate from unmatchable ones, and allow decomposition only of matchable applications. The two application forms comprise different nodes in the PICO grammar. Decomposing only matchable applications is a backward-compatible treatment, as today's Haskell has only matchable applications. In turn, keeping the application forms separate requires tracking the matchability of the abstractions themselves.

PICO's support of the application decomposition while allowing unsaturated λ-abstractions is one of the key improvements PICO makes over Gundry's *evidence* language [37]. See Section 8.1 for more discussion of the comparison of my work to Gundry's.

Matching on partially applied constants PICO does not contain type families. Instead, it uses λ-abstractions and **case** expressions, as these are more familiar to functional programmers. And yet, I wish for PICO to support the variety of ways in which type families are used in today's Haskell. One curiosity of today's Haskell is that it allows matching on partially applied data constructors:

```
type family IsLeft a where
  IsLeft 'Left  = 'True
  IsLeft 'Right = 'False
```

The type family *IsLeft* is inferred to have kind $\forall\ k.\ (k \to \mathit{Either}\ k\ k) \to \mathit{Bool}$. (Note that $k \to \mathit{Either}\ k\ k$ is what you get when unifying the kind of *Left* with that of *Right*.) That is, it matches on the *Left* and *Right* constructors, even though these are not applied to arguments. While it may seem that *IsLeft* is matching on a *function*—after all, the type of *IsLeft*'s argument appears to be an arrow type—it is not. It is matching only on constructors, because today's kind-level $\to$ classifies only type constants. That is, it really should be spelled $'\to$.

To support functions such as *IsLeft*, PICO allows **case** scrutinees to have matchable $'\Pi$-types, instead of just fully applied datatypes. As designed here, matching on partially applied data constructors is also available at the term level in PICO. However, practical considerations (e.g., how would you compile such a match?) may lead us to prevent the use of this feature from surface Haskell.

5.1.1.3 Matching on Type

Today's Haskell also has the ability, through its type families, to match on members of **Type**. For example:

[41] For example, we can prove $(\lambda x{:}_{\mathsf{Rel}}\mathit{Int}.\,3)\,4 \sim (\lambda x{:}_{\mathsf{Rel}}\mathit{Int}.\,3)\,5$ but do not wish to be able to prove $4 \sim 5$.

```
type family IntLike x where
  IntLike Integer = 'True
  IntLike Int     = 'True
  IntLike _       = 'False
```

This ability for a function to inspect the choice of a type—and not a code for a type—is unique among production languages to Haskell, as far as I am aware. With the type families in today's Haskell, discerning between types is done by simple pattern matching. However, if we compile type families to **case** statements, we need a way to deal with this construct, even though **Type** is not an algebraic datatype.

Fortunately, types like *Either* resemble data constructors like *Just*: both are classified by matchable quantification(s) over a type headed by another type constant. In the case of *Either*, we have $\mathit{Either} : \Pi\_{:\mathsf{Rel}}\mathbf{Type}, \_{:\mathsf{Rel}}\mathbf{Type}.\,\mathbf{Type}$;[42] note that the body of the Π-type is headed by the constant **Type**. For *Just*, we have $\mathit{Just}_{\{a\}} : \Pi\_{:\mathsf{Rel}}a.\,\mathit{Maybe}\ a$.[43] With this similarity, it is not hard to create a typing rule for a **case** statement that can handle both data constructors (like *Just*) and types (like *Either*).

A key feature, however, that is needed to support matching on **Type** is default patterns. For a closed datatype, where all the constructors can be enumerated, default patterns are merely a convenience; any default can be expanded to list all possible constructors. For an open type, like **Type**, the availability of the default pattern is essential. It is for this reason alone that I have chosen to include default patterns in Pico.

5.1.1.4 Hypothetical equality

Pico allows abstraction over coercions, much like any λ-calculus allows abstraction over expressions (or, in a call-by-value calculus, values). Coercion abstraction means that a type equality may be *assumed* in a given type. When we wish to evaluate a term that assumes an equality, we must apply that term to evidence that the equality holds—an actual coercion. It is this ability, to assume an equality, that allows Pico to have GADTs. See the example in Section 5.5 for the details.

5.1.2 Design requirements for Pico

In the course of any language design, there needs to be a guiding principle to aid in making free design decisions. The chief motivator for the design of Pico is that it should be suitable for use as the internal language of a Haskell compiler. This use case provides several desiderata:

Decidable, syntax-directed, efficient type checking The use of types in a compiler's intermediate language serves only as a check of the correctness of the

[42] Why Rel? See the end of Section 5.4.2.2.

[43] The $\{a\}$ subscript is explained in Section 5.4.1.

compiler. Any programmer errors are caught before the intermediate language code is emitted, and so a correct compiler should only produce well typed intermediate-language programs, if it produces such programs at all. In addition, a correct compiler performing program transformations on the intermediate language should take a well typed program to a well typed program. However, not all compilers are correct, and thus it is helpful to have a way to check that intermediate-language program generation and transformation is at least type-preserving. To check this property, we need to type-check the intermediate language, both after it is originally produced and after every transformation. It thus must be easy and efficient to do so.

Pico essentially encodes a typing derivation right in the syntax of types and coercions. It is thus very easy to write a type checker for the language. Type-checking is manifestly decidable and can be done in one pass over the program text, with no constraint solving.[44] Pico's lack of a termination requirement also significantly lowers the burden of implementation of a type checker for the language.

Erasability An intermediate-language program should make clear what information can be erased at runtime. After all, when the compiler is done performing optimizations, runtime code generation must take place, and we thus need to know what information can be dropped. It is for this reason that Pico includes the relevance annotations.

A balance between ease of proving and ease of implementation Pico serves two goals: to be a template for an implementation, and also to be a calculus used to prove type safety. These goals are sometimes at odds with each other.

These two goals of System FC have tugged in different directions since the advent of FC. Historically, published versions of the language have greatly simplified certain details. No previously published treatment of FC has included support for recursion, either through **letrec** or **fix**. In contrast, the implemented version of FC (also called GHC Core) makes certain choices for efficiency; for example, applied type constructors, such as *Either Int Bool*, have a different representation than do applied type variables, such as *a Int Bool*. The former is stored as the head constructor with a list of arguments, and the latter is stored as nested binary applications. This is convenient when implementing but meddlesome when proving properties. The divergence between published FC and the implemented version (more often called GHC Core) have led to a separate document just to track the implemented version [26].

In the design of Pico, I have aimed for balance between these two needs. Because of the risk that non-termination might cause unsoundness, I have explicitly included **fix** in the design, just to make sure that the non-termination is obvious.[45] I have

[44] I do not claim that it is strictly linear, as a formal analysis of its running time is beyond the scope of this dissertation. In particular, one rule (see Section 5.6.5) requires the use of a unification algorithm and likely breaks linearity.

[45] With **Type** : **Type**, we have the possibility of Girard's paradox [36, 48] and thus can have non-termination even without **fix**, but making the non-termination more obvious clarifies that we can achieve type safety without termination.

not, however, included an explicit **let** or **letrec** construct, as the specification of these would be quite involved, and yet desugaring these constructs into λ and **fix** is straightforward. (See Section 5.13.1.)

On the other hand, I have included **case**. Having **case** in the language also significantly complicates the presentation, but here in a useful way: the existence of **case** (over unsaturated constructors) motivates the distinction between $\underset{\sim}{\Pi}$ and Π. The desugaring of **case** into recursive types built, say, with **fix** is not nearly as simple as the desugaring of **let**.

In the end, choices such as these are somewhat arbitrary and come down to taste. I believe that the choices I have made here bring us to a useful formalization with the right points of complexity. Some of these design decisions are considered in more depth after Pico has been presented; see Section 5.12.

5.1.3 Other applications of Pico

It is my hope that Pico sees application beyond just in Haskell. In designing it, I have tried to permit certain Haskell idioms (call-by-name semantics, the extra capabilities of **case** expressions outlined above) while still retaining a general enough flavor that it could be adapted to other settings. I believe that the arguments above about Pico's design mean that it is a suitable starting point for the design of an intermediate language for any dependently typed surface language. Other uses might want call-by-value instead of call-by-name or to remove the somewhat fiddly distinction between Π and $\underset{\sim}{\Pi}$. These changes should be rather straightforward to make.

In certain areas, I have decided not to support certain existing Haskell constructs directly in Pico because doing so would clutter the language, making its applicability beyond Haskell harder to envision. Various extensions of Pico—which would likely appear in an implementation of Pico within GHC—are discussed in Section 5.13. These include representation polymorphism and support for the $(\rightarrow)$ type constructor, for example.

5.1.4 No roles in Pico

Recent versions of System FC have included *roles* [11], which distinguish between two different notions of type equality: nominal equality is the equality relation embodied in Haskell's $(\sim)$ operator, whereas representational equality relates types that have bit-for-bit identical runtime representations. Tracking these two equality relations is important for allowing zero-cost conversions between types known to have the same representation, and it is an important feature to boost performance of programs that use **newtype** to enforce abstraction.

However, roles greatly clutter the language and its proofs. Including them throughout this dissertation would distract us from the main goal of understanding a dependently typed language with **Type** : **Type** and at ease with non-termination. It is for this reason that I have chosen to omit roles entirely from this work. (See also

Section 4.4.3 for a consideration of how roles interacts with the surface language proposed here.) I am confident that, in time, roles can be integrated with the language presented here, perhaps along the lines I have articulated in a draft paper [27], though the treatment there still leaves something to be desired. Regardless of clutter, having a solid approach to combining roles with dependent types will be a prerequisite of releasing a performant implementation of dependent types in GHC.

5.2 A formal specification of Pico

The full grammar of Pico appears in Figure 5.1 on the next page and notation conventions appear in Figure 5.2 on page 77. We will cover these in detail in the following sections. Later sections of this chapter will cover portions of the typing rules, but for a full listing of all the typing rules of the language, please see Appendix B. Figure 5.3 on page 78 includes the judgment forms and two key lemmas, useful in understanding the judgments. All of the metatheory lemmas, theorems, and proofs appear in Appendix C. This chapter mentions several key lemmas and theorems, but the ordering here is intended for readability and lemma statements may be abbreviated; please see the appendix for the correct dependency ordering and full statements.

You will see that the Pico language is centered around what I call types, represented by metavariables τ, σ, and κ. As Pico is a full dependently typed language with a unified syntax for terms, types, and kinds, this production could be called "expressions" and could be assigned the metavariable e. However, I have decided to reserve e (and the moniker "expression") for *erased* expressions only, after all the types have been removed. These expressions are used only in the type erasure theorem (Section 5.11); the rest of the metatheory is about types. Nevertheless, a program written in Pico intended to be run will technically be a type, and types in Pico have an operational semantics (Section 5.7).

As previewed in Section 5.1.1.2, Pico supports two different forms of Π-type: the matchable $\dot{\Pi}$ and the unmatchable $\tilde{\Pi}$. It also supports two forms of application: τ_ψ is a matchable application and $\tau\_\sim\psi$ is an unmatchable one. However, labeling all applications would grossly clutter this presentation, and so I just write $\tau\,\psi$ for both kinds of applications, where we can discern between them by looking at τ's kind. Indeed, the only reason that the grammar has to distinguish between the two applications at all is in the consistency proof (Section 5.10), a portion of which works in an untyped setting. (See, in particular, the end of Section 5.10.2 for the one place where labeling the applications is used.) It is not expected that an implementation of Pico would need to mark the applications, as this mark is redundant with the typing information.

Note also the definition for arguments ψ: the application form $\tau\,\psi$ applies a type to an argument, which can be a type, an irrelevant type, or a coercion. It would be equivalent to have six[46] productions in the definition for types, but having a separate

[46]Product of two application modes (matchable vs. unmatchable) and three relevance modes (type

Metavariables:

T	algebraic datatype	K	data constructor
$a, b, x, \_$	type/term variable	c	coercion variable
i, j, k, n	natural number/index		

Π	::=	Π	matchable dep. quantifier
	|	$\underset{\sim}{\Pi}$	unmatchable dep. quantifier
z	::=	$a \mid c$	type or coercion variable
H	::=	$T \mid K \mid \mathbf{Type}$	constant
ρ	::=	$\mathsf{Rel} \mid \mathsf{Irrel}$	relevance annotation
δ	::=	$a{:}_\rho \kappa \mid c{:}\phi$	binder
ϕ	::=	$\tau_1 \ {}^{\kappa_1}\!\sim^{\kappa_2} \tau_2$	heterogeneous equality
τ, σ, κ	::=	$a \mid \tau \_ \psi \mid \tau \underset{\sim}{} \psi \mid \Pi\delta.\,\tau \mid \lambda\delta.\,\tau$	dependent types
	|	$H_{\{\overline{\tau}\}}$	constant applied to universals
	|	$\tau \rhd \gamma$	kind cast
	|	$\mathbf{case}_\kappa\, \tau\, \mathbf{of}\, \overline{alt}$	case-splitting
	|	$\mathbf{fix}\, \tau$	recursion
	|	$\mathbf{absurd}\, \gamma\, \tau$	absurdity elimination
ψ	::=	$\tau \mid \{\tau\} \mid \gamma$	argument
alt	::=	$\pi \to \tau$	case alternative
π	::=	$H \mid \_$	pattern
γ, η	::=	c	coercion assumption
	|	$\langle \tau \rangle \mid \mathbf{sym}\, \gamma \mid \gamma_1 \fatsemi \gamma_2$	equivalence
	|	$H_{\{\overline{\gamma}\}} \mid \gamma\, \omega \mid \Pi a{:}_\rho \eta.\, \gamma \mid \Pi c{:}(\eta_1, \eta_2).\, \gamma$	congruence
	|	$\mathbf{case}_\eta\, \gamma\, \mathbf{of}\, \overline{calt} \mid \mathbf{fix}\, \gamma \mid \lambda a{:}_\rho \eta.\, \gamma \mid \lambda c{:}(\eta_1, \eta_2).\, \gamma \mid \mathbf{absurd}\, (\eta_1, \eta_2)\, \gamma$	
	|	$\tau_1 \approx_\eta \tau_2$	coherence
	|	$\mathbf{argk}\, \gamma \mid \mathbf{argk}_n\, \gamma \mid \mathbf{res}^n\, \gamma \mid \gamma @ \omega$	Π-type decomposition
	|	$\mathbf{nth}_n\, \gamma \mid \mathbf{left}_\eta\, \gamma \mid \mathbf{right}_\eta\, \gamma$	generativity & injectivity
	|	$\mathbf{kind}\, \gamma$	"John Major" equality
	|	$\mathbf{step}\, \tau$	β-equivalence
$calt$	::=	$\pi \to \gamma$	case alternative in coercion
ω	::=	$\gamma \mid \{\gamma\} \mid (\gamma_1, \gamma_2)$	coercion argument
Σ	::=	$\varnothing$	signature
	|	$\Sigma, T{:}(\overline{a}{:}\overline{\kappa})$	algebraic datatype
	|	$\Sigma, K{:}(\Delta;\, T)$	data constructor
Γ, Δ	::=	$\varnothing \mid \Gamma, \delta$	context/telescope
θ	::=	$\varnothing \mid \theta, \tau/a \mid \theta, \gamma/c$	substitution

Figure 5.1: The grammar of Pico

$\overline{}$	$\triangleq$	(an overbar) indicates a list
$\_$	$\triangleq$	a fresh variable whose name is not used
$\mathsf{dom}(\overline{\Delta})$	$\triangleq$	the list of variables bound in Δ
$\mathsf{prefix}(\cdot)$	$\triangleq$	a prefix of a list; length specified elsewhere
$\mathsf{fv}(\cdot)$	$\triangleq$	extract all free variables, as a set
H	$\triangleq$	$H_{\{\}}$ (when appearing in a type)
$\tau\,\psi$	$\triangleq$	τ_ψ or $\tau\_\sim\psi$, depending on τ's kind
$\Pi\Delta.\,\tau$	$\triangleq$	nested Πs
$\Pi\!\!\!\!-\Delta.\,\tau$	$\triangleq$	nested Πs, where the individual Πs used might differ
$\lambda\Delta.\,\tau$	$\triangleq$	nested λs
$\tau_1 \sim \tau_2$	$\triangleq$	$\tau_1 \,^{\kappa_1}\!\sim^{\kappa_2} \tau_2$ (when the kinds are obvious or unimportant)
$\bullet$	$\triangleq$	an erased coercion
$\#$	$\triangleq$	the sets of free variables of two entities are distinct
$\lfloor\cdot\rfloor$	$\triangleq$	coercion erasure (Section 5.8.3)
$\lVert\cdot\rVert$	$\triangleq$	type erasure (Section 5.11)
let	is	used in the metatheory only and should be eagerly inlined

Figure 5.2: Notation conventions of Pico

definition for arguments allows us to easily discuss what I call *vectors*,[47] which are lists of arguments $\overline{\psi}$. Similarly to the redundancy of application forms, tracking relevant types as compared to irrelevant types is also redundant with the kind of the function type; an implementation would not need to store this distinction.

Coercions are the most distinctive and most intricate part of Pico. Because the formation rules for coercions necessarily refer to many other parts of the language, a thorough treatment of coercions is delayed until the other constructs are covered. However, it may be helpful to readers unfamiliar with System FC to learn a few quick facts about coercions: see Figure 5.4 on page 79.

As you will see in Figure 5.2, my presentation of Pico uses several abbreviations and elisions in its typesetting. In particular, I frequently write types like $\Pi\Delta.\,\tau$ to represents a nested Π-type, binding the variables listed in Δ (which, as you can see, is just a list of binders δ). An equality proposition in Pico lists both the related types and their kinds. Often, the kinds are redundant, obvious, or unimportant, and so I elide them in those cases.

All of the metatheory in this dissertation is typeset using ott [82]. This tool effectively type-checks my work, preventing me from writing, say, the nonsense $a{:}\phi$, which is rightly a ott parsing error.[48] In addition, I have configured my use of ott to require me to write the kinds of an equality proposition even when I intend for them

vs. irrelevant type vs. coercion)

[47] I have adopted this terminology from Gundry [37].

[48] Indeed, to include that example in the text, I had to avoid rendering it in ott syntax.

$\Sigma \vdash_{\mathsf{tc}} H : \Delta_1; \Delta_2; H'$	Constant H has universals Δ_1, existentials Δ_2, and belongs to parent type H'.
$\Sigma; \Gamma \vdash_{\mathsf{ty}} \tau : \kappa$	Type τ has kind κ.
$\Sigma; \Gamma; \sigma \vdash^{\tau_0}_{\mathsf{alt}} \pi \to \tau : \kappa$	Case alternative $\pi \to \tau$ yields something of kind κ when used with a scrutinee τ_0 of type σ.
$\Sigma; \Gamma \vdash_{\mathsf{co}} \gamma : \phi$	Coercion γ proves proposition ϕ.
$\Sigma; \Gamma \vdash_{\mathsf{prop}} \phi\ \mathsf{ok}$	Proposition ϕ is well formed.
$\Sigma; \Gamma \vdash_{\mathsf{vec}} \overline{\psi} : \Delta$	Vector $\overline{\psi}$ is classified by telescope Δ.
$\Sigma; \Gamma \vdash_{\mathsf{cev}} \overline{\psi} : \Delta$	Vector $\overline{\psi}$ is classified by telescope Δ (with induction defined from the end).
$\vdash_{\mathsf{sig}} \Sigma\ \mathsf{ok}$	Signature Σ is well formed.
$\Sigma \vdash_{\mathsf{ctx}} \Gamma\ \mathsf{ok}$	Context Γ is well formed.
$\Sigma; \Gamma \vdash_{\mathsf{s}} \tau \longrightarrow \tau'$	Type τ reduces to type τ' in one step.

Lemma (Kind regularity [Lemma C.43]). *If* $\Sigma; \Gamma \vdash_{\mathsf{ty}} \tau : \kappa$, *then* $\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{ty}} \kappa : \mathbf{Type}$.

Lemma (Prop. regularity [Lemma C.44]). *If* $\Sigma; \Gamma \vdash_{\mathsf{co}} \gamma : \phi$, *then* $\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{prop}} \phi\ \mathsf{ok}$.

Figure 5.3: Judgments used in the definition of Pico

to be elided in the rendered output, as a check to make sure these parameters can indeed be written with the information to hand.

This chapter proceeds by explaining all of the various typing judgments individually. Section 5.3 explains contexts Γ, along with relevance annotations. Section 5.4 explains signatures Σ, which contain specifications for constants H. Having covered the more unexpected aspects of the syntax, Section 5.5 then presents examples of Pico programs. Types come next, in Section 5.6, followed by the operational semantics in Section 5.7. Now having an thorough understanding of the rest of Pico, we are prepared to tackle coercions, the thorniest part, in Section 5.8. Section 5.9 covers one final rule from the operational semantics (S_KPush), too challenging to describe before coercions are fully explained. Sections 5.10 and 5.11 cover the metatheory. Section 5.12 describes certain, perhaps unexpected design decisions. The chapter concludes in Section 5.13 by considering a variety of extensions to Pico that are needed for full, backward-compatible support for Haskell as embodied in GHC 8.

Coercions define the equivalence relation $\sim$ that is used in PICO's analogue of a traditional conversion rule, as presented in Section 5.1. Here is a brief introduction to coercions. The full definition of coercion formation rules appears in Appendix B.3. The rules are explicated in Section 5.8.

- Coercions are heterogeneous (Section 5.8.1). If a coercion γ proves $\tau_1 \ {}^{\kappa_1}\!\sim^{\kappa_2} \tau_2$, then we know that τ_1 is convertible with τ_2 and also that κ_1 is convertible with κ_2. The form $\mathbf{kind}\,\gamma$ extracts the kind equality from the type equality. I often elide the kinds when writing propositions, however.
- Equality may be assumed via a λ-abstraction over a coercion variable c, proving any arbitrary equality proposition. (Section 5.8.2)
- Equality is coherent (Section 5.8.3), meaning that a coercion relates any two types that are identical except for the coercions and casts within them. The coercion form $\tau_1 \approx_\eta \tau_2$ proves that $\tau_1 \sim \tau_2$ and is valid whenever τ_1 and τ_2 are identical, ignoring internal coercions. (The coercion η relates the types' kinds.)
- Equality is an equivalence (Section 5.8.4): $\langle\tau\rangle$ is reflexive coercion over τ; $\mathbf{sym}\,\gamma$ represents symmetry; and $\gamma_1 \fatsemi \gamma_2$ represents transitivity.
- Equality is (almost) congruent (Section 5.8.5), meaning that if we have a proof of $\tau_1 \sim \tau_2$, then we can derive a proof relating larger types containing τ_1 and τ_2 but are otherwise identical. The "almost" qualifier is due to a technical restriction that can be ignored on a first reading.
- Coercions can be decomposed (Section 5.8.6). For example, if γ proves $(\Pi a_1{:}_\rho\kappa_1.\,\tau_1) \sim (\Pi a_2{:}_\rho\kappa_2.\,\tau_2)$, then $\mathbf{argk}\,\gamma$ proves $\kappa_1 \sim \kappa_2$. Other coercion forms decompose other type forms.
- The $\mathbf{step}\,\tau$ coercion relates τ to its small-step reduct. (Section 5.8.7)

Figure 5.4: A brief introduction to coercions

5.3 Contexts Γ and relevance annotations

One of the distinctive aspects of PICO is its use of relevance annotations on binders. Every variable binding $a{:}_\rho\kappa$ comes with a relevance annotation ρ, which can be either Rel or Irrel. A typing context Γ is just a list of such binders (along with, perhaps, coercion variable binders) and so retains the relevance annotation. These annotations

come into play only in the rule for checking variable occurrences:

$$\frac{\Sigma \vdash_{\mathsf{ctx}} \Gamma\ \mathsf{ok} \qquad a{:}_{\mathsf{Rel}}\kappa \in \Gamma}{\Sigma; \Gamma \vdash_{\mathsf{ty}} a : \kappa}\ \textsc{Ty\_Var}$$

Note that this rule requires $a{:}_{\mathsf{Rel}}\kappa \in \Gamma$, with a relevant binder. Thus, only variables that are considered relevant—that is, variables that will remain at runtime—can be used in an expression. As described briefly above, when we "go up a level", we reset the context, marking all variables relevant. This resetting is done by the $\mathsf{Rel}(\Gamma)$ operation, defined recursively on the structure of Γ as follows:

$$\begin{aligned} \mathsf{Rel}(\varnothing) &= \varnothing \\ \mathsf{Rel}(\Gamma, a{:}_{\rho}\kappa) &= \mathsf{Rel}(\Gamma), a{:}_{\mathsf{Rel}}\kappa \\ \mathsf{Rel}(\Gamma, c{:}\phi) &= \mathsf{Rel}(\Gamma), c{:}\phi \end{aligned}$$

The $\mathsf{Rel}(\Gamma)$ operation is used, for example, in the judgment to check contexts for validity:

$\boxed{\Sigma \vdash_{\mathsf{ctx}} \Gamma\ \mathsf{ok}}$ Context formation

$$\frac{\vdash_{\mathsf{sig}} \Sigma\ \mathsf{ok}}{\Sigma \vdash_{\mathsf{ctx}} \varnothing\ \mathsf{ok}}\ \textsc{Ctx\_Nil}$$

$$\frac{\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{ty}} \kappa : \mathbf{Type} \qquad a \# \Gamma \qquad \Sigma \vdash_{\mathsf{ctx}} \Gamma\ \mathsf{ok}}{\Sigma \vdash_{\mathsf{ctx}} \Gamma, a{:}_{\rho}\kappa\ \mathsf{ok}}\ \textsc{Ctx\_TyVar}$$

$$\frac{\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{prop}} \phi\ \mathsf{ok} \qquad c \# \Gamma \qquad \Sigma \vdash_{\mathsf{ctx}} \Gamma\ \mathsf{ok}}{\Sigma \vdash_{\mathsf{ctx}} \Gamma, c{:}\phi\ \mathsf{ok}}\ \textsc{Ctx\_CoVar}$$

Here, we see that a binding $a{:}_{\rho}\kappa$ can be appended onto a context Γ when the a is fresh and the κ is well typed at **Type** in $\mathsf{Rel}(\Gamma)$. The reason for using $\mathsf{Rel}(\Gamma)$ instead of Γ here is that the kind κ does not exist at runtime, regardless of the relevance annotation on a. We are thus free to essentially ignore the relevance annotations on Γ, which is what $\mathsf{Rel}(\Gamma)$ does. The same logic applies to the use of $\mathsf{Rel}(\Gamma)$ in the Ctx_CoVar rule. Indeed, all premises involving coercions use $\mathsf{Rel}(\Gamma)$, as all coercions are erased and are thus irrelevant.

In order for premises that use $\mathsf{Rel}(\Gamma)$ to work in the metatheory, we must frequently use the following lemma:

Lemma (Increasing relevance [Lemma C.6])**.** *Let* Γ *and* Γ' *be the same except that some bindings in* Γ' *are labeled* Rel *where those same bindings in* Γ *are labeled* Irrel*. Any judgment about* Γ *is also true about* Γ'*.*

Regularity Regularity is an important property of Pico, allowing us to easily assume well-formed contexts and signatures:

Lemma (Context regularity [Lemma C.9])**.** *If*

1. $\Sigma; \Gamma \vdash_{\mathsf{ty}} \tau : \kappa$*, or*
2. $\Sigma; \Gamma \vdash_{\mathsf{co}} \gamma : \phi$*, or*
3. $\Sigma; \Gamma \vdash_{\mathsf{prop}} \phi\ \mathsf{ok}$*, or*
4. $\Sigma; \Gamma; \sigma_0 \vdash^{\tau_0}_{\mathsf{alt}} alt : \kappa$*, or*
5. $\Sigma; \Gamma \vdash_{\mathsf{vec}} \overline{\psi} : \Delta$*, or*
6. $\Sigma \vdash_{\mathsf{ctx}} \Gamma\ \mathsf{ok}$*,*

then $\Sigma \vdash_{\mathsf{ctx}} \mathsf{prefix}(\Gamma)\ \mathsf{ok}$ *and* $\vdash_{\mathsf{sig}} \Sigma\ \mathsf{ok}$*, where* $\mathsf{prefix}(\Gamma)$ *is an arbitrary prefix of* Γ*. Furthermore, both resulting derivations are no larger than the input derivations.*

5.4 Signatures Σ and type constants H

The typing rules in Pico are all parameterized by both a signature Σ and a context Γ. Signatures contain bindings for all global constants: type and data constructors. In contrast, contexts contain local bindings, for type and coercion variables. Several treatments of System FC assume a fixed, global signature, but I find it more precise here to make dependency on this signature explicit.

5.4.1 Signature validity

The judgment to check the validity of a signature follows:

$\boxed{\vdash_{\mathsf{sig}} \Sigma\ \mathsf{ok}}$ Signature formation

$$\frac{}{\vdash_{\mathsf{sig}} \varnothing\ \mathsf{ok}}\ \textsc{Sig\_Nil}$$

$$\frac{\Sigma \vdash_{\mathsf{ctx}} \overline{a}{:}_{\mathsf{Irrel}}\overline{\kappa}\ \mathsf{ok} \qquad T \# \Sigma}{\vdash_{\mathsf{sig}} \Sigma, T{:}(\overline{a}{:}\overline{\kappa})\ \mathsf{ok}}\ \textsc{Sig\_ADT}$$

$$\frac{T{:}(\overline{a}{:}\overline{\kappa}) \in \Sigma \qquad \Sigma \vdash_{\mathsf{ctx}} \overline{a}{:}_{\mathsf{Irrel}}\overline{\kappa}, \Delta\ \mathsf{ok} \qquad K \# \Sigma}{\vdash_{\mathsf{sig}} \Sigma, K{:}(\Delta; T)\ \mathsf{ok}}\ \textsc{Sig\_DataCon}$$

We see here the two different entities that can belong to a signature, an algebraic datatype (ADT) T or a data constructor K.

An ADT is classified only by its list of universally quantified variables (often shortened to *universals*), as this is the only piece of information that varies between ADTs. For example, the Haskell type *Int* contains no universals, while *Either* contains two (both of kind **Type**), and *Proxy*'s universals are $(a : \mathbf{Type}, b : a)$. The relevance of universals is predetermined (see Section 5.4.2.2) and so no relevance annotations appear on ADT specifications. Additionally, coercion variables are not permitted here—coercion variables would be very much akin to Haskell's misfeature of datatype contexts[49] and so are excluded.

A data constructor is classified by a telescope Δ of existentially bound variables (or *existentials*) and the ADT to which it belongs. The grammar for telescopes is the same as that for contexts, but we use the metavariables Γ and Δ in distinct ways: Γ is used as the context for typing judgments, whereas Δ is more often used as some component of a type. A telescope is a list of binders—both type variables and coercion variables—where later binders may depend on earlier ones. A data constructor's existentials are the data that cannot be determined from an applied data constructor's type. In this formulation, the term *existential* also includes what would normally be considered term-level arguments.

For example, let's consider these Haskell definitions:

```
data Tuple a where
  MkTuple :: ∀ a. Int → Char → a → Tuple a
data Ex a where
  MkEx :: ∀ a b. b → a → Ex a
```

If I have a value of type *Tuple Double*, then I know the types of the data stored in a *MkTuple*, but I do not know the *Int*, the *Char*, or the *Double*—these are the existentials. Similarly, if I have a value of type *Ex Char*, then I know the type of one argument to *MkEx*, but I do not know the type of the other; I also know neither value. In this case, the second type, *b*, is existential, as are both values (of types *b* and *a*, respectively).

The use of the term *existential* to refer to term-level arguments may be non-standard, but it is quite convenient (while remaining technically accurate) in the context of a pure type system with ADTs.

5.4.2 Looking up type constants

Information about type constants is retrieved via the $\Sigma \vdash_{\mathsf{tc}} H : \Delta_1; \Delta_2; H'$ judgment, presented in Figure 5.5 on the following page. This judgment retrieves three pieces of data about a type constant H: its universals, its existentials, and the head of the result type. It is best understood in concert with the typing rule that handles type constants, which also uses the typing judgment on vectors—ordered lists of arguments—also presented in Figure 5.5 on the next page. Let's tackle this all in order of complexity.

[49] See discussion of how this is a misfeature at `https://prime.haskell.org/wiki/NoDatatypeContexts`.

$$\frac{\Sigma \vdash_{\mathsf{tc}} H : \Delta_1; \Delta_2; H' \qquad \Sigma \vdash_{\mathsf{ctx}} \Gamma\ \mathsf{ok} \qquad \Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{vec}} \overline{\tau} : \mathsf{Rel}(\Delta_1)}{\Sigma; \Gamma \vdash_{\mathsf{ty}} H_{\{\overline{\tau}\}} : \Pi(\Delta_2[\overline{\tau}/\mathsf{dom}(\Delta_1)]).\, H'\,\overline{\tau}} \quad \textsc{Ty\_Con}$$

$\boxed{\Sigma \vdash_{\mathsf{tc}} H : \Delta_1; \Delta_2; H'}$ Type constant kinds, with universals Δ_1, existentials Δ_2, and result H'

$$\frac{}{\Sigma \vdash_{\mathsf{tc}} \mathbf{Type} : \varnothing; \varnothing; \mathbf{Type}} \quad \textsc{Tc\_Type}$$

$$\frac{T{:}(\overline{a}{:}\overline{\kappa}) \in \Sigma}{\Sigma \vdash_{\mathsf{tc}} T : \varnothing; \overline{a}{:}_{\mathsf{Rel}}\overline{\kappa}; \mathbf{Type}} \quad \textsc{Tc\_ADT}$$

$$\frac{K{:}(\Delta;\, T) \in \Sigma \qquad T{:}(\overline{a}{:}\overline{\kappa}) \in \Sigma}{\Sigma \vdash_{\mathsf{tc}} K : \overline{a}{:}_{\mathsf{Irrel}}\overline{\kappa}; \Delta;\, T} \quad \textsc{Tc\_DataCon}$$

$\boxed{\Sigma; \Gamma \vdash_{\mathsf{vec}} \overline{\psi} : \Delta}$ Type vector formation

$$\frac{\Sigma \vdash_{\mathsf{ctx}} \Gamma\ \mathsf{ok}}{\Sigma; \Gamma \vdash_{\mathsf{vec}} \varnothing : \varnothing} \quad \textsc{Vec\_Nil}$$

$$\frac{\Sigma; \Gamma \vdash_{\mathsf{ty}} \tau : \kappa \qquad \Sigma; \Gamma \vdash_{\mathsf{vec}} \overline{\psi} : \Delta[\tau/a]}{\Sigma; \Gamma \vdash_{\mathsf{vec}} \tau, \overline{\psi} : a{:}_{\mathsf{Rel}}\kappa, \Delta} \quad \textsc{Vec\_TyRel}$$

$$\frac{\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{ty}} \tau : \kappa \qquad \Sigma; \Gamma \vdash_{\mathsf{vec}} \overline{\psi} : \Delta[\tau/a]}{\Sigma; \Gamma \vdash_{\mathsf{vec}} \{\tau\}, \overline{\psi} : a{:}_{\mathsf{Irrel}}\kappa, \Delta} \quad \textsc{Vec\_TyIrrel}$$

$$\frac{\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{co}} \gamma : \phi \qquad \Sigma; \Gamma \vdash_{\mathsf{vec}} \overline{\psi} : \Delta[\gamma/c]}{\Sigma; \Gamma \vdash_{\mathsf{vec}} \gamma, \overline{\psi} : c{:}\phi, \Delta} \quad \textsc{Vec\_Co}$$

Figure 5.5: Type constants H and vectors $\overline{\psi}$

5.4.2.1 The constant Type

The constant **Type** has no universals, no existentials, and **Type**'s type is **Type**, as TC_TYPE tells us. Thus, in the use of TY_CON when $H_{\{\overline{\tau}\}}$ is just $\mathbf{Type}_{\{\}}$ (normally, we omit such empty braces), we see that Δ_1, Δ_2, and $\overline{\tau}$ are all empty, meaning that we get $\Sigma; \Gamma \vdash_{\mathsf{ty}} \mathbf{Type} : \mathbf{Type}$, as desired.

5.4.2.2 Algebraic datatypes

Let's consider *Maybe* as an example. We see that the list of universals Δ_1 is empty for all ADTs. Thus, the list of universal arguments $\overline{\tau}$ must be empty in TY_CON. The list of existentials Δ_2 is $a{:}_{\mathsf{Rel}}\mathbf{Type}$ and the result type root is **Type**, both by TC_ADT. We thus get $\Sigma; \Gamma \vdash_{\mathsf{ty}} \mathit{Maybe} : \text{'}\Pi a{:}_{\mathsf{Rel}}\mathbf{Type}.\, \mathbf{Type}$, as desired. (Note that a is unused in the body of the 'Π and thus that this type could also be written as $\mathbf{Type} \to \mathbf{Type}$.)

I have argued here how the rules work out this case correctly, but it may surprise the reader to see that the argument to *Maybe* is treated as an *existential* here—part of Δ_2—and not a universal. This could best be understood if we consider **Type** itself to be an open ADT (that is, an extensible ADT) with no universal parameters. To make this even more concrete, here is how it might look in Haskell:

```
data Type where
  Bool  :: Type
  Int   :: Type
  Maybe :: Type → Type
  Proxy :: ∀ (k :: Type). k → Type
  ...
```

Thinking of ADTs this way, we can see why the argument to *Maybe* is existential, just like other arguments to constructors (see Section 5.4.1 for an explanation of the unusual use of the word *existential* here). We can also see that the kind parameter *k* to *Proxy* is also considered an existential in this context.

The last detail to cover here is the relevance annotation on the $\overline{a}$, as assigned in TC_ADT: all the variables are considered relevant. This is a free choice in the design of PICO. Any choice of relevance annotations would work, including allowing the user to decide on a case-by-case basis. I have chosen to mark them as relevant, however, with the consideration that these ADTs might be present at runtime. There is nothing in PICO that restricts ADTs to be present only at compile time; the user might write a runtime computation that returns *Bool*, for example.[50] (Such a facility replaces Haskell's current *TypeRep* facility [75].) By marking the ADT parameters as relevant, a runtime decision can be made between, say, *Maybe Int* and *Maybe Bool*. This seems useful, and so I have decided to make these parameters relevant.

[50] This statement does not mean that you can extract the value *Maybe Int* from *Just* 3, which would require preserving all types for runtime.

5.4.2.3 Data constructors

The most involved case is that for data constructors, where both the universals and the existentials can be non-empty. We'll try to understand Ty_Con first by an example inspired by the Haskell expression $\mathit{Left}\ \mathit{True} :: \mathit{Either}\ \mathit{Bool}\ \mathit{Char}$. Let's recall the definition of *Either*, a basic sum type:

$$\begin{aligned}&\textbf{data}\ \mathit{Either} :: \mathbf{Type} \to \mathbf{Type} \to \mathbf{Type}\ \textbf{where}\\ &\quad \mathit{Left}\ \ :: a \to \mathit{Either}\ a\ b\\ &\quad \mathit{Right} :: b \to \mathit{Either}\ a\ b\end{aligned}$$

In Pico this looks like the following:

$$\Sigma = \mathit{Either}{:}(a : \mathbf{Type}, b : \mathbf{Type}), \mathit{Left}{:}(x{:}_{\mathsf{Rel}} a; \mathit{Either}), \mathit{Right}{:}(x{:}_{\mathsf{Rel}} b; \mathit{Either}),\\ \mathit{Bool}{:}(\varnothing), \mathit{True}{:}(\varnothing; \mathit{Bool}), \mathit{False}{:}(\varnothing; \mathit{Bool}), \mathit{Char}{:}(\varnothing)$$

$$\Sigma; \varnothing \vdash_{\mathsf{ty}} \mathit{Left}_{\{\mathit{Bool}, \mathit{Char}\}}\ \mathit{True} : \mathit{Either}\ \mathit{Bool}\ \mathit{Char}$$

We see how the universal arguments *Bool* and *Char* to the constructor *Left* are specified in the subscript; without these arguments, there would be no way to get the type of $\mathit{Left}\ \mathit{True}$ in a syntax-directed way.

Universal argument saturation The grammar for type constant occurrences in types requires them to appear fully saturated with respect to universals but perhaps unsaturated with respect to existentials. There are several reasons for this seemingly peculiar design:

- It is helpful to separate universals from existentials in a variety of contexts. For example, existentials are brought into scope on a **case**-match, while universals are not. Separating out these arguments is also essential in the step rule S_KPush.
- If Pico did not allow matching on unsaturated constants, it might be most natural to require saturation with respect to *both* universals and existentials (while still keeping these different arguments separate). This would allow, for example, for a simple statement of the canonical forms lemma (Lemma C.75), because only a λ-expression would have a Π-type.

  However, since Pico does allow matching on unsaturated constants, the grammar must permit this form. Because Pico tracks the difference between matchable Π and unmatchable $\underset{\sim}{\Pi}$, we retain the simplicity of the canonical forms lemma, as any expression classified by a Π must be a partially applied constant and any expression classified by a $\underset{\sim}{\Pi}$ must be a λ.
- All universal arguments are always irrelevant and erased during type erasure (Section 5.11). It is thus natural to separate these from existentials in the grammar.

As with many design decisions, it is possible to redesign Pico and avoid this unusual choice, but in my opinion, this design pays its weight nicely.

Typing rules for data constructors The Tc_DataCon rule looks up a data constructor K in the signature Σ to find its telescope of existentials Δ and parent datatype T. The second premise of the rule then looks up T to get the universals. The universals are annotated with Irrel, as universals are always irrelevant in data constructors—universal arguments are properly part of the type of a data constructor and are thus not needed at runtime. The telescope of existentials Δ and datatype T are also returned from $\vdash_{\mathsf{tc}}$.

Rule Ty_Con checks the supplied arguments $\overline{\tau}$ against the telescope of universals, here named Δ_1. Note that $\overline{\tau}$ are checked against $\mathsf{Rel}(\Delta_1)$; the braces that appear in the production $H_{\{\overline{\tau}\}}$ are part of the concrete syntax and do not represent wrapping each individual $\tau \in \overline{\tau}$ in braces (cf. Section 5.6.2). Rule Ty_Con then builds the result type, a 'Π-type binding the existentials and producing H'—that is, the parent type T—applied to all of the universals.

5.5 Examples

Though these examples may make sense more fully after reading the sections below, it may be helpful at this point to see a few short examples of Pico programs.

We will work with a definition of length-indexed vectors, a tried-and-true example of the design of GADTs. Here is how they are declared in Haskell (further explanation is available in Section 3.1.1):

```
data Nat = Zero | Succ Nat
data Vec :: Type → Nat → Type where
  VNil  :: Vec a 0
  VCons :: a → Vec a n → Vec a ('Succ n)
```

If Pico had a concrete syntax, these declarations would be transformed roughly into the following:

```
Nat  :: Type
Zero :: Nat
Succ :: Nat → Nat

Vec   :: Type → Nat → Type
VNil  :: ∀ (a :: Type) (n :: Nat). (n ∼ Zero) → Vec a n
VCons :: ∀ (a :: Type) (n :: Nat).
           ∀ (m :: Nat). (n ∼ Succ m) → a → Vec a m → Vec a n
```

The change seen here is just the transformation between specifying a GADT equality constraint via a return type in a declaration to using an explicit existential variable with an explicit equality constraint.

In the abstract syntax of Pico, these declarations are represented by this signature Σ_0:

$$\begin{aligned}\Sigma_0 = \; & \mathit{Nat}{:}(\varnothing),\\ & \mathit{Zero}{:}(\varnothing; \mathit{Nat}),\\ & \mathit{Succ}{:}(\_ :_{\mathsf{Rel}} \mathit{Nat}; \mathit{Nat}),\\ & \mathit{Vec}{:}(a : \mathbf{Type}, n : \mathit{Nat}),\\ & \mathit{VNil}{:}(c{:}n \sim 0; \mathit{Vec}),\\ & \mathit{VCons}{:}(m{:}_{\mathsf{Irrel}} \mathit{Nat}, c{:}n \sim \mathit{Succ}\ m, \_ :_{\mathsf{Rel}} a, \_ :_{\mathsf{Rel}} \mathit{Vec}\ a\ m; \mathit{Vec})\end{aligned}$$

Let's walk through these declarations. Our binding for *Nat* includes an empty list of universally quantified type variables. This binding is followed by specifications for *Zero*, which lists no existential variables and is a constructor of the datatype *Nat*, and *Succ*, which has one (anonymous) existential variable and also belongs to *Nat*. The bindings for *Vec* and its constructors are similar, but with more parameters. Note the coercion bindings in the telescopes associated with *VNil* and *VCons*, as well as the irrelevant binding for the existential *m* of *VCons*. The design we see here, echoing the Haskell, does not permit runtime extraction of the length of a vector. If we changed the *m* to be relevant, then runtime length extraction would be trivial.

We will now look at a few simple operations on vectors, first in Haskell and then in Pico.[51]

5.5.1 *isEmpty*

First, a very simple test for emptiness, in order to familiarize ourselves with pattern-match syntax in Pico:

$$\begin{aligned}&\mathit{isEmpty} :: \mathit{Vec}\ a\ n \to \mathit{Bool}\\ &\mathit{isEmpty}\ \mathit{VNil} \qquad\quad = \mathit{True}\\ &\mathit{isEmpty}\ (\mathit{VCons}\ \{\}) = \mathit{False}\end{aligned}$$

Translated to Pico, we get the following:

$$\begin{aligned}&\mathit{isEmpty} \;:\; \tilde{\Pi}(a{:}_{\mathsf{Irrel}}\mathbf{Type}), (n{:}_{\mathsf{Irrel}}\mathit{Nat}), (v{:}_{\mathsf{Rel}}\mathit{Vec}\ a\ n).\ \mathit{Bool}\\ &\mathit{isEmpty} = \lambda(a{:}_{\mathsf{Irrel}}\mathbf{Type}), (n{:}_{\mathsf{Irrel}}\mathit{Nat}), (v{:}_{\mathsf{Rel}}\mathit{Vec}\ a\ n).\\ &\qquad \mathbf{case}_{\mathit{Bool}}\ v\ \mathbf{of}\\ &\qquad\quad \mathit{VNil} \;\to \lambda(c{:}n \sim 0), (c_0{:}v \sim \mathit{VNil}_{\{a,n\}}\ c).\ \mathit{True}\\ &\qquad\quad \mathit{VCons} \to \lambda(m{:}_{\mathsf{Irrel}}\mathit{Nat}), (c{:}n \sim \mathit{Succ}\ m), (x{:}_{\mathsf{Rel}}a), (xs{:}_{\mathsf{Rel}}\mathit{Vec}\ a\ m),\\ &\qquad\qquad\qquad (c_0{:}v \sim \mathit{VCons}_{\{a,n\}}\ m\ c\ x\ xs).\\ &\qquad\qquad\qquad \mathit{False}\end{aligned}$$

The most striking feature about this Pico code is the form of the **case** expression. Unlike the concrete syntax of Haskell, patterns in Pico do not directly bind any

[51] In these examples, I assume the use of numerals to specify elements of type *Nat*, and I also assume the existence of, e.g., *Bool*.

arguments. Note that there are no variable bindings to the left of the arrows in the case-branches. Instead, I have chosen to have λs to the right of the arrow. This design choice greatly simplifies the typing and scoping rules for pattern matches, because it removes a binding site in the grammar (leaving us with two: Π and λ). Because of the typing rule for **case** expressions (Section 5.6.5), we *still* must bind all of the existentials of a data constructor when matching against it—even when these existentials are ignored, as we see here.

The matches also bind a variable not mentioned in the data constructors' existentials: the coercion variable c_0. This coercion witnesses the equality between the scrutinee (v, in this case) and the applied data constructor that introduces the case branch. This coercion variable is bound in all matches, meaning that all pattern matching in Pico is dependent pattern matching.[52]

The behavior of **case** can also be viewed through its operational semantics, as captured in the following rule, excerpted from Section 5.7.2:

$$\frac{alt_i \;=\; H \to \tau_0}{\Sigma; \Gamma \vdash_{\mathsf{s}} \mathbf{case}_\kappa \; H_{\{\overline{\tau}\}} \; \overline{\psi} \; \mathbf{of} \; \overline{alt} \longrightarrow \tau_0 \; \overline{\psi} \; \langle H_{\{\overline{\tau}\}} \; \overline{\psi} \rangle} \quad \text{S\_Match}$$

Note that the body of the match, τ_0, is applied to the existential arguments to $H_{\{\overline{\tau}\}}$ and a coercion witnessing the equality between the scrutinee and the pattern. In the case of a successful match, this coercion is reflexive, as denoted by the angle brackets $\langle H_{\{\overline{\tau}\}} \; \overline{\psi} \rangle$.

5.5.2 *replicate*

Let's now look at *replicate*, one of the simplest functions that requires a proper Π-type. First, in Haskell:

```
replicate :: Π n → a → Vec a n
replicate Zero     _ = VNil
replicate (Succ m) x = VCons x (replicate m x)
```

[52] Contrast to Gundry [37], who use two separate constructs, **case** and **dcase**, only the latter of which does dependent matching. This separation is necessary in his language because not all expressions can be used in types and thus in dependent pattern matching. In particular, Gundry prevents λ-expressions in types, a limitation I have avoided by maintaining the distinction between matchable and unmatchable Π-types.

Now, in PICO:

$$
\begin{array}{l}
\mathit{replicate} \;:\; \underset{\sim}{\Pi}(\mathit{a}{:}_{\mathsf{Irrel}}\mathbf{Type}), (\mathit{n}{:}_{\mathsf{Rel}}\mathit{Nat}), (\mathit{x}{:}_{\mathsf{Rel}}\mathit{a}).\ \mathit{Vec}\ \mathit{a}\ \mathit{n} \\
\mathit{replicate} = \lambda \mathit{a}{:}_{\mathsf{Irrel}}\mathbf{Type}. \\
\qquad\qquad \mathbf{fix}\ \lambda(\mathit{r}{:}_{\mathsf{Rel}}\underset{\sim}{\Pi}(\mathit{n}{:}_{\mathsf{Rel}}\mathit{Nat}), (\mathit{x}{:}_{\mathsf{Rel}}\mathit{a}).\ \mathit{Vec}\ \mathit{a}\ \mathit{n}), \\
\qquad\qquad\quad (\mathit{n}{:}_{\mathsf{Rel}}\mathit{Nat}), (\mathit{x}{:}_{\mathsf{Rel}}\mathit{a}). \\
\qquad\qquad\quad \mathbf{case}_{\mathit{Vec}\ \mathit{a}\ \mathit{n}}\ \mathit{n}\ \mathbf{of} \\
\qquad\qquad\qquad \mathit{Zero} \to \lambda c_0{:}(\mathit{n} \sim \mathit{Zero}).\ \mathit{VNil}_{\{\mathit{a},\mathit{n}\}}\ c_0 \\
\qquad\qquad\qquad \mathit{Succ} \to \lambda \mathit{m}{:}_{\mathsf{Rel}}\mathit{Nat}, c_0{:}(\mathit{n} \sim \mathit{Succ}\ \mathit{m}).\ \mathit{VCons}_{\{\mathit{a},\mathit{n}\}}\ \{\mathit{m}\}\ c_0\ \mathit{x}\ (\mathit{r}\ \mathit{m}\ \mathit{x})
\end{array}
$$

This example shows the (standard) use of **fix** as well as some of the more exotic features of PICO. In the case branches, we see how we pass universal arguments to the data constructors *VNil* and *VCons*. We also see how we have to wrap irrelevant arguments (the $\{m\}$ in the last line) in braces. This example also shows where the coercion variable c_0 comes into play: it's needed to provide the coercion to the *VNil* and *VCons* constructors to prove that the universal argument *n* is indeed of the shape required for these constructors. Without the ability to do a dependent pattern match, this example would be impossible to write, unless you fake dependent types using singletons or some other technique.

5.5.3 *append*

We'll now examine how to append two vectors. This operation will also require the use of an addition operation, defined using prefix notation so as not to pose a parsing challenge:

```
plus :: Nat → Nat → Nat
plus Zero     n = n
plus (Succ m) n = Succ (plus m n)

append :: Vec a m → Vec a n → Vec a ('plus m n)
append VNil         ys = ys
append (VCons x xs) ys = VCons x (append xs ys)
```

And in PICO (where I elide the uninteresting *plus* for brevity):

$$
\begin{array}{l}
\mathit{append} \;:\; \underset{\sim}{\Pi}(\mathit{a}{:}_{\mathsf{Irrel}}\mathbf{Type}), (\mathit{m}{:}_{\mathsf{Irrel}}\mathit{Nat}), (\mathit{n}{:}_{\mathsf{Irrel}}\mathit{Nat}), (\mathit{xs}{:}_{\mathsf{Rel}}\mathit{Vec}\ \mathit{a}\ \mathit{m}), (\mathit{ys}{:}_{\mathsf{Rel}}\mathit{Vec}\ \mathit{a}\ \mathit{n}). \\
\qquad\quad \mathit{Vec}\ \mathit{a}\ (\mathit{plus}\ \mathit{m}\ \mathit{n}) \\
\mathit{append} = \lambda(\mathit{a}{:}_{\mathsf{Irrel}}\mathbf{Type}). \\
\qquad\quad \mathbf{fix}\ \lambda(\mathit{app}{:}_{\mathsf{Rel}}\underset{\sim}{\Pi}(\mathit{m}{:}_{\mathsf{Irrel}}\mathit{Nat}), (\mathit{n}{:}_{\mathsf{Irrel}}\mathit{Nat}), (\mathit{xs}{:}_{\mathsf{Rel}}\mathit{Vec}\ \mathit{a}\ \mathit{m}), (\mathit{ys}{:}_{\mathsf{Rel}}\mathit{Vec}\ \mathit{a}\ \mathit{n}). \\
\qquad\qquad\qquad\quad \mathit{Vec}\ \mathit{a}\ (\mathit{plus}\ \mathit{m}\ \mathit{n})), \\
\qquad\qquad\quad (\mathit{m}{:}_{\mathsf{Irrel}}\mathit{Nat}), (\mathit{n}{:}_{\mathsf{Irrel}}\mathit{Nat}), (\mathit{xs}{:}_{\mathsf{Rel}}\mathit{Vec}\ \mathit{a}\ \mathit{m}), (\mathit{ys}{:}_{\mathsf{Rel}}\mathit{Vec}\ \mathit{a}\ \mathit{n}). \\
\qquad\qquad\quad \mathbf{case}_{\mathit{Vec}\ \mathit{a}\ (\mathit{plus}\ \mathit{m}\ \mathit{n})}\ \mathit{xs}\ \mathbf{of} \\
\qquad\qquad\qquad \mathit{VNil} \quad \to \lambda(c{:}\mathit{m} \sim \mathit{Zero}), (c_0{:}\mathit{xs} \sim \mathit{VNil}_{\{\mathit{a},\mathit{m}\}}\ c). \\
\qquad\qquad\qquad\qquad\qquad \mathbf{let}\ c_1 := \langle \mathit{plus} \rangle\ c\ \langle \mathit{n} \rangle\ \mathbf{in} \\
\qquad\qquad\qquad\qquad\qquad \mathbf{let}\ c_2 := \mathbf{step}^{j}\ (\mathit{plus}\ \mathit{Zero}\ \mathit{n})\ \mathbf{in} \\
\qquad\qquad\qquad\qquad\qquad \mathit{ys} \rhd \mathbf{sym}\ (\mathit{Vec}\ \langle \mathit{a} \rangle\ (c_1 \fatsemi c_2)) \\
\qquad\qquad\qquad \mathit{VCons} \to \lambda(\mathit{m'}{:}_{\mathsf{Irrel}}\mathit{Nat}), (c{:}\mathit{m} \sim \mathit{Succ}\ \mathit{m'}), (\mathit{x}{:}_{\mathsf{Rel}}\mathit{a}), (\mathit{xs'}{:}_{\mathsf{Rel}}\mathit{Vec}\ \mathit{a}\ \mathit{m'}) \\
\qquad\qquad\qquad\qquad\qquad (c_0{:}\mathit{xs} \sim \mathit{VCons}_{\{\mathit{a},\mathit{m}\}}\ \{\mathit{m'}\}\ c\ \mathit{x}\ \mathit{xs'}). \\
\qquad\qquad\qquad\qquad\qquad \mathbf{let}\ c_1 := \langle \mathit{plus} \rangle\ c\ \langle \mathit{n} \rangle\ \mathbf{in} \\
\qquad\qquad\qquad\qquad\qquad \mathbf{let}\ c_2 := \mathbf{step}^{k}\ (\mathit{plus}\ (\mathit{Succ}\ \mathit{m'})\ \mathit{n})\ \mathbf{in} \\
\qquad\qquad\qquad\qquad\qquad \mathit{VCons}_{\{\mathit{a},\mathit{plus}\ \mathit{m}\ \mathit{n}\}}\ \{\mathit{plus}\ \mathit{m'}\ \mathit{n}\}\ (c_1 \fatsemi c_2)\ \mathit{x}\ (\mathit{app}\ \{\mathit{m'}\}\ \{\mathit{n}\}\ \mathit{xs'}\ \mathit{ys})
\end{array}
$$

This is the first example where we are required to write non-trivial coercions. Let's start by considering the right-hand side of the *VNil* case. As we see in the Haskell version, we wish to return *ys*. However, *ys* has type *Vec a n*, and we need to return something of type *Vec a* (*plus m n*). We must, accordingly, cast *ys* to have type *Vec a* (*plus m n*). This is what the coercion $\mathbf{sym}\ (\mathit{Vec}\ \langle \mathit{a} \rangle\ (c_1 \fatsemi c_2))$ is doing; it proves that *Vec a n* is in fact equal to *Vec a* (*plus m n*). Both the starting type *Vec a n* and the ending type *Vec a* (*plus m n*) have the same prefix of *Vec a*. We use a congruence coercion (Section 5.8.5) $\mathit{Vec}\ \langle \mathit{a} \rangle\ \gamma$ to simplify our problem. Now, we need only a coercion γ that proves *plus m n* equals *n*. (The use of **sym** helpfully has reversed our proof obligation.) This γ is built in two steps, tied together by using our transitivity operator $\fatsemi$: c_1, which uses our reflexivity operator $\langle \cdot \rangle$, proves that *plus m n* equals *plus* 0 *n* by using c, the GADT equality constraint from the *VNil* constructor; and c_2 proves that *plus* 0 *n* equals *n*.[53] For this last coercion, we use the **step** coercion that reduces a type by one step. It is fiddly (and unenlightening) to calculate the precise number of steps necessary to get from *plus* 0 *n* to *n*, so I have just written that this takes j steps. It is straightforward to calculate j in practice.

The coercion manipulations in the *VCons* case are similar.

Also of note in this example is the interplay between relevant variables and irrelevant ones. We see that the lengths *m* and *n* are irrelevant throughout this function. Indeed, we do not need lengths at runtime to append two vectors. Accordingly, we can see that all uses of *m* and *n* (or *m'*) occur in irrelevant contexts, such as coercions or irrelevant arguments to functions.

[53] Recall (Figure 5.2 on page 77) that **let** is defined by simple expansion. It is not properly a language construct but instead is just a convenient abbreviation in this writeup.

5.5.4 *safeHead*

With length-indexed vectors, we can write a safe *head* operation, allowed only when we know that the vector has a non-zero length:

$$
\begin{array}{l}
\mathit{safeHead} :: \mathit{Vec}\ a\ (\text{'}\mathit{Succ}\ n) \to a \\
\mathit{safeHead}\ (\mathit{VCons}\ x\ \_) = x
\end{array}
$$

Note that *safeHead* contains a total pattern match; the *VNil* alternative is impossible given the type signature of the function. This function translates to Pico thusly:

$$
\begin{array}{l}
\mathit{safeHead}\ :\ \tilde{\Pi}(a{:}_{\mathsf{Irrel}}\mathbf{Type}), (n{:}_{\mathsf{Irrel}}\mathit{Nat}), (v{:}_{\mathsf{Rel}}\mathit{Vec}\ a\ (\mathit{Succ}\ n)).\ a \\
\mathit{safeHead} = \lambda(a{:}_{\mathsf{Irrel}}\mathbf{Type}), (n{:}_{\mathsf{Irrel}}\mathit{Nat}), (v{:}_{\mathsf{Rel}}\mathit{Vec}\ a\ (\mathit{Succ}\ n)). \\
\qquad \mathbf{case}_a\ v\ \mathbf{of} \\
\qquad\quad \mathit{VNil} \quad \to \lambda(c{:}\mathit{Succ}\ n \sim \mathit{Zero}), (c_0{:}v \sim \mathit{VNil}_{\{a,\mathit{Succ}\ n\}}\ c).\ \mathbf{absurd}\ c\ a \\
\qquad\quad \mathit{VCons} \to \lambda(m{:}_{\mathsf{Irrel}}\mathit{Nat}), (c{:}\mathit{Succ}\ n \sim \mathit{Succ}\ m), (x{:}_{\mathsf{Rel}}a), (xs{:}_{\mathsf{Rel}}\mathit{Vec}\ a\ m), \\
\qquad\qquad\qquad (c_0{:}v \sim \mathit{VCons}_{\{a,\mathit{Succ}\ n\}}\ \{m\}\ c\ x\ xs). \\
\qquad\qquad\quad\ x
\end{array}
$$

The new feature demonstrated in this example is the **absurd** operator, which appears in the body of the *VNil* case. In order to be sure that **case** expressions do not get stuck, the typing rules require that all matches are exhaustive. However, in general, in can be undecidable to determine whether the type of a scrutinee indicates that a certain constructor can be excluded. In order to step around this potential trap, Pico supports absurdity elimination through **absurd**. The coercion passed into **absurd** (c, above) must prove that one constant equals another. This is, of course, impossible, and so we allow $\mathbf{absurd}\,\gamma\,\tau$ to have any type τ.

5.6 Types τ

Having gone through several examples explaining the flavor of Pico code, let's now walk through the remaining typing rules of the system. Recall that we have already seen the typing rules for variables, Ty_Var in Section 5.3, and constants, Ty_Con in Section 5.4.2.

5.6.1 Abstractions

The definition for types τ includes the usual productions for a pure type system, including both a Π-form and a λ-form:

$$\frac{\Sigma; \Gamma, \mathsf{Rel}(\delta) \vdash_{\mathsf{ty}} \kappa : \mathbf{Type}}{\Sigma; \Gamma \vdash_{\mathsf{ty}} \Pi\delta.\, \kappa : \mathbf{Type}} \quad \textsc{Ty\_Pi}$$

$$\frac{\Sigma; \Gamma, \delta \vdash_{\mathsf{ty}} \tau : \kappa}{\Sigma; \Gamma \vdash_{\mathsf{ty}} \lambda\delta.\, \tau : \tilde{\Pi}\delta.\, \kappa} \quad \textsc{Ty\_Lam}$$

The only novel component of these rules is the use of $\mathsf{Rel}(\delta)$ in the premise to Ty_Pi. This is done to allow the bound variable to appear in κ, regardless of whether it is relevant or not. As an example, the use of $\mathsf{Rel}(\delta)$ here is necessary to allow the type of Haskell's $\bot$: $\tilde{\Pi} a{:}_{\mathsf{Irrel}} \mathbf{Type}.\, a$.

5.6.2 Applications

Terms with a Π-type (either type constants or λ-terms) can be applied to arguments, via these rules:

$$\frac{\Sigma; \Gamma \vdash_{\mathsf{ty}} \tau_1 : \Pi a{:}_{\mathsf{Rel}} \kappa_1.\, \kappa_2 \qquad \Sigma; \Gamma \vdash_{\mathsf{ty}} \tau_2 : \kappa_1}{\Sigma; \Gamma \vdash_{\mathsf{ty}} \tau_1\, \tau_2 : \kappa_2[\tau_2/a]} \quad \textsc{Ty\_AppRel}$$

$$\frac{\Sigma; \Gamma \vdash_{\mathsf{ty}} \tau_1 : \Pi a{:}_{\mathsf{Irrel}} \kappa_1.\, \kappa_2 \qquad \Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{ty}} \tau_2 : \kappa_1}{\Sigma; \Gamma \vdash_{\mathsf{ty}} \tau_1\, \{\tau_2\} : \kappa_2[\tau_2/a]} \quad \textsc{Ty\_AppIrrel}$$

$$\frac{\Sigma; \Gamma \vdash_{\mathsf{ty}} \tau : \Pi c{:}\phi.\, \kappa \qquad \Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{co}} \gamma : \phi}{\Sigma; \Gamma \vdash_{\mathsf{ty}} \tau\, \gamma : \kappa[\gamma/c]} \quad \textsc{Ty\_CApp}$$

We see in these rules that the argument form for an abstraction over an irrelevant binder requires braces. (See the conclusion of Ty_AppIrrel.) The system would remain syntax-directed without marking off irrelevant arguments, but type erasure (Section 5.11) would then need to be type-directed. It seems easier just to separate relevant arguments from irrelevant arguments syntactically.

Note also the use of $\mathsf{Rel}(\Gamma)$ in Ty_AppIrrel and Ty_CApp; resetting the context here happens because irrelevant arguments and coercions are erased in the running program.

5.6.3 Kind casts

We can always use an equality to change the kind of a type:

$$\frac{\begin{array}{l}\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{co}} \gamma : \kappa_1 \sim \kappa_2 \\ \Sigma; \Gamma \vdash_{\mathsf{ty}} \tau : \kappa_1 \qquad \Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{ty}} \kappa_2 : \mathbf{Type}\end{array}}{\Sigma; \Gamma \vdash_{\mathsf{ty}} \tau \rhd \gamma : \kappa_2} \quad \textsc{Ty\_Cast}$$

In this rule, a type of kind κ_1 is cast by γ to have a type κ_2. As always, the coercion is checked in a reset context $\mathsf{Rel}(\Gamma)$. The final premise, $\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{ty}} \kappa_2 : \mathbf{Type}$ is implied by the first premise (which is actually $\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{co}} \gamma : \kappa_1 \ {}^{\mathbf{Type}}\!\sim^{\mathbf{Type}} \kappa_2$) via proposition regularity, but we must include it in order to prove kind regularity[54] before we prove coercion regularity.

5.6.4 fix

Pico supports fixpoints via the following rule:

$$\frac{\Sigma; \Gamma \vdash_{\mathsf{ty}} \tau : \underset{\sim}{\Pi} a{:}_{\mathsf{Rel}}\kappa.\ \kappa}{\Sigma; \Gamma \vdash_{\mathsf{ty}} \mathbf{fix}\, \tau : \kappa} \quad \textsc{Ty\_Fix}$$

The rule requires type τ to have an unmatchable $\underset{\sim}{\Pi}$ so that we can be sure that τ's canonical form is indeed a λ (as opposed to an unsaturated constant); otherwise the progress theorem (Section 5.7) would not hold.

5.6.5 case

Unsurprisingly, the typing rules to support pattern matching are the most involved and are presented in Figure 5.6 on the following page with the rules to type-check **case** branches.

Most of the premises of Ty_Case are easy enough to explain:

- The result kind of a **case**, κ is given right in the syntax; the first premise $\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{ty}} \kappa : \mathbf{Type}$ ensures that it is a valid result kind.
- We also must check the kind of the scrutinee, τ. This kind must have the form $'\Pi\Delta.\, H\, \overline{\sigma}$ (note the matchable $'\Pi$), where the $\overline{\sigma}$ cannot mention any of the variables bound in Δ. (The $\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{ty}} H\, \overline{\sigma} : \mathbf{Type}$ premise checks this scoping condition.) Note that the scrutinee's type may be a $'\Pi$-type in order to support matching against partially applied type and data constructors.
- The alternatives must be exhaustive and distinct. Exhaustivity is needed to prove that a well-typed **case** cannot get stuck, and distinctness is necessary to prove that the reduction relation is deterministic.

[54] Both regularity lemmas are stated in Figure 5.3 on page 78.

$$\frac{\begin{array}{l}\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{ty}} \kappa : \mathbf{Type} \qquad \Sigma; \Gamma \vdash_{\mathsf{ty}} \tau : \sigma \\ \sigma \;=\; \Pi\Delta.\, H\,\overline{\sigma} \qquad \Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{ty}} H\,\overline{\sigma} : \mathbf{Type} \\ \forall i,\ \Sigma; \Gamma; \sigma \vdash^{\tau}_{\mathsf{alt}} alt_i : \kappa \\ \overline{alt} \text{ are exhaustive and distinct for } H, \text{ (w.r.t. } \Sigma)\end{array}}{\Sigma; \Gamma \vdash_{\mathsf{ty}} \mathbf{case}_{\kappa}\, \tau\, \mathbf{of}\, \overline{alt} : \kappa} \quad \textsc{Ty\_Case}$$

$\boxed{\Sigma; \Gamma; \sigma \vdash^{\tau}_{\mathsf{alt}} alt : \kappa}$ Case alternatives

$$\frac{\begin{array}{l}\Sigma \vdash_{\mathsf{tc}} H : \Delta_1; \Delta_2; H' \qquad \Delta_3, \Delta_4 \;=\; \Delta_2[\overline{\sigma}/\mathsf{dom}(\Delta_1)] \\ \mathsf{dom}(\Delta_4) \;=\; \mathsf{dom}(\Delta') \\ \mathsf{match}_{\{\mathsf{dom}(\Delta_3)\}}(\mathsf{types}(\Delta_4); \mathsf{types}(\Delta')) \;=\; \mathsf{Just}\,\theta \\ \Sigma; \Gamma \vdash_{\mathsf{ty}} \tau : \Pi\Delta_3, c{:}\tau_0 \sim H_{\{\overline{\sigma}\}}\, \mathsf{dom}(\Delta_3).\, \kappa\end{array}}{\Sigma; \Gamma; \Pi\Delta'.\, H'\,\overline{\sigma} \vdash^{\tau_0}_{\mathsf{alt}} H \to \tau : \kappa} \quad \textsc{Alt\_Match}$$

$$\frac{\Sigma; \Gamma \vdash_{\mathsf{ty}} \tau : \kappa}{\Sigma; \Gamma; \sigma \vdash^{\tau_0}_{\mathsf{alt}} \_ \to \tau : \kappa} \quad \textsc{Alt\_Default}$$

$\boxed{\mathsf{types}(\Delta) \;=\; \overline{\tau}}$ Extract the types from a telescope

$$\begin{aligned}\mathsf{types}(\varnothing) &= \varnothing \\ \mathsf{types}(\Delta, a{:}_{\rho}\kappa) &= \mathsf{types}(\Delta), \kappa \\ \mathsf{types}(\Delta, c{:}\tau_1 \;^{\kappa_1}\!\sim^{\kappa_2} \tau_2) &= \mathsf{types}(\Delta), \kappa_1, \kappa_2, \tau_1, \tau_2\end{aligned}$$

Figure 5.6: Rule and auxiliary definitions for **case** expressions

We are left to consider type-checking the alternatives. This is done via the judgment with schema $\Sigma; \Gamma; \sigma \vdash^{\tau}_{\mathsf{alt}} alt : \kappa$. When $\Sigma; \Gamma; \sigma \vdash^{\tau}_{\mathsf{alt}} alt : \kappa$ holds, we know that the expression in the case alternative *alt* produces a type of kind κ when considered with signature Σ and typing context Γ and when matched against a scrutinee τ of type σ. The premises of Ty_Case indeed check that all alternatives satisfy this judgment.

5.6.5.1 Checking case alternatives

The rule Alt_Match is intricate. It assumes a scrutinee τ_0 of type $\Pi\Delta'.\, H'\,\overline{\sigma}$, and we are checking a case alternative $H \to \tau$.

First, we must verify that the constant H is classified by H'—that is, either H is a data constructor of the datatype H' or H is a datatype and H' is **Type**. We say that H' is the *parent* of H. This check is done by the $\Sigma \vdash_{\mathsf{tc}} H : \Delta_1; \Delta_2; H'$ premise, which also extracts the universals Δ_1 and existentials Δ_2.

The next premise (reading to the right) uses $\Delta_2[\overline{\sigma}/\mathsf{dom}(\Delta_1)]$ to instantiate the existentials with the known choices for the universals. These known choices $\overline{\sigma}$ are obtained from determining the type of the scrutinee; see the appearance of $\overline{\sigma}$ in the

type appearing before the $\vdash_{\mathsf{alt}}$ in the conclusion of the rule. The second premise also splits the instantiated existentials into two telescopes, Δ_3 and Δ_4.

Note that Δ' is an input to this rule; it is extracted from the type of the scrutinee. Accordingly, the third premise $\mathsf{dom}(\Delta_4) = \mathsf{dom}(\Delta')$ serves two roles: it fixes the length of Δ_4 (and, hence, Δ_3) and it also forces any renaming of bound variables necessary to line up the telescopes Δ' and Δ_4. Keeping the names of the bound variables consistent between these telescopes simplifies this rule. We see that in the event that the scrutinee is a fully saturated datatype or data constructor, $\Delta_4 = \Delta' = \varnothing$ and $\Delta_3 = \Delta_2[\overline{\sigma}/\mathsf{dom}(\Delta_1)]$; in this common case, then, unification is unnecessary.

The next premise uses a one-way unification algorithm to make sure that the bound telescope in the scrutinee's type, Δ', matches the expected shape Δ_4. (The types operation appears in Figure 5.6 on the previous page.) We will return to this in Section 5.6.5.2, below. In the common case of $\Delta' = \varnothing$ (that is, full saturation of the scrutinee), this premise is trivially satisfied. Also note that we do not use the output of this premise, θ, anywhere in the rule, so skipping it on a first reading is appropriate.

Lastly, we must check that the body of the alternative, τ, has the right type. This type must bind (by any combination of matchable Π and unmatchable $\underset{\sim}{\Pi}$—recall that this is the meaning of H from Figure 5.2 on page 77) all of the existentials in Δ_3, as well as the coercion variable witnessing the equality between τ_0 (the scrutinee) and the applied H. In this rule the use of $\mathsf{dom}(\Delta_3)$ as a list of arguments to $H_{\{\overline{\sigma}\}}$ is a small pun; we must imagine braces surrounding any variable in $\mathsf{dom}(\Delta_3)$ that is irrelevantly bound. The return type of the abstraction in τ must be κ, the result kind of the overall match.

For examples of this in action—at least in the fully saturated case—see the worked out examples above (Section 5.5).

5.6.5.2 Unification in Alt_Match

Let's examine the use of unification in Alt_Match more carefully. We will proceed by examining two examples, a simple one where unification is unnecessary and a more involved one showing why we sometimes need it.

Our first example was given above, when first describing unsaturated matching (Section 5.1.1.2):

```
type family IsLeft x where
  IsLeft 'Left  = 'True
  IsLeft 'Right = 'False
```

The translation of *Either* into Pico appears in Section 5.4.2.3. This type family translated to the following Pico function (rewritten to be lowercase according to

Haskell naming requirements):

$$
\begin{aligned}
&\mathit{isLeft} \;:\; \underset{\sim}{\Pi}(a{:}_{\mathsf{Irrel}}\mathbf{Type}), (x{:}_{\mathsf{Rel}}{}'\Pi(y{:}_{\mathsf{Rel}}a).\ \mathit{Either}\ a\ a).\ \mathit{Bool}\\
&\mathit{isLeft} = \lambda(a{:}_{\mathsf{Irrel}}\mathbf{Type}), (x{:}_{\mathsf{Rel}}{}'\Pi(y{:}_{\mathsf{Rel}}a).\ \mathit{Either}\ a\ a).\\
&\qquad \mathbf{case}_{\mathit{Bool}}\ x\ \mathbf{of}\\
&\qquad\quad \mathit{Left}\ \ \to \lambda c_0{:}(x \sim \mathit{Left}_{\{a,a\}}).\ \mathit{True}\\
&\qquad\quad \mathit{Right} \to \lambda c_0{:}(x \sim \mathit{Right}_{\{a,a\}}).\ \mathit{False}
\end{aligned}
$$

Comparing the first alternative against Alt_Match, we see the following concrete instantiations of metavariables:

$$
\begin{aligned}
H &= \mathit{Left} & \overline{\sigma} &= a, a\\
\Delta_1 &= s{:}_{\mathsf{Irrel}}\mathbf{Type}, t{:}_{\mathsf{Irrel}}\mathbf{Type} & \Delta_3 &= \varnothing\\
\Delta_2 &= y{:}_{\mathsf{Rel}}s & \Delta_4 &= y{:}_{\mathsf{Rel}}a\\
H' &= \mathit{Either} & \theta &= \varnothing\\
\tau_0 &= x & \tau &= \lambda(c_0{:}x \sim \mathit{Left}_{\{a,a\}}).\ \mathit{True}\\
\Delta' &= y{:}_{\mathsf{Rel}}a & \kappa &= \mathit{Bool}
\end{aligned}
$$

In this example, the constructor is not applied to any existential variables, and so Δ_3, the telescope of binders that are to be bound by the match, is empty. The only variable bound in the match body is c_0, the dependent-match coercion variable. Also note that Δ_4, the instantiated suffix of the telescope of existential arguments to *Left*, and Δ', the telescope of binders in the type of the scrutinee, coincide. Accordingly, the match operation succeeds with an empty substitution $\theta = \varnothing$.

In contrast, the following example shows why we need unification in Alt_Match:

```
data X where
  MkX :: a → a → X
      -- NB: a is existential; no universals here
type family UnX (x :: Bool '→ X) :: Bool where
  UnX ('MkX y) = y
```

Note that we're extracting the first (visible) argument from an unsaturated use of *MkX*. This Haskell code translates to the following Pico:

$$
\begin{aligned}
&\Sigma = X{:}(\varnothing),\\
&\qquad \mathit{MkX}{:}(a{:}_{\mathsf{Irrel}}\mathbf{Type}, y{:}_{\mathsf{Rel}}a, z{:}_{\mathsf{Rel}}a; X)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{unX} \;:\; \underset{\sim}{\Pi}(x{:}_{\mathsf{Rel}}{}'\Pi(z{:}_{\mathsf{Rel}}\mathit{Bool}).\ X).\ \mathit{Bool}\\
&\mathit{unX} = \underset{\sim}{\lambda}(x{:}_{\mathsf{Rel}}{}'\Pi(z{:}_{\mathsf{Rel}}\mathit{Bool}).\ X).\\
&\qquad \mathbf{case}_{\mathit{Bool}}\ x\ \mathbf{of}\\
&\qquad\quad \mathit{MkX} \to \lambda(a{:}_{\mathsf{Irrel}}\mathbf{Type}), (y{:}_{\mathsf{Rel}}a), (c_0{:}x\ {}^{'\Pi(z:_{\mathsf{Rel}}\mathit{Bool}).\ X}{\sim}^{'\Pi(z:_{\mathsf{Rel}}a).\ X}\ \mathit{MkX}\ a\ y).\\
&\qquad\qquad y \rhd \mathbf{sym}\,(\mathbf{argk}\,(\mathbf{kind}\ c_0))
\end{aligned}
$$

Before we get into the minutiae of Alt_Match, let's dwell a moment on the cast

necessary in the last line. According to both the type of unX and the return type provided in the **case**, the match must return something of type Bool. Yet the body of a match must bind precisely the existential variables of a data constructor; according to the definition of MkX, the variable y has type a, not Bool. We thus must cast y from a to Bool. We do this by extracting out the right coercion from c_0. This c_0 is heterogeneous; I have typeset the code above with the kinds explicit to show this. The left-hand kind is the declared type of x, binding z of type Bool. The right-hand kind is the kind of $\mathsf{MkX\ a\ y}$, which binds z of type a. By using **kind** (which extracts a kind equality from a heterogeneous coercion; see Section 5.8.1), followed by **argk** (which extracts a coercion between the kinds of the arguments of Π-types; see Section 5.8.6.1), and then **sym** (which reverses the orientation of a coercion), we get the coercion needed, of type $\mathsf{a} \sim \mathsf{Bool}$.

Now, we'll try to understand the matching in Alt_Match. Let's once again examine the concrete instantiations of the metavariables in the rule:

$$
\begin{array}{rlrl}
H = & \mathsf{MkX} & \overline{\sigma} = & \varnothing \\
\Delta_1 = & \varnothing & \Delta_3 = & \mathsf{a}{:}_{\mathsf{Irrel}}\mathbf{Type}, \mathsf{y}{:}_{\mathsf{Rel}}\mathsf{a} \\
\Delta_2 = & \mathsf{a}{:}_{\mathsf{Irrel}}\mathbf{Type}, \mathsf{y}{:}_{\mathsf{Rel}}\mathsf{a}, \mathsf{z}{:}_{\mathsf{Rel}}\mathsf{a} & \Delta_4 = & \mathsf{z}{:}_{\mathsf{Rel}}\mathsf{a} \\
H' = & \mathsf{X} & \theta = & \mathsf{Bool}/\mathsf{a} \\
\tau_0 = & \mathsf{x} & \tau = & \langle \text{as above} \rangle \\
\Delta' = & \mathsf{z}{:}_{\mathsf{Rel}}\mathsf{Bool} & \kappa = & \mathsf{Bool}
\end{array}
$$

Recall that Δ_3 and Δ_4 are the prefix and suffix, respectively, of the telescope of existentials Δ_2, after this telescope has been instantiated with the known arguments for the universals. However, with MkX, there are no universals at all (the datatype X takes no arguments), and so this instantiation is a no-op. (The lack of universals shows up in the equations above via an empty Δ_1 and an empty $\overline{\sigma}$.) We thus have $\Delta_3, \Delta_4 = \Delta_2$, where the length of Δ_4 must match the length of Δ', the telescope of variables bound in the type of the scrutinee. We see that the scrutinee x has type $\Pi(\mathsf{z}{:}_{\mathsf{Rel}}\mathsf{Bool}).\,\mathsf{X}$ and so $\Delta' = \mathsf{z}{:}_{\mathsf{Rel}}\mathsf{Bool}$. Thus Δ_3—the existentials bound by the pattern match—has two elements (a and y) and Δ_4 has one (z).

We now must make sure that the shape of the types in Δ' match the template given by the types in Δ_4. That is, Δ' must be some instance of Δ_4, as determined by a unification algorithm (discussed in more depth in Section 7.3). In this case, the unification succeeds, assigning the type variable a to be Bool, as shown in the choice for θ, above. Accordingly, the match is well typed.

Requiring this unification simply reduces the set of well typed programs. It is thus important to understand why the restriction is necessary. What goes wrong if we omit it? The problem comes up in the proof for progress, in the case where the scrutinee has a top-level cast. We will use step rule S_KPush (see Section 5.9); that rule has several typing premises[55] which can be satisfied only when this match succeeds. The restriction is quite technical in nature, but any alternative not ruled out by the type

[55]These unexpected typing premises to a small-step reduction rule are addressed in Section 5.7.4.

of the scrutinee should be acceptable. See the proof of progress in Appendix C.11 for the precise details.

5.6.5.3 Default alternatives

Pico supports *default alternatives* through the form $\_ \to \tau$. This is a catch-all case, to be used only when no other case matches. In a language with a simpler treatment for **case** statements, a default would be unnecessary; every **case** could simply enumerate all possible constructors. However, Pico has two features that makes defaults indispensable:

- When matching on a scrutinee of kind **Type** (or, say, a function returning a **Type**), it would be impossible to enumerate all possibilities of this open type. Such matches must have a default alternative.
- If a scrutinee is partially applied, the typing rules dictate a delicate unification process to make sure alternatives are well typed. (See Section 5.6.5.2.) Given the design of Alt_Match, it is possible some of the constructors of a datatype would be ill typed as patterns in an unsaturated match. It might therefore be challenging to detect whether an unsaturated match is exhaustive. To avoid this problem, unsaturated matches may use a default alternative in order to be unimpeachably exhaustive.

Happily, the typing rule Alt_Default for default alternatives could hardly be simpler.

5.6.5.4 Absurdity

We saw in the *safeHead* example (Section 5.5.4) the need for absurdity elimination via the **absurd** operator. Here is the typing rule:

$$\frac{\begin{array}{l}\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{co}} \gamma : H_{1\{\overline{\tau}_1\}}\,\overline{\psi}_1 \sim H_{2\{\overline{\tau}_2\}}\,\overline{\psi}_2 \qquad H_1 \neq H_2 \\ \Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{ty}} \tau : \mathbf{Type}\end{array}}{\Sigma; \Gamma \vdash_{\mathsf{ty}} \mathbf{absurd}\,\gamma\,\tau : \tau} \quad \textsc{Ty\_Absurd}$$

This rule requires that the coercion argument to **absurd**, γ, relate two unequal type constants H_1 and H_2. The type $\mathbf{absurd}\,\gamma\,\tau$ can have any well formed kind, as chosen by τ. Because τ is needed only to choose the overall kind of the type, it is checked a context reset by Rel.

As explained with the example, absurdity elimination is sometimes needed in the body of case alternatives that can never be reached. In a language that admits *undefined*, the **absurd** construct is not strictly necessary. Yet by including it, we can definitively mark those alternatives that are unreachable. Simply returning *undefined* would not be as informative.

5.7 Operational semantics

Now that we have seen the static semantics of types, we are well placed to explore their dynamic semantics—how the types can reduce to values. The dynamic semantics of types is expressed in Pico via a small-step operational semantics, captured in the judgment $\Sigma; \Gamma \vdash_{\mathsf{s}} \tau \longrightarrow \tau'$. Rules in this judgment are prefixed by "S_". It must be parameterized over a typing environment because of the push rules, as explained in Section 5.7.4.

The operational semantics obeys preservation and progress theorems.

Theorem (Preservation [Theorem C.46]). *If* $\Sigma; \Gamma \vdash_{\mathsf{ty}} \tau : \kappa$ *and* $\Sigma; \Gamma \vdash_{\mathsf{s}} \tau \longrightarrow \tau'$*, then* $\Sigma; \Gamma \vdash_{\mathsf{ty}} \tau' : \kappa$.

Theorem (Progress [Theorem C.78]). *Assume* Γ *has only irrelevant variable bindings. If* $\Sigma; \Gamma \vdash_{\mathsf{ty}} \tau : \kappa$*, then either* τ *is a value* v*,* τ *is a coerced value* $v \rhd \gamma$*, or there exists* τ' *such that* $\Sigma; \Gamma \vdash_{\mathsf{s}} \tau \longrightarrow \tau'$.

The progress theorem is non-standard in two different ways:

- As discussed shortly (Section 5.7.1), reduction can take place in a context with irrelevant variable bindings.
- The progress theorem guarantees that a stuck type is *either* a value v or a coerced value $v \rhd \gamma$. This statement of the theorem follows previous work (such as Weirich et al. [105]) and is applicable in the right spot in the proof of type erasure (Section 5.11).

The operational semantics are also deterministic.

Lemma (Determinacy [Lemma C.20]). *If* $\Sigma; \Gamma \vdash_{\mathsf{s}} \tau \longrightarrow \sigma_1$ *and* $\Sigma; \Gamma \vdash_{\mathsf{s}} \tau \longrightarrow \sigma_2$*, then* $\sigma_1 = \sigma_2$.

5.7.1 Values

A subset of the types τ are considered values, written with the metavariable v:

Definition (Values). *Let values* v *be defined by the following sub-grammar of* τ*:*

$$v ::= H_{\{\overline{\tau}\}}\, \overline{\psi} \mid \Pi\delta.\, \tau \mid \lambda a{:}_{\mathsf{Rel}}\kappa.\, \tau \mid \lambda a{:}_{\mathsf{Irrel}}\kappa.\, v \mid \lambda c{:}\phi.\, \tau$$

As we can see, values include applied constants, Π-types, and some λ-types. However, note a subtle but important part of this definition: the production for irrelevant abstractions is recursive. An irrelevant abstraction $\lambda a{:}_{\mathsf{Irrel}}\kappa.\, \tau$ is a value if and only if τ, the body, is also a value. This choice is important in order to prove type erasure.

Our definition of values also gives us this convenient property:

Lemma (Value types [Lemma C.76]). *If* $\Sigma; \Gamma \vdash_{\mathsf{ty}} v : \kappa$*, then* κ *is a value.*

During compilation, we erase irrelevant components of an expression completely. This includes irrelevant abstractions. Thus, the erasure operation, written $\lVert\cdot\rVert$ and further explored in Section 5.11, includes this equation,

$$\lVert \lambda a{:}_{\mathsf{Irrel}}\kappa.\, \tau \rVert \;=\; \lVert \tau \rVert,$$

erasing the abstraction entirely. Yet we must make sure to maintain the following lemma, referring to the definition of values on erased expressions:

Lemma (Expression redexes [Lemma C.84]). *If* $\lVert\tau\rVert$ *is not a value, then* τ *is not a value.*

If we have the equation above erasing irrelevant abstractions to the erasure of their bodies but call *all* irrelevant abstractions values (that is, make $\lambda a{:}_{\mathsf{Irrel}}\kappa.\, \tau$ a value for all τ), then this lemma becomes false. To wit, suppose τ is not a value. Then $\lVert\lambda a{:}_{\mathsf{Irrel}}\kappa.\, \tau\rVert$ would not be a value, but $\lambda a{:}_{\mathsf{Irrel}}\kappa.\, \tau$ would be. Thus, in order to maintain this lemma, we have a recursive definition of values for irrelevant abstractions and, accordingly, evaluate under irrelevant abstractions as well. See rule S_IrrelAbs_Cong in Section 5.7.3.

5.7.2 Reduction

Several of the small-step rules perform actual reduction in a type:

$$\frac{}{\Sigma; \Gamma \vdash_{\mathsf{s}} (\lambda a{:}_{\mathsf{Rel}}\kappa.\, \sigma_1)\_\sigma_2 \longrightarrow \sigma_1[\sigma_2/a]} \quad \text{S\_BetaRel}$$

$$\frac{}{\Sigma; \Gamma \vdash_{\mathsf{s}} (\lambda a{:}_{\mathsf{Irrel}}\kappa.\, v_1)\_\{\sigma_2\} \longrightarrow v_1[\sigma_2/a]} \quad \text{S\_BetaIrrel}$$

$$\frac{}{\Sigma; \Gamma \vdash_{\mathsf{s}} (\lambda c{:}\phi.\, \sigma)\_\gamma \longrightarrow \sigma[\gamma/c]} \quad \text{S\_CBeta}$$

$$\frac{alt_i \;=\; H \to \tau_0}{\Sigma; \Gamma \vdash_{\mathsf{s}} \mathbf{case}_\kappa\, H_{\{\overline{\tau}\}}\, \overline{\psi}\, \mathbf{of}\, \overline{alt} \longrightarrow \tau_0\, \overline{\psi}\, \langle H_{\{\overline{\tau}\}}\, \overline{\psi}\rangle} \quad \text{S\_Match}$$

$$\frac{alt_i \;=\; \_ \to \sigma \qquad \text{no alternative in } \overline{alt} \text{ matches } H}{\Sigma; \Gamma \vdash_{\mathsf{s}} \mathbf{case}_\kappa\, H_{\{\overline{\tau}\}}\, \overline{\psi}\, \mathbf{of}\, \overline{alt} \longrightarrow \sigma} \quad \text{S\_Default}$$

$$\frac{alt_i \;=\; \_ \to \sigma \qquad \text{no alternative in } \overline{alt} \text{ matches } H}{\Sigma; \Gamma \vdash_{\mathsf{s}} \mathbf{case}_\kappa\, H_{\{\overline{\tau}\}}\, \overline{\psi} \rhd \gamma\, \mathbf{of}\, \overline{alt} \longrightarrow \sigma} \quad \text{S\_DefaultCo}$$

$$\frac{\tau \;=\; \lambda a{:}_{\mathsf{Rel}}\kappa.\, \sigma}{\Sigma; \Gamma \vdash_{\mathsf{s}} \mathbf{fix}\, \tau \longrightarrow \sigma[\mathbf{fix}\, \tau/a]} \quad \text{S\_Unroll}$$

Note that S_BetaIrrel requires a value v_1 in the body of the abstraction in order to keep the rules deterministic. The only other surprising feature in these rules is the way that S_Match works by applying the body of the alternative τ_0 to the actual existential arguments to $H_{\{\overline{\tau}\}}$ and a reflexive coercion. This follows directly from my design of having **case** alternatives avoid a special binding form and use the existing forms in the language.

The Beta rules above make explicit that the application is an unmatchable application $\tau \mathrel{\tilde{}} \psi$. This is actually redundant, as all λ-abstractions are unmatchable. I have included the notation here to make it clearer how these rules line up with the rules in the parallel rewrite relation used to prove consistency (Section 5.10.2).

5.7.3 Congruence forms

Pico has several uninteresting congruence forms,

$$\frac{\Sigma; \Gamma \vdash_{\mathsf{s}} \sigma \longrightarrow \sigma'}{\Sigma; \Gamma \vdash_{\mathsf{s}} \sigma\, \psi \longrightarrow \sigma'\, \psi} \quad \text{S\_App\_Cong}$$

$$\frac{\Sigma; \Gamma \vdash_{\mathsf{s}} \sigma \longrightarrow \sigma'}{\Sigma; \Gamma \vdash_{\mathsf{s}} \sigma \rhd \gamma \longrightarrow \sigma' \rhd \gamma} \quad \text{S\_Cast\_Cong}$$

$$\frac{\Sigma; \Gamma \vdash_{\mathsf{s}} \sigma \longrightarrow \sigma'}{\Sigma; \Gamma \vdash_{\mathsf{s}} \mathbf{case}_\tau\, \sigma\, \mathbf{of}\, \overline{alt} \longrightarrow \mathbf{case}_\tau\, \sigma'\, \mathbf{of}\, \overline{alt}} \quad \text{S\_Case\_Cong}$$

$$\frac{\Sigma; \Gamma \vdash_{\mathsf{s}} \tau \longrightarrow \tau'}{\Sigma; \Gamma \vdash_{\mathsf{s}} \mathbf{fix}\, \tau \longrightarrow \mathbf{fix}\, \tau'} \quad \text{S\_Fix\_Cong}$$

and one more unusual one:

$$\frac{\Sigma; \Gamma, a{:}_{\mathsf{Irrel}}\kappa \vdash_{\mathsf{s}} \sigma \longrightarrow \sigma'}{\Sigma; \Gamma \vdash_{\mathsf{s}} \lambda a{:}_{\mathsf{Irrel}}\kappa.\, \sigma \longrightarrow \lambda a{:}_{\mathsf{Irrel}}\kappa.\, \sigma'} \quad \text{S\_IrrelAbs\_Cong}$$

This last rule allows for evaluation under irrelevant abstractions, as described in Section 5.7.1. It must add the new irrelevant variable to the context, but is otherwise unexceptional.

5.7.4 Push rules

A system with explicit coercions like Pico must deal with the possibility that coercions get in the way of reduction. For example, what happens when we try to reduce

$$((\lambda x{:}_{\mathsf{Rel}}\mathit{Bool}.\, x) \rhd \langle \mathit{Bool} \rangle)\ \mathit{True} \quad ?$$

Casting by a reflexive coercion should hardly matter, and yet no rule yet described applies here. In particular, S_BetaRel does not.

$$\frac{}{\Sigma; \Gamma \vdash_{\mathsf{s}} (v \rhd \gamma_1) \rhd \gamma_2 \longrightarrow v \rhd (\gamma_1 \fatsemi \gamma_2)} \quad \text{S\_Trans}$$

$$\frac{\begin{array}{l}\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{co}} \gamma_0 : \Pi a{:}_{\mathsf{Rel}}\kappa.\, \sigma \sim \Pi a{:}_{\mathsf{Rel}}\kappa'.\, \sigma' \\ \gamma_1 = \mathbf{sym}\,(\mathbf{argk}\,\gamma_0) \qquad \gamma_2 = \gamma_0@(\tau \rhd \gamma_1 \approx_{\mathbf{sym}\,\gamma_1} \tau)\end{array}}{\Sigma; \Gamma \vdash_{\mathsf{s}} (v \rhd \gamma_0)\,\tau \longrightarrow v\,(\tau \rhd \gamma_1) \rhd \gamma_2} \quad \text{S\_PushRel}$$

$$\frac{\begin{array}{l}\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{co}} \gamma_0 : \Pi a{:}_{\mathsf{Irrel}}\kappa.\, \sigma \sim \Pi a{:}_{\mathsf{Irrel}}\kappa'.\, \sigma' \\ \gamma_1 = \mathbf{sym}\,(\mathbf{argk}\,\gamma_0) \qquad \gamma_2 = \gamma_0@(\tau \rhd \gamma_1 \approx_{\mathbf{sym}\,\gamma_1} \tau)\end{array}}{\Sigma; \Gamma \vdash_{\mathsf{s}} (v \rhd \gamma_0)\,\{\tau\} \longrightarrow v\,\{\tau \rhd \gamma_1\} \rhd \gamma_2} \quad \text{S\_PushIrrel}$$

$$\frac{\begin{array}{l}\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{co}} \gamma_0 : \Pi c{:}\phi.\, \sigma \sim \Pi c{:}\phi'.\, \sigma' \\ \gamma_1 = \mathbf{argk}_1\,\gamma_0 \qquad \gamma_2 = \mathbf{argk}_2\,\gamma_0 \\ \eta' = \gamma_1 \fatsemi \eta \fatsemi \mathbf{sym}\,\gamma_2 \qquad \gamma_3 = \gamma_0@(\eta', \eta)\end{array}}{\Sigma; \Gamma \vdash_{\mathsf{s}} (v \rhd \gamma_0)\,\eta \longrightarrow v\,\eta' \rhd \gamma_3} \quad \text{S\_CPush}$$

$$\frac{\begin{array}{l}\gamma_1 = \underset{\sim}{\Pi} a{:}_{\mathsf{Irrel}}\langle\kappa\rangle.\, \gamma \qquad \gamma_2 = \tau_1 \approx_{\langle\mathbf{Type}\rangle} \tau_2 \\ \tau_1 = \underset{\sim}{\Pi} a{:}_{\mathsf{Irrel}}\kappa.\, (\kappa_1[a \rhd \mathbf{sym}\,\langle\kappa\rangle/a]) \qquad \tau_2 = \underset{\sim}{\Pi} a{:}_{\mathsf{Irrel}}\kappa.\, \kappa_1\end{array}}{\Sigma; \Gamma \vdash_{\mathsf{s}} \lambda a{:}_{\mathsf{Irrel}}\kappa.\, (v \rhd \gamma) \longrightarrow (\lambda a{:}_{\mathsf{Irrel}}\kappa.\, v) \rhd (\gamma_1 \fatsemi \gamma_2)} \quad \text{S\_APush}$$

$$\frac{\begin{array}{l}\gamma_1 = \gamma_0@(a \approx_{\gamma_2} a \rhd \gamma_2) \fatsemi \mathbf{sym}\,\gamma_2 \\ \gamma_2 = \mathbf{argk}\,\gamma_0\end{array}}{\Sigma; \Gamma \vdash_{\mathsf{s}} \mathbf{fix}\,((\lambda a{:}_{\mathsf{Rel}}\kappa.\, \sigma) \rhd \gamma_0) \longrightarrow (\mathbf{fix}\,(\lambda a{:}_{\mathsf{Rel}}\kappa.\, (\sigma \rhd \gamma_1))) \rhd \gamma_2} \quad \text{S\_FPush}$$

$$\frac{\begin{array}{l}\Sigma \vdash_{\mathsf{tc}} H : \overline{a}{:}_{\mathsf{Irrel}}\overline{\kappa}; \Delta; H' \qquad \Delta = \Delta_1, \Delta_2 \qquad n = |\Delta_2| \\ \kappa = \text{'}\Pi\overline{a}{:}_{\mathsf{Irrel}}\overline{\kappa}, \Delta.\, H'\,\overline{a} \\ \sigma = \text{'}\Pi(\Delta_2[\overline{\tau}/\overline{a}][\overline{\psi}/\mathsf{dom}(\Delta_1)]).\, H'\,\overline{\tau} \\ \sigma' = \text{'}\Pi(\Delta_2[\overline{\tau}'/\overline{a}][\overline{\psi}'/\mathsf{dom}(\Delta_1)]).\, H'\,\overline{\tau}' \\ \Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{co}} \eta : \sigma \sim \sigma' \\ \Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{vec}} \overline{\tau}' : \overline{a}{:}_{\mathsf{Rel}}\overline{\kappa} \\ \forall i,\ \gamma_i = \mathsf{build\_kpush\_co}(\langle\kappa\rangle@(\mathbf{nths}\,(\mathbf{res}^n\,\eta)); \overline{\psi}_{1\ldots i-1}) \\ \forall i,\ \psi_i' = \mathsf{cast\_kpush\_arg}(\psi_i; \gamma_i) \\ H \to \kappa' \in \overline{alt}\end{array}}{\Sigma; \Gamma \vdash_{\mathsf{s}} \mathbf{case}_{\kappa_0}\,(H_{\{\overline{\tau}\}}\,\overline{\psi}) \rhd \eta\,\mathbf{of}\,\overline{alt} \longrightarrow \mathbf{case}_{\kappa_0}\,H_{\{\overline{\tau}'\}}\,\overline{\psi}'\,\mathbf{of}\,\overline{alt}} \quad \text{S\_KPush}$$

Figure 5.7: Push rules

To deal with this and similar scenarios, PICO follows the System FC tradition and contains so-called *push rules*, as shown in Figure 5.7 on the previous page. These rules are fiddly but—ignoring S_KPUSH for a moment—straightforward. They simply serve to rephrase a type with a coercion in the "wrong" place to an equivalent type with the coercion moved out of the way. The rules can be derived simply by following the typing rules and a desire to push the coercion aside. Compared to previous work, the novelty here is in rules S_APUSH (which handles reduction under irrelevant abstractions and must take into account the awkward substitution in CO_PITY; see Section 5.8.5.1) and S_FPUSH (which handles **fix**, never before seen in System FC), but these rules again pose no design challenge other than the need for attention to detail.

Many of the push rules share an odd feature: they have typing judgment premises. These premises are the reason that the stepping judgment is parameterized on a typing context. In order to prove the progress theorem, it is necessary to prove *consistency* (Section 5.10), which basically says that no coercion (made without assumptions) can prove, say, $\mathit{Int} \sim \mathit{Bool}$. Still ignoring S_KPUSH, the consistency lemma is enough to admit the typing premises to the push rules. However, using consistency here would mean that the preservation theorem depends on the consistency lemma, while consistency is normally used only to prove progress. In seems to lead to cleaner proofs to avoid the dependency of preservation on consistency, and so these typing premises are necessary.

The S_KPUSH rule is very intricate and makes use of a variety of coercions. Explicating this rule in its entirety is best saved until after we have covered coercions in more depth. See Section 5.9.

5.8 Coercions γ

PICO comes with a very rich theory of equality, embodied in the large number of coercion forms. We will examine these forms in terms of the properties they imbue on the equality relation. Note that the coercion language is far from orthogonal; it is often possible to prove one thing in multiple ways. Indeed, GHC comes with a *coercion optimizer* [96] that transforms a coercion proving a certain proposition into another, simpler one proving the same proposition. Enhancing this optimizer is beyond the scope of this dissertation, however. It is needed only as an optimization in the speed of compilation and is not central to the theory or metatheory of the language.

All coercions are erased before runtime (Section 5.11). Accordingly, we check for well typed coercions (via the judgment $\Sigma; \Gamma \vdash_{\mathsf{co}} \gamma : \phi$) only in contexts reset by the $\mathsf{Rel}(\cdot)$ operator.

5.8.1 Equality is heterogeneous

The equality relation in PICO is heterogeneous, allowing $\sim$ to relate two types of different kinds. This is most clearly demonstrated in the rule for the well-formedness

of propositions:[56]

$\boxed{\Sigma; \Gamma \vdash_{\mathsf{prop}} \phi\ \mathsf{ok}}$ Proposition formation

$$\frac{\Sigma; \Gamma \vdash_{\mathsf{ty}} \tau_1 : \kappa_1 \quad \Sigma; \Gamma \vdash_{\mathsf{ty}} \tau_2 : \kappa_2}{\Sigma; \Gamma \vdash_{\mathsf{prop}} \tau_1\ {}^{\kappa_1}\!\sim^{\kappa_2}\ \tau_2\ \mathsf{ok}} \quad \textsc{Prop\_Equality}$$

Note that the kinds κ_1 and κ_2 are allowed to differ.

The particular flavor of heterogeneous equality in Pico is so-called "John Major" equality [58], where an equality between two types implies the equality between the kinds:

$$\frac{\Sigma; \Gamma \vdash_{\mathsf{co}} \gamma : \tau_1\ {}^{\kappa_1}\!\sim^{\kappa_2}\ \tau_2}{\Sigma; \Gamma \vdash_{\mathsf{co}} \mathbf{kind}\, \gamma : \kappa_1 \sim \kappa_2} \quad \textsc{Co\_Kind}$$

As we can see, the **kind** coercion form extracts a kind coercion from a type coercion.

Though I have described my equality relation following McBride [58], he uses identity proofs in quite a different way than I do here. His language confirms that an identity proof is reflexive and then brings *definitional* equalities of the types and kinds into scope. The surface Haskell version of heterogeneous equality works quite like McBride's. My invocation of "John Major" here is to recall that an equality between types implies the same relationship among the kinds.

It's worth pausing here for a moment to consider two other possible meanings, among others, of heterogeneous equality:

Trellys equality The equality relation studied in the Trellys project [13] a heterogeneous equality with no equivalent of the **kind** coercion. That is, if we have a proof of $\tau_1\ {}^{\kappa_1}\!\sim^{\kappa_2}\ \tau_2$, then there is no way to prove $\kappa_1 \sim \kappa_2$ (absent other information). Indeed, Trellys equality (that is, omitting the Co_Kind rule) would work in Pico; that coercion form is never needed in the metatheory. Omitting it would weaken Pico's equational theory, however, and so I have decided to include it.

Flexible homogeneous equality Another potential meaning of heterogeneous equality is that κ_1 and κ_2 might not be identical—as they would be in a traditional homogeneous equality relation—but they are propositionally equal.[57] Such an equality

[56] This rule is the entire judgment—there is no other form of proposition supported in Pico.

[57] I am distinguishing here between *definitional* equality and *propositional* equality. The former, in Pico, refers to α-equivalence. Definitional equality is the equality used implicitly in typing rules when we use the same metavariable twice. If written explicitly, it is sometimes written $\equiv$. Propositional equality, on the other hand, means an equality that must be accompanied by a proof; in Pico, $\sim$ is the propositional equality relation. Languages with a Conv rule (Section 5.1) import propositional equality into their definitional equality. Pico does not do this, requiring a cast to use a propositional equality.

would use this rule (not part of Pico):

$$\frac{\Sigma; \Gamma \vdash_{\mathsf{ty}} \tau_1 : \kappa_1 \qquad \Sigma; \Gamma \vdash_{\mathsf{ty}} \tau_2 : \kappa_2 \\ \Sigma; \Gamma \vdash_{\mathsf{co}} \gamma : \kappa_1 \sim \kappa_2}{\Sigma; \Gamma \vdash_{\mathsf{prop}} \tau_1 \ {}^{\kappa_1}\!\sim^{\kappa_2}_{\gamma} \ \tau_2 \ \mathsf{ok}} \quad \textsc{Prop\_Homogeneous}$$

Note how $\sim$ is indexed by γ, the proof that the kinds are equal. I call this equality homogeneous, because even to form the equality $\tau_1 \sim \tau_2$, we must know that the kinds are equal. Contrast to Prop_Equality, where the proposition itself is well formed even when the kinds and/or types are not provably equal.

5.8.2 Equality is hypothetical

A key property of equality in Pico is that programs can *assume* an equality proof. This is how GADTs are implemented, by packing an equality proof into a nugget of data and then extracting it again on pattern match. In the body of the pattern match, we can assume the packed equality. Here is the typing rule:

$$\frac{\Sigma \vdash_{\mathsf{ctx}} \Gamma \ \mathsf{ok} \qquad c{:}\phi \in \Gamma}{\Sigma; \Gamma \vdash_{\mathsf{co}} c : \phi} \quad \textsc{Co\_Var}$$

Coercion variables are brought into scope by Π and λ over coercion binders.

5.8.3 Equality is coherent

Pico's equality relation is *coherent*, in that the precise locations and structure of coercions within types is immaterial. This is a critical property because it is intended for a compiler to create and place these coercions. The type system must be agnostic to where, precisely, they are placed. Coherence is obtained through this coercion form:

$$\frac{\Sigma; \Gamma \vdash_{\mathsf{co}} \eta : \kappa_1 \sim \kappa_2 \qquad \lfloor \tau_1 \rfloor = \lfloor \tau_2 \rfloor \\ \Sigma; \Gamma \vdash_{\mathsf{ty}} \tau_1 : \kappa_1 \qquad \Sigma; \Gamma \vdash_{\mathsf{ty}} \tau_2 : \kappa_2}{\Sigma; \Gamma \vdash_{\mathsf{co}} \tau_1 \approx_\eta \tau_2 : \tau_1 \sim \tau_2} \quad \textsc{Co\_Coherence}$$

This coercion form requires two well kinded types τ_1 and τ_2 as well as a coercion η that relates their kinds. It also requires the critical premise that $\lfloor \tau_1 \rfloor = \lfloor \tau_2 \rfloor$, where $\lfloor \cdot \rfloor$ is a *coercion erasure* operation. This operation is separate from (though similar to) the type erasure operation spelled $\llbracket \cdot \rrbracket$ and discussed several times thus far. The full definition of this operation is given in Definition C.47. Briefly, coercion erasure is defined recursively on types, binders, case alternatives, and propositions by the

following equations, treating other forms homomorphically:

$$\begin{array}{rclcrcl} \lfloor \tau\, \gamma \rfloor & = & \lfloor \tau \rfloor \bullet & & \lfloor \tau \rhd \gamma \rfloor & = & \lfloor \tau \rfloor \\ \lfloor \mathbf{absurd}\, \gamma\, \tau \rfloor & = & \mathbf{absurd} \bullet \lfloor \tau \rfloor & & \lfloor (c{:}\phi) \rfloor & = & (\bullet{:}\lfloor \phi \rfloor) \end{array}$$

As we can see coercion erasure simply removes the coercions from a type. We use $\bullet$ to stand in for an erased coercion application. I sometimes use the metavariable ϵ to stand for a type that has its coercions erased, but τ and σ may also refer to a coercion-erased type, if that is clear from the context.

By using coercion erasure in its premise, the coherence coercion can relate any two types that are the same, ignoring the coercions. This is precisely what we mean by coherence.

The coherence rule implies that any two proofs of equality are considered interchangeable. In other words, Pico assumes the uniqueness of identity proofs (UIP) [44]. This choice makes Pico "anti-HoTT", that is, incompatible with homotopy type theory [91], which takes as a key premise that there may be more than one way to prove the identity between two types. While baking UIP into the language may limit its applicability, Pico's intended role as an intermediate language, where the coercions are inferred by the compiler, makes this choice necessary. We would not want the static semantics of our programs to depend on the vagaries of how the compiler placed its equality proofs.

Note that the coherence form in Pico is rather more general than the coherence form used in my prior work [105]. The way I have phrased coherence is critical for my consistency proof. See Section 5.10.5 for more discussion.

5.8.4 Equality is an equivalence

The equality relation $\sim$ is explicitly an equivalence relation, via these rules:

$$\frac{\Sigma; \Gamma \vdash_{\mathsf{ty}} \tau : \kappa}{\Sigma; \Gamma \vdash_{\mathsf{co}} \langle \tau \rangle : \tau \sim \tau} \quad \textsc{Co\_Refl}$$

$$\frac{\Sigma; \Gamma \vdash_{\mathsf{co}} \gamma : \tau_1 \sim \tau_2}{\Sigma; \Gamma \vdash_{\mathsf{co}} \mathbf{sym}\, \gamma : \tau_2 \sim \tau_1} \quad \textsc{Co\_Sym}$$

$$\frac{\Sigma; \Gamma \vdash_{\mathsf{co}} \gamma_1 : \tau_1 \sim \tau_2 \qquad \Sigma; \Gamma \vdash_{\mathsf{co}} \gamma_2 : \tau_2 \sim \tau_3}{\Sigma; \Gamma \vdash_{\mathsf{co}} \gamma_1 \fatsemi \gamma_2 : \tau_1 \sim \tau_3} \quad \textsc{Co\_Trans}$$

Note the use of $\langle \tau \rangle$ to denote a reflexive coercion over the type τ.

5.8.5 Equality is (almost) congruent

Given coercions between the component parts of two types, we often want to build a coercion relating the types themselves. For example, if we know that $\Sigma; \Gamma \vdash_{\mathsf{co}} \gamma_1 : \tau_1 \sim \sigma_1$

and $\Sigma; \Gamma \vdash_{\mathsf{co}} \gamma_2 : \tau_2 \sim \sigma_2$, then we can build $\Sigma; \Gamma \vdash_{\mathsf{co}} \gamma_1\,\gamma_2 : \tau_1\,\tau_2 \sim \sigma_1\,\sigma_2$. The form $\gamma_1\,\gamma_2$ is typed by a congruence rule; each form of type has an associated congruence rule. The rules that do not bind variables appear in Figure 5.8 on the following page; I'll call these the simple congruence rules. Rules that do bind variables are subtler; they appear in Figure 5.9 on page 109.

The simple congruence rules simply build up larger coercions from smaller ones. With the exception of Co_Absurd, they assert that the types related by the coercion are well formed; it is easier simply to check the types than to repeat all the conditions in the relevant typing rules. The typing premises for **absurd** are simple enough on their own, however.

The notation I use for congruence rules deliberately mimics that of types. However, do not be fooled: the coercion $\gamma_1\,\gamma_2$ does *not* apply a "coercion function" γ_1 to some argument. The coercion $\gamma_1\,\gamma_2$ never β-reduces to become some $\gamma[\gamma_2/c]$. Similarly, the λ-coercion (one of the binding congruence forms) does *not* define a λ-abstraction over coercions; it witnesses the equality between two λ-abstraction types.

Two of the congruence rules—Co_CApp and Co_Absurd—relate types that mention coercions. In these congruence rules, the coercion γ must explicitly mention the two coercions that appear in the respective locations in the related types, as we do not have a coercion form that relates coercions. For example, examine Co_CApp, declaring that $\gamma_0\,(\gamma_1, \gamma_2)$ relates $\tau_1\,\gamma_1$ and $\tau_2\,\gamma_2$, given that γ_0 relates τ_1 and τ_2. Instead of (γ_1, γ_2) appearing in the coercion, we might naively expect some η that relates γ_1 and γ_2; since such an η does not exist in the grammar, we just list the two coercions γ_1 and γ_2. The syntax for Co_Absurd is similar.

5.8.5.1 Binding congruence forms

The binding coercions forms (Figure 5.9 on page 109) all have a particular challenge to meet. Suppose we know that $\Sigma; \Gamma \vdash_{\mathsf{co}} \eta : \kappa_1 \sim \kappa_2$ and we wish to prove equality between $\Pi a{:}_\rho \kappa_1.\,\tau_1$ and $\Pi a{:}_\rho \kappa_2.\,\tau_2$. We surely must have a coercion γ relating τ_1 to τ_2. But in what context should we check γ? We cannot assign a both κ_1 and κ_2.

In Pico, I have chosen to favor the left-hand kind in the context and do a substitution in the result. Let's examine Co_PiTy closely. The coercion η indeed relates κ_1 and κ_2. The coercion γ is checked in the context $\Gamma, a{:}_{\mathsf{Rel}}\kappa_1$—note the use of κ_1 there. Regardless of the relevance annotation ρ on the coercion, the context is extended with a binding marked Rel, echoing the use of $\mathsf{Rel}(\delta)$ in the premise to Ty_Pi (Section 5.6.1). The types related by γ (σ_1 and σ_2) might mention a, assumed to be of type κ_1. For σ_1, that assumption is correct; the left-hand type in the result is $\Pi a{:}_\rho \kappa_1.\,\sigma_1$. However, for σ_2, this assumption is wrong: we wish a to have kind κ_2 in the right-hand result type. In order to fix up the mess, the conclusion of Co_PiTy does an unusual substitution, mentioning the type $\sigma_2[a \rhd \mathbf{sym}\,\eta/a]$. This takes σ_2—well typed in a context where a has kind κ_1—and changes it to expect a to have kind κ_2. It does this by casting a by $\mathbf{sym}\,\eta$, a coercion from κ_2 to κ_1. We can thus use the (standard) substitution lemma (Lemma C.35) to show that this result type is itself

$$\frac{\forall i,\ \Sigma;\Gamma \vdash_{\mathsf{co}} \gamma_i : \sigma_i \sim \sigma'_i \quad \Sigma;\Gamma \vdash_{\mathsf{ty}} H_{\{\overline{\sigma}\}} : \kappa_1 \qquad \Sigma;\Gamma \vdash_{\mathsf{ty}} H_{\{\overline{\sigma'}\}} : \kappa_2}{\Sigma;\Gamma \vdash_{\mathsf{co}} H_{\{\overline{\gamma}\}} : H_{\{\overline{\sigma}\}} \sim H_{\{\overline{\sigma'}\}}} \quad \textsc{Co\_Con}$$

$$\frac{\Sigma;\Gamma \vdash_{\mathsf{co}} \gamma_1 : \tau_1 \sim \tau_2 \quad \Sigma;\Gamma \vdash_{\mathsf{co}} \gamma_2 : \sigma_1 \sim \sigma_2 \quad \Sigma;\Gamma \vdash_{\mathsf{ty}} \tau_1\,\sigma_1 : \kappa_1 \qquad \Sigma;\Gamma \vdash_{\mathsf{ty}} \tau_2\,\sigma_2 : \kappa_2}{\Sigma;\Gamma \vdash_{\mathsf{co}} \gamma_1\,\gamma_2 : \tau_1\,\sigma_1 \sim \tau_2\,\sigma_2} \quad \textsc{Co\_AppRel}$$

$$\frac{\Sigma;\Gamma \vdash_{\mathsf{co}} \gamma_1 : \tau_1 \sim \tau_2 \quad \Sigma;\Gamma \vdash_{\mathsf{co}} \gamma_2 : \sigma_1 \sim \sigma_2 \quad \Sigma;\Gamma \vdash_{\mathsf{ty}} \tau_1\,\{\sigma_1\} : \kappa_1 \qquad \Sigma;\Gamma \vdash_{\mathsf{ty}} \tau_2\,\{\sigma_2\} : \kappa_2}{\Sigma;\Gamma \vdash_{\mathsf{co}} \gamma_1\,\{\gamma_2\} : \tau_1\,\{\sigma_1\} \sim \tau_2\,\{\sigma_2\}} \quad \textsc{Co\_AppIrrel}$$

$$\frac{\Sigma;\Gamma \vdash_{\mathsf{co}} \gamma_0 : \tau_1 \sim \tau_2 \quad \Sigma;\Gamma \vdash_{\mathsf{ty}} \tau_1\,\gamma_1 : \kappa_1 \qquad \Sigma;\Gamma \vdash_{\mathsf{ty}} \tau_2\,\gamma_2 : \kappa_2}{\Sigma;\Gamma \vdash_{\mathsf{co}} \gamma_0\,(\gamma_1,\gamma_2) : \tau_1\,\gamma_1 \sim \tau_2\,\gamma_2} \quad \textsc{Co\_CApp}$$

$$\frac{\Sigma;\Gamma \vdash_{\mathsf{co}} \eta : \kappa_1 \sim \kappa_2 \qquad \Sigma;\Gamma \vdash_{\mathsf{co}} \gamma_0 : \tau_1 \sim \tau_2 \quad \forall i,\ \Sigma;\Gamma \vdash_{\mathsf{co}} \gamma_i : \sigma_i \sim \sigma'_i \quad \overline{alt}_1 = \overline{\pi_i \to \sigma_i} \qquad \overline{alt}_2 = \overline{\pi_i \to \sigma'_i} \quad \Sigma;\Gamma \vdash_{\mathsf{ty}} \mathbf{case}_{\kappa_1}\,\tau_1\,\mathbf{of}\,\overline{alt}_1 : \kappa_1 \qquad \Sigma;\Gamma \vdash_{\mathsf{ty}} \mathbf{case}_{\kappa_2}\,\tau_2\,\mathbf{of}\,\overline{alt}_2 : \kappa_2}{\Sigma;\Gamma \vdash_{\mathsf{co}} \mathbf{case}_{\eta}\,\gamma_0\,\mathbf{of}\,\overline{\pi_i \to \gamma_i} : \mathbf{case}_{\kappa_1}\,\tau_1\,\mathbf{of}\,\overline{alt}_1 \sim \mathbf{case}_{\kappa_2}\,\tau_2\,\mathbf{of}\,\overline{alt}_2} \quad \textsc{Co\_Case}$$

$$\frac{\Sigma;\Gamma \vdash_{\mathsf{co}} \gamma : \tau_1 \sim \tau_2 \quad \Sigma;\Gamma \vdash_{\mathsf{ty}} \mathbf{fix}\,\tau_1 : \kappa_1 \qquad \Sigma;\Gamma \vdash_{\mathsf{ty}} \mathbf{fix}\,\tau_2 : \kappa_2}{\Sigma;\Gamma \vdash_{\mathsf{co}} \mathbf{fix}\,\gamma : \mathbf{fix}\,\tau_1 \sim \mathbf{fix}\,\tau_2} \quad \textsc{Co\_Fix}$$

$$\frac{\Sigma;\Gamma \vdash_{\mathsf{co}} \gamma_1 : H_{1\{\overline{\tau}_1\}}\,\overline{\psi}_1 \sim H'_{1\{\overline{\tau}'_1\}}\,\overline{\psi}'_1 \qquad H_1 \neq H'_1 \quad \Sigma;\Gamma \vdash_{\mathsf{co}} \gamma_2 : H_{2\{\overline{\tau}_2\}}\,\overline{\psi}_2 \sim H'_{2\{\overline{\tau}'_2\}}\,\overline{\psi}'_2 \qquad H_2 \neq H'_2 \quad \Sigma;\Gamma \vdash_{\mathsf{co}} \eta : \kappa_1 \sim \kappa_2}{\Sigma;\Gamma \vdash_{\mathsf{co}} \mathbf{absurd}\,(\gamma_1,\gamma_2)\,\eta : \mathbf{absurd}\,\gamma_1\,\kappa_1 \sim \mathbf{absurd}\,\gamma_2\,\kappa_2} \quad \textsc{Co\_Absurd}$$

Figure 5.8: Congruence rules that do not bind variables

$$\frac{\begin{array}{l}\Sigma;\Gamma \vdash_{\mathsf{co}} \eta : \kappa_1 \ {}^{\mathbf{Type}}\!\sim^{\mathbf{Type}} \kappa_2 \\ \Sigma;\Gamma, a{:}_{\mathsf{Rel}}\kappa_1 \vdash_{\mathsf{co}} \gamma : \sigma_1 \ {}^{\mathbf{Type}}\!\sim^{\mathbf{Type}} \sigma_2\end{array}}{\Sigma;\Gamma \vdash_{\mathsf{co}} \Pi a{:}_\rho \eta.\, \gamma : (\Pi a{:}_\rho \kappa_1.\, \sigma_1) \sim (\Pi a{:}_\rho \kappa_2.\, (\sigma_2[a \rhd \mathbf{sym}\,\eta/a]))} \quad \textsc{Co\_PiTy}$$

$$\frac{\begin{array}{l}\Sigma;\Gamma \vdash_{\mathsf{co}} \eta_1 : \tau_1 \sim \tau_2 \qquad \Sigma;\Gamma \vdash_{\mathsf{co}} \eta_2 : \sigma_1 \sim \sigma_2 \\ \Sigma;\Gamma, c{:}\tau_1 \sim \sigma_1 \vdash_{\mathsf{co}} \gamma : \kappa_1 \ {}^{\mathbf{Type}}\!\sim^{\mathbf{Type}} \kappa_2 \qquad c \mathbin{\tilde{\#}} \gamma \\ \eta_3 \ = \ \eta_1 \fatsemi c \fatsemi \mathbf{sym}\,\eta_2\end{array}}{\Sigma;\Gamma \vdash_{\mathsf{co}} \Pi c{:}(\eta_1,\eta_2).\, \gamma : (\Pi c{:}\tau_1 \sim \sigma_1.\, \kappa_1) \sim (\Pi c{:}\tau_2 \sim \sigma_2.\, (\kappa_2[\eta_3/c]))} \quad \textsc{Co\_PiCo}$$

$$\frac{\begin{array}{l}\Sigma;\Gamma \vdash_{\mathsf{co}} \eta : \kappa_1 \sim \kappa_2 \\ \Sigma;\Gamma, a{:}_\rho\kappa_1 \vdash_{\mathsf{co}} \gamma : \tau_1 \sim \tau_2 \\ \Sigma;\Gamma, a{:}_\rho\kappa_1 \vdash_{\mathsf{ty}} \tau_1 : \sigma_1 \qquad \Sigma;\Gamma, a{:}_\rho\kappa_1 \vdash_{\mathsf{ty}} \tau_2 : \sigma_2\end{array}}{\Sigma;\Gamma \vdash_{\mathsf{co}} \lambda a{:}_\rho\eta.\, \gamma : \lambda a{:}_\rho\kappa_1.\, \tau_1 \sim \lambda a{:}_\rho\kappa_2.\, (\tau_2[a \rhd \mathbf{sym}\,\eta/a])} \quad \textsc{Co\_Lam}$$

$$\frac{\begin{array}{l}\Sigma;\Gamma \vdash_{\mathsf{co}} \eta_1 : \tau_1 \sim \tau_2 \qquad \Sigma;\Gamma \vdash_{\mathsf{co}} \eta_2 : \sigma_1 \sim \sigma_2 \\ \Sigma;\Gamma, c{:}\tau_1 \sim \sigma_1 \vdash_{\mathsf{co}} \gamma : \kappa_1 \sim \kappa_2 \qquad c \mathbin{\tilde{\#}} \gamma \\ \eta_3 \ = \ \eta_1 \fatsemi c \fatsemi \mathbf{sym}\,\eta_2\end{array}}{\Sigma;\Gamma \vdash_{\mathsf{co}} \lambda c{:}(\eta_1,\eta_2).\, \gamma : (\lambda c{:}\tau_1 \sim \sigma_1.\, \kappa_1) \sim (\lambda c{:}\tau_2 \sim \sigma_2.\, (\kappa_2[\eta_3/c]))} \quad \textsc{Co\_CLam}$$

Figure 5.9: Congruence rules that bind variables

well typed, as needed to prove regularity (Lemma C.44). The other binding congruence forms use similar substitutions in their conclusions, for similar reasons.

This extra substitution in the conclusion is indeed asymmetric and a bit unwieldy,[58] but this treatment is, on balance, better than the only known alternative. Other type systems similar to PICO [37, 92, 105] use an entirely different way of handling congruence coercions with binders: instead of trying to treat a as a variable with two different kinds, they invent fresh variables. What I write as $\Pi a{:}_\rho\eta.\,\gamma$, they would write as $\Pi_\eta(a_1, a_2, c).\gamma$, binding $a_1 : \kappa_1$ and $a_2 : \kappa_2$, as well as a coercion $c : a_1 \sim a_2$. You can see either of those works for the details, but I have found this construction worse than the asymmetrical version. Other than the bookkeeping overhead of extra variables, the three-variable version also requires us to introduce a coercion variable even when making a congruence coercion over a Π-type over a type variable. Coercion variables in the context cause trouble (as discussed in Section 5.10.3), and my one-variable version helps to contain the trouble. See Section 5.10.5.3 for more discussion.

As a further support to my choice of a one-variable binding form with an asymmetrical rule, I have implemented both versions in GHC. Initially, I implemented the three-variable form from Weirich et al. [105]. This worked, but it was often hard to construct the coercions, and it was sometimes a struggle to find names guaranteed to

[58] See the statement of the push rule S_APUSH (Section 5.7.4) for an example of how its unwieldiness can bite.

be fresh. When I refactored the code to use the one-variable version formalized here, the code became simpler.

5.8.5.2 Congruence over coercion binders

The congruence forms over types that bind coercion variables (rules Co_PiCo and Co_CLam) have two more wrinkles. The first is that there is no equivalent of Co_PiTy's η coercion that relates two propositions; we must settle for the pair of coercions (η_1, η_2) that appear in Co_PiCo and Co_CLam. These coercions relate corresponding parts of the propositions. The second wrinkle is in the $c \mathrel{\tilde{\#}} \gamma$ premise of both of these rules.

Definition ("Almost devoid"). *Define $c \mathrel{\tilde{\#}} \gamma$ (pronounced "γ is almost devoid of c") to mean that the coercion variable c appears nowhere in γ except, perhaps, in one of the types related by a $\tau_1 \approx_\eta \tau_2$ coercion.*

The almost-devoid condition on Co_PiCo and Co_CLam restricts where the bound variable c can appear in the coercion body. This technical restriction, based on the original idea by Weirich et al. [105], is necessary for my proof of consistency (Section 5.10) to go through. The motivation for the restriction is discussed in depth in Section 5.10.3.

The key example that this restriction forbids looks like this:

$$\Sigma; \Gamma \not\vdash_{\mathsf{co}} \Pi c{:}(\langle \mathsf{Int} \rangle, \langle \mathsf{Bool} \rangle).\ c : (\Pi c{:}\mathsf{Int} \sim \mathsf{Bool}.\ \mathsf{Int}) \sim (\Pi c{:}\mathsf{Int} \sim \mathsf{Bool}.\ \mathsf{Bool})$$

It would seem that this coercion would not cause harm, yet I know of no way to prove consistency while allowing it. See Section 5.10.5 for a discussion of other approaches.

Happily, this restriction is not likely to bite when translating Dependent Haskell programs to Pico, as we can write functions witnessing the isomorphism between the two types related above:

$$\begin{aligned}
&\mathsf{to} : \tilde{\Pi}(x{:}_{\mathsf{Rel}}(\tilde{\Pi} c{:}\mathsf{Int} \sim \mathsf{Bool}.\ \mathsf{Int})).\ (\tilde{\Pi} c{:}\mathsf{Int} \sim \mathsf{Bool}.\ \mathsf{Bool}) \\
&\mathsf{to} = \lambda(x{:}_{\mathsf{Rel}}(\tilde{\Pi} c{:}\mathsf{Int} \sim \mathsf{Bool}.\ \mathsf{Int})), (c{:}\mathsf{Int} \sim \mathsf{Bool}).\ (x\ c) \rhd c \\
&\mathsf{from} : \tilde{\Pi}(x{:}_{\mathsf{Rel}}(\tilde{\Pi} c{:}\mathsf{Int} \sim \mathsf{Bool}.\ \mathsf{Bool})).\ (\tilde{\Pi} c{:}\mathsf{Int} \sim \mathsf{Bool}.\ \mathsf{Int}) \\
&\mathsf{from} = \lambda(x{:}_{\mathsf{Rel}}(\tilde{\Pi} c{:}\mathsf{Int} \sim \mathsf{Bool}.\ \mathsf{Bool})), (c{:}\mathsf{Int} \sim \mathsf{Bool}).\ (x\ c) \rhd \mathbf{sym}\ c
\end{aligned}$$

A compiler of Dependent Haskell creates functions such as these as it is compiling a subsumption relationship $\leq$, as discussed further in Section 6.4.2. In other words, while we don't have $(\Pi c{:}\mathsf{Int} \sim \mathsf{Bool}.\ \mathsf{Int}) \sim (\Pi c{:}\mathsf{Int} \sim \mathsf{Bool}.\ \mathsf{Bool})$, these two types are related by $\leq$, in both directions. This mean that a Dependent Haskell program that expects one of these types in a certain context, but gets the other type, is still well typed.

When can the lack of the equality proof bite? Only when that proof is needed as a coercion argument to some function or GADT constructor. As we've just seen, using

it to cast is unnecessary, as we can just use one component of the isomorphism. The forbidden equalities all relate Π-types over coercions. Yet, in Dependent Haskell, an abstraction over an equality constraint is considered a polytype. Passing a polytype as an argument is considered a use of impredicativity, which is not supported. (See Section 4.4.4.) In particular, the equality constraint $((\mathsf{Int} \sim \mathsf{Bool}) \Rightarrow \mathsf{Int}) \sim ((\mathsf{Int} \sim \mathsf{Bool}) \Rightarrow \mathsf{Bool})$ is malformed in Dependent Haskell, because it passes polytypes as arguments to $\sim$. I thus conjecture that no Dependent Haskell program is ruled out because of the coercion variable restriction. Proving such a claim seems challenging, however, and remains as an exercise for the reader.

5.8.5.3 (Almost) Congruence

The coercion variable restriction means that equality is not quite congruent, according to the following definition:

Definition (Congruence)**.** *Equality is congruent if, whenever* $\Sigma; \Gamma \vdash_{\mathsf{co}} \gamma : \sigma_1 \ {}^{\kappa}\!\sim^{\kappa} \sigma_2$ *and* $\Sigma; \Gamma, a{:}_{\rho}\kappa \vdash_{\mathsf{ty}} \tau : \kappa_0$*, there exists* η *such that* $\Sigma; \Gamma \vdash_{\mathsf{co}} \eta : \tau[\sigma_1/a] \ {}^{\kappa_0[\sigma_1/a]}\!\sim^{\kappa_0[\sigma_2/a]} \tau[\sigma_2/a]$.

If we were to try to prove that equality is congruent, it seems natural to proceed by induction on the typing derivation for τ. However, in the proof, we are stuck when $\tau = \lambda c{:}\phi.\, \tau_0$. The congruence form for λ-types over coercions is no help because of the coercion variable restriction.[59] If we strengthen the induction hypothesis to provide what we need in this case, then other cases fail, unable to obey the restriction.

As a concrete example, consider this: Let $\Gamma = \mathsf{y}{:}_{\mathsf{Rel}}\mathsf{Int}, c{:}3 \sim \mathsf{y}$ and $\tau = \lambda(c'{:}\mathsf{Int} \sim \mathsf{Bool}).\, \mathsf{x} \rhd c'$. We know $\Sigma; \Gamma \vdash_{\mathsf{co}} c : 3 \sim \mathsf{y}$ and $\Sigma; \Gamma, \mathsf{x}{:}_{\mathsf{Rel}}\mathsf{Int} \vdash_{\mathsf{ty}} \tau : \Pi(c'{:}\mathsf{Int} \sim \mathsf{Bool}).\, \mathsf{Bool}$. Yet there seems to be no way to construct a proof of $\tau[3/\mathsf{x}] \sim \tau[\mathsf{y}/\mathsf{x}]$.[60]

Instead of proving congruence, I am left proving almost-congruence, as follows:

Definition (Unrestricted coercion variables [Definition C.87])**.** *Define a new judgment* $\vdash^{*}_{\mathsf{co}}$ *to be identical to* $\vdash_{\mathsf{co}}$*, except with the* $c \mathrel{\tilde{\#}} \gamma$ *premises removed from rules* Co_PiCo *and* Co_CLam *and all recursive uses of* $\vdash_{\mathsf{co}}$ *replaced with* $\vdash^{*}_{\mathsf{co}}$.

Now, the proof for the following theorem is straightforward:

Theorem ((Almost) Congruence [Theorem C.90])**.** *Equality is congruent with the judgment* $\vdash^{*}_{\mathsf{co}}$.

[59] Contrast to the proof of the lifting lemma in my prior work [106]; that proof relies on a critical auxiliary lemma (their Lemma C.7) which requires a different coercion variable restriction than what I am using here. Furthermore, I show in Section 5.10.5.2 that their restriction is too weak.

[60] It is tempting to try to prove this by using the Co_CLam form and then coherence forms stitched together with transitivity; after all, the $c \mathrel{\tilde{\#}} \gamma$ restriction in Co_CLam does not affect the types in a coherence coercion. However, the η coercion in the coherence coercion (η relates the kinds of the types mentioned in the coherence coercion) must still be devoid of c, and that is where this plan falls apart.

What this means, in practice, is that we can often think of equality as congruent, and intuition about the equality relation stemming from congruence is often accurate. In particular, if the type τ in the statement of congruence has no coercion abstractions or Π-types, then congruence with respect to $\vdash_{\mathsf{co}}$ holds.[61]

5.8.5.4 Consequences of congruence

Congruence is not, thankfully, a necessary property of Pico. Nowhere in the metatheory do we rely on this result (or lack thereof).

In the implementation, however, congruence[62] is used to perform some coercion optimizations [96]. After desugaring Haskell into its Core language (currently based on the version of System FC as described in my prior work [105]), GHC optionally performs coercion optimization, in the hope of converting large coercions into smaller ones that prove the same propositions. This speeds up compilation and reduces the size of the interface files that GHC writes to disk to store information about compiled modules; the optimization has no effect at runtime, however, because coercions are fully erased before execution.

Congruence comes into play when optimizing a coercion such as $(\Pi a{:}_\rho \eta.\, \gamma_1)@\gamma_2$, where $\gamma_1@\gamma_2$ is a decomposition form that instantiates a Π-type (Section 5.8.6.2). Without going into further detail, in order to perform the instantiation requested, we must find exactly the coercion suggested in the definition of congruence above. Since Pico lacks congruence, the updated coercion optimizer sometimes fails to optimize these coercions. The troublesome case—when we would run afoul of the $c \mathbin{\tilde{\#}} \gamma$ restrictions in Co_PiCo and Co_CLam—is easy to detect, and the optimization is simply skipped when this were to happen. The lack of congruence does not otherwise bite.

5.8.6 Equality can be decomposed

Pico comes equipped with a large variety of ways of decomposing an equality to get out a smaller one—in some sense, these are the inverses of the congruence forms. We will approach these in batches.

5.8.6.1 The argk forms

The coercion form **argk** extracts a coercion between the kinds of the bound variables in a coercion relating abstractions. The rules appear in Figure 5.10 on the next page. The rules are actually straightforward; look at Co_ArgK for a typical example. This form extracts the equality between κ_1 and κ_2 from the type of γ. The other forms work

[61] This intuition is hard to state precisely, because of the possibility that the contexts have abstractions over coercions. We would somehow need a premise that states that no coercion abstractions are "reachable" from τ, but defining such a property and then proving this claim seems not to pay its way.

[62] What I call congruence here has been called the *lifting lemma* in the literature.

$$\frac{\Sigma;\Gamma \vdash_{\mathsf{co}} \gamma : (\Pi a{:}_\rho \kappa_1.\, \sigma_1) \sim (\Pi a{:}_\rho \kappa_2.\, \sigma_2)}{\Sigma;\Gamma \vdash_{\mathsf{co}} \mathbf{argk}\, \gamma : \kappa_1 \sim \kappa_2} \quad \textsc{Co\_ArgK}$$

$$\frac{\Sigma;\Gamma \vdash_{\mathsf{co}} \gamma : (\Pi c{:}(\tau_1 \sim \tau_1').\, \sigma_1) \sim (\Pi c{:}(\tau_2 \sim \tau_2').\, \sigma_2)}{\Sigma;\Gamma \vdash_{\mathsf{co}} \mathbf{argk}_1\, \gamma : \tau_1 \sim \tau_2} \quad \textsc{Co\_CArgK1}$$

$$\frac{\Sigma;\Gamma \vdash_{\mathsf{co}} \gamma : (\Pi c{:}(\tau_1 \sim \tau_1').\, \sigma_1) \sim (\Pi c{:}(\tau_2 \sim \tau_2').\, \sigma_2)}{\Sigma;\Gamma \vdash_{\mathsf{co}} \mathbf{argk}_2\, \gamma : \tau_1' \sim \tau_2'} \quad \textsc{Co\_CArgK2}$$

$$\frac{\Sigma;\Gamma \vdash_{\mathsf{co}} \gamma : (\lambda a{:}_\rho \kappa_1.\, \sigma_1) \sim (\lambda a{:}_\rho \kappa_2.\, \sigma_2)}{\Sigma;\Gamma \vdash_{\mathsf{co}} \mathbf{argk}\, \gamma : \kappa_1 \sim \kappa_2} \quad \textsc{Co\_ArgKLam}$$

$$\frac{\Sigma;\Gamma \vdash_{\mathsf{co}} \gamma : (\lambda c{:}(\tau_1 \sim \tau_1').\, \sigma_1) \sim (\lambda c{:}(\tau_2 \sim \tau_2').\, \sigma_2)}{\Sigma;\Gamma \vdash_{\mathsf{co}} \mathbf{argk}_1\, \gamma : \tau_1 \sim \tau_2} \quad \textsc{Co\_CArgKLam1}$$

$$\frac{\Sigma;\Gamma \vdash_{\mathsf{co}} \gamma : (\lambda c{:}(\tau_1 \sim \tau_1').\, \sigma_1) \sim (\lambda c{:}(\tau_2 \sim \tau_2').\, \sigma_2)}{\Sigma;\Gamma \vdash_{\mathsf{co}} \mathbf{argk}_2\, \gamma : \tau_1' \sim \tau_2'} \quad \textsc{Co\_CArgKLam2}$$

Figure 5.10: The **argk** rules of coercion formation

analogously. The forms with $\mathbf{argk}_i$ are necessary because Pico has no built-in notion of an equality between equalities: If we tried to extract a relation between propositions like we do in Co_ArgK, we would need something that looks like $\phi_1 \sim \phi_2$, which does not exist in Pico. So, we have to extract either the left side of the propositions or the right side.

Note that these rules are syntax-directed even though their conclusions overlap: we can always find the proposition a coercion proves and then decide which **argk** rule to use.

5.8.6.2 The instantiation forms

Given a coercion between abstractions, we can instantiate the bound variable and get a coercion between the instantiated bodies. The rules for these coercions are in Figure 5.11 on the following page.

These rules are essentially concrete instances of two rule schemas, one for instantiation coercions built with @, and the other for "result" coercions built with **res**. The instantiation coercions can work with one of three argument types (relevant type, irrelevant type, and coercion) and one of two forms (Π and λ), leading to six very similar rules. Along the same lines, **res** coercions work with both Π and λ, though this form is agnostic to the argument flavor, so we get only two rules.

The instantiation coercions are essential in writing the push rules (Section 5.7.4)

$$\frac{\Sigma;\Gamma \vdash_{\mathsf{co}} \gamma : \Pi a{:}_{\mathsf{Rel}}\kappa_1.\,\sigma_1 \sim \Pi a{:}_{\mathsf{Rel}}\kappa_2.\,\sigma_2 \quad \Sigma;\Gamma \vdash_{\mathsf{co}} \eta : \tau_1 \ {}^{\kappa_1}\!\sim^{\kappa_2} \tau_2}{\Sigma;\Gamma \vdash_{\mathsf{co}} \gamma@\eta : \sigma_1[\tau_1/a] \sim \sigma_2[\tau_2/a]} \quad \textsc{Co\_InstRel}$$

$$\frac{\Sigma;\Gamma \vdash_{\mathsf{co}} \gamma : \Pi a{:}_{\mathsf{Irrel}}\kappa_1.\,\sigma_1 \sim \Pi a{:}_{\mathsf{Irrel}}\kappa_2.\,\sigma_2 \quad \Sigma;\Gamma \vdash_{\mathsf{co}} \eta : \tau_1 \ {}^{\kappa_1}\!\sim^{\kappa_2} \tau_2}{\Sigma;\Gamma \vdash_{\mathsf{co}} \gamma@\{\eta\} : \sigma_1[\tau_1/a] \sim \sigma_2[\tau_2/a]} \quad \textsc{Co\_InstIrrel}$$

$$\frac{\Sigma;\Gamma \vdash_{\mathsf{co}} \eta_1 : \Pi c{:}\phi_1.\,\sigma_1 \sim \Pi c{:}\phi_2.\,\sigma_2 \quad \Sigma;\Gamma \vdash_{\mathsf{co}} \gamma_1 : \phi_1 \quad \Sigma;\Gamma \vdash_{\mathsf{co}} \gamma_2 : \phi_2}{\Sigma;\Gamma \vdash_{\mathsf{co}} \eta_1@(\gamma_1,\gamma_2) : \sigma_1[\gamma_1/c] \sim \sigma_2[\gamma_2/c]} \quad \textsc{Co\_CInst}$$

$$\frac{\Sigma;\Gamma \vdash_{\mathsf{co}} \gamma : \lambda a{:}_{\mathsf{Rel}}\kappa_1.\,\tau_1 \sim \lambda a{:}_{\mathsf{Rel}}\kappa_2.\,\tau_2 \quad \Sigma;\Gamma \vdash_{\mathsf{co}} \eta : \sigma_1 \ {}^{\kappa_1}\!\sim^{\kappa_2} \sigma_2}{\Sigma;\Gamma \vdash_{\mathsf{co}} \gamma@\eta : \tau_1[\sigma_1/a] \sim \tau_2[\sigma_2/a]} \quad \textsc{Co\_InstLamRel}$$

$$\frac{\Sigma;\Gamma \vdash_{\mathsf{co}} \gamma : \lambda a{:}_{\mathsf{Irrel}}\kappa_1.\,\tau_1 \sim \lambda a{:}_{\mathsf{Irrel}}\kappa_2.\,\tau_2 \quad \Sigma;\Gamma \vdash_{\mathsf{co}} \eta : \sigma_1 \ {}^{\kappa_1}\!\sim^{\kappa_2} \sigma_2}{\Sigma;\Gamma \vdash_{\mathsf{co}} \gamma@\{\eta\} : \tau_1[\sigma_1/a] \sim \tau_2[\sigma_2/a]} \quad \textsc{Co\_InstLamIrrel}$$

$$\frac{\Sigma;\Gamma \vdash_{\mathsf{co}} \gamma : \lambda c{:}\phi_1.\,\sigma_1 \sim \lambda c{:}\phi_2.\,\sigma_2 \quad \Sigma;\Gamma \vdash_{\mathsf{co}} \eta_1 : \phi_1 \quad \Sigma;\Gamma \vdash_{\mathsf{co}} \eta_2 : \phi_2}{\Sigma;\Gamma \vdash_{\mathsf{co}} \gamma@(\eta_1,\eta_2) : \sigma_1[\eta_1/c] \sim \sigma_2[\eta_2/c]} \quad \textsc{Co\_CInstLam}$$

$$\frac{\Sigma;\Gamma \vdash_{\mathsf{co}} \gamma : \Pi\!\!\Pi\Delta_1.\,\tau_1 \sim \Pi\!\!\Pi\Delta_2.\,\tau_2 \quad |\Delta_1| = |\Delta_2| = n \quad \Sigma;\Gamma \vdash_{\mathsf{ty}} \tau_1 : \mathbf{Type} \quad \Sigma;\Gamma \vdash_{\mathsf{ty}} \tau_2 : \mathbf{Type}}{\Sigma;\Gamma \vdash_{\mathsf{co}} \mathbf{res}^n\,\gamma : \tau_1 \sim \tau_2} \quad \textsc{Co\_Res}$$

$$\frac{\Sigma;\Gamma \vdash_{\mathsf{co}} \gamma : \lambda\Delta_1.\,\tau_1 \sim \lambda\Delta_2.\,\tau_2 \quad |\Delta_1| = |\Delta_2| = n \quad \Sigma;\Gamma \vdash_{\mathsf{ty}} \tau_1 : \kappa_1 \quad \Sigma;\Gamma \vdash_{\mathsf{ty}} \tau_2 : \kappa_2}{\Sigma;\Gamma \vdash_{\mathsf{co}} \mathbf{res}^n\,\gamma : \tau_1 \sim \tau_2} \quad \textsc{Co\_ResLam}$$

Figure 5.11: Instantiation rules of coercion formation

of the operational semantics.[63]

The **res** coercions are a form of degenerate instantiation, usable when the body of an abstraction (either Π or λ) does not mention the bound variable(s). Note that both **res** rules require that the body types (τ_1 and τ_2) are well typed without any of the bound variables in Δ_1 or Δ_2. These coercions also allow for the possibility of looking through multiple binders. This ability cannot be emulated by repeated use of **res** because of the possibility of an intermediate dependency. For example, consider the reflexive coercion $\gamma = \langle \Pi(a{:}_{\mathsf{Irrel}}\mathbf{Type}), (b{:}_{\mathsf{Rel}}\, a).\, \mathbf{Type}\rangle$. We can see that $\mathbf{res}^2\,\gamma$ is well typed, even though $\mathbf{res}^1\,\gamma$ is not (because of the appearance of a in the type of b).

We must use **res** instead of instantiation when we don't have a coercion to use for the instantiation. This situation happens in the S_KPush rule, where we need a coercion relating the bodies of two propositionally equal Π-types, but we have no coercions to hand to use in instantiation. See Section 5.9 for more details.

5.8.6.3 Type constants are injective

In Pico, all type constants are considered injective, as witnessed by the **nth** coercions, which extract an equality between arguments of a type constant:

$$\frac{\begin{array}{l}\Sigma;\Gamma \vdash_{\mathsf{co}} \gamma : H_{\{\overline{\kappa}\}}\,\overline{\psi} \sim H_{\{\overline{\kappa}'\}}\,\overline{\psi}' \\ \psi_i = \tau \qquad \psi_i' = \sigma \\ \Sigma;\Gamma \vdash_{\mathsf{ty}} \tau : \kappa_1 \qquad \Sigma;\Gamma \vdash_{\mathsf{ty}} \sigma : \kappa_2\end{array}}{\Sigma;\Gamma \vdash_{\mathsf{co}} \mathbf{nth}_i\,\gamma : \tau \sim \sigma}\ \textsc{Co\_NthRel}$$

$$\frac{\begin{array}{l}\Sigma;\Gamma \vdash_{\mathsf{co}} \gamma : H_{\{\overline{\kappa}\}}\,\overline{\psi} \sim H_{\{\overline{\kappa}'\}}\,\overline{\psi}' \\ \psi_i = \{\tau\} \qquad \psi_i' = \{\sigma\} \\ \Sigma;\mathsf{Rel}(\Gamma) \vdash_{\mathsf{ty}} \tau : \kappa_1 \qquad \Sigma;\mathsf{Rel}(\Gamma) \vdash_{\mathsf{ty}} \sigma : \kappa_2\end{array}}{\Sigma;\Gamma \vdash_{\mathsf{co}} \mathbf{nth}_i\,\gamma : \tau \sim \sigma}\ \textsc{Co\_NthIrrel}$$

Both forms above require that we extract a coercion between *type* arguments, never *coercion* arguments. As discussed in Section 5.8.3, we never need an explicit proof of equality between coercions. The last line of premises in the rules are simply to produce the kinds to put in the result proposition, where the kinds are elided in the typesetting.

Injectivity of type constants is sometimes controversial [104] and is known to be anti-classical [47]. However, in a type system with **Type** : **Type**, being able to prove absurdity by combining type constant injectivity with, say, the Law of the Excluded Middle, does not weaken any property of the language. Injectivity is vital in the S_KPush rule and is thus a part of the language.

[63] It is necessary for the system to allow instantiation on Π-types; λ-types, on the other hand, are not strictly necessary to instantiate in order to prove type safety. However, doing so is easy, and so I took the opportunity to make the equality relation stronger.

$$\frac{\begin{array}{l}\Sigma; \Gamma \vdash_{\mathsf{co}} \gamma : \tau_1 \_ \psi_1 \sim \tau_2 \_ \psi_2 \\ \Sigma; \Gamma \vdash_{\mathsf{ty}} \tau_1 : \Pi \delta_1 . \kappa_1 \qquad \Sigma; \Gamma \vdash_{\mathsf{ty}} \tau_2 : \Pi \delta_2 . \kappa_2 \\ \Sigma; \Gamma \vdash_{\mathsf{co}} \eta : \Pi \delta_1 . \kappa_1 \sim \Pi \delta_2 . \kappa_2\end{array}}{\Sigma; \Gamma \vdash_{\mathsf{co}} \mathbf{left}_\eta \, \gamma : \tau_1 \sim \tau_2} \text{ Co\_Left}$$

$$\frac{\begin{array}{l}\Sigma; \Gamma \vdash_{\mathsf{co}} \gamma : \tau_1 \_ \sigma_1 \sim \tau_2 \_ \sigma_2 \\ \Sigma; \Gamma \vdash_{\mathsf{ty}} \sigma_1 : \kappa_1 \qquad \Sigma; \Gamma \vdash_{\mathsf{ty}} \sigma_2 : \kappa_2 \qquad \Sigma; \Gamma \vdash_{\mathsf{co}} \eta : \kappa_1 \sim \kappa_2\end{array}}{\Sigma; \Gamma \vdash_{\mathsf{co}} \mathbf{right}_\eta \, \gamma : \sigma_1 \sim \sigma_2} \text{ Co\_RightRel}$$

$$\frac{\begin{array}{l}\Sigma; \Gamma \vdash_{\mathsf{co}} \gamma : \tau_1 \_ \{\sigma_1\} \sim \tau_2 \_ \{\sigma_2\} \\ \Sigma; \Gamma \vdash_{\mathsf{ty}} \sigma_1 : \kappa_1 \qquad \Sigma; \Gamma \vdash_{\mathsf{ty}} \sigma_2 : \kappa_2 \qquad \Sigma; \Gamma \vdash_{\mathsf{co}} \eta : \kappa_1 \sim \kappa_2\end{array}}{\Sigma; \Gamma \vdash_{\mathsf{co}} \mathbf{right}_\eta \, \gamma : \sigma_1 \sim \sigma_2} \text{ Co\_RightIrrel}$$

Figure 5.12: Function application decomposition coercions

5.8.6.4 Matchable types are generative and injective

In Section 4.2.4, I define *matchable* as the conjunction of generative and injective. Pico includes two coercion forms that witness the generativity (**left**) and injectivity (**right**) of matchable function types, as shown in Figure 5.12. Note that the applications in the proposition proved by γ are matchable applications $\tau \_ \psi$, distinct from unmatchable applications $\tau \sim \psi$.

Interestingly, these coercions require an extra coercion η that proves that the kinds of the output types are equal. This kind coercion is necessary to prove the consistency of the **kind** coercion (Section 5.8.1). It is curiously absent from my prior work on kind equalities [105], but I now believe that this coercion is necessary—though I have yet to find a counterexample to consistency by omitting it, I am unable to prove consistency without it.

Does adding this extra argument to **left** and **right** now weaken Pico's expressiveness, compared to its predecessors? Yes and no:

Yes, fewer coercions are available, when comparing against the system in my prior work [105]. However, I argue in Section 5.10.5.2 that the proof in that prior work is broken, precisely around its **kind** coercion. If Pico reduces expressiveness compared to an unsound system, this may be an improvement.

No fewer coercions are available, when comparing against the System FC before kind equalities (that is, the System FC in GHC 7). Prior to GHC 8, the **left** and **right** coercions required the kinds of the output types to be identical. In those cases, the η coercion in Pico's **left** and **right** would just be reflexive. Though this restriction on the kinds was overlooked in the original publication

on System FC [87], it appears in later treatments [11, 32].[64]

I thus conclude that adding these extra kind coercions is appropriate, considering that their omission in GHC 8.0 may be unsafe and that including them is conservative with respect to GHC 7.

5.8.7 Equality includes β-reduction

The last rule to consider in the $\vdash_{\mathsf{co}}$ judgment is the one that witnesses β-reduction:

$$\frac{\begin{array}{ll}\Sigma;\Gamma \vdash_{\mathsf{ty}} \tau : \kappa & \Sigma;\Gamma \vdash_{\mathsf{ty}} \tau' : \kappa \\ \Sigma;\Gamma \vdash_{\mathsf{s}} \tau \longrightarrow \tau' & \end{array}}{\Sigma;\Gamma \vdash_{\mathsf{co}} \mathbf{step}\,\tau : \tau \sim \tau'} \quad \textsc{Co\_Step}$$

This rule is in place of having β-equivalence be part of definitional equality, as is done in some other dependently typed languages, such as Coq. Instead, in order to get a type to reduce, a Pico program must invoke the **step** coercion explicitly. Generating these coercions is quite painful to do by hand (as seen in the example in Section 5.5.3), but straightforward for a compiler.[65]

You will see that the rule requires both the redex and the reduct to be well kinded at kind κ. The requirement on the reduct is implied by the preservation theorem (Theorem C.46), but omitting it from the rule means that the proofs of proposition regularity (Lemma C.44) and preservation would have to be mutually inductive. It seems simpler just to add this extra, redundant premise.

5.8.8 Discussion

The coercion language in Pico is quite extensive, boasting (or suffering from, depending on your viewpoint) 37 separate typing rules. I consider here, briefly, why this is so.

There are several coercion forms (to wit, 10) that are absolutely essential for Pico to be proven type-safe and yet remain meaningful. These include the equivalence and coherence rules, assumptions, the Π-congruence form over type variables,[66] **argk** over Π, instantiation over Π, injectivity, and β-reduction. With the exception of assumptions (Co_Var) and β-reduction (Co_Step), these forms are all needed somewhere in the push rules (Section 5.7.4).[67] Assumptions and β-reduction, however, make Pico what

[64] The **left** and **right** coercions were omitted entirely from Yorgey et al. [107]. Correspondingly, they were dropped from the implementation in GHC 7.4. However, users found that this omission prevented some programs from being accepted. See GHC ticket #7205.

[65] If a type must reduce many times, it would be more efficient to support a $\mathbf{step}^n$ coercion form that performs n steps at once. Indeed, this is what I plan to implement. It is easier, however, to prove properties about single-step reduction.

[66] This form is needed only to support reduction under irrelevant λs.

[67] I am considering here a version of Pico without unsaturated matches. If we wish to include unsaturated matches, we would also need **res** over Π.

it is; the language would be near useless as a candidate for an internal dependently typed language without these.

The rest of the forms merely enrich the equality relation, while remaining inessential. I have decided to include them to make the equality relation relate more types. Doing so makes Pico—and, in turn, Dependent Haskell—more expressive. When adding rules, we must be careful that the new forms do not violate consistency (or other proved properties), so they are not entirely free. Perhaps there are more useful, safe rules one could add later, simply by updating the relevant proofs. Because Pico never inspects the structure of a coercion, adding new rules introduces only a minimal burden on any implementation—essentially just for bookkeeping. I thus leave open the possibility of more coercions as Pico gets used in practice.

5.9 The S_KPush rule

$$\frac{\begin{array}{l}\Sigma \vdash_{\mathsf{tc}} H : \overline{a}{:}_{\mathsf{Irrel}}\overline{\kappa}; \Delta; H' \qquad \Delta = \Delta_1, \Delta_2 \qquad n = |\Delta_2| \\ \kappa = \text{'}\Pi \overline{a}{:}_{\mathsf{Irrel}}\overline{\kappa}, \Delta.\, H'\, \overline{a} \\ \sigma = \text{'}\Pi(\Delta_2[\overline{\tau}/\overline{a}][\overline{\psi}/\mathsf{dom}(\Delta_1)]).\, H'\, \overline{\tau} \\ \sigma' = \text{'}\Pi(\Delta_2[\overline{\tau}'/\overline{a}][\overline{\psi}'/\mathsf{dom}(\Delta_1)]).\, H'\, \overline{\tau}' \\ \Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{co}} \eta : \sigma \sim \sigma' \\ \Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{vec}} \overline{\tau}' : \overline{a}{:}_{\mathsf{Rel}}\overline{\kappa} \\ \forall i,\ \gamma_i = \mathsf{build\_kpush\_co}(\langle\kappa\rangle @ (\mathbf{nths}\,(\mathbf{res}^n\,\eta)); \overline{\psi}_{1\ldots i-1}) \\ \forall i,\ \psi'_i = \mathsf{cast\_kpush\_arg}(\psi_i; \gamma_i) \\ H \to \kappa' \in \overline{alt}\end{array}}{\Sigma; \Gamma \vdash_{\mathsf{s}} \mathbf{case}_{\kappa_0}\,(H_{\{\overline{\tau}\}}\,\overline{\psi}) \rhd \eta\,\mathbf{of}\,\overline{alt} \longrightarrow \mathbf{case}_{\kappa_0}\,H_{\{\overline{\tau}'\}}\,\overline{\psi}'\,\mathbf{of}\,\overline{alt}}\ \text{S\_KPush}$$

The S_KPush rule handles the case where the scrutinee of a **case** expression is headed by a cast. As in all previous work on System FC, this push rule is the most intricate. However, in this dissertation, I have taken a new approach to S_KPush that does not require the so-called "lifting lemma" of previous work.[68] This lifting lemma is a generalization of the congruence property, which does not hold in Pico (Section 5.8.5.3). Instead, I rely on instantiating the type of a type constant, and on the fact that type constant types are always closed. As the computational content of the S_KPush rule must actually be implemented as part of a compiler that uses Pico, this (slightly) simpler statement of S_KPush may prove to be a measurable optimization in practice.

A few examples can demonstrate the general idea. Firstly, note that in S_KPush, only the scrutinee matters; the alternatives remain the same before and after the reduction. With that in mind, we can see scrutinees before and after pushing in Figure 5.13 on the following page.

[68] See for example, Weirich et al. [105], which contains a good, detailed explication of the lifting lemma.

Original scrutinee	Assumptions / Notes	
Pushed scrutinee		
$\mathit{True} \rhd \langle \mathit{Bool} \rangle$	simple case; no universals	(1)
True		
$\mathit{Just}_{\{\mathit{Int}\}}\, 3 \rhd \gamma$	$\Sigma; \Gamma \vdash_{\mathsf{co}} \gamma : \mathit{Maybe\ Int} \sim \mathit{Maybe}\ b$ $b{:}_{\mathsf{Irrel}}\mathbf{Type} \in \Gamma$	(2)
$\mathit{Just}_{\{b\}}\, (3 \rhd \mathbf{argk}\, (\langle \text{'}\Pi a{:}_{\mathsf{Irrel}}\mathbf{Type}, x{:}_{\mathsf{Rel}} a.\ \mathit{Maybe}\ a \rangle @ (\mathbf{nth}_1\, \gamma)))$		
$\mathit{MkG}_{\{\mathit{Bool}\}}\, \langle \mathit{Bool} \rangle \rhd \gamma$	$\Sigma; \Gamma \vdash_{\mathsf{co}} \gamma : G\ \mathit{Bool} \sim G\ b$ $b{:}_{\mathsf{Irrel}}\mathbf{Type} \in \Gamma$	(3)
$\mathit{MkG}_{\{b\}}\, (\mathbf{sym}\, (\mathbf{argk}_1\, \eta) \fatsemi \langle \mathit{Bool} \rangle \fatsemi \mathbf{argk}_2\, \eta)$, where $\eta = \langle \text{'}\Pi(a{:}_{\mathsf{Irrel}}\mathbf{Type}), (c{:}a \sim \mathit{Bool}).\ G\ a \rangle @ (\mathbf{nth}_1\, \gamma)$		
$(\mathit{Pack}_{\{\mathit{Bool}\}}\ \mathit{True}\ \mathit{MkP}_{\{\mathit{Bool}, \mathit{True}\}}) \rhd \gamma$	$\Sigma; \Gamma \vdash_{\mathsf{co}} \gamma : \text{'}\Pi\delta_1.\ \mathit{Ex\ Bool} \sim \text{'}\Pi\delta_2.\ \mathit{Ex}\ b$ $\delta_1 = y{:}_{\mathsf{Rel}}\mathit{Proxy\ Bool\ True}$ $\delta_2 = y{:}_{\mathsf{Rel}}\mathit{Proxy}\ b\, (\mathit{True} \rhd \gamma_2)$ $\Sigma; \Gamma \vdash_{\mathsf{co}} \gamma_2 : \mathit{Bool} \sim b$ $b{:}_{\mathsf{Irrel}}\mathbf{Type} \in \Gamma$	(4)
$\mathit{Pack}_{\{b\}}\, \{\mathit{True} \rhd \eta_0'\}\, (\mathit{MkP}_{\{\mathit{Bool}, \mathit{True}\}} \rhd \eta_1')$, where $\kappa = \text{'}\Pi(k{:}_{\mathsf{Irrel}}\mathbf{Type}), (a{:}_{\mathsf{Irrel}} k), (x{:}_{\mathsf{Rel}}\mathit{Proxy\ k\ a}), (y{:}_{\mathsf{Rel}}\mathit{Proxy\ k\ a}).\ \mathit{Ex\ k}$ $\eta_0 = \langle \kappa \rangle @ (\mathbf{nth}_1\, (\mathbf{res}^1\, \gamma))$ $\eta_0' = \mathbf{argk}\, \eta_0$ $\eta_1 = \eta_0 @ (\mathit{True} \approx_{\eta_0'} \mathit{True} \rhd \eta_0')$ $\eta_1' = \mathbf{argk}\, \eta_1$		

The reductions above assume the following datatypes. In Haskell:

```
data Bool = False | True
data Maybe a = Just a | Nothing
data G a where
  MkG :: G Bool
data Proxy (a :: k) = MkP
data Ex k where
  Pack :: ∀ (a :: k). Proxy a → Proxy a → Ex k
```

And in Pico:

$$\begin{aligned}\Sigma = {} & \mathit{Bool}{:}(\varnothing), \mathit{False}{:}(\varnothing; \mathit{Bool}), \mathit{True}{:}(\varnothing; \mathit{Bool}) \\ & \mathit{Maybe}{:}(a{:}\mathbf{Type}), \mathit{Just}{:}(x{:}_{\mathsf{Rel}} a; \mathit{Maybe}), \mathit{Nothing}{:}(\varnothing; \mathit{Maybe}) \\ & G{:}(a{:}\mathbf{Type}), \mathit{MkG}{:}(c{:}a \sim \mathit{Bool}; G) \\ & \mathit{Proxy}{:}(k : \mathbf{Type}, a : k), \mathit{MkP}{:}(\varnothing; \mathit{Proxy}) \\ & \mathit{Ex}{:}(k{:}\mathbf{Type}), \mathit{Pack}{:}(a{:}_{\mathsf{Irrel}} k, x{:}_{\mathsf{Rel}}\mathit{Proxy\ k\ a}, y{:}_{\mathsf{Rel}}\mathit{Proxy\ k\ a}; \mathit{Ex})\end{aligned}$$

Figure 5.13: Examples of S_KPush

$$
\begin{aligned}
\mathsf{build\_kpush\_co}(\gamma; \varnothing) &= \gamma \\
\mathsf{build\_kpush\_co}(\gamma; \overline{\psi}, \tau) &= \mathbf{let}\ c := \mathsf{build\_kpush\_co}(\gamma; \overline{\psi})\ \mathbf{in} \\
&\quad\ c@(\tau \approx_{\mathbf{argk}\, c} \tau \rhd \mathbf{argk}\, c) \\
\mathsf{build\_kpush\_co}(\gamma; \overline{\psi}, \{\tau\}) &= \mathbf{let}\ c := \mathsf{build\_kpush\_co}(\gamma; \overline{\psi})\ \mathbf{in} \\
&\quad\ c@\{\tau \approx_{\mathbf{argk}\, c} \tau \rhd \mathbf{argk}\, c\} \\
\mathsf{build\_kpush\_co}(\gamma; \overline{\psi}, \eta) &= \mathbf{let}\ c := \mathsf{build\_kpush\_co}(\gamma; \overline{\psi})\ \mathbf{in} \\
&\quad\ c@(\eta, \mathbf{sym}\,(\mathbf{argk}_1\, c) \fatsemi \eta \fatsemi \mathbf{argk}_2\, c)
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{cast\_kpush\_arg}(\tau; \gamma) &= \tau \rhd \mathbf{argk}\, \gamma \\
\mathsf{cast\_kpush\_arg}(\{\tau\}; \gamma) &= \{\tau \rhd \mathbf{argk}\, \gamma\} \\
\mathsf{cast\_kpush\_arg}(\gamma; \eta) &= \mathbf{sym}\,(\mathbf{argk}_1\, \eta) \fatsemi \gamma \fatsemi \mathbf{argk}_2\, \eta
\end{aligned}
$$

Figure 5.14: Helper functions implementing S_KPush

Example (1) In this example, there are no universals of the type in question (*Bool*), and so "pushing" is extraordinarily simple: just drop the coercion. We can see this in terms of S_KPush in that both $\overline{\tau}$ and $\overline{\psi}$ are empty. Note that if we had a non-reflexive coercion in the scrutinee—that is, if the scrutinee were, say, $\mathit{True} \rhd \gamma$ with $\Sigma; \Gamma \vdash_{\mathsf{co}} \gamma : \mathit{Bool} \sim a$—the **case** expression would not be well typed. Rule Ty_Case requires the type of a scrutinee to be of the form $\Pi\Delta.\, H\, \overline{\sigma}$. The type a does not have this form, and so such a scrutinee is disallowed. Also note that we cannot have $\mathit{True} \rhd \gamma$ with $\Sigma; \Gamma \vdash_{\mathsf{co}} \gamma : \mathit{Bool} \sim \mathit{Int}$ due to the consistency lemma (Section 5.10).

Example (2) This is the simplest non-trivial example. We need to push a coercion γ proving $\mathit{Maybe\ Int} \sim \mathit{Maybe\ b}$ into $\mathit{Just}_{\{\mathit{Int}\}}\, 3$. This coerced scrutinee has type *Maybe b*; the pushed scrutinee must have the same type. We thus know it must start with $\mathit{Just}_{\{b\}}$. The only challenge left is to cast the argument, 3, with a coercion that proves $\mathit{Int} \sim b$. We will always be able to extract this coercion from the coercion casting the scrutinee, γ. But how, in general?

The coercion needed to cast each (existential) argument to a constructor must surely depend on the type of the constructor. Previous versions of System FC did a transformation on this type to produce the coercion. In this work, I instantiate the type using the @ operator (Section 5.8.6.2) via the helper metatheory functions build_kpush_co and cast_kpush_arg, presented in Figure 5.14.

In the present case—pushing a coercion into $\mathit{Just} : \Pi a{:}_{\mathsf{Irrel}} \mathbf{Type}, x{:}_{\mathsf{Rel}} a.\ \mathit{Maybe}\ a$—we take *Just*'s type and instantiate a by the coercion $\mathbf{nth}_1\, \gamma$, which proves $\mathit{Int} \sim b$. We are thus left with a coercion that proves

$$(\Pi x{:}_{\mathsf{Rel}} \mathit{Int}.\ \mathit{Maybe\ Int}) \sim (\Pi x{:}_{\mathsf{Rel}} b.\ \mathit{Maybe}\ b).$$

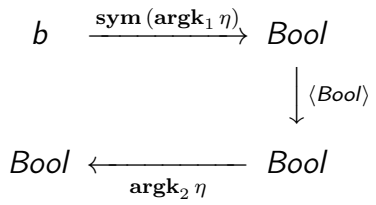

Figure 5.15: "Casting" a coercion in Example (3)

Then, all we have to do is use **argk** to extract the coercion proving $\mathit{Int} \sim b$ and we can use it to cast 3.

Seeing the above action in the definition for S_KPush may be challenging. Let's take another look, focusing on the metavariables in the definition of the rule (presented in Figure 5.7 on page 102). The type σ is the type of the underlying (uncoerced) scrutinee, and σ' is the type of the coerced scrutinee. In our example, we have $\sigma = \mathit{Maybe}\ \mathit{Int}$ and $\sigma' = \mathit{Maybe}\ b$. Note that neither of these are Π-types, and thus the telescope Δ_2 from the rule is empty, with $n = 0$. The κ metavariable in the rule is the type of *Just*, above. The coercion we are building is the one to cast the first argument, that is, γ_1. The second argument to build_kpush_co is a list of all previous existential arguments, but in our case, there are no previous arguments, so this list is empty. We thus have $\gamma_1 = \mathsf{build\_kpush\_co}(\langle\kappa\rangle@(\mathbf{nth}_1\,\gamma); \varnothing)$.[69] We can see from the definition of build_kpush_co that the function just returns its first argument when its second argument is empty, and so we get $\gamma_1 = \langle\kappa\rangle@(\mathbf{nth}_1\,\gamma)$ as desired. The use of cast_kpush_arg is to apply the right **argk** form (Section 5.8.6.1), depending on whether we are casting a type or "casting" a coercion.

We focus on understanding cast_kpush_arg on the next example.

Example (3) The datatype *G* is a simple-as-they-come GADT. In this example, we cast $\mathit{MkG} :: G\ \mathit{Bool}$ to have type $G\ b$ (for some type variable b). The action in S_KPush here is actually quite similar to the previous case, because *MkG* is quite similar to *Just*: both take one argument, whose type depends on the one universal parameter. The difference here is that *MkG*'s argument is a coercion, whereas *Just*'s is a type. We thus cannot use **argk** in exactly the same way as before, instead requiring $\mathbf{argk}_1$ and $\mathbf{argk}_2$, as diagrammed in Figure 5.15. In this example, two of the steps in the diagram are redundant, but they will not be, in general. It can be convenient to think of constructions such as this as "casting" a coercion—that is, taking the coercion $\langle\mathit{Bool}\rangle$ and changing it to connect b with *Bool*. Indeed, prior work [105] even used a special notation for this: $\gamma \rhd \eta_1 \sim \eta_2$, but I find it clearer to avoid the sugar.

[69] Technically, we should write $\mathbf{res}^0\,\gamma$, because the superscript in **res** coercions is part of the language, not the metatheory. However, a $\mathbf{res}^0$ coercion is a no-op, so I leave it out here for simplicity.

Example (4) Having warmed ourselves up on the simpler examples above, Example (4) demonstrates the full complexity of S_KPush, including dependent existential arguments and an unsaturated scrutinee. We'll take these complications one at a time.

Having dependent existentials motivates the intricacies of build_kpush_co. Since the pushed-in cast changes universal arguments (unless it's reflexive), we need to cast existential arguments that may be dependent on the universals. However, if a later existential argument is dependent upon an earlier one and we change the earlier one, we must also change that later one. In this example, the first existential argument (instantiated to *True*) depends on the universal argument (instantiated to *Bool*), and the second existential depends on the first. The first existential is cast by η'_0 and thus the second must be cast by η'_1, which essentially replaces the occurrence of *True* in the type of the applied *MkP* constructor with $\mathit{True} \rhd \eta'_0$, using a coherence coercion built with $\approx$. Indeed, this is the whole point of build_kpush_co—using coherence to alter the types of later existentials depending on earlier ones. Here is the critical correctness property of build_kpush_co:

Lemma (Correctness of build_kpush_co [Lemma C.45])**.**
Assume $\Sigma; \Gamma \vdash_{\mathsf{cev}} \overline{\psi} : \Delta[\overline{\tau}/\overline{a}]$*, and let* $\gamma_i = \mathsf{build\_kpush\_co}(\eta; \overline{\psi}_{1\ldots i-1})$ *and* $\psi'_i = \mathsf{cast\_kpush\_arg}(\psi_i; \gamma_i)$*. If* $\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{co}} \eta : (\,\Pi\Delta.\,\sigma)[\overline{\tau}/\overline{a}] \sim (\,\Pi\Delta.\,\sigma)[\overline{\tau}'/\overline{a}]$*, then:*

1. $\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{co}} \mathsf{build\_kpush\_co}(\eta; \overline{\psi}) : \sigma[\overline{\tau}/\overline{a}][\overline{\psi}/\mathsf{dom}(\Delta)] \sim \sigma[\overline{\tau}'/\overline{a}][\overline{\psi}'/\mathsf{dom}(\Delta)]$
2. $\Sigma; \Gamma \vdash_{\mathsf{cev}} \overline{\psi}' : \Delta[\overline{\tau}'/\overline{a}]$

This lemma is phrased in terms of $\vdash_{\mathsf{cev}}$; that relation includes the same elements as $\vdash_{\mathsf{vec}}$ but allows induction from right-to-left instead of the usual left-to-right. The η in the lemma statement relates the type of a constructor to itself, but with the universals instantiated with potentially different concrete arguments. These instantiations come directly from the coercion being pushed into the scrutinee, by way of **nth**. (Note that the Π quantifiers in the type of η above are not a consequence of the possibility of unsaturation; instead, these are the existentials of the data constructor.) The lemma concludes that the resulting coercion relates the instantiated coercion (that is, the one built by build_kpush_co) to itself, with substitutions for both the universals and some existentials. Along the way, it also asserts the validity of the cast existentials, via the $\vdash_{\mathsf{cev}}$ result.

The remaining detail of Example (4) is its unsaturation. This is handled more simply by a **res** coercion (Section 5.8.6.2), which looks through binders to relate the bodies of two abstract types. Indeed, S_KPush is the reason that the **res** coercion exists at all, though it is not a burden to support in the metatheory.

5.10 Metatheory: Consistency

Broadly speaking, the type safety proof proceeds along lines well established by prior work [11, 31, 106]. Indeed, the only challenge in proving the preservation theorem

$\boxed{\tau_1 \propto \tau_2}$ Type compatibility

$$\frac{\tau_1 \text{ is not a value}}{\tau_1 \propto \tau_2} \quad \text{C\_NonValue1}$$

$$\frac{\tau_2 \text{ is not a value}}{\tau_1 \propto \tau_2} \quad \text{C\_NonValue2}$$

$$\frac{}{H_{\{\overline{\tau}\}}\,\overline{\psi} \propto H_{\{\overline{\tau}'\}}\,\overline{\psi}'} \quad \text{C\_TyCon}$$

$$\frac{\tau \propto \tau'}{\Pi a{:}_\rho \kappa.\,\tau \propto \Pi a{:}_\rho \kappa'.\,\tau'} \quad \text{C\_PiTy}$$

$$\frac{}{\Pi c{:}\phi.\,\tau \propto \Pi c{:}\phi'.\,\tau'} \quad \text{C\_PiCo}$$

$$\frac{}{\lambda\delta.\,\tau \propto \lambda\delta'.\,\tau'} \quad \text{C\_Lam}$$

Figure 5.16: Type compatibility

is in dealing with S_KPush. The tricky bit is all in proving the correctness of build_kpush_co; see Section 5.9. Otherwise, the proof of preservation is as expected.

On the other hand, progress is a challenge, as it has been in previous proofs of type safety of System FC. We proceed, as before, by proving consistency and then using that to prove progress. (The definition for $\propto$ is in the next subsection.)

Lemma (Consistency [Lemma C.74])**.** *If* Γ *contains only irrelevant type variable bindings and* $\Sigma; \Gamma \vdash_{\mathsf{co}} \gamma : \tau_1 \sim \tau_2$ *then* $\tau_1 \propto \tau_2$*.*

We restrict Γ not to have any coercion variables bound. Otherwise, a coercion assumption might relate, say, *Int* and *Bool* and we would be unable to prove consistency. As consistency is needed only during the progress proof, this restriction does not pose a problem.

5.10.1 Compatibility

The statement of consistency depends on the $\tau_1 \propto \tau_2$ relation (pronounced "τ_1 is compatible with τ_2"), as given in Figure 5.16. The goal of compatibility is to relate any two values (as defined in Section 5.7.1) that have the same head; non-values are compatible with everything. Note, in particular, in C_TyCon, that we care only that the two H are the same. The universals ($\overline{\tau}/\overline{\tau}'$) and existentials ($\overline{\psi}/\overline{\psi}'$) are allowed to differ. The one exception to this general scheme is in the C_PiTy rule, where we require the bodies τ/τ' also to be compatible. This is necessary because irrelevant binders are erased, and we must thus be sure that any exposed types are also compatible.

Consistency is used in the progress proof mainly in order to establish the typing premises of the push rules (Section 5.7.4). A representative example is in the case when we are trying to show that an application $\tau_1\,\tau_2$ is either a value or can step (it is clearly not a coerced value; recall the statement of the progress theorem from Section 5.7). The induction hypothesis tells us that τ_1 is a value, a coerced value, or can step. If it can step, we are done by S_App_Cong. If τ_1 is a value, we can determine that it is a λ-abstraction and thus we can do β-reduction. The remaining case is when τ_1 is a coerced value $v \rhd \gamma$. We need to be able to show that γ relates two Π-types in order to use S_PushRel. The right-hand type must be a Π-type because it is the function in an application. But the only way we can show that the left-hand type is a Π-type is by appealing to consistency.

We know, at this point, that the type being coerced is a value; thus its type is also a value (Lemma C.76, also introduced in Section 5.7.1). At this point, now that we know that both types involved in the type of the coercion γ are values, compatibility becomes a much stronger definition, allowing us to conclude that if the types are compatible and if one is a Π-type, the other must surely also be a Π-type. Because we can rule out non-values in the places where we wish to invoke the consistency lemma, the flexibility around non-values does not get in our way.

5.10.2 The parallel rewrite relation

To prove consistency, I (following prior work) define a parallel rewrite relation, written $\tau_1 \rightsquigarrow \tau_2$, and show that this relation includes pairs of compatible types only. A small wrinkle with this definition is that the rewrite relation works over only types whose coercions have been erased, as per the $\lfloor\cdot\rfloor$ operation, initially introduced along with coherence coercions in Section 5.8.3. The operation, as you may recall, removes all casts from a type, and replaces coercion arguments with an uninformative $\bullet$. Stripping out casts and coercions is important in the rewrite relation; if the rewrite relation considered these features, the language would lose its coherence property. Going forward, I use a convention where all types written as being related by $\rightsquigarrow$ have had their coercions erased.

The rewrite relation $\rightsquigarrow$ appears in Figure 5.17 on the next page and Figure 5.18 on page 126. Following conventions in the rewriting literature, I write $\tau_1 \rightsquigarrow \tau_3 \leftsquigarrow \tau_2$ to mean that $\tau_1 \rightsquigarrow \tau_3$ and $\tau_2 \rightsquigarrow \tau_3$, and I write $\tau_1 \rightsquigarrow^* \tau_2$ to mean the reflexive, transitive closure of $\rightsquigarrow$.

Note the Beta rules, which work over only unmatchable applications τ_ψ. This fact allows us to conclude that matchable applications τ_ψ never undergo β-reduction, in turn allowing us to prove that the **left** and **right** coercions are sound.

$\boxed{\tau \rightsquigarrow \tau'}$ Type parallel reduction, over erased types

$$\frac{}{\tau \rightsquigarrow \tau} \quad \text{R\_Refl}$$

$$\frac{\overline{\tau} \rightsquigarrow \overline{\tau}'}{H_{\{\overline{\tau}\}} \rightsquigarrow H_{\{\overline{\tau}'\}}} \quad \text{R\_Con}$$

$$\frac{\tau \rightsquigarrow \tau' \qquad \sigma \rightsquigarrow \sigma'}{\tau\,\sigma \rightsquigarrow \tau'\,\sigma'} \quad \text{R\_AppRel}$$

$$\frac{\tau \rightsquigarrow \tau' \qquad \sigma \rightsquigarrow \sigma'}{\tau\,\{\sigma\} \rightsquigarrow \tau'\,\{\sigma'\}} \quad \text{R\_AppIrrel}$$

$$\frac{\tau \rightsquigarrow \tau'}{\tau\,\bullet \rightsquigarrow \tau'\,\bullet} \quad \text{R\_CApp}$$

$$\frac{\delta \rightsquigarrow \delta' \qquad \tau \rightsquigarrow \tau'}{\Pi\delta.\,\tau \rightsquigarrow \Pi\delta'.\,\tau'} \quad \text{R\_Pi}$$

$$\frac{\delta \rightsquigarrow \delta' \qquad \tau \rightsquigarrow \tau'}{\lambda\delta.\,\tau \rightsquigarrow \lambda\delta'.\,\tau'} \quad \text{R\_Lam}$$

$$\frac{\tau \rightsquigarrow \tau'}{\mathbf{fix}\,\tau \rightsquigarrow \mathbf{fix}\,\tau'} \quad \text{R\_Fix}$$

$$\frac{\tau \rightsquigarrow \tau'}{\mathbf{absurd}\,\bullet\,\tau \rightsquigarrow \mathbf{absurd}\,\bullet\,\tau'} \quad \text{R\_Absurd}$$

$$\frac{\kappa \rightsquigarrow \kappa' \qquad \tau \rightsquigarrow \tau' \qquad \overline{\sigma} \rightsquigarrow \overline{\sigma}'}{\mathbf{case}_{\kappa}\,\tau\,\mathbf{of}\,\overline{\pi \to \sigma} \rightsquigarrow \mathbf{case}_{\kappa'}\,\tau'\,\mathbf{of}\,\overline{\pi \to \sigma'}} \quad \text{R\_Case}$$

$$\frac{\tau_1 \rightsquigarrow \tau_1' \qquad \tau_2 \rightsquigarrow \tau_2'}{(\lambda a{:}_{\mathsf{Rel}}\kappa.\,\tau_1)\_\tau_2 \rightsquigarrow \tau_1'[\tau_2'/a]} \quad \text{R\_BetaRel}$$

$$\frac{\tau_1 \rightsquigarrow \tau_1' \qquad \tau_2 \rightsquigarrow \tau_2'}{(\lambda a{:}_{\mathsf{Irrel}}\kappa.\,\tau_1)\_\{\tau_2\} \rightsquigarrow \tau_1'[\tau_2'/a]} \quad \text{R\_BetaIrrel}$$

$$\frac{\tau \rightsquigarrow \tau'}{(\lambda\bullet{:}\phi.\,\tau)\_\bullet \rightsquigarrow \tau'} \quad \text{R\_CBeta}$$

$$\frac{alt_i \;=\; H \to \tau_0 \qquad \overline{\psi} \rightsquigarrow \overline{\psi}' \qquad \tau_0 \rightsquigarrow \tau_0'}{\mathbf{case}_{\kappa}\,H_{\{\overline{\tau}\}}\,\overline{\psi}\,\mathbf{of}\,\overline{alt} \rightsquigarrow \tau_0'\,\overline{\psi}'\,\bullet} \quad \text{R\_Match}$$

$$\frac{alt_i \;=\; \_ \to \sigma \qquad \text{no alternative in } \overline{alt} \text{ matches } H \qquad \sigma \rightsquigarrow \sigma'}{\mathbf{case}_{\kappa}\,H_{\{\overline{\tau}\}}\,\overline{\psi}\,\mathbf{of}\,\overline{alt} \rightsquigarrow \sigma'} \quad \text{R\_Default}$$

$$\frac{\sigma \rightsquigarrow \sigma' \qquad \kappa \rightsquigarrow \kappa'}{\mathbf{fix}\,(\lambda a{:}_{\mathsf{Rel}}\kappa.\,\sigma) \rightsquigarrow \sigma'[\mathbf{fix}\,(\lambda a{:}_{\mathsf{Rel}}\kappa'.\,\sigma')/a]} \quad \text{R\_Unroll}$$

Figure 5.17: Parallel reduction over erased types

$\boxed{\delta \rightsquigarrow \delta'}$ Parallel reduction of binders

$$\frac{\kappa \rightsquigarrow \kappa'}{a{:}_\rho \kappa \rightsquigarrow a{:}_\rho \kappa'} \quad \text{R\_TyBinder}$$

$$\frac{\tau \rightsquigarrow \tau' \qquad \kappa_1 \rightsquigarrow \kappa_1' \qquad \kappa_2 \rightsquigarrow \kappa_2' \qquad \sigma \rightsquigarrow \sigma'}{\bullet{:}\tau \ {}^{\kappa_1}\!\sim^{\kappa_2} \sigma \rightsquigarrow \bullet{:}\tau' \ {}^{\kappa_1'}\!\sim^{\kappa_2'} \sigma'} \quad \text{R\_CoBinder}$$

$\boxed{\gamma \rightsquigarrow \gamma'}$ "Reduction" of erased coercion

$$\frac{}{\bullet \rightsquigarrow \bullet} \quad \text{R\_ErasedCo}$$

Figure 5.18: Parallel reduction auxiliary relations

5.10.2.1 Substitution

The relation $\rightsquigarrow$ is almost a non-deterministic, strong version of normal reduction ($\Sigma; \Gamma \vdash_{\mathsf{s}} \tau \longrightarrow \tau'$). In all the congruence forms (toward the top of Figure 5.17 on the previous page), the relation definition recurs in every component, as necessary to support the following lemma:

Lemma (Parallel reduction substitution in parallel [Lemma C.51])**.** *Assume* $\overline{\psi} \rightsquigarrow \overline{\psi}'$.

1. *If* $\tau_1 \rightsquigarrow \tau_2$*, then* $\tau_1[\overline{\psi}/\overline{z}] \rightsquigarrow \tau_2[\overline{\psi}'/\overline{z}]$.
2. *If* $\delta_1 \rightsquigarrow \delta_2$*, then* $\delta_1[\overline{\psi}/\overline{z}] \rightsquigarrow \delta_2[\overline{\psi}'/\overline{z}]$.

Note that all of the reductions are single-step.

Beyond the congruence rules, the rewrite relation includes parallel variants of the reduction rules from the normal step relation, toward the bottom of the figure. Note that these allow the components of a type to step as the reduction happens, as required for the local diamond lemma needed to prove confluence.

5.10.2.2 Confluence

This reduction relation is confluent (that is, has the Church-Rosser property). I prove this by proving a local diamond lemma:

Lemma (Local diamond [Lemma C.54])**.**

1. *If* $\tau_0 \rightsquigarrow \tau_1$ *and* $\tau_0 \rightsquigarrow \tau_2$*, then there exists* τ_3 *such that* $\tau_1 \rightsquigarrow \tau_3 \leftsquigarrow \tau_2$.
2. *If* $\delta_0 \rightsquigarrow \delta_1$ *and* $\delta_0 \rightsquigarrow \delta_2$*, then there exists* δ_3 *such that* $\delta_1 \rightsquigarrow \delta_3 \leftsquigarrow \delta_2$.

The proof of this lemma reasons by induction on the structure of τ_0/δ_0 and makes heavy use of the substitution lemma above. It is not otherwise challenging. The local diamond lemma implies confluence.

5.10.3 Completeness of the rewrite relation

Having written a confluent rewrite relation, we must also connect this relation to our equality relation. This is done via the following lemma:

Lemma (Completeness of type reduction [Lemma C.62]). *If* $\Sigma; \Gamma \vdash_{\mathsf{co}} \gamma : \tau_1 \sim^{\kappa_1}{}^{\kappa_2} \tau_2$ *and* $c \tilde{\#} \gamma$ *for every* $c{:}\phi \in \Gamma$*, then:*

1. *There exists some erased type* ϵ *such that* $\lfloor \tau_1 \rfloor \rightsquigarrow^* \epsilon \; {}^*\!\!\leftsquigarrow \lfloor \tau_2 \rfloor$.
2. *There exists some erased type* ϵ *such that* $\lfloor \kappa_1 \rfloor \rightsquigarrow^* \epsilon \; {}^*\!\!\leftsquigarrow \lfloor \kappa_2 \rfloor$.

Both the statement and proof of this lemma are rather more challenging than the previous ones. The proof proceeds by induction on the typing derivation. It is necessary in the proof to use the induction hypothesis on a premise where the context Γ is extended with a coercion variable (say, in the case for Co_PiCo). Thus, even though we will only use this lemma in a context with no coercion variables, we must strengthen the induction hypothesis to allow for coercion variables. Critically, though, we restrict how all coercion variables in the context can appear in γ, according to the definition of $\tilde{\#}$, introduced in Section 5.8.5.2. This restriction allows us to skip the impossible Co_Var case while still allowing induction in the Co_PiCo case.

The definition of $c \tilde{\#} \gamma$ allows c to appear in the types related by a coherence $\approx$ coercion. Happily, in the Co_Coherence case (when proving clause 1 of the lemma), we do not need to use the induction hypothesis, as a premise of Co_Coherence states that the erased types are, in fact, already equal. It is for precisely this reason that $c \tilde{\#} \gamma$ can allow c in the types in a coherence coercion.

We also see that the statement of the completeness lemma requires us to prove both that the types are joinable under $\rightsquigarrow$ and also that the kinds are. Otherwise, there would be no way to handle the **kind** case.

Having strengthened the induction hypothesis appropriately, the actual proof is not too hard. The case for transitivity uses confluence—this is the only place confluence is used. The decomposition forms use the fact that when a value type reduces under $\rightsquigarrow$, the reduct has to have the same shape as the redex, with individual components in the redex reducing to those same components in the reduct. To deal with **step**, we must consider the different possibilities given by the $\Sigma; \Gamma \vdash_{\mathsf{s}} \tau \longrightarrow \tau'$ relation. The proper reduction rules all have analogues in $\rightsquigarrow$, the congruence rules all follow from the induction hypothesis, and the push rules cause no change to a type with its coercions erased. To prove that the kinds are joinable, we must rely heavily on the deterministic nature of the typing relation, but there are no other undue complications.

5.10.4 From completeness to consistency

Having established the relationship between $\Sigma; \Gamma \vdash_{\mathsf{co}} \gamma : \phi$ and joinability with respect to the rewrite relation, we must only show that the rewrite relation relates compatible types. Here are the key lemmas:

Lemma (Joinable types are consistent [Lemma C.72]). *If* $\epsilon_1 \rightsquigarrow^* \epsilon_3 \ {}^*\!\!\leftsquigarrow \epsilon_2$*, then* $\epsilon_1 \propto \epsilon_2$.

Lemma (Erasure/consistency [Lemma C.73]). *If* $\lfloor \tau_1 \rfloor \propto \lfloor \tau_2 \rfloor$*, then* $\tau_1 \propto \tau_2$.

Other than some care needed around irrelevant abstractions (which cause recursion in the rules defining $\propto$), these lemmas are not hard to prove.

With all the groundwork laid, we can now conclude our consistency lemma, stated near the top of this section.

5.10.5 Related consistency proofs

There are a few aspects of the consistency proof where it may be helpful to highlight the differences between my proof here and those in prior work. The comments below dispute other, published proofs of consistency. The authors of these proofs have conceded to me in private communication that their proofs were incorrect and do not disagree with my assertions here.

5.10.5.1 Non-linear, non-terminating rewrite systems are not confluent

As described in some detail by Eisenberg et al. [32], non-terminating rewrite systems with non-linear left-hand sides are not confluent. We can easily see that the rewrite relation $\rightsquigarrow$ is not terminating. In this presentation, however, its "left-hand side" is linear. Breaking from previous work, I have phrased type families in Pico as λ-expressions that use **case**; thus the parallel to rewrite systems is not as apparent as in previous work. In the context of my work here, a non-linear left-hand side would look like a primitive equality check, as further explored in Section 5.13.2. Because the formalization of Pico that I am presenting does not contain this equality operator, I avoid the non-confluence problem described by Eisenberg et al. [32].

Nevertheless, promising new work in the term-rewriting community [50] suggests that there is a way to prove consistency without confluence even after adding an equality check. I leave it as future work to reconcile the approach here with the recent result cited above.

5.10.5.2 The proof of consistency by Weirich et al. [105] is wrong

The type system presented in my prior work [105] is very similar to Pico, although without dependency. Its treatment of Co_PiCo is subtly different, however. Although there are numerous changes in how the syntax is structured, that work effectively loosens the definition of $c \mathbin{\tilde{\#}} \gamma$ to allow c anywhere in a coherence coercion $(\tau_1 \approx_\eta \tau_2)$. In contrast, Pico allows c only in τ_1 or τ_2, but not in η. When armed with the **kind** coercion (identical in Pico to the version in the previous work), this allows us to violate a key lemma used to prove consistency. Here is the counterexample coercion, translated into Pico:

$$\gamma \;=\; \underset{\sim}{\Pi} c{:}(\langle \mathsf{Int} \rangle, \langle \mathsf{Bool} \rangle).\, \mathbf{kind}\,(3 \approx_c (3 \rhd c))$$

In the body of the abstraction, the coercion variable c has type $\mathsf{Int} \sim \mathsf{Bool}$. We can use a coherence coercion to relate 3 and $3 \rhd c$; their kinds are also related by c. We can then extract the kinds of the types related by the coherence coercion. Putting it all together yields this fact:

$$\Sigma; \varnothing \vdash_{\mathsf{co}} \gamma : (\underset{\sim}{\Pi} c{:}\mathsf{Int} \sim \mathsf{Bool}.\, \mathsf{Int}) \sim (\underset{\sim}{\Pi} c{:}\mathsf{Int} \sim \mathsf{Bool}.\, \mathsf{Bool})$$

The problem is that we can see that no rewrite relation will join the two types related by γ. Because the prior work's type system permits γ, its consistency proof must be wrong. (PICO rules out γ for using c in an illegal spot—the kind coercion in the subscript for $\approx$.) Note that the language in that work might indeed be consistent (I have no counterexample to consistency), but its consistency surely cannot be proved via the use of a rewrite relation in the way presented in that paper.

5.10.5.3 A one-variable version of Co_PiTy simplifies the consistency proof

Weirich et al.'s language differs along a different dimension, using three binders instead of one in its version of Co_PiTy. (See discussion in Section 5.8.5.1.) Apart from the awkwardness of needing extra variable names, the three-binder approach poses another problem: it introduces a coercion variable into the context. Unlike for their Co_PiCo, Weirich et al. do not introduce a coercion variable restriction for this coercion variable, as it is always a proof of equality between two variables. This extra coercion variable cannot imperil consistency. To prove this in the consistency proof, Weirich et al. employ a notion of "Good" contexts, which must be threaded through their proofs. My one-variable version, with no bound coercion variable, avoids this complication.

5.10.5.4 The proof of consistency by Gundry [37] is wrong

Gundry, in his thesis, takes a very different approach to proving consistency of his *evidence* language, also closely related to PICO. He sets up, essentially, a step-indexed logical relation and uses it to consider only closed coercions; when, say, a coercion variable is added to the context, Gundry quantifies over all possible closing substitutions.

A key property of Gundry's logical relation is transitivity. Yet, in his proof of transitivity, the indices do not work out. Gundry was not able to spot a straightforward solution, and in unpublished work, Weirich also tackled this problem and failed. Neither Gundry nor Weirich (nor I) have a proof that the step-indexed logical relation approach is not able to work, but no one has been able to finish the proof, either.

The failure of this approach is disappointing, because Gundry's *evidence* language

does not have the coercion variable restriction inherent in PICO's CO_PICO rule. Gundry's language thus allows more coercions than does PICO.

Can a System-FC-like language be proven consistent without a coercion variable restriction on its analogue of CO_PICO? My personal belief is "yes"—given that I believe such a language is, in fact, consistent—but researchers have yet to show it.

5.11 Metatheory: Type erasure

A critical property of any intermediate language used to compile Haskell is its ability to support type erasure. Haskell takes pride in erasing all of its complicated, helpful types before runtime, and the intermediate language must show that this is possible. PICO achieves this goal through its relevance annotations, where irrelevant abstractions and applications can be erased. In previous, non-dependent intermediate languages for Haskell, irrelevant abstractions and applications were also erased, but these were easier to spot, as they dealt with types instead of terms. In PICO, types and terms are indistinguishable, so we are required to use relevance annotations.

I prove the type erasure property via defining an untyped λ-calculus with an operational semantics, defining an erasure operation that translates from PICO to the untyped calculus, and proving a simulation property between the two languages.

5.11.1 The untyped λ-calculus

The definition of our erased calculus appears in Figure 5.19 on the following page. It is an untyped λ-calculus with datatypes (allowing for default patterns) and **fix**. The language also contains two fixed constants, $\text{'}\Pi$ and $\underset{\sim}{\Pi}$, here only to have something for Π-types to erase to.

The calculus also supports "coercion abstraction" via its $\lambda\bullet.e$ and $e\,\bullet$ forms. The existence of these forms mean that coercion abstractions are not fully erased. We can see why this must be so in the following example: let $\tau = \lambda c{:}\mathsf{Int} \sim \mathsf{Bool}.\,\mathsf{not}\,(3 \rhd c)$. The type τ is a valid PICO type. We do not have to worry about the nonsense in the body of the abstraction because consistency guarantees that we will never be able to apply τ to a (closed) coercion. As an abstraction, τ is a value and a normal form. However, if our type erasure operation dropped coercion abstractions, then disaster would strike. The erased expression would be $\mathsf{not}\,3$, which is surely stuck. We thus retain coercion abstractions and applications, while dropping the coercions themselves by rewriting all coercions with the uninformative $\bullet$.

What has now happened to our claim of type erasure? Coercions exist only to alter types, so have we kept some meddlesome vestige of types around? In a sense, yes, we have kept some type information around until runtime. However, two critical facts mean that this retention does not cause harm:

- Coercion applications contain no information, and therefore can be represented by precisely 0 bits. Indeed, this is how coercions are currently compiled in GHC,

Grammar:

$$
\begin{array}{rcll}
e & ::= & a \mid H \mid e\ y \mid \Pi \mid \mathbf{case}\ e\ \mathbf{of}\ \overline{ealt} \mid \lambda a.e \mid \lambda\bullet.e \mid \mathbf{fix}\ e & \text{expression} \\
y & ::= & e \mid \bullet & \text{argument} \\
ealt & ::= & \pi \to e & \text{case alternative}
\end{array}
$$

$\boxed{e \longrightarrow e'}$ Single-step operational semantics of expressions

$$\frac{}{(\lambda a.e_1)\ e_2 \longrightarrow e_1[e_2/a]}\ \text{E\_Beta}$$

$$\frac{}{(\lambda\bullet.e)\ \bullet \longrightarrow e}\ \text{E\_CBeta}$$

$$\frac{ealt_i\ =\ H \to e}{\mathbf{case}\ H\ \overline{y}\ \mathbf{of}\ \overline{ealt} \longrightarrow e\ \overline{y}\ \bullet}\ \text{E\_Match}$$

$$\frac{ealt_i\ =\ \_ \to e \qquad \text{no alternative in } \overline{ealt} \text{ matches } H}{\mathbf{case}\ H\ \overline{y}\ \mathbf{of}\ \overline{ealt} \longrightarrow e}\ \text{E\_Default}$$

$$\frac{}{\mathbf{fix}\ (\lambda a.e) \longrightarrow e[\mathbf{fix}\ (\lambda a.e)/a]}\ \text{E\_Unroll}$$

$$\frac{e \longrightarrow e'}{e\ y \longrightarrow e'\ y}\ \text{E\_App\_Cong}$$

$$\frac{e \longrightarrow e'}{\mathbf{case}\ e\ \mathbf{of}\ \overline{ealt} \longrightarrow \mathbf{case}\ e'\ \mathbf{of}\ \overline{ealt}}\ \text{E\_Case\_Cong}$$

$$\frac{e \longrightarrow e'}{\mathbf{fix}\ e \longrightarrow \mathbf{fix}\ e'}\ \text{E\_Fix\_Cong}$$

Erasure operation, $e\ =\ \llbracket\tau\rrbracket$:

$$
\begin{array}{rcl}
\llbracket a\rrbracket & = & a \\
\llbracket H_{\{\overline{\tau}\}}\rrbracket & = & H \\
\llbracket\tau_1\ \tau_2\rrbracket & = & \llbracket\tau_1\rrbracket\ \llbracket\tau_2\rrbracket \\
\llbracket\tau_1\ \{\tau_2\}\rrbracket & = & \llbracket\tau_1\rrbracket \\
\llbracket\tau_1\ \gamma\rrbracket & = & \llbracket\tau_1\rrbracket\ \bullet \\
\llbracket\Pi\delta.\ \tau\rrbracket & = & \Pi \\
\llbracket\tau \rhd \gamma\rrbracket & = & \llbracket\tau\rrbracket
\end{array}
\qquad
\begin{array}{rcl}
\llbracket\mathbf{case}_\kappa\ \tau\ \mathbf{of}\ \overline{alt}\rrbracket & = & \mathbf{case}\ \llbracket\tau\rrbracket\ \mathbf{of}\ \llbracket\overline{alt}\rrbracket \\
\llbracket\lambda a{:}_{\mathsf{Rel}}\kappa.\ \tau\rrbracket & = & \lambda a.\llbracket\tau\rrbracket \\
\llbracket\lambda a{:}_{\mathsf{Irrel}}\kappa.\ \tau\rrbracket & = & \llbracket\tau\rrbracket \\
\llbracket\lambda c{:}\phi.\ \tau\rrbracket & = & \lambda\bullet.\llbracket\tau\rrbracket \\
\llbracket\mathbf{fix}\ \tau\rrbracket & = & \mathbf{fix}\ \llbracket\tau\rrbracket \\
\llbracket\mathbf{absurd}\ \gamma\ \tau\rrbracket & = & \Pi \\
\llbracket\pi \to \tau\rrbracket & = & \pi \to \llbracket\tau\rrbracket
\end{array}
$$

Figure 5.19: The type-erased λ-calculus

by using an unboxed representation that is 0 bits wide. Thus, no memory is taken up at runtime.

- The coercion abstractions are not, in fact, meddlesome. The way in which coercion abstractions could cause harm at runtime is by causing a program to be a value when the user is not expecting it. For example, if a compiler translated the Haskell program $1 + 2$ into the expression $\lambda\bullet.1 + 2$, then we would never get 3. I thus make this claim: no Haskell program ever evaluates to a coercion abstraction. This claim is properly a property of the type inference / elaboration algorithm and so is deferred until Section 6.10.2.

One may wonder why PICO needs coercion abstractions at all. I can provide two reasons: to preserve the simplified treatment of **case** that does not bind variables, and in order to enable floating. An optimizer may decide to common up two branches of a **case** expression (i.e., float the branches out), both of which bind the same coercion variable. If there were no coercion abstraction form, this would be impossible. It is a correctness property of the optimizer (well beyond the scope of this dissertation) to make sure that the floated coercion abstraction does not halt evaluation prematurely.

5.11.2 Simulation

Here is the simulation property we seek:

Theorem (Type erasure [Theorem C.83]). *If* $\Sigma; \Gamma \vdash_{\mathsf{s}} \tau \longrightarrow \tau'$*, then either* $[\![\tau]\!] \longrightarrow [\![\tau']\!]$ *or* $[\![\tau]\!] = [\![\tau']\!]$.

Note that the untyped language might step once or not at all. For example, when PICO steps by a push rule, the untyped language does not step. The proof of this theorem is very straightforward.

5.11.3 Types do not prevent evaluation

Proving only that the erased calculus simulates PICO is not quite enough, as it still might be possible that an expression in the erased calculus can step even though the PICO type from which it was derived is a normal form. The property we need is embodied in this theorem:

Theorem (Types do not prevent evaluation [Theorem C.86]). *Suppose* $\Sigma; \Gamma \vdash_{\mathsf{ty}} \tau : \kappa$ *and* Γ *has only irrelevant variable bindings. If* $[\![\tau]\!] \longrightarrow e'$*, then* $\Sigma; \Gamma \vdash_{\mathsf{s}} \tau \longrightarrow \tau'$ *and either* $[\![\tau']\!] = e'$ *or* $[\![\tau']\!] = [\![\tau]\!]$.

This theorem would be false if PICO did not step under irrelevant binders, for example.

The proof depends on both the progress theorem and the type erasure (simulation) theorem above, as well as this key lemma:

Lemma (Expression redexes [Lemma C.84])**.** *If $\llbracket \tau \rrbracket$ is not an expression value, then τ is neither a value nor a coerced value.*

This lemma is straightforward to prove inductively on the structure of τ, and then the proof of the theorem above simply stitches together the pieces.

5.12 Design decisions

In the course of designing Pico, I have had to make quite a number of design decisions. Some of these are forced by external constraints (such as the need for two Π-forms), but others have been relatively free choices. In this section, I revisit some of these decisions and try to motivate why I have built Pico in the way that I have. It is my hope that this section will empower readers who wish to extend or alter Pico to understand its design better.

5.12.1 Coercions are not types

One alternative I considered was to make a coercion γ a possible production of a type τ. This would allow, for example, the form $\tau_1\,\tau_2$ to encompass both type application and coercion application. Going down this route, propositions ϕ would also have to become kinds κ, and we would have a rule such as

$$\frac{\Sigma; \Gamma \vdash_{\mathsf{co}} \gamma : \phi}{\Sigma; \Gamma \vdash_{\mathsf{ty}} \gamma : \phi} \quad \textsc{Ty\_Coercion}$$

This alternative design does not cause trouble with type safety, because we are injecting the safe coercions into the unsafe types. The other way around—injecting potentially non-terminating types into coercions—would lead to chaos.

This injection would simplify aspects of the grammar and rules. For example, the $\mathbf{argk}_1$ and $\mathbf{argk}_2$ coercions could be rewritten in terms of **argk** and **nth**.

In the end, I decided against this design because it simply moves the complexity around. Instead of the syntactic complexity inherent in Pico's actual design, this injection would cause complexity in needing to rule out the presence of coercions in various places where they would not appear. For example, the scrutinee of a **case** can never be a coercion, and there is no good way to define what $\llbracket \gamma \rrbracket$ should be. The design I chose adds a little syntactic overhead to avoid these thorny proof obligations, and that seems to be a win.

5.12.2 Putting braces around irrelevant arguments

A similar design decision was to put braces around irrelevant arguments. The syntactic distinction between relevant arguments and irrelevant ones is not necessary for syntax-directedness, because we can always look up the type of the function to see whether

we should consider the type application to be relevant or irrelevant. Yet putting this distinction directly in the syntax makes certain parts of the metatheory cleaner, when relevant and irrelevant applications are treated separately. Marking relevance in the syntax also allows us to define an erasure operation that is not type-directed.

5.12.3 Including types' kinds in propositions

Given that we can always extract a type's kind from the type, why is it necessary to mark all propositions with the types' kinds, as in $\tau_1 \sim^{\kappa_1}_{\kappa_2} \tau_2$? (Recall that all propositions in Pico are so marked, even though the kinds are frequently elided in the typesetting.) Once again, having details present directly in the syntax of propositions is more convenient than having those details implicit in the kinds of types. In this case, the kinds are necessary when defining $\mathbf{argk}_1$ and $\mathbf{argk}_2$. When proving the completeness of the rewrite relation (Section 5.10.3), we must be able to show that the kinds of the two types related by a coercion are joinable. Without having the kinds in the types erased of coercions (that is, in the output of $\lfloor \cdot \rfloor$), this is not provable.

An alternative here would be to have the erased language maintain the kinds but to omit them from Pico proper, but that makes erasure type-directed and more challenging. It seems simpler (and rather less error-prone) once again to make the syntax more ornate and the proofs shorter.

5.13 Extensions

I conclude this chapter by considering several extensions one might want to make to Pico to support a few more features of Haskell.

5.13.1 let

Haskell allows binding variables with **let**, and it would be convenient to do so in Pico as well. We shall consider the non-recursive case first and then move on to the complexities of **letrec**. Below, flouting Haskell convention, I use **let** to refer exclusively to the non-recursive case and use **letrec** when considering recursive bindings.

Non-recursive **let** would be very easy to incorporate. At first blush, we could consider **let** as a derived form, much as described in the literature [77, Section 11.5], replacing $\mathbf{let}\,(x : \kappa) := \tau\,\mathbf{in}\,\sigma$ with $(\lambda x{:}_{\mathsf{Rel}}\kappa.\,\sigma)\,\tau$. However, doing so would make optimizations harder: with the explicit **let** form, the optimizer can know the value of x in σ; this connection is lost with the applied λ-expression. Nevertheless, adding **let** as a new proper type form would be straightforward. We could additionally incorporate the ability to bind a coercion variable proving that, say, $x \sim \tau$ in σ. We would also add a new rule to the operational semantics expanding out all **let** definitions directly; an implementation may wish to optimize this, however. The only real challenge we would run into is adding a congruence coercion for **let**, which would share the complications

of the other binding forms (see Section 5.8.5.1). The designer of this extension could choose, however, to omit the congruence coercion for **let**, as the coercion is not strictly necessary.

Recursive **letrec** has all of the complexities above, along with the challenge of being recursive. In an expression such as $\mathbf{letrec}\,(x : \kappa) := \tau\,\mathbf{in}\,\sigma$, we would not be able to bind a coercion variable witnessing the equality between x and τ, as that would bring us into the realm of very dependent types [42]. Even ignoring that complication, we may also wish to consider the operational semantics of **letrec**. To my surprise, I am unable to find a published account of an operational semantics that deals with **letrec**, other than my own unproven version [26]. I can imagine rewriting a **letrec** to a form where each recursive occurrence of a variable is replaced with a copy of the entire **letrec**. I believe this would hold together, though I have not worked out the details. I do not wish to begin to imagine what a congruence coercion for **letrec** would look like.

Despite these challenges, I do think an implemented version of Pico could accommodate a primitive **letrec** rather easily, as the implementation of the language in an optimizing compiler would not have to include the operational semantics rules verbatim. Indeed, despite many published versions of the operational semantics of System FC (e.g., [87]), GHC does not currently implement these rules directly. In a similar fashion, an implementation of Pico would not need to include the hideously inefficient version of **letrec** sketched above but could use existing techniques to implement recursion.

Given that Pico incorporates general recursion via **fix**, adding such constructs should not imperil type safety.

5.13.2 A primitive equality check

Haskell also supports non-linear patterns in its type families, as canonically embodied by this type function:

```
type family Equals x y where
  Equals a a = 'True
  Equals a b = 'False
```

The *Equals* type family effectively compares its two arguments. If they are identical (reducing other type families as possible and necessary), *Equals* returns *True*. On the other hand, if the two arguments are apart, in the sense described by Eisenberg et al. [32],[70] *Equals* reduces to *False*. If the arguments are neither identical nor apart, the call cannot reduce.

Equals cannot be represented in Pico as described in this chapter; no typing rule has a notion of apartness built into it. Thus we need a new primitive if we are to

[70] Briefly, two types are apart if there is no possibility of a coercion between them. Or, rather, it is a conservative approximation of non-coercibility, as non-coercibility is undecidable.

$$\frac{\Sigma;\Gamma \vdash_{\mathsf{ty}} \tau_1 : \kappa \qquad \Sigma;\Gamma \vdash_{\mathsf{ty}} \tau_2 : \kappa}{\Sigma;\Gamma \vdash_{\mathsf{ty}} \mathbf{equals}\,\tau_1\,\tau_2 : \mathit{Bool}} \quad \textsc{Ty\_Equals}$$

$$\frac{\Sigma;\Gamma \vdash_{\mathsf{ty}} \tau : \kappa}{\Sigma;\Gamma \vdash_{\mathsf{co}} \mathbf{axEquals}\,\tau : \mathbf{equals}\,\tau\,\tau \sim \mathit{True}} \quad \textsc{Co\_AxEquals}$$

$$\frac{\Sigma;\Gamma \vdash_{\mathsf{ty}} \tau_1 : \kappa \qquad \Sigma;\Gamma \vdash_{\mathsf{ty}} \tau_2 : \kappa \qquad \mathsf{apart}(\tau_1;\tau_2)}{\Sigma;\Gamma \vdash_{\mathsf{co}} \mathbf{axApart}\,\tau_1\,\tau_2 : \mathbf{equals}\,\tau_1\,\tau_2 \sim \mathit{False}} \quad \textsc{Co\_AxApart}$$

$$\frac{\Sigma;\Gamma \vdash_{\mathsf{s}} \tau_1 \longrightarrow \tau_1'}{\Sigma;\Gamma \vdash_{\mathsf{s}} \mathbf{equals}\,\tau_1\,\tau_2 \longrightarrow \mathbf{equals}\,\tau_1'\,\tau_2} \quad \textsc{S\_Equals\_Cong1}$$

$$\frac{\Sigma;\Gamma \vdash_{\mathsf{s}} \tau_2 \longrightarrow \tau_2'}{\Sigma;\Gamma \vdash_{\mathsf{s}} \mathbf{equals}\,v_1\,\tau_2 \longrightarrow \mathbf{equals}\,v_1\,\tau_2'} \quad \textsc{S\_Equals\_Cong2}$$

$$\frac{}{\Sigma;\Gamma \vdash_{\mathsf{s}} \mathbf{equals}\,v\,v \longrightarrow \mathit{True}} \quad \textsc{S\_EqTrue}$$

$$\frac{v_1 \neq v_2}{\Sigma;\Gamma \vdash_{\mathsf{s}} \mathbf{equals}\,v_1\,v_2 \longrightarrow \mathit{False}} \quad \textsc{S\_EqFalse}$$

Figure 5.20: Typing rules for primitive equality

compile *Equals*. Actually, we need three:

$$\begin{array}{rcl} \tau & ::= & \ldots \mid \mathbf{equals}\,\tau_1\,\tau_2 \\ \gamma & ::= & \ldots \mid \mathbf{axEquals}\,\tau \mid \mathbf{axApart}\,\tau_1\,\tau_2 \end{array}$$

The typing rules appear in Figure 5.20. Other than the new coercions **axEquals** and **axApart**, these rules might be what one would expect: the **equals** form evaluates its two arguments and then tests for equality. However, just having this evaluation behavior (without the two new coercions) is not quite enough to emulate Haskell's *Equals*: they cannot handle the case where *Equals a a* reduces to *True*, where *a* is locally bound type variable. In Haskell, the equality condition arising from a non-linear use of a variable in a pattern does not require that the arguments be reduced to any normal form; we thus have to handle this possibility in Pico. The same is true for the **axApart** coercion, necessary to handle the case (like **equals** *Int* (*Maybe a*), where *a* is a local type variable) where the arguments are demonstrably apart but not normal forms.

The typing rules above cause a challenge in proving the completeness of the rewrite relation (Section 5.10.3). To prove completeness for Co_AxEquals, we would need to show that **equals** $\tau\,\tau$ eventually reduces to *True*, but that requires termination. To prove completeness for Co_AxApart, we would need to show that τ_1 and τ_2

reduce to distinct values whenever $\mathsf{apart}(\tau_1; \tau_2)$. This also requires termination, in addition to certain properties of apartness. Since Pico is non-terminating, this direct approach is hopeless. Instead, we might add new rules to the rewrite relation to deal with these cases, but that moves the burden to the proof of the local diamond lemma (Section 5.10.2.2). Eisenberg et al. [32] explore this territory in some detail, but with an unsatisfying conclusion: that work assumes termination in order to get the consistency proof to go through.

As mentioned above, it is possible that recent work in this area by Kahrs and Smith [50] gives us a way to include **equals** without losing consistency, but I have yet to formally connect my work to theirs.

5.13.3 Splitting type applications

Haskell type families permit an unusual operation I will call splitting:

```
type family Split x where
  Split (a b) = 'Just '(a, b)
  Split other = 'Nothing
```

The *Split* function, inferred to have Haskell kind $\forall\ k_1\ k_2.\ k_2 \to \mathit{Maybe}\ (k_1 \to k_2, k_1)$, can detect a type application. It will return *Just* if it sees *IO Int* but *Nothing* if it sees *Bool*. This function cannot be encoded into Pico as it stands.[71] We instead must add a new primitive, **split**.

At its most basic, a **split** expression would look like this: $\mathbf{split}\,\tau\,\mathbf{into}\,\sigma_1\,\mathbf{or}\,\sigma_2$. The idea is that if τ is a type application $\tau_1\,\tau_2$, then the **split** expression reduces to σ_1 (applied to some details of τ); otherwise, the expression reduces to σ_2. The result kind of τ_1 is known: it is the type of τ. However, the argument kind of τ_1 is not apparent and thus must be passed to σ_1. The type σ_1 would thus be

$$\Pi\!\!\sim a_1{:}_{\mathsf{Irrel}}\mathbf{Type},\, b_1{:}_{\mathsf{Rel}}(\text{'}\Pi x{:}_{\mathsf{Rel}}\, a_1.\,\kappa),\, b_2{:}_{\mathsf{Rel}}\, a_1,\, c{:}\tau \sim b_1\, b_2.\,\kappa_2$$

where κ is the kind of the scrutinee τ and κ_2 is the result kind. Note the 'Π in the type of b_1, meaning that we can break apart only matchable applications. This is a good thing, because we would not want to be able to separate arbitrary functions from their arguments to inspect one or the other. In this formulation, the kind of σ_2 would just be κ_2.

Unfortunately, this "most basic" version does not quite cut it. The problem is that the scrutinee τ might also be $\tau_1\,\{\tau_2\}$ or $\tau_1\,\gamma_2$, and thus the **split** form would really need four branches (including one for the default, atomic case). Each case would need its own rule in the operational semantics. We would also need a push rule in case a coercion is in the way of examining a type application. The parallel rewrite

[71] Other type families, as long as their left-hand sides do not repeat variables, can be desugared into Pico, by adapting work by Augustsson [2].

relation would need to be extended as well, with analogues to all the new rules in the operational semantics. In the end, it seems **split** is not paying its way, and so I have kept it out of this presentation. Despite this omission, I do believe it would not be a technical challenge to add, should this feature prove necessary.

5.13.4 Levity polymorphism

In version 8, GHC supports *levity polymorphism* [28]. The idea is embodied in the following mutually recursive definitions:

```
data UnaryRep = PtrRep | IntRep | ...
type RuntimeRep = [UnaryRep]
constant TYPE :: RuntimeRep → Type   -- primitive constant
type Type = TYPE 'PtrRep
```

The idea here is that instead of having one sort, **Type**, the language would have a family of sorts, all headed by *TYPE* and indexed by an element of type *RuntimeRep*. At runtime, each sort corresponds to a different representation: values of a type of kind *TYPE* '[*PtrRep*] are represented by pointers to potentially thunked data, whereas values of a type of kind *TYPE* '[*IntRep*] are represented directly as machine integers. The use of a list to index *TYPE* is to support GHC's *unboxed tuples*, which group together values that would be passed in several registers; see a more detailed description in my concurrent work [28].

As described in my concurrent work (and too much of a diversion here to repeat in detail), abstracting over runtime representations must be quite restricted, lest the code generator be hamstrung when trying to compile code involving an unknown runtime representation.

Levity polymorphism is useful in Haskell because a number of constructs are truly flexible in which representation they work over. Two telling examples are *error* and $(\rightarrow)$. Regardless of the representation of the result of a function, *error* is always well typed, and $(\rightarrow)$ works to connect types of varying representations (like $\mathit{Int\#} \rightarrow \mathit{Bool}$, where *Int#* has kind *TYPE* '[*IntRep*] and *Bool* has kind **Type**—that is, *TYPE* '[*PtrRep*].)

Because levity polymorphism simply amounts to adding more sorts to a language, it would seem not to run into trouble with type safety. And I indeed believe this is true, that levity polymorphism does not threaten the type safety proof. However, it is very syntactically painful to add to the formalism, essentially requiring annotating every Π with the sort of its binder. This annotation becomes necessary for precisely the same reasons that we must include kinds in the types of a proposition (Section 5.12.3): we cannot prove completeness of the rewrite relation (Section 5.10.3) without it.

I thus leave adding levity polymorphism as an exercise to the reader; in my attempt to add this feature, I encountered no real challenge other than fiddliness and lots of syntactic noise.

5.13.5 The $(\to)$ type constructor

Haskell allows programmers to use the function arrow, $(\to)$, as a type constructor of kind $\mathbf{Type} \to \mathbf{Type} \to \mathbf{Type}$.[72] Here are two examples of how this works:[73]

```
  -- a class of categories
class Category (cat :: k → k → Type) where
  id  :: ∀ (a :: k). cat a a
  (∘) :: ∀ (a :: k) (b :: k) (c :: k). cat b c → cat a b → cat a c
  -- the instance for (→)
instance Category (→) where
  id x = x
  (f ∘ g) x = f (g x)
  -- a lightweight reader monad, based on (→)
instance Functor ((→) a) where
  fmap f g x = f (g x)
instance Applicative ((→) x) where
  pure x = λ_ → x
  (f <*> g) x = f x (g x)
instance Monad ((→) x) where
  (f >>= g) x = g (f x) x
```

Unfortunately, Pico cannot, as written, easily accommodate $(\to)$. A non-dependent arrow is rightly seen as a degenerate form of Π: the type $a \to b$ is the same as $\underset{\sim}{\Pi}\_{:}{}_{\mathsf{Rel}} a.\ b$. Without introducing yet a new function type (on top of the six we already have) and argument syntax, it seems hard to abstract over this degenerate form of Π.

Instead, we could add $(\to)$ as a new primitive constant with coercions relating it to $\underset{\sim}{\Pi}$:

$$\begin{array}{rcl} H & ::= & \ldots \mid (\to) \\ \gamma & ::= & \ldots \mid \mathbf{arrow}\ \tau_1\ \tau_2 \end{array}$$

$$\frac{}{\Sigma \vdash_{\mathsf{tc}} (\to) : \varnothing;\ a{:}_{\mathsf{Rel}}\mathbf{Type},\ b{:}_{\mathsf{Rel}}\mathbf{Type};\ \mathbf{Type}} \quad \textsc{Tc\_Arrow}$$

$$\frac{\Sigma;\Gamma \vdash_{\mathsf{ty}} \tau_1 : \mathbf{Type} \qquad \Sigma;\Gamma \vdash_{\mathsf{ty}} \tau_2 : \mathbf{Type}}{\Sigma;\Gamma \vdash_{\mathsf{co}} \mathbf{arrow}\ \tau_1\ \tau_2 : (\to)\ \tau_1\ \tau_2 \sim \underset{\sim}{\Pi} a{:}_{\mathsf{Rel}}\tau_1.\ \tau_2} \quad \textsc{Co\_Arrow}$$

The problem we are faced with at this point is consistency. Specifically, we will surely be unable to prove completeness of the rewrite relation (Section 5.10.3) with the Co_Arrow rule. To repair the damage, we can alter the coercion erasure operation

[72] The kind of $(\to)$ really is restricted to be $\mathbf{Type} \to \mathbf{Type} \to \mathbf{Type}$, even though a saturated use of it can relate unlifted types as well. This oddity is due to be explored, among other dark corners of lifted vs. unlifted types, in a paper I am hoping to write in the next year.

[73] Recall that, in $((\to)\ x)$, x is the parameter that is normally written to the *left* of the arrow.

to also rewrite saturated arrow forms to be Π forms, where the following equation is tried before other application forms:

$$\lfloor (\rightarrow)\, \tau_1\, \tau_2 \rfloor \;=\; \underset{\sim}{\Pi} a{:}_{\mathsf{Rel}} \lfloor \tau_1 \rfloor .\, \lfloor \tau_2 \rfloor$$

Now, completeness for Co_Arrow is trivial.

The problem will last surface in the erasure/consistency lemma (Section 5.10.4), which states that whenever $\lfloor \tau_1 \rfloor \propto \lfloor \tau_2 \rfloor$, we have $\tau_1 \propto \tau_2$. This is now plainly false. We must assert that arrow forms are consistent with Π-types:

$$\frac{}{(\rightarrow)\, \tau_1\, \tau_2 \propto \underset{\sim}{\Pi} a{:}_{\mathsf{Rel}} \tau_1 .\, \tau_2} \quad \text{C\_Arrow1}$$

$$\frac{}{\underset{\sim}{\Pi} a{:}_{\mathsf{Rel}} \tau_1 .\, \tau_2 \propto (\rightarrow)\, \tau_1\, \tau_2} \quad \text{C\_Arrow2}$$

The definition of $\propto$ is used in the proof of progress, where now we must consider the possibility of encountering unexpected arrow types. This possibility, though, is dispatched by adding one clause to the canonical forms lemma:

Lemma (Canonical form of arrow types)**.** $\Sigma; \Gamma \not\vdash_{\mathsf{fy}} v : (\rightarrow)\, \tau_1\, \tau_2$

That is, no value has an arrow type, because all λ-forms have Π-types instead. With this in hand, the progress proof should go through unimpeded.

5.14 Conclusion

This chapter is a full consideration of Pico. The detail presented here is intended to be useful to implementors of the language and researchers interested in adapting Pico to be used as the internal language for a surface language other than Haskell. I believe Pico is a viable candidate as a general-purpose intermediate language for dependently typed surface languages.

Chapter 6

Type inference and elaboration, or How to BAKE a PICO

Chapter 4 presents the additions to modern Haskell to make it Dependent Haskell, and Chapter 5 presents PICO, the internal language to which we compile Dependent Haskell programs. This chapter formally relates the two languages by defining a type inference/elaboration algorithm,[74] BAKE, checking Dependent Haskell code and producing a well typed PICO program.

At a high level, BAKE is unsurprising. It simply combines the ideas of several pieces of prior work [33, 37, 99] and targets PICO as its intermediate language. Despite its strong basis in prior work, BAKE exhibits a few novelties:

- Perhaps its biggest innovation is how it decides between dependent and non-dependent pattern matching depending on whether the algorithm is in checking or synthesis mode. (See also Section 6.4.)
- It turns out that checking the annotated expression $(\lambda(x :: s) \to ...) :: \forall\, x \to ...$ depends on whether or not the type annotation describes a dependent function. This came as a surprise. See Section 6.6.4.
- The subsumption relation allows an unmatchable function to be subsumed by a matchable one. That is, a function expecting an unmatchable function $a \to b$ can also accept a matchable one $a \,'\!\to b$.

After presenting the elaboration algorithm, I discuss the metatheory in Section 6.8. This section include a soundness result that the PICO program produced by BAKE is well typed. It also relates BAKE both to OUTSIDEIN and the bidirectional type system ("System SB") from Eisenberg et al. [33], arguing that BAKE is a conservative extension of both.

[74] I refer to BAKE variously as an elaboration algorithm, a type inference algorithm, and a type checking algorithm. This is appropriate, as it is all three. In general, I do not differentiate between these descriptors.

Full statements of all judgments appear in Appendix D, while theorems and definitions, with proofs, appear in Appendix E.

6.1 Overview

Bake is a bidirectional [78] constraint-generation algorithm [79]. It walks over the input syntax tree and generates constraints, which are later solved. It can operate in either a synthesis mode (when the expected type of an expression is unknown) or in checking mode (when the type is known). Like prior work [37, 99], I leave the details of the solver unspecified; any solver that obeys the properties described in Section 6.10.1 will do. In practice, the solver will be the one currently implemented in GHC. Despite the fact that the dependency tracking described here is omitted from Vytiniotis et al. [99], the most detailed description of GHC's solver,[75] the solver as implemented does indeed do dependency tracking and should support all of the innovations described in this chapter.

Constraints in Bake are represented by *unification telescopes*, which are lists of possibly dependent unification variables,[76] with their types. Naturally, there are two sorts of unification variables: types α and coercions ι. The solver finds concrete types to substitute in for unification variables α and concrete coercions to substitute in for unification variables ι. Implication constraints [84, 99] are handled by classifying unification variables by quantified kinds and propositions. See Section 6.3.

The algorithm is stated as several judgments of the following general form:[77]

$$\Sigma; \Psi \mapsto \mathit{inputs} \rightsquigarrow \mathit{outputs} \dashv \Omega$$

Most judgments are parameterized by a fixed signature Σ that defines the datatypes that are in scope.[78] The context Ψ is a generalization of contexts Γ; a context Ψ contains both Pico variables and unification variables. Because this is an algorithmic treatment of type inference, the notation is careful to separate inputs from outputs. Everything to the left of $\rightsquigarrow$ is an input; everything to the right is an output. Most judgments also produce an output Ω, which is a unification telescope, containing bindings for only unification variables. This takes the place of the emitted constraints

[75] In the paper describing OutsideIn [99], the authors separate out the constraint generation from the solver. They call the constraint-generation algorithm OutsideIn and the solver remains unnamed. I use the moniker OutsideIn to refer both to the constraint-generation algorithm and the solver.

[76] Depending on the source, various works in the literature refer to unification variables as existential variables (e.g., [24]) or metavariables (e.g., [37] and the GHC source code). I prefer unification variables here, as I do not wish to introduce confusion with existentials of data constructors nor the metavariables of my metatheory.

[77] The definitions for Ψ and Ω appear in Figure 6.2 on page 147.

[78] I do not consider in this dissertation how these signatures are formed. To my knowledge, there is no formal presentation of the type-checking of datatype declarations, and I consider formalizing this process and presenting an algorithm to be important future work.

seen in other constraint-generation algorithms. It also serves as a context in which to type-check the remainder of the syntax tree.

The solver's interface looks like this:

$$\Sigma; \Psi \upharpoonright\!\!\!\rightarrow_{\mathsf{solv}} \Omega \rightsquigarrow \Delta; \Theta$$

That is, it takes as inputs the current environment and a unification telescope. It produces outputs of Δ, a telescope of variables to quantify over, and Θ, the *zonker* (Section 6.3.1), which is an idempotent substitution from unification variables to other types/coercions. To understand the output Δ, consider checking the declaration $y = \lambda x \to x$. The variable x gets assigned a unification variable type α. No constraints then get put on that type. When trying to solve the unification telescope $\alpha{:}_{\mathsf{Irrel}}\mathbf{Type}$, we have nothing to do. The way forward is, of course, to generalize. So we get $\Delta = a{:}_{\mathsf{Irrel}}\mathbf{Type}$ and $\Theta = a/\alpha$. In the constraint-generation rules for declarations, the body of a declaration and its type are generalized over Δ. (See IDECL_SYNTHESIZE in Section 6.7.)

Writing a type inference algorithm for a dependently typed language presents a challenge in that the type of an expression can be very intricate. Yet we still wish to infer types for unannotated expressions. To resolve this tension, BAKE adheres to the following:

Guiding Principle. *In the absence of other information, infer a simple type.*

Guiding Principle. *Never make a guess.*

For example, consider inferring a type for

$$\mathit{compose}\ f\ g = \lambda x \to f\ (g\ x)$$

The function *compose* could naively be given either of the following types:

$$\begin{aligned}
&\mathit{compose}\ ::\ (b \to c) \to (a \to b) \to (a \to c)\\
&\mathit{compose}\ ::\ \forall\ (a :: \mathbf{Type})\\
&\qquad\qquad (b :: a \to \mathbf{Type})\\
&\qquad\qquad (c :: \forall\ (x :: a) \to b\ x \to \mathbf{Type})\\
&\qquad .\ \ \Pi\ (f :: \forall\ (x :: a).\ \Pi\ (y :: b\ x) \to c\ x\ y)\\
&\qquad\qquad (g :: \Pi\ (x :: a) \to b\ x)\\
&\qquad\qquad (x :: a)\\
&\qquad\to c\ x\ (g\ x)
\end{aligned}$$

However, we surely want inference to produce the first one. If inference did not tend toward simple types, there would be no hope of retaining principal types in the system. I do not prove that BAKE infers principal types, as doing so is meaningless without some non-deterministic specification of the type system, which is beyond the scope of this work. However, I wish to design Dependent Haskell with an eye toward

establishing a principal types result in the future. Inferring only rank-1 types still allows for higher-rank types in a bidirectional type system [74]. Accordingly, it is my hope that inferring only simple types will allow for Dependent Haskell to retain principal types.

The second guiding principle is that BAKE should never make guesses. Guesses, after all, are sometimes wrong. By "guess" here, I mean that the algorithm and solver should never set the value of a unification variable unless doing so is the only possible way an expression can be well typed. Up until this point, GHC's type inference algorithm has resolutely refused to guess. This decision manifests itself, among other places, in GHC's inability to work with a function $f :: F\ a \to F\ a$, where F is a type function.[79] The problem is that, from $f\ 3$, there is no way to figure out what a should be, and GHC will not guess the answer.

A key consequence of not making any guesses is that BAKE (more accurately, the solver it calls) does no higher-order unification. Consider this example:

```
fun :: a → (f $ a)
    -- NB: The use of $ means that f is not a matchable function
bad :: Bool → Bool
bad x = fun x
```

In the body of *bad*, it is fairly clear that we should unify f with the identity function. Yet the solver flatly refuses, because doing so amounts to a guess, given that there are many ways to write the identity function.[80]

In my choice to avoid higher-order unification, my design diverges from the designs of other dependently typed languages, where higher-order unification is common. Time will tell whether the predictability gotten from avoiding guesses is worth the potential annoyance of lacking higher-order unification. Avoiding guesses is also critical for principal types. See Vytiniotis et al. [99, Section 3.6.2] for some discussion.

Now that we've seen the overview, let's get down to details.

6.2 Haskell grammar

I must formalize a slice of Dependent Haskell in order to describe an elaboration procedure over it. The subset of Haskell I will consider is presented in Figure 6.1 on the next page. Note that all Haskell constructs are typeset in upright Latin letters; this is to distinguish these from PICO constructs, typeset in italics and often using Greek letters.

The version of Dependent Haskell presented here differs in a few details from the language presented in Chapter 4. These differences are to enable an easier specification

[79] Unless F is known to be injective [86].

[80] Note that my development does not natively support functional extensionality, so that these different ways of writing an identity function are not equal to one another.

$$
\begin{array}{rcll}
\mathrm{t}, \mathrm{k} & ::= & a \mid \lambda \mathrm{qvar}.\, \mathrm{t} \mid \Lambda \mathrm{qvar}.\, \mathrm{t} \mid \mathrm{t}_1\, \mathrm{t}_2 \mid \mathrm{t}_1\, @\mathrm{t}_2 \mid \mathrm{t} :: \mathrm{s} & \text{type/kind} \\
& \mid & \mathbf{case}\, \mathrm{t}\, \mathbf{of}\, \overline{\mathrm{alt}} \mid \mathrm{t}_1 \to \mathrm{t}_2 \mid \mathrm{t}_1\, {}'\!\to \mathrm{t}_2 \mid \mathbf{fix}\, \mathrm{t} & \\
& \mid & \mathbf{let}\, x := \mathrm{t}_1\, \mathbf{in}\, \mathrm{t}_2 & \\
\mathrm{qvar} & ::= & \mathrm{aqvar} \mid @\mathrm{aqvar} & \text{quantified variable} \\
\mathrm{aqvar} & ::= & a \mid a :: \mathrm{s} & \text{quantified variable (w/o vis.)} \\
\mathrm{alt} & ::= & \mathrm{p} \to \mathrm{t} & \text{case alternative} \\
\mathrm{p} & ::= & H\, \overline{x} \mid \_ & \text{pattern} \\
\mathrm{s} & ::= & \mathrm{quant}\, \mathrm{qvar}.\, \mathrm{s} \mid \mathrm{t} \Rightarrow \mathrm{s} \mid \mathrm{t} & \text{type scheme/polytype} \\
\mathrm{quant} & ::= & \forall \mid {}'\forall \mid \Pi \mid {}'\Pi & \text{quantifier} \\
\mathrm{decl} & ::= & x :: \mathrm{s} := \mathrm{t} \mid x := \mathrm{t} & \text{declaration} \\
\mathrm{prog} & ::= & \varnothing \mid \mathrm{decl}; \mathrm{prog} & \text{program}
\end{array}
$$

Figure 6.1: Formalized subset of Dependent Haskell

of the elaboration algorithm. Translating between the "real" Dependent Haskell of Chapter 4 and this version can be done by a preprocessing step. Critically, (but with one exception) no part of this preprocessor needs type information. For example, $\forall\, a\, b.\, \ldots$ is translated to $\forall\, @a.\, \forall\, @b.\, \ldots$ so that it is easier to consider individual bound variables.

The exception to the irrelevance of type information is in dealing with pattern matches. Haskell pattern matches can be nested, support guards, perhaps view patterns, perhaps pattern synonyms [76], etc. However, translating such a rich pattern syntax into a simple one is a well studied problem with widely used solutions [2, 101] and I thus consider the algorithm as part of the preprocessor and do not consider this further.

6.2.1 Dependent Haskell modalities

Let's now review some of the more unusual annotations in Dependent Haskell, originally presented in Chapter 4. Each labeled paragraph below describes an orthogonal feature (visibility, matchability, relevance).

The @ prefix Dependent Haskell uses an @ prefix to denote an argument that would normally be invisible. It is used in two places in the grammar:

- An @-sign before an argument indicates that the argument is allowed to be omitted, yet the user has written it explicitly. This follows the treatment in my prior work on invisible arguments [33].
- An @-sign before a quantified variable (in the definition for qvar) indicates that the actual argument may be omitted when calling a function. In a λ-expression,

this would indicate a pattern that matches against an invisible argument (Section 4.2.3.1). In a Π- or ∀-expression, the @-sign is produced by the preprocessor when it encounters a ∀ ... . or Π ... . quantification.

Ticked quantifiers Three of the quantifiers that can be written in Dependent Haskell come in two varieties: ticked and unticked. A ticked quantifier introduces matchable (that is, generative and injective) functions, whereas the unticked quantifier describes an unrestricted function space. Recall that type constructors and data constructors are typed by matchable functions, whereas ordinary λ-expressions are not.

Relevance The difference between ∀ and Π in Dependent Haskell is that the former defines an irrelevant abstraction (fully erased during compilation) while the latter describes a relevant abstraction (retained at runtime). In terms, an expression introduced by λ is a relevant abstraction; one introduced by Λ is an irrelevant one.

6.2.2 let should not be generalized

Though the formalized Haskell grammar includes **let**, I will take the advice of Vytiniotis et al. [98] that **let** should not be generalized. As discussed at some length in the work cited, local, generalized **let**s are somewhat rare and can easily be generalized by a type signature. For all the same reasons articulated in that work, generalizing **let** poses a problem for Bake. We thus live with an ungeneralized **let** construct.

6.2.3 Omissions from the Haskell grammar

There are two notable omissions from the grammar in Figure 6.1 on the preceding page.

Type constants The Haskell grammar contains no production for H, a type constant. This is chiefly because type constants must be saturated with respect to universals in Pico, whereas we do not need this restriction in Haskell. Accordingly, type constants are considered variables that expand to type constants that have been η-expanded to take their universal arguments in a curried fashion. For example, *Just* in Haskell, which can appear fully unsaturated, becomes $\lambda a{:}_{\mathsf{Irrel}}\mathbf{Type}.\, \mathit{Just}_{\{a\}}$ in Pico.

Recursive let Following the decision not to include a **letrec** construct in Pico (Section 5.1.2), the construct is omitted from the formalized subset of Haskell as well. Having a formal treatment of **letrec** would require a formalization of Haskell's consideration of polymorphic recursion [41, 62, 67], whereby definitions with type signatures can participate in polymorphic recursion while other definitions cannot. In turn, this would require a construct where a polymorphic function is treated

Metavariables:

α, β unification type variable $\quad \iota$ unification coercion variable

Grammar extensions:

τ	$::=$	$\ldots \mid \alpha_{\overline{\psi}}$	type/kind
γ	$::=$	$\ldots \mid \iota_{\overline{\psi}}$	coercion
ζ	$::=$	$\alpha \mid \iota$	unification variable
Θ	$::=$	$\varnothing \mid \Theta, \forall\, \overline{z}.\tau/\alpha \mid \Theta, \forall\, \overline{z}.\gamma/\iota$	zonker (Section 6.3.1)
ξ	$::=$	$\varnothing \mid \xi, \zeta \mapsto \overline{\psi}$	generalizer (Section 6.5)
u	$::=$	$\alpha :_\rho \forall\, \Delta.\kappa \mid \iota : \forall\, \Delta.\phi$	unif. var. binding
Ω	$::=$	$\varnothing \mid \Omega, u$	unification telescope
Ψ	$::=$	$\varnothing \mid \Psi, \delta \mid \Psi, u$	typing context

I elide the $\forall$ when the list of variables or telescope quantified over would be empty.

Figure 6.2: Additions to the grammar to support Bake.

monomorphically in a certain scope and polymorphically beyond that scope.[81] The problems faced here are not unique to (nor made particularly worse by) dependent types. I thus have chosen to exclude this construct for simplicity.

We have now reviewed the source language of Bake, and the previous chapter described its target language, Pico. I'll now fill in the gap by introducing the additions to the grammar needed to describe the inference algorithm.

6.3 Unification variables

The extensions to the grammar to support inference are in Figure 6.2. These extensions all revolve around supporting unification variables, which are rather involved. One might think that unification variables need not be so different from ordinary variables; constraint generation could produce a telescope of these unification variables and solving simply produces a substitution. However, this naive view does not work out because of unification variable generalization.[82]

Consider a λ-abstraction over the variable x. When doing constraint generation inside of the λ, the kinds of fresh unification variables might mention x. Here is a case in point, which will serve as a running example:

[81] Readers familiar with the internals of GHC may recognize its *AbsBinds* data constructor in this description. Formalizing all of its intricacies would indeed be required to infer the type of a **letrec**.

[82] The treatment of unification variables throughout Bake is essentially identical to the treatment by Gundry [37], which is itself closely based on the work of Dunfield and Krishnaswami [24].

$\mathit{poly} :: \forall\ j\ (b :: j) \to \ldots$
$\mathit{example} = \lambda k\ a \to \mathit{poly}\ k\ a$

Type inference can easily discover that the kind of a is k. But in order for the inference algorithm to do this, it must be aware that k is in scope before a is. Note that when we call the solver (after type-checking the entire body of *example*), k is *not* in scope. Thus, as we produce the unification telescope during constraint generation over the body of *example*, we must somehow note that the unification variable α (the type of a) can mention k.

This means that unification variable bindings are quantified over a telescope Δ. (You can see this in the definition for u in Figure 6.2 on the preceding page.) In the vocabulary of OutsideIn, the bindings in Δ are the *givens* under which a unification variable should be solved for and a unification variable binding $\alpha :_\rho \forall\, \Delta.\kappa$ or $\iota : \forall\, \Delta.\phi$ with a non-empty Δ is an implication constraint.

6.3.1 Zonking

Solving produces a substitution from unification variables to types/coercions. Following the nomenclature within GHC, I call applying this substitution *zonking*. The substitution itself, written Θ, is called a *zonker*.

Zonkers pose a naming problem. Consider solving to produce the zonker for *example*, above. Suppose the type of a is assigned to be α. We would like to zonk α to k. However, as before, k is out of scope when solving for α. We thus cannot just write k/α, as that would violate the Barendregt convention, where we can never name a variable that is out of scope (as it might arbitrarily change due to α-renaming).

The solution to this problem is to have all occurrences of unification variables applied to vectors $\overline{\psi}$.[83] When we zonk a unification variable occurrence $\alpha_{\overline{\psi}}$, the vector $\overline{\psi}$ is substituted for the variables in the telescope Δ that α's kind is quantified over.

Here is the formal definition of zonking:

Definition (Zonking [Definition E.19])**.** *A zonker can be used as a postfix function. It operates homomorphically on all recursive forms and as the identity operation on leaves other than unification variables. Zonking unification variables is defined by these equations:*

$$\begin{array}{rcl} \forall\, \overline{z}.\tau/\alpha \in \Theta & \Longrightarrow & \alpha_{\overline{\psi}}[\Theta] = \tau[\overline{\psi}[\Theta]/\overline{z}] \\ \textit{otherwise} & & \alpha_{\overline{\psi}}[\Theta] = \alpha_{\overline{\psi}[\Theta]} \\ \forall\, \overline{z}.\gamma/\iota \in \Theta & \Longrightarrow & \iota_{\overline{\psi}[\Theta]} = \gamma[\overline{\psi}[\Theta]/\overline{z}] \\ \textit{otherwise} & & \iota_{\overline{\psi}[\Theta]} = \iota_{\overline{\psi}[\Theta]} \end{array}$$

Continuing the example from above, we would say that a has the type α_k, where we have $\alpha :_{\mathsf{Irrel}} \forall\, k{:}_{\mathsf{Irrel}}\mathbf{Type}.\mathbf{Type}$. The solver will create a zonker with the mapping

[83] Recall that ψ is a metavariable that can stand for either a type or a coercion. Thus $\overline{\psi}$ is a mixed list of types and coercions, suitable for substituting in for a list of type/coercion variables $\overline{z}$.

$\boxed{\Sigma; \Psi \models_{\mathsf{ty}} \tau : \kappa}$ Extra rule to support unification variables in types

$$\frac{\alpha :_{\mathsf{Rel}} \forall\, \Delta.\kappa \in \Psi \qquad \Sigma \models_{\mathsf{ctx}} \Psi\ \mathsf{ok} \qquad \Sigma; \Psi \models_{\mathsf{vec}} \overline{\psi} : \Delta}{\Sigma; \Psi \models_{\mathsf{ty}} \alpha_{\overline{\psi}} : \kappa[\overline{\psi}/\mathsf{dom}(\Delta)]}\ \textsc{Ty\_UVar}$$

$\boxed{\Sigma; \Psi \models_{\mathsf{co}} \gamma : \phi}$ Extra rule to support unification variables in coercions

$$\frac{\iota : \forall\, \Delta.\phi \in \Psi \qquad \Sigma \models_{\mathsf{ctx}} \Psi\ \mathsf{ok} \qquad \Sigma; \Psi \models_{\mathsf{vec}} \overline{\psi} : \Delta}{\Sigma; \Psi \models_{\mathsf{co}} \iota_{\overline{\psi}} : \phi[\overline{\psi}/\mathsf{dom}(\Delta)]}\ \textsc{Co\_UVar}$$

$\boxed{\Sigma \models_{\mathsf{ctx}} \Psi\ \mathsf{ok}}$ Extra rules to support binding unification variables

$$\frac{\Sigma; \mathsf{Rel}(\Psi, \Delta) \models_{\mathsf{ty}} \kappa : \mathbf{Type} \qquad \Sigma \models_{\mathsf{ctx}} \Psi\ \mathsf{ok}}{\Sigma \models_{\mathsf{ctx}} \Psi, \alpha :_{\rho} \forall\, \Delta.\kappa\ \mathsf{ok}}\ \textsc{Ctx\_UTyVar}$$

$$\frac{\Sigma; \mathsf{Rel}(\Psi, \Delta) \models_{\mathsf{prop}} \phi\ \mathsf{ok} \qquad \Sigma \models_{\mathsf{ctx}} \Psi\ \mathsf{ok}}{\Sigma \models_{\mathsf{ctx}} \Psi, \iota : \forall\, \Delta.\phi\ \mathsf{ok}}\ \textsc{Ctx\_UCoVar}$$

Figure 6.3: Extra rules in Pico judgments to support unification variables

$\forall j.j/\alpha$ (where I have changed the variable name for demonstration). This will zonk α_k to become $j[k/j]$ which is, of course k as desired.

Note that the quantification we see here is very different from normal Π-quantification in Pico. These quantifications are fully second class and may be viewed almost as suspended substitutions.

6.3.2 Additions to Pico judgments

The validity and typing judgments in Pico all work over signatures Σ and contexts Γ. In Bake, however, we need to be able to express these judgments in an environment where unification variables are in scope. I thus introduce mixed contexts Ψ, containing both Pico variables and unification variables.

Accordingly, I must redefine all of the Pico judgments to support unification variables in the context. These judgments are written with a ⊨ turnstile in place of Pico's ⊢ turnstile. There are also several new rules that must be added to support unification variables. These rules appear in Figure 6.3.

Note the rules Ty_UVar and Co_UVar that support unification variable occurrences. The unification variables are applied to vectors $\overline{\psi}$ which must match the telescope Δ in the classifier for the unification variable. In addition, this vector is substituted directly into the unification variable's kind.

These definitions support all of the properties proved about the original Pico

judgments, such as substitution and regularity. The statements and proofs are in Appendix E.

6.3.3 Untouchable unification variables

Vytiniotis et al. [99, Section 5.2] introduces the notion of *touchable* unification variables, as distinct from *untouchable* variables. Their observation is that it is harmful to assign a value to a "global" unification variable when an equality constraint is in scope. "Global" here means that the unification variable has a larger scope than the equality constraint. We call the "local" unification variables touchable, and the "global" ones untouchable. OutsideIn must manually keep track of touchability; the set of touchable unification variables is an extra input to its solving judgment.

In Bake, on the other hand, tracking touchability is very easy with its use of unification telescopes: all unification variables quantified by the same equality constraints as the constraint under consideration are touchable; the rest are untouchable.

To make this all concrete, let's look at a concrete example (taken from Vytiniotis et al. [99]) where the notion of touchable variables is beneficial.

Suppose we have this definition:

$$\begin{array}{l}\textsf{data } T\ a \textsf{ where}\\ \quad K :: (\mathit{Bool} \sim a) \Rightarrow \mathit{Maybe}\ \mathit{Int} \to T\ a\end{array}$$

I have written this GADT with an explicit equality constraint in order to make the use of this constraint clearer. The definition for K is entirely equivalent to saying $K :: \mathit{Maybe}\ \mathit{Int} \to T\ \mathit{Bool}$.

We now wish to infer the type of

$$\lambda x \to \textsf{case } x \textsf{ of } \{K\ n \to \mathit{isNothing}\ n\}$$

where $\mathit{isNothing} :: \forall\ a.\ \mathit{Maybe}\ a \to \mathit{Bool}$ checks for an empty *Maybe*. Consider any mention of a new unification variable to be fresh. We assign x to have type α_0 and the result of the function to have type β_0. By the existence of the constructor K in the **case**-match, we learn that α_0 should really be $T\,\alpha_1$. Inside the **case** alternative, we now have a given constraint $\mathit{Bool} \sim \alpha_1$. We then instantiate the polymorphic *isNothing* with a unification variable β_1, so that the type of *isNothing* is $\mathit{Maybe}\ \beta_1 \to \mathit{Bool}$. We can now emit two equality constraints:

- The argument type to *isNothing*, $\mathit{Maybe}\ \beta_1$, must match the type of n, $\mathit{Maybe}\ \mathit{Int}$.
- The return type of the **case** expression (β_0) is the return type of *isNothing* (*Bool*).

Pulling this all together, we get the following unification telescope:

$$\begin{aligned}\Omega = [\ & \alpha_0{:}_{\mathsf{Irrel}}\mathbf{Type},\\ & \beta_0{:}_{\mathsf{Irrel}}\mathbf{Type},\\ & \alpha_1{:}_{\mathsf{Irrel}}\mathbf{Type},\\ & \iota_0{:}\alpha_0 \sim \mathit{T}\ \alpha_1,\\ & \beta_1 :_{\mathsf{Irrel}} \forall\,(c{:}\mathit{Bool} \sim \alpha_1).\mathbf{Type},\\ & \iota_1 : \forall\,(c{:}\mathit{Bool} \sim \alpha_1).(\mathit{Maybe}\,\beta_{1\,c} \sim \mathit{Maybe}\ \mathit{Int}),\\ & \iota_2 : \forall\,(c{:}\mathit{Bool} \sim \alpha_1).(\beta_0 \sim \mathit{Bool})\end{aligned}$$

Before we walk through what the solver does with such a telescope, what *should* it do? That is, what's the type of our original expression? It turns out that this is not an easy question to answer! The expression has no principal type. Both of the following are true:

$$\begin{aligned}&(\lambda x \to \mathsf{case}\ x\ \mathsf{of}\ \{K\ n \to \mathit{isNothing}\ n\}) :: \forall\ a.\ \mathit{T}\ a \to a\\ &(\lambda x \to \mathsf{case}\ x\ \mathsf{of}\ \{K\ n \to \mathit{isNothing}\ n\}) :: \forall\ a.\ \mathit{T}\ a \to \mathit{Bool}\end{aligned}$$

Note that neither $\mathit{T}\ a \to a$ nor $\mathit{T}\ a \to \mathit{Bool}$ is more general than the other.

We would thus like the solver to fail when presented with this unification telescope. This is true, even though there is a solution to the inference problem (that is, a valid zonker Θ with a telescope of quantified variables Δ; see the specification of $\mathrel{\overset{\rightarrow}{\mathsf{solv}}}$, Section 6.1):

$$\begin{aligned}\Delta &= a{:}_{\mathsf{Irrel}}\mathbf{Type}\\ \Theta &= \mathit{T}\ a/\alpha_0,\\ &\quad\ \mathit{Bool}/\beta_0,\\ &\quad\ a/\alpha_1,\\ &\quad\ \langle \mathit{T}\ a\rangle/\iota_0,\\ &\quad\ \forall\, c.\mathit{Int}/\beta_1,\\ &\quad\ \forall\, c.\langle \mathit{Maybe}\ \mathit{Int}\rangle/\iota_1,\\ &\quad\ \forall\, c.\langle \mathit{Bool}\rangle/\iota_2\end{aligned}$$

The problem is that here is another valid substitution for β_0 and ι_2:

$$\begin{aligned}\Theta = &\ldots,\\ &\quad a/\beta_0,\\ &\quad \ldots,\\ &\quad \forall\, c.\mathbf{sym}\ c/\iota_2\end{aligned}$$

These zonkers correspond to the overall type $\mathit{T}\ a \to \mathit{Bool}$ and $\mathit{T}\ a \to a$, respectively.

We must thus ensure that the solver rejects Ω outright. This is achieved by making β_0 untouchable when considering solving the ι_2 constraint.[84] As described

[84]Why this particular mechanism works is discussed in some depth by Vytiniotis et al. [99, Section

by Vytiniotis et al. [99, Section 5.5], the solver considers the constraints individually. When simplifying (OutsideIn's terminology for solving a simple, non-implication constraint) the ι_1 and ι_2 constraints, any unification variable not quantified by c is considered untouchable.[85] Thus, β_0 is untouchable when simplifying ι_2, so the solver will never set β_0 to anything at all. It will remain an ambiguous variable and a type error will be issued.

Contrast this with α_1, which is also not set by the solver. This variable, however, is fully unconstrained and can be quantified over and turned into the non-unification variable a. There is no way to quantify over β_0, however.

Despite not setting β_0, the solver is free to set β_1 which is considered touchable, as it is also quantified by c. The unification variable β_1 is fully local to the **case** alternative body, and setting it can have no effect outside of the **case** expression. In the terminology of OutsideIn, that unification would be introduced by $\exists\beta_1$ in an implication constraint. In our example, the ability to set β_1 means that we get only one type error reported, not two.

6.4 Bidirectional type-checking

Like previous algorithms for GHC/Haskell [33, 37, 74], Bake takes a bidirectional approach [78]. The fundamental idea to bidirectional type-checking is that, sometimes, the type inference algorithm knows what type to look for. When this happens, the algorithm should take advantage of this knowledge.

Bidirectional type-checking works by defining two mutually recursive algorithms: a type synthesis algorithm and a type checking algorithm. The former is used when we have no information about the type of an expression, and the latter is used when we do indeed know an expression's expected type. The algorithms are mutually recursive because of function applications: knowing the result type of a function call does not tell you about the type of the function (meaning the checking algorithm must use synthesis on the function), but once we know the function's type, we know the type of its arguments (allowing the synthesis algorithm to use the more informative checking algorithm).

Historically, bidirectional type-checking in Haskell has been most useful when considering higher-rank polymorphism—for example, in a type like $(\forall\ a.\ a \to a) \to \mathit{Int}$. Motivating higher-rank types would bring us too far afield, but the literature has helpful examples [33, 74] and there is a brief introduction in Section 2.5. Naturally, Dependent Haskell continues to use bidirectional type-checking to allow for higher-rank types, but there is now even more motivation for bidirectionality.

5.2].

[85] To make this a bit more formal, I would need to label the quantification by c by some label drawn from an enumerable set of labels. The touchable unification variables would be those quantified by the same label as the constraint being simplified. We cannot just use the name c, as names are fickle due to potential α-variation.

As discussed above (Section 6.3.3), bringing equality constraints into scope makes some unification variables untouchable. In practice, this means that the result type of a GADT pattern match must be known; programmers must put type annotations on functions that perform a GADT pattern match.

In a dependently typed language, however, *any* pattern match might bring equality constraints into scope, where the equality relates the scrutinee with the pattern. For example, if I say something as simple as **case** b **of** $\{ \mathit{True} \to x; \mathit{False} \to y \}$, I may want to use the fact that $b \sim \mathit{True}$ when type-checking x or $b \sim \mathit{False}$ when type-checking y. This is, of course, dependent pattern matching (Section 4.3.3). Our problem now is that it seems that *every* pattern match introduces an equality constraint, meaning that the basic type inference of Haskell might no longer work, stymied by untouchable variables.

The solution is to take advantage of the equality available by dependent pattern matching only when the result type of the **case** expression is being propagated downwards—that is, when the inference algorithm is in checking mode. If we do not know a **case** expression's overall type, then the pattern match is treated as a traditional, non-dependent pattern match. Without bidirectional type-checking, the user might have to annotate which kind of match is intended.[86]

6.4.1 Invisibility

As discussed in Section 4.2.3, Dependent Haskell programmers can choose the visibility of their arguments: A visible argument must be provided at every function call, while an invisible one may be elided. If the programmer wants to write an explicit value to use for an invisible argument, prefixing the argument with @ allows it to stand for the invisible parameter.

In the context of type inference, though, we must be careful. As explored in my prior work [33], invisible arguments are sometimes introduced at the whim of the compiler. For example, consider

$$\begin{aligned} &\text{-- } \mathit{isShorterThan} :: [a] \to [b] \to \mathit{Bool} \\ &\mathit{isShorterThan}\ \mathit{xs}\ \mathit{ys} = \mathit{length}\ \mathit{xs} < \mathit{length}\ \mathit{ys} \end{aligned}$$

Note that the type signature is commented out. The function *isShorterThan* takes two invisible arguments, a, and b. Which order should they appear in? Without the type signature for guidance, it is, in general, impossible to predict what order these will be generalized. See Eisenberg et al. [33, Section 3.1] for more discussion on this point.

Despite the existence of functions like *isShorterThan* with fully inferred type signatures, we wish to retain principal types in our type system—at least in the subset of the language that does not work with equality constraints. We thus must have *three* different levels of visibility:

[86] The Dependent Haskell described by Gundry [37] indeed has the user annotate this choice for **case** expressions. Due to Gundry's restrictions on the availability of terms in types (see his Section 6.2.3), however, the bidirectional approach would have been inappropriate in his design.

Required parameters (also called visible) must be provided at function call sites.

Specified parameters are invisible, but their order is user-controlled. These parameters are to functions with type signatures or with an explicit ∀....

Inferred parameters (called "generalized" in Eisenberg et al. [33]) are ones invented by the type inference algorithm (like the parameter a in the example used to explain untouchable variables; see Section 6.3.3). They cannot ever be instantiated explicitly. All coercion abstractions are inferred.

Note that these three levels of visibility are not a consequence of dependent types, but of having an invisibility override mechanism; these three levels of visibility are fully present in GHC 8. In the judgments that form Bake, I often write a subscript Req, Spec, or Inf to Π symbols indicating the visibility of the binders quantified over. These subscripts have no effect on well-formedness of types and are completely absent from pure Pico.

Following my prior work, both the synthesis and checking algorithms are split into two judgments apiece: one written $\overset{}{\underset{\mathsf{ty}}{\Rightarrow}}$ and one written $\overset{*}{\underset{\mathsf{ty}}{\Rightarrow}}$. The distinction is that the latter works with types that may have invisible binders, while the former does not. For example, a type produced by the $\overset{}{\underset{\mathsf{ty}}{\Rightarrow}}$ judgment in synthesis mode is guaranteed not to have any invisible (that is, specified or inferred) binders at the outermost level. Thus when synthesizing the type of t_1 in the expression $\mathrm{t}_1\,\mathrm{t}_2$, we use the $\overset{}{\underset{\mathsf{ty}}{\Rightarrow}}$ judgment, as we want any invisible arguments to be inferred before applying t_1 to t_2. Considering the algorithm in checking mode, when processing a traditional λ-expression, we want the rule to be part of the $\overset{}{\underset{\mathsf{ty}}{\Rightarrow}}$ judgment, to be sure that the algorithm has already skolemized (Section 6.4.3) the known type down to one that accepts a visible argument. Conversely, the rule for an expression like $\lambda\ @a \to ...$ must belong in the $\overset{*}{\underset{\mathsf{ty}}{\Rightarrow}}$ judgment, as we want to see the invisible binders in the type to match against the invisible argument the programmer wishes to bind.

The interplay between the starred judgments and the unstarred nudges this system toward principal types. Having these two different judgments is indeed one of the main innovations in my prior work [33], where the separation is necessary to have principal types.

6.4.2 Subsumption

Certain expression forms do not allow inward propagation of a type. As mentioned above, if we are checking an expression $f\ x$ against a type τ, we have no way of usefully propagating information about τ into f or x. Instead, we use the synthesis judgment for f and then check x's type against the argument type found for f. After all of this, we will get a type τ' for $f\ x$. We then must check τ' against τ—but they do not have to match exactly. For example, if τ' is $\forall\ a.\ a \to a$ and τ is $\mathit{Int} \to \mathit{Int}$, then we're fine, as any expression of the former type can be used at the latter.

$\boxed{\vdash^{\rightarrow}_{\mathsf{pre}} \kappa \rightsquigarrow \kappa'}$ Convert a kind into prenex form.

$$\frac{\nu \leq \mathsf{Spec} \qquad \vdash^{\rightarrow}_{\mathsf{pre}} \kappa_2 \rightsquigarrow \prod\!\!\!\!\sim \Delta.\, \kappa_2'}{\vdash^{\rightarrow}_{\mathsf{pre}} \prod\!\!\!\!\sim_{\nu} \delta.\, \kappa_2 \rightsquigarrow \prod\!\!\!\!\sim \delta, \Delta.\, \kappa_2'} \textsc{Prenex\_Invis}$$

$$\frac{\vdash^{\rightarrow}_{\mathsf{pre}} \kappa_2 \rightsquigarrow \prod\!\!\!\!\sim \Delta.\, \kappa_2'}{\vdash^{\rightarrow}_{\mathsf{pre}} \prod\!\!\!\!\sim_{\mathsf{Req}} \delta.\, \kappa_2 \rightsquigarrow \prod\!\!\!\!\sim \Delta.\, \prod\!\!\!\!\sim_{\mathsf{Req}} \delta.\, \kappa_2'} \textsc{Prenex\_Vis}$$

$$\frac{}{\vdash^{\rightarrow}_{\mathsf{pre}} \kappa \rightsquigarrow \kappa} \textsc{Prenex\_NoPi}$$

$\boxed{\kappa_1 \leq^* \kappa_2}$ "κ_1 subsumes κ_2." (κ_2 is in prenex form)

$$\frac{\neg(\rho_1 = \mathsf{Rel} \wedge \rho_2 = \mathsf{Irrel}) \qquad \kappa_3 \leq \kappa_1 \rightsquigarrow \tau \qquad \kappa_2[\tau_1\ b/a] \leq \kappa_4}{\Pi_{\mathsf{Req}}\, a{:}_{\rho_1}\kappa_1.\, \kappa_2 \leq^* \prod\!\!\!\!\sim_{\mathsf{Req}} b{:}_{\rho_2}\kappa_3.\, \kappa_4} \textsc{Sub\_Fun}$$

$$\frac{\mathsf{fresh}\ \iota{:}\tau_1 \sim \tau_2}{\tau_1 \leq^* \tau_2} \textsc{Sub\_Unify}$$

$\boxed{\kappa_1 \leq \kappa_2}$ "κ_1 subsumes κ_2."

$$\frac{\vdash^{\mathsf{Spec}}_{\mathsf{inst}} \kappa_1 \rightsquigarrow \kappa_1' \qquad \vdash^{\rightarrow}_{\mathsf{pre}} \kappa_2 \rightsquigarrow \prod\!\!\!\!\sim \Delta.\, \kappa_2' \qquad \kappa_1' \leq^* \kappa_2'}{\kappa_1 \leq \kappa_2} \textsc{Sub\_DeepSkol}$$

Figure 6.4: Subsumption in Bake (simplified)

What we need here is a notion of *subsumption*, whereby we say that $\forall\ a.\ a \to a$ *subsumes* $\mathit{Int} \to \mathit{Int}$, written

$$\forall\ a.\ a \to a \leq \mathit{Int} \to \mathit{Int}$$

For reasons well articulated in prior work [74, Section 4.6], my choice for the subsumption relation does *deep skolemization*. This means that the types $\forall\ a.\ \mathit{Int} \to a \to a$ and $\mathit{Int} \to \forall\ a.\ a \to a$ are fully equivalent. This choice is furthermore backward compatible with the current treatment of non-prenex types in GHC.

Bake's subsumption relation is in Figure 6.4. The rules in this figure are simplified from the full rules (which appear in Section D.9), omitting constraint generation and elaboration. The rules in each judgment are meant to be understood as an algorithm, trying earlier rules before later ones. Thus, for example, rule Sub_Unify is not as universal as it appears.

The entry point is the bottom, unstarred subsumption judgment. It computes

the prenex form of κ_2 using the auxiliary judgment $\vdash^{\mathsf{Spec}}_{\mathsf{pre}}$ and instantiates κ_1. (The Spec superscript to $\vdash^{\mathsf{Spec}}_{\mathsf{inst}}$ says to instantiate any argument that is no more visible than Spec—that is, either Inf or Spec arguments.) The instantiated κ_1' and prenexed κ_2' are then compared using the starred subsumption judgment.[87]

The starred judgment has the usual contravariance rule for functions. This rule, however, has three interesting characteristics.

Dependency We cannot simply compare $\kappa_2 \leq \kappa_4$. The problem is that κ_2 has a variable a of type κ_1 in scope, whereas κ_4 has a variable b of type κ_3 in scope. Contrast this rule to a rule for non-dependent functions where no such bother arises. In the fully detailed versions of these judgments, learning that $\kappa_1 \leq \kappa_2$ gives us a term τ such that $\tau : \tilde{\Pi}\_ :_{\mathsf{Rel}} \kappa_1.\, \kappa_2$—that is, a way of converting a κ_1 into a κ_2. I include such a τ when checking whether $\kappa_3 \leq \kappa_1$. This τ is then used to convert $b : \kappa_3$ into a value of type κ_1, suitable for substitution in for a. With this substitution completed, we can perform the subsumption comparison against κ_4 as desired.

Matchable functions subsume unmatchable ones Rule Sub_Fun includes a subsumptive relationship among the two flavors of Π. Whenever an unmatchable $\tilde{\Pi}$-type is expected, surely a matchable Π-type will do. Thus we allow either Π on the left of the $\leq$. Note that the other way would be wrong: not only might an unmatchable $\tilde{\Pi}$-type not work where a matchable Π-type is expected, but we also have no way of creating the Π-type during elaboration. Our need to elaborate correctly keeps us from getting this wrong.

Irrel subsumes Rel Finally, the rule also includes a subsumptive relationship among relevances. If the relevances ρ_1 and ρ_2 match up, then all is well. But also if ρ_1 is Irrel and ρ_2 is Rel, we are OK. If ρ_2 is Rel, that says that the expression we are checking is allowed to use its argument relevantly, but nothing goes wrong if the expression, in fact, does not (that is, if ρ_1 is Irrel). Once again, elaboration keeps us honest here; if the rule is written the wrong way around, there is no sound way to elaborate.

6.4.3 Skolemization

In checking mode, the $\vdash^{*}_{\mathsf{ty}}$ judgment *skolemizes* any invisible quantifiers in the known type.[88] As an example, consider

$$(\lambda x \to x) :: \forall\ a.\ a \to a$$

[87] The stars on these judgments have a different meaning than the star on $\vdash^{*}_{\mathsf{ty}}$; they are borrowed from the notation by Peyton Jones et al. [74], not Eisenberg et al. [33].

[88] I am following Peyton Jones et al. [74] in my use of the word "skolem". I understand that this word may have slightly different connotation in a logical context, but my use here has become standard in the study of GHC/Haskell.

When checking the λ-expression against that type, we first must dispose of the $\forall\ a$. This is done by essentially making a a fresh type constant, equal to no other. This act is called skolemization; a becomes a skolem. The variable x is then given this type a, and the body of the λ indeed has type a as desired.

As we look at more complicated examples, a question arises about how deeply to skolemize. Here is an illustrative example, taken from prior work [34]:

$$
\begin{aligned}
&x = \lambda 5\ z \to z \\
&\quad \text{-- } x \text{ is inferred to have type } \forall\ a.\ \mathit{Int} \to a \to a \\
&y :: \mathit{Int} \to \forall\ b.\ b \to b \\
&y = x
\end{aligned}
$$

In this example, we are checking x of type $\forall\ a.\ \mathit{Int} \to a \to a$ against the type $\mathit{Int} \to \forall\ b.\ b \to b$. We must be a bit careful here, though: x's type is fully inferred, and thus its quantification over a is Inf, not Spec. With the right flags,[89] GHC prints x's type as $\forall\ \{a\}.\ \mathit{Int} \to a \to a$ to denote that it is not available for a visibility override.

The type we are checking against does not have any invisible binders at the top (its first binder is the visible one for Int), so we do not initially skolemize. We instead discover that there is no checking rule for variables and have to use the fall-through case for checking, which does synthesis and then a subsumption check. However, a naive approach would be wrong here: if we synthesize the type of x, we will get the instantiated $\mathit{Int} \to \alpha \to \alpha$. This is because Inf binders are always instantiated immediately, much like in the original syntax-directed version of the Hindley-Milner type system [18, 20]. In the subsumption check, we will want to set α to be b, the skolem created from y's type signature. We will be unable to do so, however, because doing so would be ill scoped: α occurs in the unification telescope before b is ever brought into scope. This means that it would be ill scoped for the value chosen for α to refer to b.[90] It would quite unfortunate to reject this example, because the subsumption judgment, with its deep skolemization, would have this work out if only we didn't instantiate that Inf binder so eagerly.

Instead, I have written the ITyC_Infer rule (details in Section 6.6.2) to eagerly skolemize the known type deeply, effectively *before* ever looking at the expression. This puts b firmly into scope when consider α, and the subsumption check (and later solver) succeeds.

The solution to this problem proposed in prior work is to do deep skolemization in the checking $\overset{*}{\underset{\mathsf{ty}}{\mapsto}}$ judgment. This works in the System SB of Eisenberg et al. [33]. However, it fails us here. The problem is that Dependent Haskell allows for constructs like $\lambda n\ @a \to ....$ If we check that expression against $\mathit{Int} \to \forall\ a.\ a \to a$, we want

[89] `-fprint-explicit-foralls`, specifically

[90] Saying that this example fails because of scoping is a vast improvement over the state of affairs in Eisenberg et al. [34], where a delicate line of reasoning based on the subtleties of the Barendregt convention is necessary to show how this example goes awry. By tracking our unification variables in a telescope, problems like this become much clearer.

the as to match up. Yet deeply skolemizing the type we are checking against will eliminate the a and our algorithm will reject the code. We thus instead do shallow skolemization in $\overset{*}{\underset{\mathsf{ty}}{\mapsto}}$ and instead save the deep skolemization until we are forced to switch into synthesis mode.

Returning to the x/y example, here is how it plays out:

1. The variable x is inferred to have type $\forall\,\{a\}.\ \mathit{Int} \to a \to a$ when processing the declaration for x.

2. We then check the body of y against the type $\mathit{Int} \to \forall\ b.\ b \to b$. As there are no invisible binders, no skolemization happens right away.

3. We quickly find that no checking rules apply. We then deeply skolemize the expected type, getting $\mathit{Int} \to b \to b$ for a skolem b.

4. Now, we synthesize the type for the expression x, getting $\mathit{Int} \to \alpha \to \alpha$.

5. The subsumption relation checks whether $\mathit{Int} \to \alpha \to \alpha$ subsumes $\mathit{Int} \to b \to b$. This is indeed true with $\alpha := b$, and the definition for y is accepted.[91]

We have thus accepted our problem example and remain in line with the declarative system proposed in my prior work [33, Section 6.2].

6.5 Generalization

There is one final aspect of the inference algorithm that requires study before we look at the individual pieces: the generalization operation.[92] That said, in terms of understanding the Bake algorithm, having a strong grasp on generalization is not terribly important; this is merely a technical step needed to make the mathematics hold together.

Suppose we are synthesizing the type of a λ-expression $\lambda x \to \tau$. We choose a unification variable α for the type of x. We then must put $x{:}_{\mathsf{Rel}}\alpha$ into the context when synthesizing the type for τ. Synthesizing this type will produce a unification telescope Ω. Now we have a problem: what unification telescope will we return from synthesizing the type of the entire λ-expression? It looks something like $\alpha{:}_{\mathsf{Irrel}}\mathbf{Type}, x{:}_{\mathsf{Rel}}\alpha, \Omega$ but, critically, that is not a unification telescope, as that context contains a binding for an ordinary Pico variable, x.

[91] Although not visible in the simplified presentation of Sub_DeepSkol in Figure 6.4 on page 155, it is critical that κ_2 is skolemized *before* κ_1 is instantiated, lest we end up with the same scoping problem. This can be seen in the full rule (Section D.9) with the fact that we include Ω_1 in the final generalization step. In contrast to other potential pitfalls mentioned earlier, leaving Ω_1 out of this line does not imperil the soundness of elaboration; it is only a matter of expressiveness of the source Haskell.

[92] What I call generalization here is precisely what Gundry [37, Section 7.5] calls "parameterisation" and writes with $\nearrow$.

$\boxed{\Omega \hookrightarrow \Delta \rightsquigarrow \Omega'; \xi}$ Generalize Ω over Δ.

$$\frac{}{\varnothing \hookrightarrow \Delta \rightsquigarrow \varnothing; \varnothing} \text{ IGen\_Nil}$$

$$\frac{\xi_0 = \alpha \mapsto \mathsf{dom}(\Delta) \qquad \Omega[\xi_0] \hookrightarrow \Delta \rightsquigarrow \Omega'; \xi}{\alpha :_\rho \forall\, \Delta'.\kappa, \Omega \hookrightarrow \Delta \rightsquigarrow \alpha :_\rho \forall\, \Delta, \Delta'.\kappa, \Omega'; \xi_0, \xi} \text{ IGen\_TyVar}$$

$$\frac{\xi_0 = \iota \mapsto \mathsf{dom}(\Delta) \qquad \Omega[\xi_0] \hookrightarrow \Delta \rightsquigarrow \Omega'; \xi}{\iota : \forall\, \Delta'.\phi, \Omega \hookrightarrow \Delta \rightsquigarrow \iota : \forall\, \Delta, \Delta'.\phi, \Omega'; \xi_0, \xi} \text{ IGen\_CoVar}$$

Figure 6.5: Bake's generalization operation

It might be tempting at this point simply to return a mixed telescope of unification variables and Pico variables, and just to carry on. The problem here is that we will lose track of the local scope of x. Perhaps something later, outside of the λ-expression, will end up unifying with x—which would be a disaster. No, we must get rid of it.

The solution is to generalize Ω over x. This operation is written $\Omega \hookrightarrow x{:}_{\mathsf{Rel}}\mathbf{Type} \rightsquigarrow \Omega'; \xi$. (The mnemonic behind the choice of $\hookrightarrow$ is that we are essentially moving the $x{:}_{\mathsf{Rel}}\mathbf{Type}$ binding to the right, past Ω.) The output unification telescope Ω' binds the same unification variables as Ω, but each one will be generalized with respect to x. The definition of this judgment appears in Figure 6.5. The rules are a bit complicated by the fact that we may generalize a unification variable binding multiple times; both recursive rules thus assume a telescope Δ' that has already been generalized.

The new construct ξ is a *generalizer*. It is a substitution-like construct that maps unification variables to vectors, which you may recall are lists of arguments $\overline{\psi}$. In this case, we simply use the domain of Δ as the vector, where my use of $\mathsf{dom}(\Delta)$ as a list of arguments means to insert the irrelevance braces around irrelevantly bound variables. Generalizers are necessary because generalizing changes the type of unification variables; we must then change the occurrences of them as well.

Generalizers operate like this:

Definition (Generalizing [Definition E.31])**.** *A generalizer is applied postfix as a function. It operates homomorphically on all recursive forms and as the identity operation on leaves other than unification variables. Generalizing unification variables is defined by these equations:*

$$\begin{array}{rcl} \alpha \mapsto \overline{\psi}_1 \in \xi & \Rightarrow & \alpha_{\overline{\psi}_2}[\xi] = \alpha_{\overline{\psi}_1, \overline{\psi}_2} \\ \textit{otherwise} & & \alpha_{\overline{\psi}}[\xi] = \alpha_{\overline{\psi}[\xi]} \\ \iota \mapsto \overline{\psi}_1 \in \xi & \Rightarrow & \iota_{\overline{\psi}_2}[\xi] = \iota_{\overline{\psi}_1, \overline{\psi}_2} \\ \textit{otherwise} & & \iota_{\overline{\psi}}[\xi] = \iota_{\overline{\psi}[\xi]} \end{array}$$

Just like the generalization judgment (Figure 6.5), the generalization operation $[\xi]$ prepends the newly generalized variables to those already there.

$\Sigma; \Psi \overset{\rightarrow}{\vdash}_{\mathsf{ty}} \mathrm{t} \rightsquigarrow \tau : \kappa \dashv \Omega$	synthesize a type (no invis. binders)
$\Sigma; \Psi \overset{\rightarrow}{\vdash}^{*}_{\mathsf{ty}} \mathrm{t} \rightsquigarrow \tau : \kappa \dashv \Omega$	synthesize a type
$\Sigma; \Psi \overset{\leftarrow}{\vdash}_{\mathsf{ty}} \mathrm{t} : \kappa \rightsquigarrow \tau \dashv \Omega$	check a type (no invis. binders)
$\Sigma; \Psi \overset{\leftarrow}{\vdash}^{*}_{\mathsf{ty}} \mathrm{t} : \kappa \rightsquigarrow \tau \dashv \Omega$	check a type
$\Sigma; \Psi \overset{\rightarrow}{\vdash}_{\mathsf{pt}} \mathrm{s} \rightsquigarrow \tau \dashv \Omega$	check a polytype (always with kind **Type**)
$\Sigma; \Psi; \rho \overset{*}{\vdash}_{\mathsf{arg}} \mathrm{t} : \kappa \rightsquigarrow \psi; \tau \dashv \Omega$	check an argument at relevance ρ
$\Sigma; \Psi; \kappa_0; \tau_0 \overset{\rightarrow}{\vdash}_{\mathsf{alt}} \mathrm{alt} : \kappa \rightsquigarrow \mathit{alt} \dashv \Omega$	check a case alt. against an unknown type
$\Sigma; \Psi; \kappa_0; \tau_0 \overset{\rightarrow}{\vdash}_{\mathsf{altc}} \mathrm{alt} : \kappa \rightsquigarrow \mathit{alt} \dashv \Omega$	check a case alt. against a known type
$\Sigma; \Psi \overset{\rightarrow}{\vdash}_{\mathsf{q}} \mathrm{qvar} \rightsquigarrow a : \kappa; \nu \dashv \Omega$	synth. type of a bound var.
$\Sigma; \Psi \overset{\rightarrow}{\vdash}_{\mathsf{aq}} \mathrm{aqvar} \rightsquigarrow a : \kappa \dashv \Omega$	synth. type of a bound var. (w/o vis. marker)
$\Sigma; \Psi \overset{\rightarrow}{\vdash}_{\mathsf{aq}} \mathrm{aqvar} : \kappa \rightsquigarrow a : \kappa'; x.\tau \dashv \Omega$	check type of a bound var. (w/o vis. marker)
$\overset{\rightarrow}{\vdash}_{\mathsf{pi}} \mathrm{quant} \rightsquigarrow \Pi; \rho$	interpret a quantifier
$\overset{\rightarrow}{\vdash}_{\mathsf{fun}} \kappa; \rho_1 \rightsquigarrow \gamma; \Pi; a; \rho_2; \kappa_1; \kappa_2 \dashv \Omega$	extract components of a function type
$\Sigma; \Psi \overset{\rightarrow}{\vdash}_{\mathsf{scrut}} \overline{\mathrm{alt}}; \kappa \rightsquigarrow \gamma; \Delta; H; \overline{\tau} \dashv \Omega$	extract components of a scrutinee type
$\overset{\nu}{\vdash}_{\mathsf{inst}} \kappa \rightsquigarrow \overline{\psi}; \kappa' \dashv \Omega$	instantiate a type
$\Sigma; \Gamma \overset{\rightarrow}{\vdash}_{\mathsf{decl}} \mathrm{decl} \rightsquigarrow x : \kappa := \tau$	check a declaration
$\Sigma; \Gamma \overset{\rightarrow}{\vdash}_{\mathsf{prog}} \mathrm{prog} \rightsquigarrow \Gamma'; \theta$	check a program

Figure 6.6: Bake judgments

6.6 Type inference algorithm

The schema of the judgments that define Bake appear in Figure 6.6. I will not walk through each rule of each judgment to explain its inner workings. As discussed in the introduction to this chapter, the individual rules are largely predictable. They can be reviewed in their entirety in Appendix D, and the statements of lemmas that assert the soundness of many of these judgments appear in Section 6.8.1.4. Instead of a thorough review of the algorithm, this section will call out individual rules with interesting characteristics.

6.6.1 Function application

As discussed above (Section 6.4) function applications can only synthesize their type. The two rules for synthesizing the type of a function application (one for regular application and one for application with @) appear in Figure 6.7 on the next page, along with auxiliary judgments.

Walking through the ITy_App rule, we see that Bake first infers the type κ_0 for the Haskell expression t_1, elaborating t_1 to become τ_1 and producing a unification telescope Ω_1. The type for τ_1, though, might not manifestly be a function. This would happen, for example, when inferring the type of $\lambda x\ y \to x\ y$, where the type initially assigned to x is just a unification variable α. Instead of writing κ_0 as a function, Bake

$$\frac{\begin{array}{l}\Sigma; \Psi \overset{\rightarrow}{\vdash}_{\mathsf{ty}} \mathrm{t}_1 \rightsquigarrow \tau_1 : \kappa_0 \dashv \Omega_1 \\ \overset{\rightarrow}{\vdash}_{\mathsf{fun}} \kappa_0; \mathsf{Rel} \rightsquigarrow \gamma; \Pi; a; \rho; \kappa_1; \kappa_2 \dashv \Omega_2 \\ \Sigma; \Psi, \Omega_1, \Omega_2; \rho \overset{*}{\rightarrow}_{\mathsf{arg}} \mathrm{t}_2 : \kappa_1 \rightsquigarrow \psi_2; \tau_2 \dashv \Omega_3\end{array}}{\Sigma; \Psi \overset{*}{\rightarrow}_{\mathsf{ty}} \mathrm{t}_1\, \mathrm{t}_2 \rightsquigarrow (\tau_1 \rhd \gamma)\, \psi_2 : \kappa_2[\tau_2/a] \dashv \Omega_1, \Omega_2, \Omega_3} \quad \textsc{ITy\_App}$$

$$\frac{\begin{array}{l}\Sigma; \Psi \overset{*}{\rightarrow}_{\mathsf{ty}} \mathrm{t}_1 \rightsquigarrow \tau_1 : \Pi_{\mathsf{Spec}}\, a{:}_\rho \kappa_1.\, \kappa_2 \dashv \Omega_1 \\ \Sigma; \Psi, \Omega_1; \rho \overset{*}{\rightarrow}_{\mathsf{arg}} \mathrm{t}_2 : \kappa_1 \rightsquigarrow \psi_2; \tau_2 \dashv \Omega_2\end{array}}{\Sigma; \Psi \overset{*}{\rightarrow}_{\mathsf{ty}} \mathrm{t}_1\, @\mathrm{t}_2 \rightsquigarrow \tau_1\, \psi_2 : \kappa_2[\tau_2/a] \dashv \Omega_1, \Omega_2} \quad \textsc{ITy\_AppSpec}$$

$\boxed{\overset{\rightarrow}{\vdash}_{\mathsf{fun}} \kappa; \rho_1 \rightsquigarrow \gamma; \Pi; a; \rho_2; \kappa_1; \kappa_2 \dashv \Omega}$ Extract out the parts of a function kind.

$$\frac{}{\overset{\rightarrow}{\vdash}_{\mathsf{fun}} \Pi_{\mathsf{Req}}\, a{:}_\rho \kappa_1.\, \kappa_2; \rho_0 \rightsquigarrow \langle \Pi_{\mathsf{Req}}\, a{:}_\rho \kappa_1.\, \kappa_2 \rangle; \Pi; a; \rho; \kappa_1; \kappa_2 \dashv \varnothing} \quad \textsc{IFun\_Id}$$

$$\frac{\begin{array}{ll}\mathsf{fresh}\, \iota & \mathsf{fresh}\, \beta_1, \beta_2 \\ \Omega = \beta_1{:}_{\mathsf{Irrel}}\mathbf{Type}, \beta_2{:}_{\mathsf{Irrel}}\mathbf{Type}, \iota{:}\kappa_0 \sim \underset{\sim}{\Pi}_{\mathsf{Req}}\, a{:}_\rho \beta_1.\, \beta_2 & \end{array}}{\overset{\rightarrow}{\vdash}_{\mathsf{fun}} \kappa_0; \rho \rightsquigarrow \iota; \underset{\sim}{\Pi}; a; \rho; \beta_1; \beta_2 \dashv \Omega} \quad \textsc{IFun\_Cast}$$

$\boxed{\Sigma; \Psi; \rho \overset{*}{\rightarrow}_{\mathsf{arg}} \mathrm{t} : \kappa \rightsquigarrow \psi; \tau \dashv \Omega}$ Check a function argument against its known type.

$$\frac{\Sigma; \Psi \overset{*}{\rightarrow}_{\mathsf{ty}} \mathrm{t} : \kappa \rightsquigarrow \tau \dashv \Omega}{\Sigma; \Psi; \mathsf{Rel} \overset{*}{\rightarrow}_{\mathsf{arg}} \mathrm{t} : \kappa \rightsquigarrow \tau; \tau \dashv \Omega} \quad \textsc{IArg\_Rel}$$

$$\frac{\Sigma; \mathsf{Rel}(\Psi) \overset{*}{\rightarrow}_{\mathsf{ty}} \mathrm{t} : \kappa \rightsquigarrow \tau \dashv \Omega}{\Sigma; \Psi; \mathsf{Irrel} \overset{*}{\rightarrow}_{\mathsf{arg}} \mathrm{t} : \kappa \rightsquigarrow \{\tau\}; \tau \dashv \Omega} \quad \textsc{IArg\_Irrel}$$

Figure 6.7: Function applications in Bake

instead uses its $\overset{\rightarrow}{\vdash}_{\mathsf{fun}}$ judgment, which extracts out the component parts of a function type.

It may be helpful in understanding the $\overset{\rightarrow}{\vdash}_{\mathsf{fun}}$ judgment to see its correctness property, as proved in Section E.9:

Lemma (Function position [Lemma E.37])**.** *If* $\Sigma; \Psi \vDash_{\mathsf{ty}} \kappa : \mathbf{Type}$ *and* $\overset{\rightarrow}{\vdash}_{\mathsf{fun}} \kappa; \rho_1 \rightsquigarrow \gamma; \Pi; a; \rho_2; \kappa_1; \kappa_2 \dashv \Omega$*, then* $\Sigma; \Psi, \Omega \vDash_{\mathsf{co}} \gamma : \kappa \sim \Pi_{\mathsf{Req}}\, a{:}_{\rho_2} \kappa_1.\, \kappa_2$.

We can see here that $\overset{\rightarrow}{\vdash}_{\mathsf{fun}}$ produces a coercion γ that relates the input type κ to the output type $\Pi a{:}_{\rho_2} \kappa_1.\, \kappa_2$. The input relevance ρ_1 is to be used as a default—Bake will assume that a function uses its argument relevantly unless told otherwise. Note that relevance of arguments is not denoted in the user-written source code.

Looking at the definition of $\overset{\rightarrow}{\vdash}_{\mathsf{fun}}$, we see two cases:

- If the input type κ is manifestly a Π-type, Bake just returns its component pieces along with a reflexive coercion.

- Otherwise, it invents fresh unification variables as emits a constraint relating this variables to the input.

It might be tempting to define $\overset{}{\vdash}_{\mathsf{fun}}$ only by the second rule, IFun_Cast, but this would greatly weaken Bake's power. Doing so would mean that the bidirectional algorithm would never be able to take advantage of knowing a function's argument type. Furthermore, note that β_2, the result type of the function in IFun_Cast, is not generalized with respect to $a{:}_\rho\beta_1$; a function type inferred via IFun_Cast will surely be non-dependent. This decision was made in keeping with the guiding principle that only simple types should be inferred.

Once we have extracted the component parts of the function type, we can check the argument with the $\overset{*}{\vdash}_{\mathsf{arg}}$ judgment. This judgment takes the relevance of the argument as an input; it simply uses the $\overset{*}{\vdash}_{\mathsf{ty}}$ checking judgment and insert braces as appropriate.

Contrast the behavior of ITy_App to that of ITy_AppSpec, which, crucially, does not use $\vdash_{\mathsf{fun}}$. Consider what would happen if the function's type is not manifestly a Π-type. We could, like in IFun_Cast invent unification variables and emit a constraint. But this would mean that the argument is *inferred*, not *specified*. Using an inferred argument with a visibility override violates the inference principles set forth by Eisenberg et al. [33] and would surely eliminate the possibility of principal types. Accordingly, ITy_AppSpec avoids such behavior and simply looks to make sure that the function's type is of the appropriate shape. If it is not, Bake issues an error.

6.6.2 Mediating between checking and synthesis

The two modes of Bake meet head-on when we are checking an expression (such as a function application) that has no rules in the checking judgment. The fall-through case of the checking judgment is this rule:

$$\frac{\begin{array}{l}\Sigma; \Psi \overset{*}{\vdash}_{\mathsf{ty}} \mathrm{t} \rightsquigarrow \tau : \kappa_1 \dashv \Omega \\ \vdash_{\mathsf{pre}} \kappa_2 \rightsquigarrow \Delta; \kappa_2'; \tau_2 \\ \Omega \hookrightarrow \Delta \rightsquigarrow \Omega'; \xi_1 \\ \kappa_1[\xi_1] \leq^* \kappa_2' \rightsquigarrow \tau_2' \dashv \Omega_2 \\ \Omega_2 \hookrightarrow \Delta \rightsquigarrow \Omega_2'; \xi_2\end{array}}{\Sigma; \Psi \vdash_{\mathsf{ty}} \mathrm{t} : \kappa_2 \rightsquigarrow \tau_2\,(\lambda\Delta.\, \tau_2'[\xi_2]\,\tau[\xi_1]) \dashv \Omega', \Omega_2'} \quad \text{ITyC\_Infer}$$

We are checking that t has type κ_2. First, Bake synthesizes t's type κ_1, producing unification telescope Ω. We then must, as described in Section 6.4.3, deeply skolemize κ_2. Pulling out the quantifiers in κ_2 (according to the $\vdash_{\mathsf{pre}}$ judgment) gives us $\underset{\sim}{\Pi}\Delta.\, \kappa_2'$. We then *generalize* Ω by Δ. It is this generalization step that allows the solver to solve unification variables in Ω with skolems in Δ and allows the example from Section 6.4.3 to be accepted. Having generalized, we then do the subsumption check. We now must generalize Ω_2, the output unification telescope from the subsumption check, as Ω_2 might refer to skolems bound in Δ.

Once again, the key interesting part of this rule is the first generalization step. It is not necessary to do this in order to get correct elaboration, but the analysis in my prior work [33, end of Section 6.1] suggests that this is necessary in order to have principal types.

6.6.3 case expressions

We see in Figure 6.6 on page 160 that there are two judgments for checking **case** alternatives. These correspond to the two rules for checking **case** expressions, one for synthesis (ITy_Case) and one for checking (ITyC_Case). I refrain from including the actual rules here, as their myriad and ornate details would be distracting; the overly curious can see Appendix D for these details.

As discussed previously (Sections 4.3.3 and 6.4), a **case** expression is treated differently depending on whether we can know its result type. In the case where we do not (ITy_Case), Bake invents a new unification variable β for the result type and checks each case alternative against it. This is why the $\vdash\!\!\!\rightarrow_{\mathsf{alt}}$ judgment takes a result type, even though it is used during synthesis. After all, we do require all alternatives to produce the *same* result type. Producing the unification variable within each alternative would risk running into a skolem escape, whereby the result type might mention a variable locally bound within the alternative. It is simpler just to propagate the β down into $\vdash\!\!\!\rightarrow_{\mathsf{alt}}$. The $\vdash\!\!\!\rightarrow_{\mathsf{alt}}$ judgment, in turn, does not use the equality gotten from dependent pattern matching when checking alternatives. Recall that doing so during synthesis mode would cause trouble because the equality assumption would make the β unification variable untouchable when solving constraints emitted while processing the alternatives.

On the other hand, the $\vdash\!\!\!\rightarrow_{\mathsf{altc}}$ judgment is used from ITyC_Case, in checking mode. This judgment is almost identical to $\vdash\!\!\!\rightarrow_{\mathsf{alt}}$ except that it allows the alternatives to make use of the dependent-pattern-match equality.

6.6.4 Checking λ-expressions

Consider checking this expression:

$$(\lambda(f :: \mathit{Int} \to \mathit{Int}) \to f\ 5) :: (\forall\ a.\ a \to a) \to \mathit{Int} \tag{6.6.1}$$

This expression should be accepted. The λ takes a function over *Int*s and applies it. The type signature then says that the λ should actually be applicable to any polymorphic endofunction. Of course, such a function can be specialized to *Int*, so all is well. Indeed, the expression above is accepted by GHC.

The example above, however, is not dependent. Surprisingly, the intuition in the above paragraph does not generalize to the dependent case. Consider this (contrived)

example:

$$(\lambda(f :: \mathit{Bool} \to \mathit{Bool}) \to P) :: \Pi\ (g :: \forall\ a.\ a \to a) \to \mathit{Proxy}\ '(g\ 5, g\ '\mathit{True}) \qquad (6.6.2)$$

where we have

```
data Proxy :: ∀ k. k → Type where
  P :: ∀ k (a :: k). Proxy a
  -- equivalent to data Proxy a = P
```

Once again, the annotation on the λ argument is a specialized version of the argument's type as given in the type signature. And yet, this expression must be rejected.

One way to boil this problem down is to consider what type we check the expression P against. When we are checking P, we clearly have $f :: \mathit{Bool} \to \mathit{Bool}$ in scope. Yet the natural type to check P against is $\mathit{Proxy}\ '(g\ 5, g\ '\mathit{True})$, which mentions g, not f. Even if the names were to be fixed, we would still have the problem that $g\ 5$ is certainly not well typed if g has type $\mathit{Bool} \to \mathit{Bool}$. We are stuck.

Another way to see this problem is to think about elaborating the subsumption judgment. In example (6.6.1), type inference will check whether $\forall\ a.\ a \to a \leq \mathit{Int} \to \mathit{Int}$. When it discovers that this is true, the subsumption algorithm will also produce a function that takes something of type $\forall\ a.\ a \to a$ to something of type $\mathit{Int} \to \mathit{Int}$. If the expression in example (6.6.1) is applied to an argument (naturally, of type $\forall\ a.\ a \to a$), then this conversion function readies the argument to pass to the λ-expression.

In example (6.6.2), however, we need conversions both ways. We still need the conversion from $\forall\ a.\ a \to a$ to $\mathit{Bool} \to \mathit{Bool}$, for exactly the same reason that we need it for example (6.6.1). We also need the conversion in the other direction (in this case, the impossible conversion from $\mathit{Bool} \to \mathit{Bool}$ to $\forall\ a.\ a \to a$) when checking that P, with $f :: \mathit{Bool} \to \mathit{Bool}$ in scope, has type $\mathit{Proxy}\ '(g\ 5, g\ '\mathit{True})$, using $g :: \forall\ a.\ a \to a$.

The solution to this is to have two separate rules, one in the non-dependent case and one in the dependent case. Bake looks at the type being checked against (let's call it τ). If τ uses its argument dependently, then Bake requires that the annotation on the λ argument and the function type as found in τ can be proved equal—that is, that there is a coercion between them. Otherwise, we use subsumption, just as in example (6.6.1). You can view the two rules in Section D.5; as usual, the rules are a bit cluttered to present here.

6.7 Program elaboration

Up until now, this chapter has focused more on the gate-keeping services provided by Bake, preventing ill formed programs from being accepted. In this section, we will discuss elaboration, the process of creating the Pico program that corresponds to an input Haskell program. Let's look in particular on the highest levels of elaboration, processing Haskell declarations and programs. See Figure 6.8 on the next page for the two judgments of interest.

$\boxed{\Sigma; \Gamma \mathrel{\overset{\rightarrow}{\vdash}}_{\mathsf{decl}} \mathrm{decl} \rightsquigarrow x : \kappa := \tau}$ Check a Haskell declaration.

$$\frac{\begin{array}{l}\Sigma; \Gamma \mathrel{\overset{\rightarrow}{\vdash}}_{\mathsf{ty}} \mathrm{t} \rightsquigarrow \tau : \kappa \dashv \Omega \\ \Sigma; \Gamma \mathrel{\overset{\rightarrow}{\vdash}}_{\mathsf{solv}} \Omega \rightsquigarrow \Delta; \Theta \\ \tau' = \lambda \Delta.\, (\tau[\Theta]) \qquad \kappa' = \underset{\sim}{\Pi}_{\mathsf{Inf}} \Delta.\, (\kappa[\Theta])\end{array}}{\Sigma; \Gamma \mathrel{\overset{\rightarrow}{\vdash}}_{\mathsf{decl}} x := \mathrm{t} \rightsquigarrow x : \kappa' := \tau'} \quad \textsc{IDecl\_Synthesize}$$

$$\frac{\begin{array}{l}\Sigma; \Gamma \mathrel{\overset{\rightarrow}{\vdash}}_{\mathsf{pt}} \mathrm{s} \rightsquigarrow \sigma \dashv \Omega_1 \\ \Sigma; \mathsf{Rel}(\Gamma) \mathrel{\overset{\rightarrow}{\vdash}}_{\mathsf{solv}} \mathsf{Rel}(\Omega_1) \rightsquigarrow \Delta_1; \Theta_1 \\ \sigma' = \underset{\sim}{\Pi}_{\mathsf{Inf}} \Delta_1.\, (\sigma[\Theta_1]) \\ \Sigma; \Gamma \mathrel{\overset{*}{\underset{\rightarrow}{\vdash}}}_{\mathsf{ty}} \mathrm{t} : \sigma' \rightsquigarrow \tau \dashv \Omega_2 \\ \Sigma; \Gamma \mathrel{\overset{\rightarrow}{\vdash}}_{\mathsf{solv}} \Omega_2 \rightsquigarrow \varnothing; \Theta_2 \\ \tau' = \tau[\Theta_2]\end{array}}{\Sigma; \Gamma \mathrel{\overset{\rightarrow}{\vdash}}_{\mathsf{decl}} x :: \mathrm{s} := \mathrm{t} \rightsquigarrow x : \sigma' := \tau'} \quad \textsc{IDecl\_Check}$$

$\boxed{\Sigma; \Gamma \mathrel{\overset{\rightarrow}{\vdash}}_{\mathsf{prog}} \mathrm{prog} \rightsquigarrow \Gamma'; \theta}$ Check a Haskell program.

$$\frac{}{\Sigma; \Gamma \mathrel{\overset{\rightarrow}{\vdash}}_{\mathsf{prog}} \varnothing \rightsquigarrow \varnothing; \varnothing} \quad \textsc{IProg\_Nil}$$

$$\frac{\begin{array}{c}\Sigma; \Gamma \mathrel{\overset{\rightarrow}{\vdash}}_{\mathsf{decl}} \mathrm{decl} \rightsquigarrow x : \kappa := \tau \\ \Sigma; \Gamma, x{:}_{\mathsf{Rel}}\kappa, c{:}x \sim \tau \mathrel{\overset{\rightarrow}{\vdash}}_{\mathsf{prog}} \mathrm{prog} \rightsquigarrow \Gamma'; \theta\end{array}}{\Sigma; \Gamma \mathrel{\overset{\rightarrow}{\vdash}}_{\mathsf{prog}} \mathrm{decl}; \mathrm{prog} \rightsquigarrow x{:}_{\mathsf{Rel}}\kappa, c{:}x \sim \tau, \Gamma'; (\tau/x, \langle\tau\rangle/c) \circ \theta} \quad \textsc{IProg\_Decl}$$

Figure 6.8: Elaborating declarations and programs

6.7.1 Declarations

The $\mathrel{\overset{\rightarrow}{\vdash}}_{\mathsf{decl}}$ judgment processes the two forms of declaration included in the Haskell subset formalized here: unannotated variable declarations and annotated variable declarations. It outputs the name of the new variable, its type κ and its value τ. Note that the environment used in $\mathrel{\overset{\rightarrow}{\vdash}}_{\mathsf{decl}}$ is $\Sigma; \Gamma$, with a context containing only Pico variables, no unification variables. These are top-level declarations only.

Rule IDecl_Synthesize simply ties together the pieces of using the synthesis judgment and the solver. Note that the definitions of τ' and κ' in the rule generalize over the telescope Δ produced by the solver, and that the Π-type formed marks the binders as inferred, never specified.

Rule IDecl_Check is a bit more involved. It first must check the type signature using the $\mathrel{\overset{\rightarrow}{\vdash}}_{\mathsf{pt}}$ judgment, to make sure s it is a well formed polytype. This process might emit constraints, and we must solve these before tackling the term-level expression. This would happen, for example, in the type $\forall$ *a*. *Proxy a* $\rightarrow$ (), as *a*'s kind is unspecified. The solver may produce a telescope Δ_1 to generalize by. In our example, this telescope would include $k{:}_{\mathsf{Irrel}}\mathbf{Type}$, the type of *a*. Having sorted out the type

signature, we can now proceed to the expression t, which is checked against σ' the generalized Pico translation of the user-written polytype s. We must solve once again. In this invocation of the solver, we insist that no further generalization be done because the user has already written the entire type of the expression. This decision is in keeping with standard Haskell, where a declaration like

$$\begin{aligned} &\mathit{bad} :: a \to \mathit{String} \\ &\mathit{bad}\ x = \mathit{show}\ x \end{aligned}$$

is rejected, because accepting the function body requires generalizing over an extra *Show a* constraint.

6.7.2 Programs

The elaboration of whole programs is generally straightforward. This algorithm appears in Figure 6.8 on the preceding page. The judgment $\Sigma; \Gamma \vdash\!\!\!\Rightarrow_{\mathsf{prog}} \mathrm{prog} \rightsquigarrow \Gamma'; \theta$ produces as output an extension to the context, Γ', as well as a closing substitution θ which maps the newly bound variable to its definition. (Recall that this formalization of Bake ignores recursion; thus no variable can be mentioned in its own declaration.)

The one non-trivial rule, IProg_Decl, checks a declaration and then incorporates this declaration into the context Γ used to check later declarations. There is one small twist here, though: because declared variables can be used in types as well as in terms, we wish the typing context to remember the equality between the variable and its definition. This is done via the coercion variable c included in the context in the second premise to IProg_Decl.

6.8 Metatheory

This chapter has explained the Bake algorithm in some detail, but what theoretical properties does it have? A type inference algorithm is often checked for soundness and completeness against a specification. However, as argued by Vytiniotis et al. [99, Section 6.3], lining up an algorithm such as Bake against a declarative specification is a challenge. Instead of writing a separate, non-algorithmic form of Bake, I present three results in this section:

- I prove that the elaborated Pico program produced by Bake is indeed a well typed Pico program. This result—which I call soundness—marks an upper limit on the set of programs that Bake accepts. If it cannot be typed in Pico, Bake must reject.[93]

[93] I do not prove a correspondence between the Haskell program and the Pico program produced by elaboration. It would thus theoretically be possible to design Bake to accept all input texts and produce a trivial elaborated program. But that wouldn't be nearly as much fun, and I have not done so.

- In two separate subsections, I argue that BAKE is a conservative extension both of the OUTSIDEIN algorithm and the SB algorithm of Eisenberg et al. [33]. That is, if OUTSIDEIN or SB accepts a program, so does BAKE. This results suggests that a version of GHC based on BAKE will accept all Haskell programs currently accepted. These arguments—I dare not quite call them proofs—are stated in less formal terms than other proofs in this dissertation. While it is likely possible to work out the details fully, the presentation of the other systems and of BAKE/PICO differ enough that the translation between the systems would be fiddly, and artifacts of the translation would obscure the main point. The individual differences are discussed below.

  These conservativity results provide a lower bound on the power of BAKE, declaring that some set of Haskell programs must be accepted by the algorithm.

The results listed above bound the power of the algorithm both from below and from above, serving roughly as soundness and completeness results. It is left as future work to define a precise specification of BAKE and prove that it meets the specification.

6.8.1 Soundness

Here is the fundamental soundness result:

Theorem (Soundness of BAKE elaboration [Theorem E.44]). *If* $\Sigma \vdash_{\mathsf{ctx}} \Gamma\ \mathsf{ok}$ *and* $\Sigma; \Gamma \Vdash_{\mathsf{prog}} \text{prog} \rightsquigarrow \Gamma'; \theta$*, then:*

1. $\Sigma \vdash_{\mathsf{ctx}} \Gamma, \Gamma'\ \mathsf{ok}$
2. $\Sigma; \Gamma \vdash_{\mathsf{subst}} \theta : \Gamma'$
3. $\mathsf{dom}(\text{prog}) \subseteq \mathsf{dom}(\Gamma')$

This theorem assumes that the starting environment is well formed $\Sigma \vdash_{\mathsf{ctx}} \Gamma\ \mathsf{ok}$ and that BAKE accepts the source language program prog. In return, the theorem claims that the context extension Γ' is well formed (assuming it is appended after Γ), that the substitution θ is a valid closing substitution (see below), and that indeed the new context Γ' binds the variables declared in prog.

Closing substitutions are recognized by the new judgment $\vdash_{\mathsf{subst}}$, which appears in Figure 6.9 on the next page. (Note the turnstile $\vdash$; this is a pure PICO judgment with no unification variables in sight.) It uses a new notation $\theta|_{\overline{z}}$ which restricts the domain of a substitution θ to operate only on the variables $\overline{z}$. Informally, $\Sigma; \Gamma \vdash_{\mathsf{subst}} \theta : \Delta$ holds when the substitution θ eliminates the appearance of any of the variables in Δ. Here is the key lemma that asserts the correctness of the judgment:

Lemma (Closing substitution [Lemma E.30]). *If* $\Sigma; \Gamma \vdash_{\mathsf{subst}} \theta : \Delta$ *and* $\Sigma; \Gamma, \Delta, \Gamma' \vdash \mathcal{J}$*, then* $\Sigma; \Gamma, \Gamma'[\theta|_{\mathsf{dom}(\Delta)}] \vdash \mathcal{J}[\theta|_{\mathsf{dom}(\Delta)}]$.

$\boxed{\Sigma; \Gamma \vdash_{\mathsf{subst}} \theta : \Delta}$ "θ substitutes the variables in Δ away."

$$\frac{}{\Sigma; \Gamma \vdash_{\mathsf{subst}} \theta : \varnothing} \textsc{Subst\_Nil}$$

$$\frac{\Sigma; \Gamma \vdash_{\mathsf{ty}} a[\theta] : \kappa \quad \Sigma; \Gamma \vdash_{\mathsf{subst}} \theta : \Delta[\theta|_a]}{\Sigma; \Gamma \vdash_{\mathsf{subst}} \theta : a{:}_{\mathsf{Rel}}\kappa, \Delta} \textsc{Subst\_TyRel}$$

$$\frac{\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{ty}} a[\theta] : \kappa \quad \Sigma; \Gamma \vdash_{\mathsf{subst}} \theta : \Delta[\theta|_a]}{\Sigma; \Gamma \vdash_{\mathsf{subst}} \theta : a{:}_{\mathsf{Irrel}}\kappa, \Delta} \textsc{Subst\_TyIrrel}$$

$$\frac{\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{co}} c[\theta] : \phi \quad \Sigma; \Gamma \vdash_{\mathsf{subst}} \theta : \Delta[\theta|_c]}{\Sigma; \Gamma \vdash_{\mathsf{subst}} \theta : c{:}\phi, \Delta} \textsc{Subst\_Co}$$

Figure 6.9: Validity of closing substitutions

Here, I use a notation where $\mathcal{J}$ stands for a judgment chosen from $\vdash_{\mathsf{ty}}$, $\vdash_{\mathsf{co}}$, $\vdash_{\mathsf{prop}}$, $\vdash_{\mathsf{alt}}$, $\vdash_{\mathsf{vec}}$, $\vdash_{\mathsf{ctx}}$, or $\vdash_{\mathsf{s}}$.

The use of $\vdash_{\mathsf{subst}}$ in the conclusion of the elaboration soundness theorem means that the variable values stored in θ actually have the types as given in Γ'.

Naturally, proving this theorem requires proving the soundness of all the individual judgments that form Bake. These proofs all appear in Section E.9.

6.8.1.1 Adapting lemmas on $\vdash$ to $\vDash$

The first step in establishing the soundness result is to ensure that the structural lemmas proved for $\vdash$ judgments still hold over the $\vDash$ judgments. While doing this for the definitions as given does not pose a challenge, it is in getting these proofs to work that all of the complications around unification variables (to wit, zonkers and generalizers) arise.

Relating the two sets of judgments is accomplished by this key lemma:

Lemma (Extension [Lemma E.3])**.** $\Sigma; \Gamma \vdash \mathcal{J}$ *if and only if* $\Sigma; \Gamma \vDash \mathcal{J}$.

Note that the context must contain only Pico variables, never unification variables. This fact is what allows the larger $\Sigma; \Gamma \vDash \mathcal{J}$ to imply the smaller $\Sigma; \Gamma \vdash \mathcal{J}$.

$\boxed{\Sigma; \Psi \models_{\mathsf{z}} \Theta : \Omega}$ "Θ zonks all the unification variables in Ω."

$$\frac{}{\Sigma; \Psi \models_{\mathsf{z}} \varnothing : \varnothing} \quad \textsc{Zonk\_Nil}$$

$$\frac{\begin{array}{c}\Sigma; \Psi, \Delta \models_{\mathsf{ty}} \tau : \kappa \\ \Sigma; \Psi \models_{\mathsf{z}} \Theta : \Omega[\forall\, \mathsf{dom}(\Delta).\tau/\alpha]\end{array}}{\Sigma; \Psi \models_{\mathsf{z}} \forall\, \mathsf{dom}(\Delta).\tau/\alpha, \Theta : \alpha :_{\mathsf{Rel}} \forall\, \Delta.\kappa, \Omega} \quad \textsc{Zonk\_TyVarRel}$$

$$\frac{\begin{array}{c}\Sigma; \mathsf{Rel}(\Psi, \Delta) \models_{\mathsf{ty}} \tau : \kappa \\ \Sigma; \Psi \models_{\mathsf{z}} \Theta : \Omega[\forall\, \mathsf{dom}(\Delta).\tau/\alpha]\end{array}}{\Sigma; \Psi \models_{\mathsf{z}} \forall\, \mathsf{dom}(\Delta).\tau/\alpha, \Theta : \alpha :_{\mathsf{Irrel}} \forall\, \Delta.\kappa, \Omega} \quad \textsc{Zonk\_TyVarIrrel}$$

$$\frac{\begin{array}{c}\Sigma; \Psi, \Delta \models_{\mathsf{co}} \gamma : \phi \\ \Sigma; \Psi \models_{\mathsf{z}} \Theta : \Omega[\forall\, \mathsf{dom}(\Delta).\gamma/\iota]\end{array}}{\Sigma; \Psi \models_{\mathsf{z}} \forall\, \mathsf{dom}(\Delta).\gamma/\iota, \Theta : \iota : \forall\, \Delta.\phi, \Omega} \quad \textsc{Zonk\_CoVar}$$

Figure 6.10: Zonker validity

6.8.1.2 Soundness of the solver

The solver $\Sigma; \Psi \models_{\mathsf{solv}} \Omega \rightsquigarrow \Delta; \Theta$ produces a generalization telescope and a zonker. In order to define a correctness property for this solver, we first need a judgment that asserts the validity of the zonker. This judgment appears in Figure 6.10. The judgment is quite similar to the judgment classifying closing substitutions ($\models_{\mathsf{subst}}$, in Figure 6.9 on the previous page), but it deals also with the complexity of having unification variables quantified over telescopes.

Naturally, we must require that the solver produce a valid zonker. We also require that the zonker be idempotent, as that is a necessary requirement to prove the zonking lemma, below. Here is the soundness property we are assuming of the solver. Note that this property is the *only* one we need to prove soundness of elaboration.

Property (Solver is sound [Property E.24])**.** *If* $\Sigma \models_{\mathsf{ctx}} \Psi, \Omega\ \mathsf{ok}$ *and* $\Sigma; \Psi \models_{\mathsf{solv}} \Omega \rightsquigarrow \Delta; \Theta$*, then* Θ *is idempotent,* $\Sigma \models_{\mathsf{ctx}} \Psi, \Delta\ \mathsf{ok}$*, and* $\Sigma; \Psi, \Delta \models_{\mathsf{z}} \Theta : \Omega$*.*

Lemma (Zonking [Lemma E.23])**.** *If* Θ *is idempotent,* $\Sigma; \Psi \models_{\mathsf{z}} \Theta : \Omega$*, and* $\Sigma; \Psi, \Omega, \Delta \vDash \mathcal{J}$*, then* $\Sigma; \Psi, \Delta[\Theta] \vDash \mathcal{J}[\Theta]$*.*

6.8.1.3 Soundness of generalization

The following lemma asserts the correctness of the generalization judgment:

Lemma (Generalization [Lemma E.35])**.** *If* $\Omega \hookrightarrow \Delta \rightsquigarrow \Omega'; \xi$ *and* $\Sigma; \Psi, \Delta, \Omega \vDash \mathcal{J}$*, then* $\Sigma; \Psi, \Omega', \Delta \vDash \mathcal{J}[\xi]$*.*

The proof of this lemma relies on the following smaller lemma (and its counterpart for coercion variables):

Lemma (Generalization by type variable [Lemma E.32]). *If* $\Sigma; \Psi, \Delta, \alpha :_\rho \forall\, \Delta'.\kappa, \Psi' \vDash \mathcal{J}$*, then* $\Sigma; \Psi, \alpha :_\rho \forall\, \Delta, \Delta'.\kappa, \Delta, \Psi'[\alpha \mapsto \mathsf{dom}(\Delta)] \vDash \mathcal{J}[\alpha \mapsto \mathsf{dom}(\Delta)]$.

6.8.1.4 Soundness lemmas for individual judgments

Lemma (Instantiation [Lemma E.36]). *If* $\Sigma; \Psi \vDash_{\mathsf{ty}} \tau : \kappa$ *and* $\overset{\nu}{\Rightarrow}_{\mathsf{inst}} \kappa \rightsquigarrow \overline{\psi}; \kappa' \dashv \Omega$*, then* $\Sigma; \Psi, \Omega \vDash_{\mathsf{ty}} \tau\, \overline{\psi} : \kappa'$ *and* κ' *is not a* Π*-type with a binder (with visibility* ν_2*) such that* $\nu_2 \leq \nu$.

Lemma (Scrutinee position [Lemma E.38]). *If* $\Sigma; \Psi \vDash_{\mathsf{ty}} \tau : \kappa$ *and* $\Sigma; \Psi \Rightarrow_{\mathsf{scrut}} \overline{\mathrm{alt}}; \kappa \rightsquigarrow \gamma; \Delta; H'; \overline{\tau} \dashv \Omega$*, then* $\Sigma; \Psi, \Omega \vDash_{\mathsf{ty}} \tau \rhd \gamma : \Pi\Delta.\, H'\, \overline{\tau}$ *and* $\Sigma; \mathsf{Rel}(\Psi, \Omega) \vDash_{\mathsf{ty}} H'\, \overline{\tau} : \mathbf{Type}$.

Lemma (Prenex [Lemma E.40]). *If* $\Sigma; \mathsf{Rel}(\Psi) \vDash_{\mathsf{ty}} \kappa : \mathbf{Type}$ *and* $\Rightarrow_{\mathsf{pre}} \kappa \rightsquigarrow \Delta; \kappa'; \tau$*, then* $\Sigma; \Psi \vDash_{\mathsf{ty}} \tau : \underset{\sim}{\Pi} x{:}_{\mathsf{Rel}}(\underset{\sim}{\Pi}\Delta.\, \kappa').\, \kappa$.

Lemma (Subsumption [Lemma E.41]). *Assume* $\Sigma; \mathsf{Rel}(\Psi) \vDash_{\mathsf{ty}} \kappa_1 : \mathbf{Type}$ *and* $\Sigma; \mathsf{Rel}(\Psi) \vDash_{\mathsf{ty}} \kappa_2 : \mathbf{Type}$*. If either*

1. $\kappa_1 \leq^* \kappa_2 \rightsquigarrow \tau \dashv \Omega$*, or*
2. $\kappa_1 \leq \kappa_2 \rightsquigarrow \tau \dashv \Omega$*,*

then $\Sigma; \Psi, \Omega \vDash_{\mathsf{ty}} \tau : \underset{\sim}{\Pi} x{:}_{\mathsf{Rel}} \kappa_1.\, \kappa_2$.

Lemma (Type elaboration is sound [Lemma E.42]).

1. *If any of the following:*
   (a) $\Sigma \vDash_{\mathsf{ctx}} \Psi\ \mathsf{ok}$ *and* $\Sigma; \Psi \Rightarrow_{\mathsf{ty}} \mathrm{t} \rightsquigarrow \tau : \kappa \dashv \Omega$*, or*
   (b) $\Sigma \vDash_{\mathsf{ctx}} \Psi\ \mathsf{ok}$ *and* $\Sigma; \Psi \overset{*}{\Rightarrow}_{\mathsf{ty}} \mathrm{t} \rightsquigarrow \tau : \kappa \dashv \Omega$*, or*
   (c) $\Sigma; \mathsf{Rel}(\Psi) \vDash_{\mathsf{ty}} \kappa : \mathbf{Type}$ *and* $\Sigma; \Psi \Rightarrow_{\mathsf{ty}} \mathrm{t} : \kappa \rightsquigarrow \tau \dashv \Omega$*, or*
   (d) $\Sigma; \mathsf{Rel}(\Psi) \vDash_{\mathsf{ty}} \kappa : \mathbf{Type}$ *and* $\Sigma; \Psi \overset{*}{\Rightarrow}_{\mathsf{ty}} \mathrm{t} : \kappa \rightsquigarrow \tau \dashv \Omega$*,*

   then $\Sigma; \Psi, \Omega \vDash_{\mathsf{ty}} \tau : \kappa$.
2. *If* $\Sigma \vDash_{\mathsf{ctx}} \Psi\ \mathsf{ok}$ *and* $\Sigma; \Psi \Rightarrow_{\mathsf{pt}} \mathrm{s} \rightsquigarrow \sigma \dashv \Omega$*, then* $\Sigma; \mathsf{Rel}(\Psi, \Omega) \vDash_{\mathsf{ty}} \sigma : \mathbf{Type}$.
3. *If* $\Sigma; \Psi \vDash_{\mathsf{ty}} \tau_1 : \Pi_\nu a{:}_\rho \kappa_1.\, \kappa_2$ *and* $\Sigma; \Psi; \rho \overset{*}{\Rightarrow}_{\mathsf{arg}} \mathrm{t}_2 : \kappa_1 \rightsquigarrow \psi_2; \tau_2 \dashv \Omega$*, then* $\Sigma; \Psi, \Omega \vDash_{\mathsf{ty}} \tau_1\, \psi_2 : \kappa_2[\tau_2/a]$.
4. *If* $\Sigma; \mathsf{Rel}(\Psi) \vDash_{\mathsf{ty}} \kappa : \mathbf{Type}$*,* $\Sigma; \Psi \vDash_{\mathsf{ty}} \tau_0 : \Pi\Delta.\, H\, \overline{\tau}$*,* $\Sigma; \mathsf{Rel}(\Psi) \vDash_{\mathsf{ty}} H\, \overline{\tau} : \mathbf{Type}$*, and* $\Sigma; \Psi; \Pi\Delta.\, H\, \overline{\tau}; \tau_0 \Rightarrow_{\mathsf{alt}} \mathrm{alt} : \kappa \rightsquigarrow \mathit{alt} \dashv \Omega$*, then* $\Sigma; \Psi, \Omega; \Pi\Delta.\, H\, \overline{\tau} \overset{\tau_0}{\Rightarrow}_{\mathsf{alt}} \mathit{alt} : \kappa$.
5. *If* $\Sigma; \mathsf{Rel}(\Psi) \vDash_{\mathsf{ty}} \kappa : \mathbf{Type}$*,* $\Sigma; \Psi \vDash_{\mathsf{ty}} \tau_0 : \Pi\Delta.\, H\, \overline{\tau}$*,* $\Sigma; \mathsf{Rel}(\Psi) \vDash_{\mathsf{ty}} H\, \overline{\tau} : \mathbf{Type}$*, and* $\Sigma; \Psi; \kappa_0; \tau_0 \Rightarrow_{\mathsf{altc}} \mathrm{alt} : \kappa \rightsquigarrow \mathit{alt} \dashv \Omega$*, then* $\Sigma; \Psi, \Omega; \kappa_0 \overset{\tau_0}{\Rightarrow}_{\mathsf{alt}} \mathit{alt} : \kappa$.
6. *If* $\Sigma \vDash_{\mathsf{ctx}} \Psi\ \mathsf{ok}$ *and* $\Sigma; \Psi \Rightarrow_{\mathsf{q}} \mathrm{qvar} \rightsquigarrow a : \kappa; \nu \dashv \Omega$*, then* $\Sigma; \mathsf{Rel}(\Psi, \Omega) \vDash_{\mathsf{ty}} \kappa : \mathbf{Type}$.
7. *If* $\Sigma \vDash_{\mathsf{ctx}} \Psi\ \mathsf{ok}$ *and* $\Sigma; \Psi \Rightarrow_{\mathsf{aq}} \mathrm{aqvar} \rightsquigarrow a : \kappa \dashv \Omega$*, then* $\Sigma; \mathsf{Rel}(\Psi, \Omega) \vDash_{\mathsf{ty}} \kappa : \mathbf{Type}$.
8. *If* $\Sigma; \Psi \vDash_{\mathsf{ty}} \tau_0 : \kappa$ *and* $\Sigma; \Psi \Rightarrow_{\mathsf{aq}} \mathrm{aqvar} : \kappa \rightsquigarrow a : \kappa'; x.\tau \dashv \Omega$*, then* $\Sigma; \Psi, \Omega \vDash_{\mathsf{ty}} \tau[\tau_0/x] : \kappa'$.

OutsideIn construct		Pico form	Notes
Axiom scheme	$\mathcal{Q}$	Γ	instances, etc.; implications are functions; type family instances are via unfoldings
Given constraint	Q_{g}, Q_{r}	Δ	constraints are named in Pico
Wanted constraint	Q_{w}	Ω	we must separate wanteds & givens

Figure 6.11: Translation from OutsideIn to Pico

6.8.2 Conservativity with respect to OutsideIn

I do not endeavor to give a full accounting of the OutsideIn algorithm here, instead referring readers to the original [99]. I will briefly explain judgments, etc., as they appear and refer readers to Figure numbers from the original text.

There are several mismatches between concepts in OutsideIn and in Pico. Chief among these is that OutsideIn does not track unification variables in any detail. All unification variables (and type variables, in general) in OutsideIn have kind **Type**, and thus there is no need for dependency tracking. In effect, many judgments in OutsideIn are parameterized by an unwritten set of in-scope unification variables. We have no such luxury of concision available in Bake, and so there must be consideration given to tracking the unification variables.

To partly bridge the gap between OutsideIn and Bake, I define encode which does the translation, according to Figure 6.11. encodeing a construct from the left column results in a member of the syntactic class depicted in the middle column.

OutsideIn differentiates between algorithm-generated constraints C and user-written ones Q; the former contain implication constraints. I do not discern between these classes, considering implication constraints simply as functions. I will use Q metavariables in place of OutsideIn's C.[94]

A further difference between OutsideIn and Bake is that the latter is bidirectional. When OutsideIn knows the type which it wishes to assign to a term, it synthesizes the term's type and then emits an equality constraint. In the comparison between the systems, we will pretend that Bake's checking judgments do the same.

The fact that I must change my judgments does not imperil the practical impact of the conservativity result—namely, programs that GHC accepts today will still be accepted tomorrow. GHC already uses bidirectional type-checking and so has already obviated the unidirectional aspect of OutsideIn. However, in order to make a formal comparisons between that published algorithm, it is helpful to restrict ourselves to a unidirectional viewpoint.

A final difference is that Bake does elaboration, while OutsideIn does not. I

[94]This conflation of Q and C does not mean that Dependent Haskell is now required to implement implication constraints; it would be easy to add a post-type-checking pass (a "validity" check, in the vocabulary of the GHC implementation) that ensures that no constraints have implications.

shall use the symbol $\cdot$ to denote an elaborated type that is inconsequential in this comparison.

6.8.2.1 Expressions

Claim (Expressions [Claim E.45]). *If* $\Gamma \overset{\text{OI}}{\Rightarrow} \mathrm{t} : \kappa \rightsquigarrow Q_{\mathrm{w}}$ *under axiom set* $\mathcal{Q}$ *and signature* Σ*, then* $\Sigma; \Gamma, \mathsf{encode}(\mathcal{Q}) \overset{}{\underset{\mathsf{ty}}{\Rightarrow}} \mathrm{t} \rightsquigarrow \cdot : \kappa \dashv \overline{\alpha}{:}_{\mathsf{Irrel}}\mathbf{Type}, \mathsf{encode}(Q_{\mathrm{w}})$ *where* $\overline{\alpha} = \mathsf{fuv}(\kappa) \cup \mathsf{fuv}(Q_{\mathrm{w}})$.

This claim relates OutsideIn's $\Gamma \overset{\text{OI}}{\Rightarrow} \mathrm{t} : \kappa \rightsquigarrow Q_{\mathrm{w}}$ judgment (Figures 6 and 13 from Vytiniotis et al. [99]) to Bake's synthesis $\overset{}{\underset{\mathsf{ty}}{\Rightarrow}}$ judgment. Note that the output Ω from Bake's judgment must include both the wanteds ($\mathsf{encode}(Q_{\mathrm{w}})$) and also any unification variables required during synthesis ($\overline{\alpha}$).

To argue this claim, we examine the different rules that make up OutsideIn's judgment, using structural induction. The details appear in Section E.10.

6.8.2.2 The solver

Property (Solver). *If* $\mathcal{Q}; Q_{\mathrm{g}}; \overline{\alpha}_1 \overset{\text{OI}}{\underset{\mathsf{solv}}{\Rightarrow}} Q_{\mathrm{w}} \rightsquigarrow Q_{\mathrm{r}}; \Theta$ *where* Σ *and* Γ *capture the signature and typing context for the elements of that judgment, then*

$$\Sigma; \Gamma, \mathsf{encode}(\mathcal{Q}), \mathsf{encode}(Q_{\mathrm{g}}) \overset{}{\underset{\mathsf{solv}}{\Rightarrow}} \overline{\alpha}_1{:}_{\mathsf{Irrel}}\mathbf{Type}, \mathsf{encode}(Q_{\mathrm{w}}) \rightsquigarrow \overline{a}_2{:}_{\mathsf{Irrel}}\mathbf{Type}, \mathsf{encode}(Q_{\mathrm{r}})[\overline{a}_2/\overline{\alpha}_2]; \overline{a}_2/\overline{\alpha}_2, \Theta,$$

where the $\overline{a}_2$ *are fresh replacements for the* $\overline{\alpha}_2$ *which are free in* Q_{r} *or unconstrained variables in* $\overline{\alpha}_1$.

This property is a bit more involved than we would hope, but all of the complication deals with Bake's requirement of tracking unification variables more carefully than does OutsideIn. Underneath all of the faffing about with unification variables, the key point here is that Bake's solver will produce the same residual constraint Q_{r} as OutsideIn's and the same zonking substitution Θ.

I do not try to argue this property directly, as I do not present the implementation for the solver. However, this property shows a natural generalization of the solver in an environment that includes dependencies among variables. Indeed, GHC's implementation of the solver already handles such dependency.

6.8.2.3 Programs

Claim (Bind). *If* $\Gamma \overset{\text{OI}}{\Rightarrow} \mathrm{t} : \kappa \rightsquigarrow Q_{\mathrm{w}}$ *and* $\mathcal{Q}; \epsilon; \mathsf{fuv}(\kappa) \cup \mathsf{fuv}(Q_{\mathrm{w}}) \overset{\text{OI}}{\underset{\mathsf{solv}}{\Rightarrow}} Q_{\mathrm{w}} \rightsquigarrow Q_{\mathrm{r}}; \Theta$*, then* $\Sigma; \Gamma, \mathsf{encode}(\mathcal{Q}) \overset{}{\underset{\mathsf{decl}}{\Rightarrow}} x := \mathrm{t} \rightsquigarrow x : \underset{\sim}{\Pi}_{\mathsf{Inf}}\overline{a}{:}_{\mathsf{Irrel}}\mathbf{Type}.\, (\underset{\sim}{\Pi}_{\mathsf{Inf}}\mathsf{encode}(Q_{\mathrm{r}}).\, \kappa[\Theta])[\overline{a}/\overline{\alpha}] := \tau$ *for some* τ*, where* $\overline{\alpha} = \mathsf{fuv}(\kappa[\Theta]) \cup \mathsf{fuv}(Q_{\mathrm{r}})$ *and* $\overline{a}$ *are fresh replacements for the* $\overline{\alpha}$.

This claim relates OutsideIn's Bind rule (Figure 12) to Bake's IDecl_Synthesize rule. It is a consequence of the claim on expressions and the property above of the solver.

Claim (Conservativity over OutsideIn)**.** *If* $\mathcal{Q}; \Gamma \overset{\text{OI}}{\vdash}$ prog, prog *contains no annotated bindings, and* Σ *captures the signature of the environment* prog *is checked in, then* $\Sigma; \Gamma, \mathsf{encode}(\mathcal{Q}) \overset{\rightarrow}{\vdash}_{\mathsf{prog}}$ prog $\rightsquigarrow \Gamma'; \theta$.

This claim relates the overall action of the OutsideIn algorithm (Figure 12) to Bake's algorithm for checking programs. It follows directly from the previous claim.

Because of this, I believe that any program without top-level annotations accepted by OutsideIn is also accepted by Bake.

6.8.3 Conservativity with respect to System SB

Here, I compare Bake with the bidirectional algorithm (called SB) in Figure 8 of Eisenberg et al. [33]. That algorithm is proven to be a conservative extension both of Hindley-Milner inference and also of the bidirectional algorithm presented by Peyton Jones et al. [74]. This SB algorithm, along with OutsideIn, is part of the basis for the algorithm currently implemented in GHC 8.

Before we can successfully relate these systems, we must tweak both a bit to bring their approaches more in line with one another:

- System SB assumes an ability to guess monotypes. This is evident, for example, in the SB_Abs rule, where an unannotated λ-expression is processed and the monotype of the argument is guessed. Bake, of course, uses unification variables. I thus modify System SB to always guess a unification variable when it guesses. The modified rules are SB_Abs, SB_InstS, and SB_Var.

- Because of the previous change, it is now unfair in rule SB_App to insist that the result of synthesis be a function type. Instead, the result of synthesizing the type of e_1 is an arbitrary monotype, and the $\overset{\rightarrow}{\vdash}_{\mathsf{fun}}$ judgment is used to expand this out to a proper function type. Note that we do *not* make a similar change in SB_TApp; doing so would be tantamount to saying that a unification variable might unify with a type with an invisible binder, something we have forbidden. (See Section 6.10.1.3.) We similarly must modify SB_DAbs to allow for the possibility of a unification variable being checked against.

- There is no convenient equivalent of integers in Bake; I omit the rule SB_Int.

- Bake does not do **let**-generalization. I thus modify SB_Let and SB_DLet to use the $\vdash^*_{\mathsf{sb}}$ judgment instead of the generalizing judgment.

- System SB skolemizes deeply in its checking $\vdash^*_{\mathsf{sb}}$ judgment, while Bake skolemizes only shallowly. We thus move the prenex operation from SB_DeepSkol to SB_Infer. I claim that this change does not alter the set of programs that System SB accepts, due to the fact that neither non-Infer rule in the $\vdash_{\mathsf{sb}}$ judgment interacts with $\forall$s.

- BAKE expends a great deal of effort tracking telescopes of unification variables, requiring the notion of a generalizer ξ. However, in the language supported by System SB, all type variables always have kind **Type** and so these telescopes are unnecessary. We thus simply ignore generalizers and the generalization judgment (which always succeeds, regardless).

The theorem below also needs to relate a context Ψ used in BAKE with the more traditional context Γ used in System SB. In the claim below, I use $\Psi \approx \Gamma$ to mean that all Ψ has no coercion bindings, that all irrelevant bindings in Ψ are of kind **Type**, and that no relevant bindings depend on any other. Furthermore, all unification variables bound in Ψ are absent from Γ.

I can now make the following claim:

Claim (Conservativity with respect to System SB [Claim E.47])**.** *Assume* $\Psi \approx \Gamma$.

1. *If* $\Gamma \vdash_{\mathsf{sb}} \mathrm{t} \Rightarrow \kappa$, *then* $\Sigma; \Psi \Vdash_{\mathsf{ty}} \mathrm{t} \rightsquigarrow \cdot : \kappa \dashv \Omega$.
2. *If* $\Gamma \vdash^{*}_{\mathsf{sb}} \mathrm{t} \Rightarrow \kappa$, *then* $\Sigma; \Psi \Vdash^{*}_{\mathsf{ty}} \mathrm{t} \rightsquigarrow \cdot : \kappa \dashv \Omega$.
3. *If* $\Gamma \vdash_{\mathsf{sb}} \mathrm{t} \Leftarrow \kappa$, *then* $\Sigma; \Psi \Vdash_{\mathsf{ty}} \mathrm{t} : \kappa \rightsquigarrow \cdot \dashv \Omega$.
4. *If* $\Gamma \vdash^{*}_{\mathsf{sb}} \mathrm{t} \Leftarrow \kappa$, *then* $\Sigma; \Psi \Vdash^{*}_{\mathsf{ty}} \mathrm{t} : \kappa \rightsquigarrow \cdot \dashv \Omega$.

A detailed argument for this claim appears in Section E.11.

6.9 Practicalities

I have designed BAKE with an eye toward implementing this algorithm directly in GHC. This section discusses some of the practical opportunities and challenges in integrating BAKE with the rest of GHC/Haskell.

6.9.1 Class constraints

In both PICO and BAKE, I conspicuously ignore the possibility of Haskell's type classes and instances. However, this is because classes and instances are already subsumed by these formalizations' handling of regular variables.

Classes in Haskell are already compiled into record types that store the implementations of methods, and instances are record values (often called *dictionaries*) (Section 2.1). As PICO supports datatypes, it also supports classes. Nothing about the type class system should matter at all in PICO. Indeed, System FC as currently implemented in does not GHC 8 cares about type classes, to no ill effect.

During type inference, on the other hand, we need to care a bit about classes and instances, because these are values that the type inference mechanism fills in for us. However, with BAKE's ability to distinguish visible arguments from invisible ones and

its orthogonal ability to work with variables of different relevances, the answer is right in front of us: an instance is simply an inferred, relevant argument. That's it! These are handled in the following rule, part of the judgment that converts a user-written polytype into Pico:

$$\frac{\begin{array}{l}\Sigma; \Psi \Rightarrow_{\mathsf{ty}} \mathrm{t} : \mathbf{Type} \rightsquigarrow \tau \dashv \Omega_1 \\ \Sigma; \Psi, \Omega_1, \$a{:}_{\mathsf{Rel}}\tau \Rightarrow_{\mathsf{pt}} \mathrm{s} \rightsquigarrow \sigma \dashv \Omega_2 \\ \Omega_2 \hookrightarrow \$a{:}_{\mathsf{Rel}}\tau \rightsquigarrow \Omega_2'; \xi\end{array}}{\Sigma; \Psi \Rightarrow_{\mathsf{pt}} \mathrm{t} \Rightarrow \mathrm{s} \rightsquigarrow \underset{\sim}{\Pi}_{\mathsf{Inf}} \$a{:}_{\mathsf{Rel}}\tau.\, (\sigma[\xi]) \dashv \Omega_1, \Omega_2'} \quad \textsc{IPtC\_Constrained}$$

This rule checks the constraint t, making sure it is well typed as a constraint (see Section 6.9.5) and then checks the rest of the type, assuming the constraint. The use of a \$ sign in the name of the constraint ($\$a$) is meant to convey that the variable $\$a$ cannot appear in the Haskell source.

Note that "given" class constraints (that is, a user-written context on a function type signature) are also handled without any effort, as a member of a telescope that unification variables are quantified over.

In contrast to the Bake constraint generation algorithm, the *solver* must treat instances separately and have a way of finding instances in the global set. However, this remains out of scope for this dissertation.

6.9.2 Scoped type variables

Scoped type variables in GHC/Haskell have an idiosyncratic set of rules detailing when variables are to be brought into scope [72]. Consider the following two examples, where t is an arbitrary term:

$$\begin{array}{l}\mathit{example}_1 = (\mathrm{t} :: \forall\ \mathit{a}.\ \mathit{a} \to \mathit{a}) \\ \mathit{higherRank} :: (\forall\ \mathit{a}.\ \mathit{a} \to \mathit{a}) \to () \\ \mathit{example}_2 = \mathit{higherRank}\ \mathrm{t}\end{array}$$

In $\mathit{example}_1$, the type variable a is in scope in t. In $\mathit{example}_2$, however, a is not in scope. This is true despite the fact that, in both cases, Bake would check t against the same Pico type.

Instead of trying to track all of this in the constraint generation algorithm, however, Bake divides its pool of variable names into those names that can appear in a source program (a, b, x) and those that cannot ($\$a, \$b, \$x$). When Bake must put a variable in the context that should not be available in Haskell, it uses the $\$a$ variant. Scoped type variables are explicitly brought into scope by λ or Λ. It is thus up to the preprocessor which must introduce abstractions as necessary to bring the scoped type variables into scope; as this process is not type-directed, incorporated this into the preprocessor should not be a challenge.

BAKE judgment	GHC function
$\overset{\Rightarrow}{\vdash}_{\mathsf{fun}}$	*matchExpectedFunTys*
$\overset{\Rightarrow}{\vdash}_{\mathsf{scrut}}$	*matchExpectedTyConApp*
$\overset{\Rightarrow}{\vdash}_{\mathsf{inst}}$	*topInstantiate*
$\overset{\Rightarrow}{\vdash}_{\mathsf{pre}}$	*tcDeepSplitSigmaTy_maybe*
$\leq^*$	*tcSubTypeDS*
$\leq$	*tcSubType*
$\overset{\Rightarrow}{\vdash}_{\mathsf{prog}}$	*tcPolyBinds*

Figure 6.12: GHC functions that already implement BAKE judgments

6.9.3 Correspondence between BAKE and GHC

The design of BAKE is already quite close to that of GHC's constraint-generation algorithm. Figure 6.12 lists correspondences between BAKE judgments and functions already existent in GHC.

Notably absent from Figure 6.12 are the main judgments such as $\overset{\Rightarrow}{\vdash}_{\mathsf{ty}}$. These are implemented in GHC via its *tcExpr* function, which handles both directions of the bidirectional type system at the same time through its use of *expected types*, a mechanism where the synthesis judgment is implemented by checking against a *hole*—essentially, a unification variable that can unify with a polytype. A full accounting of GHC's expected types and holes is out of scope here, but there should be no trouble adapting BAKE's bidirectional algorithm to GHC as previous bidirectional algorithms have been adapted.

6.9.4 Unification variables in GHC

The GHC implementation takes a very different approach to unification variables and zonking than does BAKE. A GHC unification variable (called a metavariable in the source code) is a mutable cell. The solver fills in the mutable cells. Though the implementation details differ a bit, the same is currently true for unification coercion variables (called coercion holes in GHC)—they are still mutable cells. The *zonking* operation walks through a type (or coercion or expression) and replaces pointers to mutable cells with the cells' contents. On the other hand, BAKE's treatment of filling in unification variables requires building up an explicit zonker Θ; in effect, the implicit substitution GHC builds in the heap using mutable cells is made explicit in BAKE.

Another key difference between GHC and my formalization (and every other) is that GHC variables track their own kinds. The implementation does track a context used in looking up user-written variable occurrences, but no context is needed to, say, extract a type's kind from the type itself. Because of this design, GHC does not need to track unification telescopes, even though GHC 8 already can have arbitrarily long chains of variables that depend on others. Instead, the solver takes (essentially) the

set of unification variables to solve for. Dependency checking is done after the fact as a simple pass making sure all variables in kinds are in scope.

A further consequence of GHC's design is that there is no need for the concept of generalizers ξ as I have described. Unification variable occurrences are not, in fact, applied to vectors. Along with the fact that GHC does not track contexts, it also uses stable names powered by a enumerable collection of *Unique*s. We thus do not have to worry about arbitrary α-renaming during constraint generation and solving. Taken together, the need for generalizers is lost, and thus the generalization operation $\hookrightarrow$ disappears.

6.9.5 *Constraint* vs. Type

Haskell includes the kind *Constraint* that classifies all constraints; we thus have $\mathit{Show} :: \mathbf{Type} \to \mathit{Constraint}$. However, due to the datatype encoding of classes and the dictionary encoding of instances, PICO manipulates constraints just as it does ordinary types. For this reason, PICO makes no distinction between *Constraint* and **Type**. This choice follows GHC's current practice, where *Constraint* and **Type** are distinct in the source language but indistinguishable in the intermediate language. This design has some unfortunate consequences; see GHC ticket #11715 for a considerable amount of discussion.

Extending the language with dependent types is orthogonal to the problems presented there, however. For simplicity, BAKE as described here does not recognize *Constraint*, putting all constraints in the kind **Type** with all the other types.

6.10 Discussion

6.10.1 Further desirable properties of the solver

Thus far, I have stated only one property (in Section 6.8.1.2) that the solver must maintain, that it must output a valid zonker. However, it is helpful to describe further properties of the solver in order to make type inference more predictable and to maintain the properties stated by Vytiniotis et al. [99], such as the fact that all inferred types are principal and that the solver makes no guesses. The full set of extra properties are listed in Figure 6.13 on the next page.

6.10.1.1 Entailment

These properties are stated with respect to an entailment relation, defined as follows:

Definition (Entailment)**.** *We say that an environment $\Sigma; \Psi$ entails a telescope Δ, written $\Sigma; \Psi \Vdash \Delta$, if there exists a vector $\overline{\psi}$ such that $\Sigma; \Psi \vDash_{\mathsf{vec}} \overline{\psi} : \Delta$.*

Property 6.1 (Solver is guess-free)**.** *If* $\Sigma; \Psi \mathrel{\mathsf{sólv}} \Omega \rightsquigarrow \Delta; \Theta$*, then* $\Sigma; \Psi, \Omega \models \Delta, \mathcal{E}_\Theta$*, where* $\mathcal{E}_\Theta = \{\_ : \alpha \sim \tau \mid \forall\, \overline{z}.\tau/\alpha \in \Theta\}$ *is the equational constraint induced by the zonker* Θ*.*

The above property is adapted from Vytiniotis et al. [99, Definition 3.2 (P1)].

Property 6.2 (Solver avoids non-simple types)**.** *If* $\Sigma; \Psi \mathrel{\mathsf{sólv}} \Omega \rightsquigarrow \Delta; \Theta$ *and* $\forall\, \overline{z}.\tau/\alpha \in \Theta$*, then* τ *is a simple type, with no invisible binders (at any level of structure) and no dependency.*

Property 6.3 (Solver does not generalize over coercions)**.** *If* $\Sigma; \Psi \mathrel{\mathsf{sólv}} \Omega \rightsquigarrow \Delta; \Theta$*, then* Δ *binds no coercion variables.*

Figure 6.13: Additional solver properties

As this section expands upon the ideas from Vytiniotis et al. [99], it is necessary to check whether this definition of entailment satisfies the entailment requirements from that work. These requirements are presented in Figure 6.14 on the following page.

All of this properties are easily satisfied, except for property (R8) (both components) which requires congruence. As explored in some depth in Section 5.8.5.3, Pico simply does not have this property. However, that same section argues that equality in Pico is "almost congruent", suggesting that the equality relation truly is congruent in the absence of coercion abstractions. The proof that the OutsideIn algorithm infers principal types does require property R8 [99, Theorem 3.2], and so it is possible that Pico's lack of congruence prevents Bake from inferring principal types. The details have yet to be worked out.

6.10.1.2 A guess-free solver

One of the guiding principles I set forth at the beginning of this chapter is that the algorithm and solver be guess-free. We thus must assert that the solver is guess-free, an important step along the way to the proof of principal types in Vytiniotis et al. [99]. See Property 6.1.

6.10.1.3 Solver does not introduce impredicativity

An important but previously unstated property is that that solver must not set a unification variable to anything but a simple type, one with no invisible binders nor dependency. (Such types are sometimes called τ-types, referring to the τ/σ split in the typical presentation of the Hindley-Milner type system.) In the context of Dependent Haskell, impredicativity has perhaps an unusual definition: no type variable is ever instantiated with a non-simple type. For this to hold, however, we must make sure that this property extends to unification variables as well, as those are sometimes used to instantiate regular variables.

Reflexivity	$\Sigma; \Psi, \Delta \Vdash \Delta$	(R1)
Transitivity	$\Sigma; \Psi, \Delta_1 \Vdash \Delta_2 \wedge \Sigma; \Psi, \Delta_2 \Vdash \Delta_3 \implies \Sigma; \Psi, \Delta_1 \Vdash \Delta_3$	(R2)
Substitution	$\Sigma; \Psi, \Delta_1, \Psi' \Vdash \Delta_2 \wedge \Sigma; \Psi \vDash_{\mathsf{subst}} \theta : \Delta_1$	(R3)
	$\implies \Sigma; \Psi, \Psi'[\theta] \Vdash \Delta_2[\theta]$	
Type eq. reflexivity	$\Sigma; \Psi \vDash_{\mathsf{ty}} \tau : \kappa \implies \Sigma; \Psi \Vdash \_ : \tau \sim \tau$	(R4)
Type eq. symmetry	$\Sigma; \Psi \Vdash \_ : \tau_1 \sim \tau_2 \implies \Sigma; \Psi \Vdash \_ : \tau_2 \sim \tau_1$	(R5)
Type eq. transitivity	$\Sigma; \Psi \Vdash \_ : \tau_1 \sim \tau_2 \wedge \Sigma; \Psi \Vdash \_ : \tau_2 \sim \tau_3$	(R6)
	$\implies \Sigma; \Psi \Vdash \_ : \tau_1 \sim \tau_3$	
Conjunctions	$\Sigma; \Psi \Vdash \Delta_1 \wedge \Sigma; \Psi \Vdash \Delta_2 \implies \Sigma; \Psi \Vdash \Delta_1, \Delta_2$	(R7)
Substitutivity	$\Sigma; \Psi \Vdash \_ : \tau_1 \sim \tau_2 \wedge \Sigma; \Psi, a{:}_{\mathsf{Rel}}\kappa_0 \vDash_{\mathsf{ty}} \tau : \kappa$	(R8a)
	$\implies \Sigma; \Psi \Vdash \_ : \tau[\tau_1/a] \sim \tau[\tau_2/a]$	
	$\Sigma; \mathsf{Rel}(\Psi) \Vdash \_ : \tau_1 \sim \tau_2 \wedge \Sigma; \Psi, a{:}_{\mathsf{Irrel}}\kappa_0 \vDash_{\mathsf{ty}} \tau : \kappa$	(R8b)
	$\implies \Sigma; \Psi \Vdash \_ : \tau[\tau_1/a] \sim \tau[\tau_2/a]$	

Figure 6.14: Required properties of entailment, following [99, Figure 3]

Solving unification variables with simple types is also important in the context of the theory around principal types developed in my prior work [33]. Specifically, we must ensure that there are no invisible binders that are hidden underneath a unification variable. By forbidding filling a unification variable with a non-simple type, we have achieved this goal. See Property 6.2.

6.10.2 No coercion abstractions

In stating that Pico supports type erasure (Section 5.11), I admit that type erasure does not mean that we can erase coercion abstractions or applications, even though we can erase the coercions themselves. Nevertheless, I argue that Pico can claim to support full type erasure because Bake never produces a Pico program that evaluates to a coercion abstraction. To support this claim, we can look at the elaborated program produced by Bake and where coercion abstractions can be inserted:

Around subsumption: Three rules extract out a telescope of binders using the $\vDash_{\mathsf{pre}}$ judgment and then use these binders in the elaboration. If the telescope includes a coercion binder, the elaboration will include a coercion abstraction. However, I am arguing that there should be no coercion binders there in the first place, so we can handle this case essentially by induction. (Rules affected: ITyC_Infer, rules in the $\vDash_{\mathsf{pre}}$ judgment, and ISub_DeepSkol)

During generalization after running the solver: If the solver produces a telescope that binds coercions, Bake will similarly include a coercion abstract in its elaboration. We must thus assert Property 6.3. This property is not as restrictive

as it may seem, as the solver may still abstract over a class constraint whose instances store a coercion.[95] (Rule affected: IDECL_SYNTHESIZE)

Elaborating case alternatives: When elaborating a **case** alternative, coercion abstractions are inserted. This is necessary for two reasons:

- GADT equalities can be brought into scope in a **case** alternative. These are bound by coercion abstractions.
- The dependent-pattern-match equality (Section 4.3.3) must be brought into scope by a coercion abstraction.

However, when a **case** expression is evaluated (by evaluation rule S_MATCH), these coercion abstractions will be applied to arguments and thus cannot be the final value of evaluating the outer PICO expression. (Rules affected: IALT_CON, IALTC_CON)

These are the only BAKE rules that can include a coercion abstraction in their elaborated types. I thus conclude that type erasure is valid, with no possibility of having evaluation be stuck on a coercion abstraction.

6.10.3 Comparison to Gundry [37]

The BAKE algorithm presented here is very similar to the type inference algorithm presented by Gundry [37, Chapter 7]. Here I review some of the salient differences.

- Gundry includes both a non-deterministic elaboration process and a deterministic one, proving that the deterministic process is sound with respect to the non-deterministic process (at least, in the absence of **case**). I have omitted a non-deterministic version of the algorithm, instead using the soundness of the resultant PICO program to set an upper limit on the programs that BAKE can accept.
- Gundry's *inch* source language and his *evidence* intermediate language have two forms of **case** statement: one for traditional, non-dependent pattern matching; and one for dependent pattern matching. BAKE chooses between these possibilities using the difference between checking and synthesis modes.
- While Gundry uses two separate judgments in synthesis mode, he uses only one checking judgment. The need for two checking judgments here is an innovation that derives from the need for principal types, as explored in my prior work [33].
- The *inch* language does not allow annotations on the binders of a λ-abstraction and so Gundry did not encounter the thorny case detailed in Section 6.6.4.

[95] For example, the Haskell equality constraint $\sim$ is such a class, distinct from the primitive equality operator in PICO. In the terminology of Vytiniotis et al. [100], the Haskell equality is lifted while the PICO equality is unlifted.

- Gundry's approach to delayed instantiation for function arguments follows along the lines of Dunfield and Krishnaswami [24], using an auxiliary judgment to control function application. While BAKE has its $\overset{*}{\Rightarrow}_{\mathsf{arg}}$ judgment, which is superficially similar, BAKE's judgment can only handle one argument at a time.

- Gundry's algorithm does not do deep skolemization. It would thus not be backward compatible with GHC's current treatment of higher-rank types.

- Gundry gives more details about the solver in his algorithm [37, Section 7.5.1]. However, this solver is a novel algorithm that remains to be implemented. Instead, BAKE targets the OUTSIDEIN solver. Nevertheless, I do not think it would be hard for Gundry's general approach to target OUTSIDEIN, as well.

- As a point of similarity, Gundry's and BAKE's treatment of unification variables are very closely aligned. This is not actually intentional—after reading Gundry's approach, I believed I could make the whole treatment of unification variables much simpler. Yet despite a variety of attempts, I was unable to make the basic lemmas that hold together a type system (e.g., substitution, regularity) go through without something as ornate as we have both used. I would love to see a simpler treatment in the future, but I do not hold out much hope.

6.11 Conclusion

This chapter has presented BAKE, a type checking / inference / elaboration algorithm that converts type-correct Dependent Haskell types and expressions into PICO. It is proven to produce type-correct PICO code, and it is designed in the hope of supporting principal types. Formulating a statement and proof of principal types in BAKE is important future work.

This algorithm is also designed to work well with GHC's existing type checker infrastructure, and in particular, its constraint solver. It is my hope and plan to implement this algorithm, quite closely to how it is stated here, in GHC in the near future.

Chapter 7

Implementation

This chapter reviews a number of practical issues that arise in the course of implementing the theory presented in this dissertation. Perhaps the most interesting of these is that the function that computes equality in GHC does not simply check for α-equivalence; see Section 7.2.

7.1 Current state of implementation

As of this writing (August 2016), only a portion of the improvements to Haskell described in this dissertation are implemented. This section describes the current state of play and future plans.

7.1.1 Implemented in GHC 8

The language supported by GHC 8 is already a large step toward the language in this dissertation. The features beyond those available in GHC 7 are enabled by GHC's `TypeInType` extension. I personally implemented essentially all aspects of this extension and merged my work in with the development stream. I have had feedback and bug reports from many users,[96] indicating that my new features are already gaining traction in the community. Here are its features, in summary:

- The core language is very closely as described in my prior work [105].
- The kind of types $\star$ is now treated as described in Section 7.4.
- Types and kinds are indistinguishable and fully interchangeable.
- Kind variables may be explicitly quantified:

[96] According to the GHC bug tracker, 19 users (excluding myself) have posted bugs against my implementation.

```
data Proxy :: ∀ k. k → Type where
  Proxy :: ∀ k (a :: k). Proxy a
```

- The same variable can be used in a type and in a kind:

```
data T where
  MkT :: ∀ k (a :: k). k → Proxy a → T
```

- Type families can be used in kinds.
- Kind-indexed GADTs:

```
data G :: ∀ k. k → Type where
  GInt   :: G Int
  GMaybe :: G Maybe
data (:≈:) :: ∀ k1 k2. k1 → k2 → Type where
  HRefl :: a :≈: a
```

- Higher-rank kinds are now possible:

```
class HTestEquality (f :: ∀ k. k → Type) where
  hTestEquality :: ∀ k1 k2 (a :: k1) (b :: k2). f a → f b → Maybe (a :≈: b)
instance HTestEquality TypeRep where    -- from Section 3.2.2
  hTestEquality = eqT
```

- GADT data constructors can now be used in types.
- The type inference algorithm used in GHC over types directly corresponds to those rules in Bake that deal with the constructs that are available in types (that is, missing **case**, **let**, and λ). This algorithm in GHC is bidirectional, as is Bake.

7.1.2 Implemented in singletons

Alongside my work implementing dependent types in GHC, I have also continued the development of my singletons package [29, 30]. This package has some enduring popularity: it has over 7,000 downloads, 31 separate users reporting bugs, is the primary subject of several blog posts[97] and has even made its way into a textbook on Haskell [81, Chapter 13]. The singletons package uses Template Haskell [83], GHC's

[97] Here are a sampling:

meta-programming facility, to transform normal term-level declarations into type-level equivalents.

I use my work in singletons as a proof-of-concept for implementing dependent types. My goal with the dependent types work is to make this package fully obsolete. In the meantime, it is an invaluable playground of ideas both for me and other Haskellers who do not wish to wait for dependent types proper.

Because of its function as a proof-of-concept, I include here a list of features supported by singletons. By their support in the library, we can be confident that these features can also be supported in GHC without terrible difficulty. The singletons currently supports code using the following features in types:

- All term-level constructs supported by Template Haskell except: view patterns, **do**, list comprehensions, arithmetic sequences. (Template Haskell does not support GHC's arrow notation.) The library specifically *does* support **case**, **let** (including recursive definitions) and λ-expressions. See my prior work for the details [29].
- Unsaturated type families and the distinction between matchable and unmatchable arrows
- Type classes and instances
- Constrained types
- Pattern guards
- Overloaded numeric literals
- Deriving of *Eq*, *Ord*, *Bounded*, and *Enum*
- Record syntax, including record updates
- Scoped type variables

The latest major effort at improving singletons targeted GHC 7, though the library continues to work with GHC 8. I am confident more constructs could be supported with a thorough update to GHC 8—in particular, **do**-notation cannot be supported in GHC 7 because it would require a higher-kinded type variable. Such type variables are fully supported in GHC 8, and so I believe singletons could support **do**-notation

- `https://www.schoolofhaskell.com/user/konn/prove-your-haskell-for-great-safety/dependent-types-in-haskell`
- `https://ocharles.org.uk/blog/posts/2014-02-25-dependent-types-and-plhaskell.html`
- `http://lambda.jstolarek.com/2014/09/promoting-functions-to-type-families-in-haskell/`
- `https://blog.jle.im/entry/practical-dependent-types-in-haskell-1.html`

all by different authors—not to mention my own posts.

and list/monad comprehensions relatively easily now. However, I wish to spend my implementation efforts on getting dependent types in Haskell for real instead of faking it with singletons, and so may not complete these upgrades.

7.1.3 Implementation to be completed

There is still a fair amount of work to be done before the implementation of dependent types in Haskell is complete. Here I provide a listing of the major tasks to be completed and my thoughts on each task:

- Implement Pico as written in this dissertation. The biggest change over the current implementation of GHC's intermediate language is that Pico combines the grammar of types and of terms. The current intermediate language already supports, for example, heterogeneous equality and the asymmetric binding coercion forms (Section 5.8.5.1). While combining the internal datatypes for types and terms will be the furthest reaching change, I think the most challenging change will be the addition of the many different quantifier forms in Pico (with relevance markers, visibility markers, and matchability markers).

- Combine the algorithms that infer the types of terms and the kinds of types. Currently, GHC maintains two separate, but similar, algorithms: one that type-checks terms and one that kind-checks types. These would be combined, as prescribed by Bake. I expect this to be a *simplification* when it is all done, as one algorithm will serve where there is currently two—and both are quite complex.

- Interleave type-checking with desugaring. Currently, GHC maintains two separate phases when compiling terms: type-checking ensures that the source expression is well typed and also produces information necessary for elaboration into its intermediate language. Afterwards, GHC desugars the type-checked program, translating it to the intermediate language. Desugaring today is done only after the whole module is type-checked. However, if some declarations depend on evaluating other declarations (because the latter are used in the former's types), then desugaring and type-checking will have to be interleaved. I do not expect this to be a challenge, however, for two reasons:

  - Type-checking and desugaring are *already* interleaved, at least in types. Indeed, the kind checker for types produces a type in the intermediate language today, effectively type-checking and desugaring all at once.
  - Type-checking happens by going in order through a sequence of mutually recursive groups. One expression cannot depend on another within the same group, and so we can just process each group one at a time, type-checking and then desugaring.

- The source language will have to be changed to accept the new features. To be honest, I am a little worried about this change, as it will require updating the parser. Currently, the parsers for types and expressions are separate, but this task would require combining them. Will this be possible? I already know of one conflict: the ' used in Template Haskell quoting (which made a brief appearance in Section 3.1.3.2) and the ' used in denoting a namespace change. Both of these elements of the syntax are pre-existing, and so I will have to find some way of merging them.

At this point, I do not foresee realistically beginning these implementation tasks before the summer of 2017. If that process goes swimmingly, then perhaps we will see Dependent Haskell released in early 2018. More likely, it will be delayed until 2019.

During the process of writing this dissertation, I worked on merging my implementation of `TypeInType` into the GHC main development stream. This process was *much* harder than I anticipated, taking up two more months than expected, working nearly full-time. I am thus leery of over-promising about the rest of the implementation task embodied in this dissertation. However, my success in emulating so many of the features in Dependent Haskell in singletons gives me hope that the worst of the implementation burden is behind me.

Despite not having fully implemented Dependent Haskell, I still have learned much by implementing one portion of the overall plan. The rest of this chapter shares this hard-won knowledge.

7.2 Type equality

The notion of type equality used in the definition of Pico is quite restrictive: it is simple α-equivalence. This equality relation is very hard to work with in practice, because it is *not* proof-irrelevant. That is, $\mathit{Int} \triangleright \langle \mathbf{Type} \rangle \not\equiv \mathit{Int}$. This is true despite the fact that the $\sim$ relation *is* proof-irrelevant.

The proof-relevant nature of $=$ poses a challenge in transforming Pico expressions into other well typed Pico expressions. This challenge comes to a head in the unifier (Section 7.3) where, given τ_1 and τ_2, we must find a substitution θ such that $\tau_1 = \tau_2$. Unification is used, for example, when matching class instances. However, with proof-relevant equality, such a specification is wrong; it would fail to find an instance $\mathsf{C}\ (\mathit{Maybe}\ \mathsf{a})$ when we seek an instance for $\mathsf{C}\ (\mathit{Maybe}\ \mathit{Int} \triangleright \langle \mathbf{Type} \rangle)$. Instead, we want θ and γ such that $\Sigma; \Gamma \vdash_{\mathsf{co}} \gamma : \tau_1[\theta] \sim \tau_2[\theta]$ (for an appropriate Σ and Γ). Experience has shown that constructing the γ is a real challenge.[98]

[98] When I attempted this implementation, the coercion language was a bit different than presented in Pico. In particular, I did not have the $\approx$ coercion form, instead having the much more restricted version of coherence that appears in my prior work [105]. The new form $\approx$ is admissible given the older form, but it is not easy to derive. It is conceivable that, with $\approx$, this implementation task would now be easier.

7.2.1 Properties of a new definitional equality $\equiv$

The problem, as noted, is that the $=$ relation is too small. How can we enlarge this relation? Since the relation we seek deviates both from α-equivalence and from $\sim$, we need a new name: let's call it $\equiv$, as it will be the form of definitional equality in the implementation. (The relation is checked by the GHC function *eqType*, called whenever two types must be compared for equality.) We will define a new type system, PICO$^{\equiv}$, based on $\equiv$. Here are several properties we require of $\equiv$, if we are to adapt the existing metatheory for PICO:

Property 7.1. *The $\equiv$ relation must be an equivalence. That is, it must be reflexive, symmetric, and transitive.*

Property 7.2. *The $\equiv$ relation must be a superset of $=$. That is, if $\tau_1 = \tau_2$, then $\tau_1 \equiv \tau_2$.*

Property 7.3. *The $\equiv$ relation must be a subset of $\sim$. That is, if $\tau_1 \equiv \tau_2$, then there must be a proof of $\tau_1 \sim \tau_2$ (in appropriate contexts).*

Property 7.4. *The $\equiv$ relation must be congruent. That is, if corresponding components of two types are $\equiv$, then so are the two types.*

Property 7.5. *The $\equiv$ relation must be proof-irrelevant. That is, $\tau \equiv \tau \rhd \gamma$ for all τ.*

Property 7.6. *The $\equiv$ relation must be homogeneous. That is, it can relate two types of the same kind only.*

Property 7.7. *Computing whether $\tau_1 \equiv \tau_2$ must be quick.*

We need Properties 7.1-7.4 for soundness. I will argue below that we can transform the typing rules for PICO to use $\equiv$ where they currently use $=$. This argument relies on these first four properties.

Property 7.5 means that our new definition of $\equiv$ indeed simplifies the implementation. After all, seeking a proof-irrelevant (that is, coherent) equality is what started this whole line of inquiry. However, despite Property 7.5 masquerading as only a *desired* property, it turns out that with my proof technique, this is a *necessary* property. Indeed, it seems that once $\equiv$ is any relation strictly larger than $=$, it must be proof irrelevant. This is because the translation from a derivation in PICO$^{\equiv}$ to one in PICO (see next subsection) will use coercions as obtained through Property 7.3. These coercions must not interfere with $\equiv$-equivalence.

Property 7.6 arises from the use of $\equiv$ (that is, *eqType*) in the implementation. There are many places where we compare two types for equality and, if they are equal, arbitrarily choose one or the other. Thus, $\equiv$ must be substitutive and accordingly homogeneous.

Property 7.7 arises because we use *eqType* very frequently. A slow computation or a search simply is not feasible.

Beyond these requirements, a larger $\equiv$ relation is better. Having a larger $\equiv$ makes implementing Pico easier, as we will be able to replace one type with another type, as long as the two are $\equiv$. Thus, having more types be related makes the system more flexible.

7.2.2 Replacing $=$ with $\equiv$

We can take the typing rules of Pico and mechanically replace uses of $=$ (over types) with $\equiv$ to form the rules of Pico$^{\equiv}$. This is done by looking for every duplicated use of a type in the premises of a rule, and putting in a $\equiv$ instead.

For example, the application rule is transformed from

$$\frac{\Sigma;\Gamma \vdash_{\mathsf{ty}} \tau_1 : \Pi a{:}_{\mathsf{Rel}}\kappa_1.\,\kappa_2 \qquad \Sigma;\Gamma \vdash_{\mathsf{ty}} \tau_2 : \kappa_1}{\Sigma;\Gamma \vdash_{\mathsf{ty}} \tau_1\,\tau_2 : \kappa_2[\tau_2/a]} \quad \textsc{Ty\_AppRel}$$

to

$$\frac{\begin{array}{ll}\Sigma;\Gamma \Vdash_{\mathsf{ty}} \tau_1 : \kappa_0 & \kappa_0 \overset{\rightarrow}{\equiv} \Pi a{:}_{\mathsf{Rel}}\kappa_1.\,\kappa_2 \\ \Sigma;\Gamma \Vdash_{\mathsf{ty}} \tau_2 : \kappa_1' & \kappa_1 \equiv \kappa_1'\end{array}}{\Sigma;\Gamma \Vdash_{\mathsf{ty}} \tau_1\,\tau_2 : \kappa_2[\tau_2/a]} \quad \textsc{DTy\_AppRel}$$

This new rule allows κ_1 and κ_1' not to be α-equivalent, as long as they are $\equiv$. It also makes use of an *extraction operator* $\overset{\rightarrow}{\equiv}$ that pulls out the component pieces of a type, respecting $\equiv$-equivalence. The full set of rules that define Pico$^{\equiv}$ appear in Appendix F.

Continuing the notational convention where $\mathcal{J}$ can stand for any of the judgments $\vdash_{\mathsf{ty}}$, $\vdash_{\mathsf{co}}$, $\vdash_{\mathsf{alt}}$, $\vdash_{\mathsf{prop}}$, $\vdash_{\mathsf{ctx}}$, $\vdash_{\mathsf{vec}}$, or $\vdash_{\mathsf{s}}$, we have following lemmas, relating Pico$^{\equiv}$ to Pico:

Lemma (Pico$^{\equiv}$ is an extension of Pico)**.** *If* $\Sigma;\Gamma \vdash \mathcal{J}$ *then* $\Sigma;\Gamma \Vdash \mathcal{J}$.

Proof. Corollary of Property 7.2 of the definition of $\equiv$. □

We also need a lemma where a result in Pico$^{\equiv}$ implies one in Pico. This is harder to state, as it requires an operation that translates a term τ that is well typed in Pico$^{\equiv}$ into one well typed in Pico. We write the latter as $\lceil\tau\rceil$. The translation operation $\lceil\cdot\rceil$ is actually a deterministic operation on the typing derivation in Pico$^{\equiv}$; the conversion is valid only when the original type is well formed in Pico$^{\equiv}$. The full statement of the lemma relating Pico$^{\equiv}$ to Pico appears in Appendix F, but the following informal statement will serve us well here:

Lemma (Pico$^{\equiv}$ is sound [Lemma F.10])**.** *If* $\Sigma;\Gamma \Vdash \mathcal{J}$*, then* $\Sigma;\lceil\Gamma\rceil \vdash \lceil\mathcal{J}\rceil$.

With both of these lemmas in hand, we can see that Pico and Pico$^{\equiv}$ are equivalent systems and that all of the results from Pico carry over to Pico$^{\equiv}$.

7.2.3 Implementation of $\equiv$

Having laid out the properties we require of $\equiv$, my choice of implementation of $\equiv$ is this:

Definition (Definitional equality $\equiv$)**.** *We have* $\tau_1 \equiv \tau_2$ *whenever* $\lfloor \kappa_1 \rfloor = \lfloor \kappa_2 \rfloor$ *and* $\lfloor \tau_1 \rfloor = \lfloor \tau_2 \rfloor$*, where* $\tau_1 : \kappa_1$ *and* $\tau_2 : \kappa_2$*.*

The operation $\lfloor \tau \rfloor$ here is the coercion erasure operation from Section 5.8.3. It simply removes all casts and coercions from a type. In the implementation, we can easily go from a type to its kind, as all type variables in GHC store their kinds directly (as also described in Section 6.9.4), with no need for a separate typing context. The implementation actually optimizes this equality check a bit, by comparing the kinds only when the type contains a cast—this avoids the extra check in the common case of a simple type.

This equality check easily satisfies the properties described above. It also supports the extraction operation, which simply looks through casts.

7.3 Unification

It is often necessary to unify two types. This is done in rule Alt_Match in Pico but is also necessary in several places during type inference—for example, when matching up a class instance with a constraint that must be solved. With dependent types, however, how should such a unifier work? For example, should $(a\ b)$ unify with $(\tau\ \sigma) \rhd \gamma$? The top-level forms of these are different, and yet, intuitively, we would want them to unify. In other words, we want an algorithm that does unification up to $\equiv$.

I have thus implemented a novel unification algorithm in GHC that does indeed unify the forms above. To first order, this algorithm simply ignores casts and coercions. The problem if we ignore coercions altogether is that the resulting substitution might not be well kinded. As a simple example, consider unifying a with $\tau \rhd \gamma$. If we just ignore casts, then we get the substitution τ/a—but τ and a might have different kinds. In the type application example, we similarly do not want the substitution $\tau/a, \sigma/b$ but instead $(\tau \rhd \gamma_1)/a, (\sigma \rhd \gamma_2)/b$ for appropriate γ_1 and γ_2.

My approach, then, is for the algorithm to take three inputs: the two types to unify and a coercion between their kinds. At the leaves (matching a variable against a type), we insert this coercion to make the substitution well kinded. At interior nodes, we simply ensure that we have a new kind coercion to pass to recursive calls.

The unification algorithm is in Figure 7.1 on the next page. It works in the context of a *UM* monad that can handle failure and stores the ambient substitution produced by unification. I will highlight a few interesting points in this algorithm:

- The *unify* function considers only those types which might be values. It specifically avoids treating **case** or **fix**. This is because non-values are *flattened* away before

$$
\begin{array}{l}
\mathit{unify} :: \mathit{Type} \to \mathit{Type} \to \mathit{Coercion} \to \mathit{UM}\ () \\
\mathit{unify}\ (\tau_1 \triangleright \gamma) \quad \tau_2 \quad \eta = \mathit{unify}\ \tau_1\ \tau_2\ \ (\gamma \fatsemi \eta) \\
\mathit{unify}\ \tau_1 \quad (\tau_2 \triangleright \gamma) \quad \eta = \mathit{unify}\ \tau_1\ \tau_2\ \ (\eta \fatsemi \mathbf{sym}\ \gamma) \\
\mathit{unify}\ a \quad \tau_2 \quad \eta = \mathit{unifyVar}\ a\ \tau_2\ \eta \\
\mathit{unify}\ \tau_1 \quad a \quad \eta = \mathit{unifyVar}\ a\ \tau_1\ (\mathbf{sym}\ \eta) \\
\mathit{unify}\ H_{\{\overline{\tau}_1\}} \quad H_{\{\overline{\tau}_2\}} \quad \_ = \mathit{unifyTys}\ \overline{\tau}_1\ \overline{\tau}_2 \\
\mathit{unify}\ (\tau_1\ \sigma_1) \quad (\tau_2\ \sigma_2) \quad \_ = \mathit{unifyTyApp}\ \tau_1\ \sigma_1\ \tau_2\ \sigma_2 \\
\mathit{unify}\ (\tau_1\ \{\sigma_1\}) \quad (\tau_2\ \{\sigma_2\}) \quad \_ = \mathit{unifyTyApp}\ \tau_1\ \sigma_1\ \tau_2\ \sigma_2 \\
\mathit{unify}\ (\tau_1\ \_) \quad (\tau_2\ \_) \quad \_ = \mathit{unifyApp}\ \tau_1\ \tau_2 \\
\mathit{unify}\ (\Pi a{:}_\rho \kappa_1.\tau_1)\ (\Pi a{:}_\rho \kappa_2.\tau_2)\ \_ = \mathbf{do}\ \mathit{unify}\ \kappa_1\ \kappa_2\ \langle \mathbf{Type} \rangle \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{unify}\ \tau_1\ \tau_2\ \langle \mathbf{Type} \rangle \\
\mathit{unify}\ (\Pi c{:}\phi_1.\tau_1)\ \ (\Pi c{:}\phi_2.\tau_2)\ \ \_ = \mathbf{do}\ \mathit{unifyProp}\ \phi_1\ \phi_2 \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{unify}\ \tau_1\ \tau_2\ \langle \mathbf{Type} \rangle \\
\mathit{unify}\ (\lambda a{:}_\rho \kappa_1.\tau_1)\ (\lambda a{:}_\rho \kappa_2.\tau_2)\ \_ = \mathbf{do}\ \mathit{unify}\ \kappa_1\ \kappa_2\ \langle \mathbf{Type} \rangle \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{unify}\ \tau_1\ \tau_2\ \langle \mathit{typeKind}\ \tau_1 \rangle \\
\mathit{unify}\ (\lambda c{:}\phi_1.\tau_1)\ \ (\lambda c{:}\phi_2.\tau_2)\ \ \_ = \mathbf{do}\ \mathit{unifyProp}\ \phi_1\ \phi_2 \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{unify}\ \tau_1\ \tau_2\ \langle \mathit{typeKind}\ \tau_1 \rangle \\
\mathit{unify}\ \_ \quad \_ \quad \_ = \mathit{mzero}
\end{array}
$$

$$
\begin{array}{l}
\mathit{unifyVar} :: \mathit{TyVar} \to \mathit{Type} \to \mathit{Coercion} \to \mathit{UM}\ () \\
\mathit{unifyVar}\ a\ \tau_2\ \eta = \mathbf{do}\ \mathit{mt1} \leftarrow \mathit{substTyVar}\ a \\
\qquad\qquad\qquad\qquad\quad \mathbf{case}\ \mathit{mt1}\ \mathbf{of} \\
\qquad\qquad\qquad\qquad\qquad \mathit{Nothing} \to \mathit{bindTv}\ a\ (\tau_2 \triangleright \mathbf{sym}\ \eta) \\
\qquad\qquad\qquad\qquad\qquad \mathit{Just}\ \tau_1\ \ \to \mathit{unify}\ \tau_1\ \tau_2\ \eta
\end{array}
$$

$$
\begin{array}{l}
\mathit{unifyTys} :: [\mathit{Type}] \to [\mathit{Type}] \to \mathit{UM}\ () \\
\mathit{unifyTys}\ [\,] \quad [\,] \quad = \mathit{return}\ () \\
\mathit{unifyTys}\ (\tau_1{:}\overline{\tau}_1)\ (\tau_2{:}\overline{\tau}_2) = \mathbf{do}\ \mathit{unify}\ \tau_1\ \tau_2\ \langle \mathit{typeKind}\ \tau_1 \rangle \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{unifyTys}\ \overline{\tau}_1\ \overline{\tau}_2 \\
\mathit{unifyTys}\ \_ \quad \_ \quad = \mathit{mzero}
\end{array}
$$

$$
\begin{array}{l}
\mathit{unifyTyApp} :: \mathit{Type} \to \mathit{Type} \to \mathit{Type} \to \mathit{Type} \to \mathit{UM}\ () \\
\mathit{unifyTyApp}\ \tau_1\ \sigma_1\ \tau_2\ \sigma_2 = \mathbf{do}\ \mathit{unifyApp}\ \tau_1\ \tau_2 \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathit{unify}\ \sigma_1\ \sigma_2\ \langle \mathit{typeKind}\ \sigma_1 \rangle
\end{array}
$$

$$
\begin{array}{l}
\mathit{unifyApp} :: \mathit{Type} \to \mathit{Type} \to \mathit{UM}\ () \\
\mathit{unifyApp}\ \tau_1\ \tau_2 = \mathbf{do} \quad \mathbf{let}\ \kappa_1 = \mathit{typeKind}\ \tau_1 \\
\qquad\qquad\qquad\qquad\qquad\quad\ \ \kappa_2 = \mathit{typeKind}\ \tau_2 \\
\qquad\qquad\qquad\qquad\quad \mathit{unify}\ \kappa_1\ \kappa_2\ \langle \mathbf{Type} \rangle \\
\qquad\qquad\qquad\qquad\quad \mathit{unify}\ \tau_1\ \tau_2\ \langle \kappa_1 \rangle
\end{array}
$$

$$
\mathit{unifyProp} :: \mathit{Prop} \to \mathit{Prop} \to \mathit{UM}\ ()
$$

$$
\mathit{unifyProp}\ (\tau_1\ {}^{\kappa_1}{\sim}^{\kappa_1'}\ \tau_1')\ (\tau_2\ {}^{\kappa_2}{\sim}^{\kappa_2'}\ \tau_2') = \mathit{unifyTys}\ [\kappa_1, \kappa_1', \tau_1, \tau_1']\ [\kappa_2, \kappa_2', \tau_2, \tau_2']
$$

Figure 7.1: A unification algorithm up to $\equiv$

the unification algorithm runs, as described in my prior work Eisenberg et al. [32, Section 3.3].

- Examine *unifyApp*. After unifying the types' kinds, it just passes a reflexive coercion when unifying the types themselves. This is correct because, by the time we are unifying the types, we know that the ambient substitution unifies the kinds. The coercion relating the types' kinds is thus now reflexive.

- In the $H_{\{\overline{\tau}\}}$ case, the algorithm does not make a separate call to unify kinds. This is because the $\overline{\tau}$ are always well typed under a *closed* telescope. Since *unifyTys* works left-to-right, the kinds of any later arguments must be unified by the time those types are considered.

I claim, but do not prove, that this unification algorithm satisfies the properties necessary for type safety. See Section C.3. For further discussion about the necessary properties of this algorithm, see `Note [Specification of Unification]` in `compiler/types/Unify.hs` in the GHC source code repository at `https://github.com/ghc/ghc`.

7.4 Parsing $\star$

As described in Section 2.3.1, the kind of types in Haskell has long been denoted as $\star$. This choice poses a parsing challenge in a language where types and kinds are intermixed. Types can include binary type operators (via the `TypeOperators` extension), and Haskellers have been using $\star$ as a binary infix operator on types for some time. (For example, in the standard library *GHC.TypeLits*.) The parsing problem is thus: is $\star$ an infix operator, or is it the kind of types?

GHC 8 offers two solutions to this problem, both already fully implemented. Firstly, forward-looking code should use the new constant **Type** to classify types. That is, we have *Int* :: **Type**. So as not to conflict with existing uses of datatypes named **Type**, this new **Type** is not always available but must be imported, from the new standard module *Data.Kind*. **Type** is available whether or not `TypeInType` is specified.

The other solution to this problem is to let the parsing of $\star$ depend on what $\star$ is in scope. This approach is to enable a smoother migration path for legacy code. Without `TypeInType` specified, $\star$ is available under its traditional meaning in code that is syntactically obviously a kind (for example, after a :: in a datatype declaration). When `TypeInType` is turned on, $\star$ is no longer available but must be imported from *Data.Kind*. This way, a module can choose to import *Data.Kind*'s $\star$ or a different $\star$, depending on its needs. Of course, the module could import these symbols qualified and use a module prefix at occurrence sites to choose which $\star$ is meant. Because $\star$ is treated as an ordinary imported symbol under `TypeInType`, module authors can now use standard techniques for managing name conflicts and migration.

In order to implement this second solution, the parser treats a space-separated sequence of type tokens as just that, without further interpretation. Only later, when we have a symbol table available, can we figure out how to deal with $\star$. This extra step of converting a sequence of tokens to a structured type expression outside of the parser actually dovetails with the existing step of fixity resolution, which similarly must happen only after a symbol table is available.

7.5 Promoting base types

This dissertation has dwelt a great deal on using algebraic datatypes in types and kinds. What about non-algebraic types, like *Int*, *Double*, or *Char*? These can be used in types just as easily as other values. The problem is in reducing operations on these types. For example, if a type mentions $5 - 8$, the normal type reduction process in the type-checker can replace this with (-3). However, what if we see $5 + x - x$ for an unknown x? We would surely like to be able to discover that $(5 + x - x) \sim 5$. Proving such equalities is difficult however.

It is here that a new innovation in GHC will come in quite handy: type-checker plugins. Diatchki [22] has already used the plugin interface (also described by Gundry [38]) to integrate an SMT solver into GHC's type-checker, in order to help with GHC's existing support for some type-level arithmetic. As more capabilities are added to types, the need for a powerful solver to deal with arithmetic equalities will grow. By having a plugin architecture, it is possible that individual users can use solvers tailored to their needs, and it will be easy for the community to increase the power of type-level reasoning in a distributed way. These plugins can easily be distributed with application code and so are appropriate for use even in deployment.

Chapter 8

Related and future work

There is a great deal of work related to this dissertation, looking at designs of similar surface languages, designs of similar intermediate languages, and similar type inference algorithms. This chapter reviews this related work, starting with a thorough comparison with the work of Gundry [37], which covers all of the areas above.

8.1 Comparison to Gundry's thesis

The most apt comparison of my work is to that of Gundry [37]. His dissertation is devoted to much the same goal as mine: adding dependent types to Haskell. I have tried to compare my work to his as this has been topical throughout this work. Here I summarize some of the key points of difference and explain how my work expands upon what he has done.

8.1.1 Unsaturated functions in types

Gundry's intermediate language uses one element of the grammar to represent both terms and types. But he offers separate typing judgments, as controlled by his use of a phase modality. In Gundry's type system, every typing judgment holds at one of three *phases*:[99] runtime, compile time, or shared (Gundry's Section 6.2). Gundry describes an *access policy* (Gundry's Section 6.2.1) whereby an expression well typed at the shared phase can also be used in either the runtime or compile-time phases. Gundry's use of phases is not unlike my use of relevance, where an expression well typed at Gundry's compile-time phase would be irrelevant in my formulation.

The big difference between my treatment and Gundry's is that I essentially combine the shared and runtime phases. That is, anything that is allowed at runtime is also allowed in types. Gundry prevents λ-expressions and unsaturated functions from being used in types. These constructs can be typed only at the runtime phase, never

[99] Actually, one of four, but both Gundry and I keep coercion typing so separate from other typing judgments that I am excluding it here.

the shared or compile-time phases. Because of this restriction around unsaturated functions, Gundry's system must carefully track where unsaturated functions appear and prevent any expression containing one from being used in a type or a dependent context.

I avoid Gundry's restriction by tracking matchable functions separate from unmatchable ones (Sections 4.2.4 and 5.8.6.4). This innovation permits me to allow unsaturated functions while retaining the useful **left** and **right** coercions. As a part of this aspect of my work, I also lift the matchable/unmatchable distinction into surface Dependent Haskell, giving the user access to the $'\rightarrow$, $'\Pi$, and $'\forall$ quantifiers.

8.1.2 Support for type families

Both Gundry's and my treatments favor λ-abstractions and **case** expressions over type families. In my case, I would support type families via compilation into those more primitive forms. Gundry's work, however, explicitly does not support type families (Gundry's Section 6.7.4). This lack of support is revealed in two missing features:

Matching on Type Through the way I have constructed my **case** expressions—specifically, treating **Type** as just another type constant—I allow pattern-matching on elements of **Type**. Gundry's treatment requires a scrutinee to be a member of a closed algebraic datatype.

Unsaturated matching Haskell type families can match on unsaturated uses of data and type constructors, something not supported in Gundry's work but supported in Pico.

8.1.3 Axioms

Gundry's *evidence* language includes support for axioms. While the notion of type-level axioms has been used in much prior work to represent type families, Gundry uses them to represent notions beyond those possible in type families, such as the commutativity of some primitive addition operation. In order to set up his consistency proof, he needs to establish that the axioms are *good*, as defined in Gundry's Definition 6.4 of his Section 6.5.1. Gundry does not provide an algorithm for determining whether a set of axioms are *good*, however.

Pico, in contrast, has no built-in support for axioms. One could try adding axioms as global coercion variables available in every context, but that would interfere with the current consistency proof (Section 5.10) which severely limits the use of coercion variables. It is conceivable that adding axioms to Pico is possible by establishing some condition, like Gundry's *good*, that claims that the axioms do not interfere with consistency. This remains as future work, however.

8.1.4 Type erasure

Gundry proves a type erasure property similar to mine. However, there is one key difference: my type erasure erases irrelevant abstractions (as does today's implementation of System FC in GHC), while Gundry's does not. It is not clear, however, that this change is significant, in that it might easily be possible to tweak Gundry's system to allow erasure of irrelevant abstractions, too.

See also Section 5.10.5.4 and Section 6.10.3 for further comments comparing my work to Gundry's.

8.2 Comparison to Idris

Of the available dependently typed language implementations, Idris is the most like Dependent Haskell. Idris was designed explicitly to answer the question "What if Haskell had *full* dependent types?" [9, Introduction] The Idris implementation is available[100] and is actively developed. So, how does Idris compare with Dependent Haskell? I review the main points of difference, below.

8.2.1 Backward compatibility

From a practical standpoint, the biggest difference between Dependent Haskell and Idris is that the former joins an already existing ecosystem of Haskell libraries and developers. Dependent Haskell is a conservative extension over existing implementations of Haskell, and all legacy programs will continue to work under Dependent Haskell. Although Idris is certainly Haskell-like (and has a foreign-function interface available to call Haskell code from Idris and vice versa) it is still not Haskell.

Pushing on this idea a bit more, for a project to be started in Idris, the programmers must decide, at the outset, that they wish to use dependent types, as its type system is Idris's most distinctive feature. With Dependent Haskell, on the other hand, developers can choose to take a part of a larger Haskell application and rewrite just that part with dependent types. This allows for gradual adoption, something that is much easier for the general public to swallow.

8.2.2 Type erasure

Dependent Haskell and Idris take different approaches to type erasure. Idris's approach is explained by Tejiščák and Brady [90] as a whole-program analysis, seeking out places where an expression is needed and ensuring that all such expressions are available at runtime. Naturally, such an approach hinders separate compilation, which the authors admit is important future work (Tejiščák and Brady's Section 8.1).

[100] `http://www.idris-lang.org/`

By contrast, Dependent Haskell depends on user-written choices—specifically, whether to use Π or $\forall$ when writing a type.

Which approach is better? It is hard to say at this point. The Idris approach has the advantage of automation. It may be hard for a user to know what expressions (especially those stored in datatypes) will be necessary at runtime. The choice between Π and $\forall$ may also motivate library-writers to duplicate their data structures providing both options. This is much like the fact that many current libraries provide both strict and lazy implementations of core data structures, as the better choice depends on a client's usage. Perhaps the option for library-writers to provide multiple versions of a datatype is an advantage, however: in Idris, a datatype's parameter may be marked as relevant even if it is used only once. In that case, the Idris programmer is perhaps better served by using one data structure (with the field irrelevant) in most places and the other data structure (with the field relevant) just where necessary. Time will tell whether the Dependent Haskell approach or the Idris approach is better.

8.2.3 Type inference

All Idris top-level definitions must be accompanied with type annotations. Even local definitions must have type annotations, sometimes requiring scoped type variables. One might say, then, that Idris does no type inference, only type checking. For this reason, studying the type inference properties of the language might be less compelling. Indeed, Brady claims [9, Section 6] that Idris "avoid[s] such difficulties since, in general, type inference is undecidable for full dependent types. Indeed, it is not clear that type inference is even desirable in many cases..."

While I admit that considering a principal-types property is much less compelling when all bindings are annotated, I still believe that writing a type inference algorithm or specification is helpful. I am unaware of a description in the literature of Idris's algorithm beyond Brady [9, Section 4], describing the elaboration of an Idris program in terms of the tactics that generate code in Idris's intermediate language, TT. Accordingly, it is hard to predict when an Idris program will be accepted. I tested the following program against the latest version of Idris (0.12.1):

```
ty:Bool → Type
ty x = case x of True ⇒ Integer; False ⇒ Char
f:(x:Bool) → ty x
f x = case x of True ⇒ 5; False ⇒ 'x'
g:(x:Bool) → ty x
g x = the (ty x)
          (case x of True ⇒ 5; False ⇒ 'x')
h:(x:Bool) → ty x
h x = the (case x of True ⇒ Integer; False ⇒ Char)
          (case x of True ⇒ 5; False ⇒ 'x')
```

Idris's *the* is its form of type annotation, with *the*:$(a$:**Type**$) \to a \to a$. Both *f* and *g* are accepted, while *h* is rejected. Note that the only difference between *g* and *h* is that the body of *ty* is expanded in *h*. Is this a bug or the correct behavior? It is hard to know.

In contrast, Chapter 6 describes a bidirectional inference algorithm that details how to treat such expressions. (All of *f*, *g*, and *h* are accepted in Dependent Haskell and today's approximation thereof using singletons.)

Beyond just having a specification, Dependent Haskell also retains Damas-Milner **let**-generalization for top-level expressions (as implemented by the IDecl_Synthesize rule of Bake). This means that simply typed functions and local declarations need not have type ascriptions. Indeed, in translating Idris's Effects library to Dependent Haskell (Section 3.2.3), I was able to eliminate several type annotations, needed in Idris but redundant in Haskell. Having **let**-generalization also powers examples like inferring the schema from the use of a dependently typed database access library (Section 3.1.3), the equivalent of which would be impossible in Idris.

8.2.4 Editor integration

One arena where Idris is clearly out ahead is in its user interface. Indeed, despite the fact that Idris is considerably younger, GHC has been clamoring to catch up to Idris's user interface for some time now. Its emacs integration means that users can interactively peruse error messages, expanding out the parts of interest and easily ignoring the unhelpful parts [17]. Dependent Haskell and GHC have much to learn from Idris in this respect; dependently typed programming in Haskell will demand improvement.

8.3 Comparison to Cayenne

Beyond Idris, there are many other languages one might want a comparison against. The most frequent comparison I have been asked for, however, is to compare against Cayenne [3], which I shall do here.

Cayenne is a language introduced in 1998 by Augustsson essentially as a dependently typed variant of Haskell. Of particular interest, it shares Dependent Haskell's cavalier attitude toward termination: Cayenne supports general recursion and all types are thus inhabited by $\bot$. Accordingly, Augustsson admits that Cayenne is not useful as a proof assistant. However, he also argues that this admission does not mean it is useless as a programming language. My argument in support of allowing general recursion in a dependently typed language (Section 4.4.5) broadly echoes Augustsson's Section 5, claiming that a verification of partial correctness is better than no verification at all.

Despite the similarities between my work here and Augustsson's, there are a number of key differences:

8.3.1 Type erasure

Augustsson's approach to type erasure is much simpler than mine. Cayenne erases all expressions of type **Type**—that's the full description of type erasure in Cayenne. This simplistic view has two shortcomings, however:

Cayenne erases too much Because every expression of type **Type** is lost, Cayenne must restrict its pattern-match facility not to work over scrutinees of type **Type**. Dependent Haskell allows matching on **Type**.

Cayenne erases too little Sometimes expressions of a type other than **Type** can be erased. For example, consider this function over length-indexed vectors (Section 3.1.1):

$$\begin{aligned} &\mathit{safeHead} :: \mathit{Vec}\ a\ (\text{'}\mathit{Succ}\ n) \to a \\ &\mathit{safeHead}\ (x :> \_) = x \end{aligned}$$

The n parameter to *safeHead* has type *Nat* and yet it can be erased in the call to *safeHead*. Cayenne would have no way of erasing this parameter.

8.3.2 Coercion assumptions

Cayenne has no support for equality assumptions. This means that it does not support GADTs (Section 2.4) or dependent pattern matching (Section 4.3.3). Lacking these features significantly simplifies the design of the language and implementation, meaning that many of the type inference issues (specifically, untouchability of type variables) described by Vytiniotis et al. [99] are avoided. The lack of equality assumptions also severely weakens Cayenne's ability to support intrinsic proofs—that is, types whose structure ensure that all values of those types are valid (like *Vec*, which ensures that the vector is of the given length). Cayenne thus truly supports only extrinsic proofs: proofs written separately from the functions and data structures they reason about. These proofs must be written explicitly (intrinsic proofs are often encoded into the structure of a function) and offer more opportunity to accidentally use a non-terminating proof.

8.3.3 A hierarchy of sorts

Cayenne uses an infinite hierarchy of sorts, similar to many other dependently typed languages, but in contrast to Dependent Haskell, with its **Type**:**Type** axiom. Augustsson describes this design decision as working in support of Cayenne's treatment as logical framework (if the user takes on the burden of termination checking) as well as to support Cayenne's implementation of type erasure.

8.3.4 Metatheory

While Augustsson presents typing rules for Cayenne, he offers no metatheory analysis for Cayenne beyond proving that the evaluation of a type-erased program simulates the evaluation of the original. Similarly, Augustsson does not describe any type inference properties in detail. The language requires top-level type annotations on all definitions, but inference is still necessary to check a dependently typed expression. Instead, Augustsson claims that "Type signatures can be omitted in many places" but does not elaborate [3, fourth-to-last bullet in Section 3.2]. Cayenne does syntactically require all function arguments to be annotated, however.

8.3.5 Modules

Cayenne has a robust module system, more advanced than Haskell's. As such, its module system is more advanced also than Dependent Haskell's. Cayenne uses dependent records as its modules, as a dependent record can store types as easily as other expressions. It remains as future work to see whether or not Dependent Haskell can incorporate these ideas and use records as modules.

8.3.6 Conclusion

As an early attempt to bring dependent types to Haskell, Cayenne deserves much credit. Despite being declared dead in 2005[101], Haskellers still discuss this language. It may have been the first thought-out vision of what a Haskell-like dependently typed language would look like and thus serves as an inspiration for both Agda and Idris.

8.4 Comparison to Liquid Haskell

Liquid Haskell [93–95] is an ongoing project seeking to add *refinement types* to Haskell. A refinement type specifies a head type and a condition; any value of the refinement type is asserted to meet the condition. For example, we might write the type of the *length* function thus:

$$\mathit{length} :: [a] \to \{\, n{:}\mathit{Int} \mid n \geqslant 0 \,\}$$

The return type tells us that the return value will always be non-negative.

The Liquid Haskell implementation works by reading in such annotations with a Haskell file and checking that the refinements are satisfied. The check is done via an SMT solver. No user intervention—other than writing the refinements in the first place—is required.

[101] `http://lambda-the-ultimate.org/node/802`

Liquid Haskell and Dependent Haskell are, in some ways, two different solutions to (nearly) the same problem: the desire to rule out erroneous programs. By specifying tight refinements on our function types, we can have Liquid Haskell check the correctness of our programs. And doing so is easy, thanks to the power of the SMT solver working in the background.

However, the refinement types of Liquid Haskell exist outside of the type system proper: it is not possible to write a type-level program that can manipulate refinements, and it is also not possible to write refinements that can reason about Haskell's type classes or other advanced type-level features. Along similar lines, it is not possible to use refinement types to write a program inadmissible in regular Haskell; for example, refinement types are not powerful enough to encode something like Idris's algebraic effects library (Section 3.2.3).

The beauty of Liquid Haskell is in its user interface. Proving that a program matches its specification is fully automatic—something very much not true of Dependent Haskell programs. The project has shown without a doubt that using an SMT solver to help type-checking will lessen users' proof burden. (Liquid Haskell is hardly the only tool that uses an SMT solver for type-checking. See also, for example, Leino [54] and Swamy et al. [89], among others.)

It is my hope that, someday, Dependent Haskell can be the backend for Liquid Haskell. The merged language would have the type refinement syntax much like Liquid Haskell's current syntax, but it would desugar to proper dependent types under the hood. An SMT solver would remain as part of the system, possibly as a type-checker plugin. For function arguments, supporting refinement types is already possible: a type like $\{ n{:}\mathit{Int} \mid n \geqslant 0 \}$ can be encoded as a dependent parameter n and a Haskell constraint. Much more problematic is a refined return type. For that same refinement, we would need a existential package, saying that a function returns *some* n with $n \geqslant 0$. While Dependent Haskell supports existentials, packing and unpacking these must be done manually. In practice, this packing and unpacking clutters the code considerably and makes the refinement approach distasteful. Perhaps worse, the packing and unpacking would be performed at runtime, making end users pay a cost for this compile-time checking. Overcoming these barriers—coming up with a lightweight syntax for existentials as well as zero runtime overhead—is important future work, perhaps my highest-priority new research direction.

8.5 Comparison to Trellys

The Trellys project [13, 14, 85] aims toward a similar goal to my work here: including dependent types in a language with non-termination. However, the Trellys approach is quite different from what I have done here, in that the language is formed of two fragments: a logical fragment and a programmatic fragment. The two halves share a syntax, but some constructs (such as general recursion) are allowed only in the programmatic fragment. Proofs in the logical fragment can be trusted (and never have

to be run) but can still mention definitions in the programmatic fragment in limited ways.

Zombie [85], one of the languages of the Trellys project, allows potentially non-terminating functions in types but retains decidable type-checking by forcing the user to indicate how much to β-reduce the types. This stands in contract to Dependent Haskell, where type-checking is undecidable.

8.6 Invisibility in other languages

Section 4.2.3 describes how Dependent Haskell deals with both visible and invisible function arguments. Here, I review how this feature is handled in several other dependently typed languages.

Agda In Agda, an argument in single braces $\{...\}$ is invisible and is instantiated via unification. An argument in double braces $\{\{...\}\}$ is invisible and is instantiated by looking for an in-scope variable of the right type. One Agda encoding of, say, the *Show* class and its *Show Bool* instance would be to make *Show* a record containing a *show* field (much like GHC's dictionary for *Show*) and a top-level variable of type *Show Bool*. The lookup process for $\{\{...\}\}$ arguments would then find this top-level variable.

Thus, *show*'s type in Agda might look like $\forall\ \{a\} \to \{\{\ \mathit{Show}\ a\ \}\} \to a \to \mathit{String}$.

Idris Idris supports type classes in much the same way as Haskell. A constraint listed before a ($\Rightarrow$) is solved just like a Haskell type class is. However, other invisible arguments can also have custom solving tactics. An Idris argument in single braces $\{...\}$ is solved via unification, just like in Agda. But a programmer may insert a proof script in the braces as well to trigger that proof script whenever the invisible parameter needs to be instantiated. For example, a type signature like *func*:{**default proof** {**trivial**} *pf*:τ} $\to$ ... names a (possibly dependent) parameter *pf*, of type τ. When *func* is called, Idris will run the **trivial** tactic to solve for a value of type τ. This value will then be inserted in for *pf*. Because a default proof script of **trivial** is so common, Idris offers an abbreviation **auto** which means **default proof** {**trivial**}.

Coq Coq has quite a different view of invisible arguments than do Dependent Haskell, Agda, and Idris. In all three of those languages, the visibility of an argument is part of a type. In Coq, top-level directives allow the programmer to change the visibility of arguments to already-defined functions. For example, if we have the definition

Definition *id* *A* (*x*:*A*) := *x*.

(without having used **Set Implicit Arguments**) both the *A* and *x* parameters are visible. Thus the following line is accepted:

Definition $mytrue_1 := id\ bool\ true$.

However, we can now change the visibility of the arguments to *id* with the directive

Arguments *id* $\{A\}\ x$.

allowing the following to be accepted:

Definition $mytrue_2 := id\ true$.

Although Coq does not allow the programmer to specify an instantiation technique for invisible arguments, it does allow the programmer to specify whether or not invisible arguments should be *maximally inserted*. A maximally inserted invisible argument is instantiated whenever possible; a non-maximally inserted argument is only instantiated when needed. For example, if the A argument to *id* were invisible and maximally inserted, then any use of *id* would immediately try to solve for A; if this were not possible, Coq would report a type error. If A were not maximally inserted, than a use of *id* would simply have the type **forall** $A, A \to A$, with no worry about invisible argument instantiation.

The issue of maximal insertion in Dependent Haskell is solved via its bidirectional type system (Section 6.4). The subsumption relation effectively ensures that the correct number of invisible parameters are provided, depending on the context.

8.7 Type erasure and relevance in other languages

Pico's approach to relevance and type erasure is distinctive and pervasive in its definition. Here I review several other approaches to type erasure in other languages and calculi.

Gundry's *evidence* language, Idris, and Cayenne See Sections 8.1.4, 8.2.2, and 8.3.1, respectively.

Agda The Agda wiki contains a comprehensive page on Agda's support for irrelevance annotations.[102] The user can annotate certain definitions and parameters as irrelevant, by preceding them with a . prefix. Irrelevant values can be used in irrelevant contexts only, much like how Pico treats irrelevantly bound variables. Irrelevant fields to a data constructor are ignored in an equality check, a feature that Pico does not currently support. For example, consider the following Agda program:

data T:*Set* **where**
  mkT: .(n:$\mathbb{N}$) $\to T$
data S:$T \to$ *Set* **where**

[102] `http://wiki.portal.chalmers.se/agda/pmwiki.php?n=ReferenceManual.Irrelevance`

$$
\begin{aligned}
&\mathit{mkS}\colon .(n{:}\mathbb{N}) \to S\ (\mathit{mkT}\ n)\\
&x{:}S\ (\mathit{mkT}\ 3)\\
&x = \mathit{mkS}\ 3\\
&y{:}S\ (\mathit{mkT}\ 4)\\
&y = x
\end{aligned}
$$

This program is accepted despite the fact that x and y have manifestly different types. Yet because the parameter to mkT is denoted as irrelevant, the types are considered equal. Note that, due to the restrictions around irrelevant contexts, if we remove the . prefix to the parameter to mkT, the constructor type for mkS would fail to type-check, because it uses its irrelevant argument n in a relevant context (as the argument to the now-relevant mkT constructor). Conversely, dropping the . in the type of mkS would not affect type checking.

It would be interesting future work to see how using relevance in this way might affect Dependent Haskell.

Despite having support for these irrelevance annotations, it seems that Agda does not have a well articulated type erasure property, instead depending on the extraction mechanism used to run Agda code.

Coq Coq uses an altogether different approach to relevance and erasure. Coq has two primary sorts, **Prop** and **Set**. (I am ignoring the infinite hierarchy of **Type**s that exist above **Prop** and **Set**.) All inhabitants of **Prop** are considered irrelevant and are erased during extraction. Coq thus enforces restrictions on the use of elements of types in **Prop**: chiefly, in the definition of an element of a type in **Set**, a program may not pattern-match on an element of a type in **Prop** unless that type has exactly 0 or 1 constructors. In other words, the choice of a value of a type in **Set** may not depend on any information from a type in **Prop**. This is sensible, because that information will disappear during extraction.

Because of Coq's separation between **Set** and **Prop**, it is sometimes necessary to have duplicate data structures, some with **Set** types and some with **Prop** types. (For example, the Coq standard library has three different variants of an existential package—*ex*, *sig* and *sigT*—depending on which parts are in **Prop** vs. **Set**.) Such duplication might also appear in Dependent Haskell, as I argue in Section 8.2.2.

ICC* Barras and Bernardo [7] introduce ICC* as a variant of Miquel's Implicit Calculus of Constructions [64]. ICC* contains two forms of Π-type as well as two forms of λ-extraction, in much the same way as Pico. The ICC literature uses "implicit" and "explicit" to refer to the concepts I call "irrelevant" and "relevant", respectively; I will continue to use my own terminology here. (Further muddying these waters, the original ICC also makes irrelevant arguments invisible. I have endeavored to keep visibility and relevance quite separate in this dissertation.) ICC* includes an erasure operation that converts ICC* expressions to ICC expressions by erasing irrelevant

arguments. In order to enforce appropriate use of irrelevant arguments, irrelevantly bound variables are forbidden from appearing in the erased, ICC-form of the body of an abstraction. This restriction is enforced by a simple check for free variables in the typing rule of the irrelevant λ-abstraction, in contrast to Pico's approach of tracking relevance in contexts. The Pico equivalent to ICC*'s approach would resemble this rule:

$$\frac{\Sigma; \Gamma, a{:}\sigma \vdash_{\mathsf{ty}} \tau : \kappa \qquad a \notin \mathsf{fv}(\lVert \tau \rVert)}{\Sigma; \Gamma \vdash_{\mathsf{ty}} \lambda a{:}_{\mathsf{Irrel}}\sigma.\, \tau : \Pi a{:}_{\mathsf{Irrel}}\sigma.\, \kappa} \quad \textsc{Ty\_Lam'}$$

It is possible that such a rule would simplify the statement of Pico, but I imagine it would complicate the proofs—especially of type erasure—as there would have to be a way of propagating the information about where irrelevant variables can appear.

8.8 Future directions

With the design for Dependent Haskell laid out here, what work is left to do? First and foremost, I must tackle the remainder of the implementation as sketched in Section 7.1.3. However, beyond that, there are many more research questions left unanswered:

- With the added complexity of dependent types, type error messages will surely become even harder to read and act on. How can these be improved? Idris's technique of displaying interactive error messages (Section 8.2.4) may be a step in the right direction, but it would be even better to have some theory of error messages to use as a guiding principle in solving this problem.

- Relatedly, dependent types work wonders for authors who wish to write an embedded domain-specific language. Programs might be written in such an EDSL by practitioners who do not know much type theory or Haskell. How can we expose a way for the DSL writer to customize the type error messages?

- What editor support is necessary to make dependent types in Haskell practical? Leading dependently typed languages (specifically, Coq, Agda, and Idris) all have quite advanced editor integration in order to make development more interactive. Haskell has some integration, but likely not enough to make dependently typed programming comfortable. What is missing here?

- Some dependently typed languages have found *tactics* a useful way of constructing proofs. Would such a technique be feasible in Dependent Haskell? What would such a facility look like?

- One of GHC's chief strengths is its optimizer. Once we have dependent types, can type-level information inform optimization in any meaningful way? In particular, using dependent types, an author might be able to write down "proofs" that a *Monad* instance is lawful. Can the optimizer take advantage of these proofs? Will we have to trust that they terminate to do so?

- How will dependent types interact with type-checker plugins? How can we use an SMT solver to make working with dependent types easier?
- Dependent types will allow for proper dependent pairs (Σ-types). Is it worth introducing new syntax to support these useful constructs directly? Would this new syntax also pave the way for better integration with Liquid Haskell (Section 8.4)?
- This dissertation has proved that the output of the Bake algorithm is a type-correct Pico program. It has not rigorously established, however, a principal types property or conservativity over today's Haskell. What steps are missing before we can prove these?
- One might reasonably ask whether all the fancy type-level bells and whistles affect parametricity. I do not believe they do, but it would be informative to try to prove this directly.

8.9 Conclusion

This chapter has really only scraped the surface of related work. There are simply too many dependently typed languages and calculi available to compare against all of them. In this crowd, however, Dependent Haskell stands out chiefly for its unapologetic embrace of non-termination and partial correctness. Dependent Haskell is, first and foremost, a programming language, and many valuable programs are indeed non-terminating or hard to prove to be total. These programs are welcome as first-class citizens in Dependent Haskell.

Appendix A

Typographical conventions

This dissertation is typeset using LaTeX with considerable help from lhs2TeX[103] and ott [82]. The lhs2TeX software allows Haskell code to be rendered more stylistically than a simple `verbatim` environment would allow. The table below maps Haskell source to glyphs appearing in this dissertation:

Haskell	Typeset	Description
`->`	$\rightarrow$	function arrow and other arrows
`=>`	$\Rightarrow$	constraint arrow
`*`	$\star$	the kind of types
`forall`	$\forall$	dependent irrelevant quantifier
`pi`	Π	dependent relevant quantifier
`++`	$+\!\!+$	list concatenation
`:~~:`	$:\approx:$	heterogeneous propositional equality
`:~>`	$:\rightsquigarrow$	lambda-calculus arrow (from Section 3.1.2)
`undefined`	$\bot$	canonical looping term

Figure A.1: Typesetting of Haskell constructs

In addition to the special formatting above, I assume a liberal overloading of number literals, including in types. For example, I write 2 where I really mean *Succ* (*Succ Zero*), depending on the context.

[103] `http://www.andres-loeh.de/lhs2tex/`

Appendix B

PICO typing rules, in full

B.1 Type constants

$\boxed{\Sigma \vdash_{\mathsf{tc}} H : \Delta_1; \Delta_2; H'}$ Type constant kinds, with universals Δ_1, existentials Δ_2, and result H'

$$\frac{}{\Sigma \vdash_{\mathsf{tc}} \mathbf{Type} : \varnothing; \varnothing; \mathbf{Type}} \quad \textsc{Tc\_Type}$$

$$\frac{T{:}(\overline{a}{:}\overline{\kappa}) \in \Sigma}{\Sigma \vdash_{\mathsf{tc}} T : \varnothing; \overline{a}{:}_{\mathsf{Rel}}\overline{\kappa}; \mathbf{Type}} \quad \textsc{Tc\_ADT}$$

$$\frac{K{:}(\Delta;\, T) \in \Sigma \qquad T{:}(\overline{a}{:}\overline{\kappa}) \in \Sigma}{\Sigma \vdash_{\mathsf{tc}} K : \overline{a}{:}_{\mathsf{Irrel}}\overline{\kappa}; \Delta;\, T} \quad \textsc{Tc\_DataCon}$$

B.2 Types

$\boxed{\Sigma; \Gamma \vdash_{\mathsf{ty}} \tau : \kappa}$ Type formation

$$\frac{\Sigma \vdash_{\mathsf{ctx}} \Gamma\ \mathsf{ok} \qquad a{:}_{\mathsf{Rel}}\kappa \in \Gamma}{\Sigma; \Gamma \vdash_{\mathsf{ty}} a : \kappa} \quad \textsc{Ty\_Var}$$

$$\frac{\begin{array}{l}\Sigma \vdash_{\mathsf{tc}} H : \Delta_1; \Delta_2; H' \qquad \Sigma \vdash_{\mathsf{ctx}} \Gamma\ \mathsf{ok} \\ \Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{vec}} \overline{\tau} : \mathsf{Rel}(\Delta_1)\end{array}}{\Sigma; \Gamma \vdash_{\mathsf{ty}} H_{\{\overline{\tau}\}} : \Pi(\Delta_2[\overline{\tau}/\mathsf{dom}(\Delta_1)]).\, H'\,\overline{\tau}} \quad \textsc{Ty\_Con}$$

$$\frac{\Sigma; \Gamma \vdash_{\mathsf{ty}} \tau_1 : \Pi a{:}_{\mathsf{Rel}}\kappa_1.\, \kappa_2 \qquad \Sigma; \Gamma \vdash_{\mathsf{ty}} \tau_2 : \kappa_1}{\Sigma; \Gamma \vdash_{\mathsf{ty}} \tau_1\, \tau_2 : \kappa_2[\tau_2/a]} \quad \textsc{Ty\_AppRel}$$

$$\frac{\Sigma; \Gamma \vdash_{\mathsf{ty}} \tau_1 : \Pi a{:}_{\mathsf{Irrel}} \kappa_1.\, \kappa_2 \qquad \Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{ty}} \tau_2 : \kappa_1}{\Sigma; \Gamma \vdash_{\mathsf{ty}} \tau_1\, \{\tau_2\} : \kappa_2[\tau_2/a]} \quad \textsc{Ty\_AppIrrel}$$

$$\frac{\Sigma; \Gamma \vdash_{\mathsf{ty}} \tau : \Pi c{:}\phi.\, \kappa \qquad \Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{co}} \gamma : \phi}{\Sigma; \Gamma \vdash_{\mathsf{ty}} \tau\, \gamma : \kappa[\gamma/c]} \quad \textsc{Ty\_CApp}$$

$$\frac{\Sigma; \Gamma, \mathsf{Rel}(\delta) \vdash_{\mathsf{ty}} \kappa : \mathbf{Type}}{\Sigma; \Gamma \vdash_{\mathsf{ty}} \Pi \delta.\, \kappa : \mathbf{Type}} \quad \textsc{Ty\_Pi}$$

$$\frac{\begin{array}{l}\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{co}} \gamma : \kappa_1 \sim \kappa_2 \\ \Sigma; \Gamma \vdash_{\mathsf{ty}} \tau : \kappa_1 \qquad \Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{ty}} \kappa_2 : \mathbf{Type}\end{array}}{\Sigma; \Gamma \vdash_{\mathsf{ty}} \tau \rhd \gamma : \kappa_2} \quad \textsc{Ty\_Cast}$$

$$\frac{\begin{array}{l}\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{ty}} \kappa : \mathbf{Type} \qquad \Sigma; \Gamma \vdash_{\mathsf{ty}} \tau : \sigma \\ \sigma \;=\; \tilde{\Pi}\Delta.\, H\, \overline{\sigma} \qquad \Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{ty}} H\, \overline{\sigma} : \mathbf{Type} \\ \forall i,\ \Sigma; \Gamma; \sigma \vdash^{\tau}_{\mathsf{alt}} alt_i : \kappa \\ \overline{alt} \text{ are exhaustive and distinct for } H, \text{ (w.r.t. } \Sigma)\end{array}}{\Sigma; \Gamma \vdash_{\mathsf{ty}} \mathbf{case}_{\kappa}\, \tau\, \mathbf{of}\, \overline{alt} : \kappa} \quad \textsc{Ty\_Case}$$

$$\frac{\Sigma; \Gamma, \delta \vdash_{\mathsf{ty}} \tau : \kappa}{\Sigma; \Gamma \vdash_{\mathsf{ty}} \lambda \delta.\, \tau : \tilde{\Pi} \delta.\, \kappa} \quad \textsc{Ty\_Lam}$$

$$\frac{\Sigma; \Gamma \vdash_{\mathsf{ty}} \tau : \tilde{\Pi} a{:}_{\mathsf{Rel}} \kappa.\, \kappa}{\Sigma; \Gamma \vdash_{\mathsf{ty}} \mathbf{fix}\, \tau : \kappa} \quad \textsc{Ty\_Fix}$$

$$\frac{\begin{array}{l}\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{co}} \gamma : H_{1\{\overline{\tau}_1\}}\, \overline{\psi}_1 \sim H_{2\{\overline{\tau}_2\}}\, \overline{\psi}_2 \qquad H_1 \neq H_2 \\ \Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{ty}} \tau : \mathbf{Type}\end{array}}{\Sigma; \Gamma \vdash_{\mathsf{ty}} \mathbf{absurd}\, \gamma\, \tau : \tau} \quad \textsc{Ty\_Absurd}$$

$\boxed{\Sigma; \Gamma; \sigma \vdash^{\tau}_{\mathsf{alt}} alt : \kappa}$ Case alternatives

$$\frac{\begin{array}{l}\Sigma \vdash_{\mathsf{tc}} H : \Delta_1; \Delta_2; H' \qquad \Delta_3, \Delta_4 \;=\; \Delta_2[\overline{\sigma}/\mathsf{dom}(\Delta_1)] \\ \mathsf{dom}(\Delta_4) \;=\; \mathsf{dom}(\Delta') \\ \mathsf{match}_{\{\mathsf{dom}(\Delta_3)\}}(\mathsf{types}(\Delta_4); \mathsf{types}(\Delta')) \;=\; \mathsf{Just}\, \theta \\ \Sigma; \Gamma \vdash_{\mathsf{ty}} \tau : \tilde{\Pi} \Delta_3, c{:}\tau_0 \sim H_{\{\overline{\sigma}\}}\, \mathsf{dom}(\Delta_3).\, \kappa\end{array}}{\Sigma; \Gamma; \tilde{\Pi}\Delta'.\, H'\, \overline{\sigma} \vdash^{\tau_0}_{\mathsf{alt}} H \to \tau : \kappa} \quad \textsc{Alt\_Match}$$

$$\frac{\Sigma; \Gamma \vdash_{\mathsf{ty}} \tau : \kappa}{\Sigma; \Gamma; \sigma \vdash^{\tau_0}_{\mathsf{alt}} \_ \to \tau : \kappa} \quad \textsc{Alt\_Default}$$

B.3 Coercions

$\boxed{\Sigma; \Gamma \vdash_{\mathsf{co}} \gamma : \phi}$ Coercion formation

$$\frac{\Sigma \vdash_{\mathsf{ctx}} \Gamma \ \mathsf{ok} \qquad c{:}\phi \in \Gamma}{\Sigma; \Gamma \vdash_{\mathsf{co}} c : \phi} \quad \textsc{Co\_Var}$$

$$\frac{\Sigma; \Gamma \vdash_{\mathsf{ty}} \tau : \kappa}{\Sigma; \Gamma \vdash_{\mathsf{co}} \langle \tau \rangle : \tau \sim \tau} \quad \textsc{Co\_Refl}$$

$$\frac{\Sigma; \Gamma \vdash_{\mathsf{co}} \gamma : \tau_1 \sim \tau_2}{\Sigma; \Gamma \vdash_{\mathsf{co}} \mathbf{sym}\, \gamma : \tau_2 \sim \tau_1} \quad \textsc{Co\_Sym}$$

$$\frac{\Sigma; \Gamma \vdash_{\mathsf{co}} \gamma_1 : \tau_1 \sim \tau_2 \qquad \Sigma; \Gamma \vdash_{\mathsf{co}} \gamma_2 : \tau_2 \sim \tau_3}{\Sigma; \Gamma \vdash_{\mathsf{co}} \gamma_1 \fatsemi \gamma_2 : \tau_1 \sim \tau_3} \quad \textsc{Co\_Trans}$$

$$\frac{\begin{array}{l}\Sigma; \Gamma \vdash_{\mathsf{co}} \eta : \kappa_1 \sim \kappa_2 \qquad \lfloor \tau_1 \rfloor = \lfloor \tau_2 \rfloor \\ \Sigma; \Gamma \vdash_{\mathsf{ty}} \tau_1 : \kappa_1 \qquad \Sigma; \Gamma \vdash_{\mathsf{ty}} \tau_2 : \kappa_2\end{array}}{\Sigma; \Gamma \vdash_{\mathsf{co}} \tau_1 \approx_\eta \tau_2 : \tau_1 \sim \tau_2} \quad \textsc{Co\_Coherence}$$

$$\frac{\begin{array}{l}\forall i,\ \Sigma; \Gamma \vdash_{\mathsf{co}} \gamma_i : \sigma_i \sim \sigma'_i \\ \Sigma; \Gamma \vdash_{\mathsf{ty}} H_{\{\overline{\sigma}\}} : \kappa_1 \qquad \Sigma; \Gamma \vdash_{\mathsf{ty}} H_{\{\overline{\sigma'}\}} : \kappa_2\end{array}}{\Sigma; \Gamma \vdash_{\mathsf{co}} H_{\{\overline{\gamma}\}} : H_{\{\overline{\sigma}\}} \sim H_{\{\overline{\sigma'}\}}} \quad \textsc{Co\_Con}$$

$$\frac{\begin{array}{l}\Sigma; \Gamma \vdash_{\mathsf{co}} \gamma_1 : \tau_1 \sim \tau_2 \\ \Sigma; \Gamma \vdash_{\mathsf{co}} \gamma_2 : \sigma_1 \sim \sigma_2 \\ \Sigma; \Gamma \vdash_{\mathsf{ty}} \tau_1\, \sigma_1 : \kappa_1 \qquad \Sigma; \Gamma \vdash_{\mathsf{ty}} \tau_2\, \sigma_2 : \kappa_2\end{array}}{\Sigma; \Gamma \vdash_{\mathsf{co}} \gamma_1\, \gamma_2 : \tau_1\, \sigma_1 \sim \tau_2\, \sigma_2} \quad \textsc{Co\_AppRel}$$

$$\frac{\begin{array}{l}\Sigma; \Gamma \vdash_{\mathsf{co}} \gamma_1 : \tau_1 \sim \tau_2 \\ \Sigma; \Gamma \vdash_{\mathsf{co}} \gamma_2 : \sigma_1 \sim \sigma_2 \\ \Sigma; \Gamma \vdash_{\mathsf{ty}} \tau_1\, \{\sigma_1\} : \kappa_1 \qquad \Sigma; \Gamma \vdash_{\mathsf{ty}} \tau_2\, \{\sigma_2\} : \kappa_2\end{array}}{\Sigma; \Gamma \vdash_{\mathsf{co}} \gamma_1\, \{\gamma_2\} : \tau_1\, \{\sigma_1\} \sim \tau_2\, \{\sigma_2\}} \quad \textsc{Co\_AppIrrel}$$

$$\frac{\begin{array}{l}\Sigma;\Gamma \vdash_{\mathsf{co}} \gamma_0 : \tau_1 \sim \tau_2 \\ \Sigma;\Gamma \vdash_{\mathsf{ty}} \tau_1\,\gamma_1 : \kappa_1 \qquad \Sigma;\Gamma \vdash_{\mathsf{ty}} \tau_2\,\gamma_2 : \kappa_2\end{array}}{\Sigma;\Gamma \vdash_{\mathsf{co}} \gamma_0\,(\gamma_1,\gamma_2) : \tau_1\,\gamma_1 \sim \tau_2\,\gamma_2}\ \textsc{Co\_CApp}$$

$$\frac{\begin{array}{l}\Sigma;\Gamma \vdash_{\mathsf{co}} \eta : \kappa_1\ {}^{\mathbf{Type}}\!\sim^{\mathbf{Type}}\ \kappa_2 \\ \Sigma;\Gamma, a{:}_{\mathsf{Rel}}\kappa_1 \vdash_{\mathsf{co}} \gamma : \sigma_1\ {}^{\mathbf{Type}}\!\sim^{\mathbf{Type}}\ \sigma_2\end{array}}{\Sigma;\Gamma \vdash_{\mathsf{co}} \Pi a{:}_\rho \eta.\,\gamma : (\Pi a{:}_\rho \kappa_1.\,\sigma_1) \sim (\Pi a{:}_\rho \kappa_2.\,(\sigma_2[a \rhd \mathbf{sym}\,\eta/a]))}\ \textsc{Co\_PiTy}$$

$$\frac{\begin{array}{l}\Sigma;\Gamma \vdash_{\mathsf{co}} \eta_1 : \tau_1 \sim \tau_2 \qquad \Sigma;\Gamma \vdash_{\mathsf{co}} \eta_2 : \sigma_1 \sim \sigma_2 \\ \Sigma;\Gamma, c{:}\tau_1 \sim \sigma_1 \vdash_{\mathsf{co}} \gamma : \kappa_1\ {}^{\mathbf{Type}}\!\sim^{\mathbf{Type}}\ \kappa_2 \qquad c\ \tilde{\#}\ \gamma \\ \eta_3\ =\ \eta_1 \fatsemi c \fatsemi \mathbf{sym}\,\eta_2\end{array}}{\Sigma;\Gamma \vdash_{\mathsf{co}} \Pi c{:}(\eta_1,\eta_2).\,\gamma : (\Pi c{:}\tau_1 \sim \sigma_1.\,\kappa_1) \sim (\Pi c{:}\tau_2 \sim \sigma_2.\,(\kappa_2[\eta_3/c]))}\ \textsc{Co\_PiCo}$$

$$\frac{\begin{array}{l}\Sigma;\Gamma \vdash_{\mathsf{co}} \eta : \kappa_1 \sim \kappa_2 \qquad \Sigma;\Gamma \vdash_{\mathsf{co}} \gamma_0 : \tau_1 \sim \tau_2 \\ \forall i,\ \Sigma;\Gamma \vdash_{\mathsf{co}} \gamma_i : \sigma_i \sim \sigma_i' \\ \overline{alt}_1\ =\ \overline{\pi_i \to \sigma_i} \qquad \overline{alt}_2\ =\ \overline{\pi_i \to \sigma_i'} \\ \Sigma;\Gamma \vdash_{\mathsf{ty}} \mathbf{case}_{\kappa_1}\,\tau_1\,\mathbf{of}\,\overline{alt}_1 : \kappa_1 \qquad \Sigma;\Gamma \vdash_{\mathsf{ty}} \mathbf{case}_{\kappa_2}\,\tau_2\,\mathbf{of}\,\overline{alt}_2 : \kappa_2\end{array}}{\Sigma;\Gamma \vdash_{\mathsf{co}} \mathbf{case}_\eta\,\gamma_0\,\mathbf{of}\,\overline{\pi_i \to \gamma_i} : \mathbf{case}_{\kappa_1}\,\tau_1\,\mathbf{of}\,\overline{alt}_1 \sim \mathbf{case}_{\kappa_2}\,\tau_2\,\mathbf{of}\,\overline{alt}_2}\ \textsc{Co\_Case}$$

$$\frac{\begin{array}{l}\Sigma;\Gamma \vdash_{\mathsf{co}} \eta : \kappa_1 \sim \kappa_2 \\ \Sigma;\Gamma, a{:}_\rho\kappa_1 \vdash_{\mathsf{co}} \gamma : \tau_1 \sim \tau_2 \\ \Sigma;\Gamma, a{:}_\rho\kappa_1 \vdash_{\mathsf{ty}} \tau_1 : \sigma_1 \qquad \Sigma;\Gamma, a{:}_\rho\kappa_1 \vdash_{\mathsf{ty}} \tau_2 : \sigma_2\end{array}}{\Sigma;\Gamma \vdash_{\mathsf{co}} \lambda a{:}_\rho\eta.\,\gamma : \lambda a{:}_\rho\kappa_1.\,\tau_1 \sim \lambda a{:}_\rho\kappa_2.\,(\tau_2[a \rhd \mathbf{sym}\,\eta/a])}\ \textsc{Co\_Lam}$$

$$\frac{\begin{array}{l}\Sigma;\Gamma \vdash_{\mathsf{co}} \eta_1 : \tau_1 \sim \tau_2 \qquad \Sigma;\Gamma \vdash_{\mathsf{co}} \eta_2 : \sigma_1 \sim \sigma_2 \\ \Sigma;\Gamma, c{:}\tau_1 \sim \sigma_1 \vdash_{\mathsf{co}} \gamma : \kappa_1 \sim \kappa_2 \qquad c\ \tilde{\#}\ \gamma \\ \eta_3\ =\ \eta_1 \fatsemi c \fatsemi \mathbf{sym}\,\eta_2\end{array}}{\Sigma;\Gamma \vdash_{\mathsf{co}} \lambda c{:}(\eta_1,\eta_2).\,\gamma : (\lambda c{:}\tau_1 \sim \sigma_1.\,\kappa_1) \sim (\lambda c{:}\tau_2 \sim \sigma_2.\,(\kappa_2[\eta_3/c]))}\ \textsc{Co\_CLam}$$

$$\frac{\begin{array}{l}\Sigma;\Gamma \vdash_{\mathsf{co}} \gamma : \tau_1 \sim \tau_2 \\ \Sigma;\Gamma \vdash_{\mathsf{ty}} \mathbf{fix}\,\tau_1 : \kappa_1 \qquad \Sigma;\Gamma \vdash_{\mathsf{ty}} \mathbf{fix}\,\tau_2 : \kappa_2\end{array}}{\Sigma;\Gamma \vdash_{\mathsf{co}} \mathbf{fix}\,\gamma : \mathbf{fix}\,\tau_1 \sim \mathbf{fix}\,\tau_2}\ \textsc{Co\_Fix}$$

$$\frac{\begin{array}{l}\Sigma;\Gamma \vdash_{\mathsf{co}} \gamma_1 : H_{1\{\overline{\tau}_1\}}\,\overline{\psi}_1 \sim H'_{1\{\overline{\tau}'_1\}}\,\overline{\psi}'_1 \qquad H_1 \neq H'_1 \\ \Sigma;\Gamma \vdash_{\mathsf{co}} \gamma_2 : H_{2\{\overline{\tau}_2\}}\,\overline{\psi}_2 \sim H'_{2\{\overline{\tau}'_2\}}\,\overline{\psi}'_2 \qquad H_2 \neq H'_2 \\ \Sigma;\Gamma \vdash_{\mathsf{co}} \eta : \kappa_1 \sim \kappa_2\end{array}}{\Sigma;\Gamma \vdash_{\mathsf{co}} \mathbf{absurd}\,(\gamma_1,\gamma_2)\,\eta : \mathbf{absurd}\,\gamma_1\,\kappa_1 \sim \mathbf{absurd}\,\gamma_2\,\kappa_2}\ \textsc{Co\_Absurd}$$

$$\frac{\Sigma;\Gamma \vdash_{\mathsf{co}} \gamma : (\Pi a{:}_{\rho}\kappa_1.\,\sigma_1) \sim (\Pi a{:}_{\rho}\kappa_2.\,\sigma_2)}{\Sigma;\Gamma \vdash_{\mathsf{co}} \mathbf{argk}\,\gamma : \kappa_1 \sim \kappa_2}\ \textsc{Co\_ArgK}$$

$$\frac{\Sigma;\Gamma \vdash_{\mathsf{co}} \gamma : (\Pi c{:}(\tau_1 \sim \tau_1').\,\sigma_1) \sim (\Pi c{:}(\tau_2 \sim \tau_2').\,\sigma_2)}{\Sigma;\Gamma \vdash_{\mathsf{co}} \mathbf{argk}_1\,\gamma : \tau_1 \sim \tau_2}\ \textsc{Co\_CArgK1}$$

$$\frac{\Sigma;\Gamma \vdash_{\mathsf{co}} \gamma : (\Pi c{:}(\tau_1 \sim \tau_1').\,\sigma_1) \sim (\Pi c{:}(\tau_2 \sim \tau_2').\,\sigma_2)}{\Sigma;\Gamma \vdash_{\mathsf{co}} \mathbf{argk}_2\,\gamma : \tau_1' \sim \tau_2'}\ \textsc{Co\_CArgK2}$$

$$\frac{\Sigma;\Gamma \vdash_{\mathsf{co}} \gamma : (\lambda a{:}_{\rho}\kappa_1.\,\sigma_1) \sim (\lambda a{:}_{\rho}\kappa_2.\,\sigma_2)}{\Sigma;\Gamma \vdash_{\mathsf{co}} \mathbf{argk}\,\gamma : \kappa_1 \sim \kappa_2}\ \textsc{Co\_ArgKLam}$$

$$\frac{\Sigma;\Gamma \vdash_{\mathsf{co}} \gamma : (\lambda c{:}(\tau_1 \sim \tau_1').\,\sigma_1) \sim (\lambda c{:}(\tau_2 \sim \tau_2').\,\sigma_2)}{\Sigma;\Gamma \vdash_{\mathsf{co}} \mathbf{argk}_1\,\gamma : \tau_1 \sim \tau_2}\ \textsc{Co\_CArgKLam1}$$

$$\frac{\Sigma;\Gamma \vdash_{\mathsf{co}} \gamma : (\lambda c{:}(\tau_1 \sim \tau_1').\,\sigma_1) \sim (\lambda c{:}(\tau_2 \sim \tau_2').\,\sigma_2)}{\Sigma;\Gamma \vdash_{\mathsf{co}} \mathbf{argk}_2\,\gamma : \tau_1' \sim \tau_2'}\ \textsc{Co\_CArgKLam2}$$

$$\frac{\begin{array}{l}\Sigma;\Gamma \vdash_{\mathsf{co}} \gamma : \Pi\Delta_1.\,\tau_1 \sim \Pi\Delta_2.\,\tau_2 \qquad |\Delta_1| = |\Delta_2| = n \\ \Sigma;\Gamma \vdash_{\mathsf{ty}} \tau_1 : \mathbf{Type} \qquad \Sigma;\Gamma \vdash_{\mathsf{ty}} \tau_2 : \mathbf{Type}\end{array}}{\Sigma;\Gamma \vdash_{\mathsf{co}} \mathbf{res}^n\,\gamma : \tau_1 \sim \tau_2}\ \textsc{Co\_Res}$$

$$\frac{\begin{array}{l}\Sigma;\Gamma \vdash_{\mathsf{co}} \gamma : \lambda\Delta_1.\,\tau_1 \sim \lambda\Delta_2.\,\tau_2 \qquad |\Delta_1| = |\Delta_2| = n \\ \Sigma;\Gamma \vdash_{\mathsf{ty}} \tau_1 : \kappa_1 \qquad \Sigma;\Gamma \vdash_{\mathsf{ty}} \tau_2 : \kappa_2\end{array}}{\Sigma;\Gamma \vdash_{\mathsf{co}} \mathbf{res}^n\,\gamma : \tau_1 \sim \tau_2}\ \textsc{Co\_ResLam}$$

$$\frac{\begin{array}{l}\Sigma;\Gamma \vdash_{\mathsf{co}} \gamma : \Pi a{:}_{\mathsf{Rel}}\kappa_1.\,\sigma_1 \sim \Pi a{:}_{\mathsf{Rel}}\kappa_2.\,\sigma_2 \\ \Sigma;\Gamma \vdash_{\mathsf{co}} \eta : \tau_1\ {}^{\kappa_1}{\sim}^{\kappa_2}\ \tau_2\end{array}}{\Sigma;\Gamma \vdash_{\mathsf{co}} \gamma@\eta : \sigma_1[\tau_1/a] \sim \sigma_2[\tau_2/a]}\ \textsc{Co\_InstRel}$$

$$\frac{\begin{array}{l}\Sigma;\Gamma \vdash_{\mathsf{co}} \gamma : \Pi a{:}_{\mathsf{Irrel}}\kappa_1.\,\sigma_1 \sim \Pi a{:}_{\mathsf{Irrel}}\kappa_2.\,\sigma_2 \\ \Sigma;\Gamma \vdash_{\mathsf{co}} \eta : \tau_1\ {}^{\kappa_1}{\sim}^{\kappa_2}\ \tau_2\end{array}}{\Sigma;\Gamma \vdash_{\mathsf{co}} \gamma@\{\eta\} : \sigma_1[\tau_1/a] \sim \sigma_2[\tau_2/a]}\ \textsc{Co\_InstIrrel}$$

$$\frac{\begin{array}{l}\Sigma;\Gamma \vdash_{\mathsf{co}} \eta_1 : \Pi c{:}\phi_1.\,\sigma_1 \sim \Pi c{:}\phi_2.\,\sigma_2 \\ \Sigma;\Gamma \vdash_{\mathsf{co}} \gamma_1 : \phi_1 \qquad \Sigma;\Gamma \vdash_{\mathsf{co}} \gamma_2 : \phi_2\end{array}}{\Sigma;\Gamma \vdash_{\mathsf{co}} \eta_1@(\gamma_1,\gamma_2) : \sigma_1[\gamma_1/c] \sim \sigma_2[\gamma_2/c]}\ \textsc{Co\_CInst}$$

$$\frac{\begin{array}{l}\Sigma;\Gamma \vdash_{\mathsf{co}} \gamma : \lambda a{:}_{\mathsf{Rel}}\kappa_1.\,\tau_1 \sim \lambda a{:}_{\mathsf{Rel}}\kappa_2.\,\tau_2 \\ \Sigma;\Gamma \vdash_{\mathsf{co}} \eta : \sigma_1 \ {}^{\kappa_1}\!\sim^{\kappa_2} \sigma_2\end{array}}{\Sigma;\Gamma \vdash_{\mathsf{co}} \gamma@\eta : \tau_1[\sigma_1/a] \sim \tau_2[\sigma_2/a]}\ \textsc{Co\_InstLamRel}$$

$$\frac{\begin{array}{l}\Sigma;\Gamma \vdash_{\mathsf{co}} \gamma : \lambda a{:}_{\mathsf{Irrel}}\kappa_1.\,\tau_1 \sim \lambda a{:}_{\mathsf{Irrel}}\kappa_2.\,\tau_2 \\ \Sigma;\Gamma \vdash_{\mathsf{co}} \eta : \sigma_1 \ {}^{\kappa_1}\!\sim^{\kappa_2} \sigma_2\end{array}}{\Sigma;\Gamma \vdash_{\mathsf{co}} \gamma@\{\eta\} : \tau_1[\sigma_1/a] \sim \tau_2[\sigma_2/a]}\ \textsc{Co\_InstLamIrrel}$$

$$\frac{\begin{array}{l}\Sigma;\Gamma \vdash_{\mathsf{co}} \gamma : \lambda c{:}\phi_1.\,\sigma_1 \sim \lambda c{:}\phi_2.\,\sigma_2 \\ \Sigma;\Gamma \vdash_{\mathsf{co}} \eta_1 : \phi_1 \qquad \Sigma;\Gamma \vdash_{\mathsf{co}} \eta_2 : \phi_2\end{array}}{\Sigma;\Gamma \vdash_{\mathsf{co}} \gamma@(\eta_1,\eta_2) : \sigma_1[\eta_1/c] \sim \sigma_2[\eta_2/c]}\ \textsc{Co\_CInstLam}$$

$$\frac{\begin{array}{l}\Sigma;\Gamma \vdash_{\mathsf{co}} \gamma : H_{\{\overline{\kappa}\}}\,\overline{\psi} \sim H_{\{\overline{\kappa}'\}}\,\overline{\psi}' \\ \psi_i = \tau \qquad \psi_i' = \sigma \\ \Sigma;\Gamma \vdash_{\mathsf{ty}} \tau : \kappa_1 \qquad \Sigma;\Gamma \vdash_{\mathsf{ty}} \sigma : \kappa_2\end{array}}{\Sigma;\Gamma \vdash_{\mathsf{co}} \mathbf{nth}_i\,\gamma : \tau \sim \sigma}\ \textsc{Co\_NthRel}$$

$$\frac{\begin{array}{l}\Sigma;\Gamma \vdash_{\mathsf{co}} \gamma : H_{\{\overline{\kappa}\}}\,\overline{\psi} \sim H_{\{\overline{\kappa}'\}}\,\overline{\psi}' \\ \psi_i = \{\tau\} \qquad \psi_i' = \{\sigma\} \\ \Sigma;\mathsf{Rel}(\Gamma) \vdash_{\mathsf{ty}} \tau : \kappa_1 \qquad \Sigma;\mathsf{Rel}(\Gamma) \vdash_{\mathsf{ty}} \sigma : \kappa_2\end{array}}{\Sigma;\Gamma \vdash_{\mathsf{co}} \mathbf{nth}_i\,\gamma : \tau \sim \sigma}\ \textsc{Co\_NthIrrel}$$

$$\frac{\begin{array}{l}\Sigma;\Gamma \vdash_{\mathsf{co}} \gamma : \tau_1\_\psi_1 \sim \tau_2\_\psi_2 \\ \Sigma;\Gamma \vdash_{\mathsf{ty}} \tau_1 : \text{'}\Pi\delta_1.\,\kappa_1 \qquad \Sigma;\Gamma \vdash_{\mathsf{ty}} \tau_2 : \text{'}\Pi\delta_2.\,\kappa_2 \\ \Sigma;\Gamma \vdash_{\mathsf{co}} \eta : \text{'}\Pi\delta_1.\,\kappa_1 \sim \text{'}\Pi\delta_2.\,\kappa_2\end{array}}{\Sigma;\Gamma \vdash_{\mathsf{co}} \mathbf{left}_\eta\,\gamma : \tau_1 \sim \tau_2}\ \textsc{Co\_Left}$$

$$\frac{\begin{array}{l}\Sigma;\Gamma \vdash_{\mathsf{co}} \gamma : \tau_1\_\sigma_1 \sim \tau_2\_\sigma_2 \\ \Sigma;\Gamma \vdash_{\mathsf{ty}} \sigma_1 : \kappa_1 \qquad \Sigma;\Gamma \vdash_{\mathsf{ty}} \sigma_2 : \kappa_2 \qquad \Sigma;\Gamma \vdash_{\mathsf{co}} \eta : \kappa_1 \sim \kappa_2\end{array}}{\Sigma;\Gamma \vdash_{\mathsf{co}} \mathbf{right}_\eta\,\gamma : \sigma_1 \sim \sigma_2}\ \textsc{Co\_RightRel}$$

$$\frac{\begin{array}{l}\Sigma;\Gamma \vdash_{\mathsf{co}} \gamma : \tau_{1\_}\{\sigma_1\} \sim \tau_{2\_}\{\sigma_2\} \\ \Sigma;\Gamma \vdash_{\mathsf{ty}} \sigma_1 : \kappa_1 \qquad \Sigma;\Gamma \vdash_{\mathsf{ty}} \sigma_2 : \kappa_2 \qquad \Sigma;\Gamma \vdash_{\mathsf{co}} \eta : \kappa_1 \sim \kappa_2\end{array}}{\Sigma;\Gamma \vdash_{\mathsf{co}} \mathbf{right}_\eta\, \gamma : \sigma_1 \sim \sigma_2}\ \textsc{Co\_RightIrrel}$$

$$\frac{\Sigma;\Gamma \vdash_{\mathsf{co}} \gamma : \tau_1 \ {}^{\kappa_1}\!\sim^{\kappa_2} \tau_2}{\Sigma;\Gamma \vdash_{\mathsf{co}} \mathbf{kind}\, \gamma : \kappa_1 \sim \kappa_2}\ \textsc{Co\_Kind}$$

$$\frac{\begin{array}{l}\Sigma;\Gamma \vdash_{\mathsf{ty}} \tau : \kappa \qquad \Sigma;\Gamma \vdash_{\mathsf{ty}} \tau' : \kappa \\ \Sigma;\Gamma \vdash_{\mathsf{s}} \tau \longrightarrow \tau'\end{array}}{\Sigma;\Gamma \vdash_{\mathsf{co}} \mathbf{step}\, \tau : \tau \sim \tau'}\ \textsc{Co\_Step}$$

$\boxed{\Sigma;\Gamma \vdash_{\mathsf{prop}} \phi\ \mathsf{ok}}$ Proposition formation

$$\frac{\begin{array}{c}\Sigma;\Gamma \vdash_{\mathsf{ty}} \tau_1 : \kappa_1 \\ \Sigma;\Gamma \vdash_{\mathsf{ty}} \tau_2 : \kappa_2\end{array}}{\Sigma;\Gamma \vdash_{\mathsf{prop}} \tau_1 \ {}^{\kappa_1}\!\sim^{\kappa_2} \tau_2\ \mathsf{ok}}\ \textsc{Prop\_Equality}$$

B.4 Vectors

$\boxed{\Sigma;\Gamma \vdash_{\mathsf{vec}} \overline{\psi} : \Delta}$ Type vector formation

$$\frac{\Sigma \vdash_{\mathsf{ctx}} \Gamma\ \mathsf{ok}}{\Sigma;\Gamma \vdash_{\mathsf{vec}} \varnothing : \varnothing}\ \textsc{Vec\_Nil}$$

$$\frac{\begin{array}{c}\Sigma;\Gamma \vdash_{\mathsf{ty}} \tau : \kappa \\ \Sigma;\Gamma \vdash_{\mathsf{vec}} \overline{\psi} : \Delta[\tau/a]\end{array}}{\Sigma;\Gamma \vdash_{\mathsf{vec}} \tau, \overline{\psi} : a{:}_{\mathsf{Rel}}\kappa, \Delta}\ \textsc{Vec\_TyRel}$$

$$\frac{\begin{array}{c}\Sigma;\mathsf{Rel}(\Gamma) \vdash_{\mathsf{ty}} \tau : \kappa \\ \Sigma;\Gamma \vdash_{\mathsf{vec}} \overline{\psi} : \Delta[\tau/a]\end{array}}{\Sigma;\Gamma \vdash_{\mathsf{vec}} \{\tau\}, \overline{\psi} : a{:}_{\mathsf{Irrel}}\kappa, \Delta}\ \textsc{Vec\_TyIrrel}$$

$$\frac{\begin{array}{c}\Sigma;\mathsf{Rel}(\Gamma) \vdash_{\mathsf{co}} \gamma : \phi \\ \Sigma;\Gamma \vdash_{\mathsf{vec}} \overline{\psi} : \Delta[\gamma/c]\end{array}}{\Sigma;\Gamma \vdash_{\mathsf{vec}} \gamma, \overline{\psi} : c{:}\phi, \Delta}\ \textsc{Vec\_Co}$$

$\boxed{\Sigma;\Gamma \vdash_{\mathsf{cev}} \overline{\psi} : \Delta}$ Vector formation, reversed

$$\frac{\Sigma \vdash_{\mathsf{ctx}} \Gamma\ \mathsf{ok}}{\Sigma; \Gamma \vdash_{\mathsf{cev}} \varnothing : \varnothing}\ \textsc{Cev\_Nil}$$

$$\frac{\begin{array}{l}\Sigma; \Gamma \vdash_{\mathsf{cev}} \overline{\psi} : \Delta \\ \Sigma; \Gamma \vdash_{\mathsf{ty}} \tau : \kappa[\overline{\psi}/\mathsf{dom}(\Delta)]\end{array}}{\Sigma; \Gamma \vdash_{\mathsf{cev}} \overline{\psi}, \tau : \Delta, a{:}_{\mathsf{Rel}}\kappa}\ \textsc{Cev\_TyRel}$$

$$\frac{\begin{array}{l}\Sigma; \Gamma \vdash_{\mathsf{cev}} \overline{\psi} : \Delta \\ \Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{ty}} \tau : \kappa[\overline{\psi}/\mathsf{dom}(\Delta)]\end{array}}{\Sigma; \Gamma \vdash_{\mathsf{cev}} \overline{\psi}, \tau : \Delta, a{:}_{\mathsf{Irrel}}\kappa}\ \textsc{Cev\_TyIrrel}$$

$$\frac{\begin{array}{l}\Sigma; \Gamma \vdash_{\mathsf{cev}} \overline{\psi} : \Delta \\ \Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{co}} \gamma : \phi[\overline{\psi}/\mathsf{dom}(\Delta)]\end{array}}{\Sigma; \Gamma \vdash_{\mathsf{cev}} \overline{\psi}, \gamma : \Delta, c{:}\phi}\ \textsc{Cev\_Co}$$

B.5 Contexts

$\boxed{\vdash_{\mathsf{sig}} \Sigma\ \mathsf{ok}}$ Signature formation

$$\frac{}{\vdash_{\mathsf{sig}} \varnothing\ \mathsf{ok}}\ \textsc{Sig\_Nil}$$

$$\frac{\Sigma \vdash_{\mathsf{ctx}} \overline{a}{:}_{\mathsf{Irrel}}\overline{\kappa}\ \mathsf{ok} \qquad T \# \Sigma}{\vdash_{\mathsf{sig}} \Sigma, T{:}(\overline{a}{:}\overline{\kappa})\ \mathsf{ok}}\ \textsc{Sig\_ADT}$$

$$\frac{T{:}(\overline{a}{:}\overline{\kappa}) \in \Sigma \qquad \Sigma \vdash_{\mathsf{ctx}} \overline{a}{:}_{\mathsf{Irrel}}\overline{\kappa}, \Delta\ \mathsf{ok} \qquad K \# \Sigma}{\vdash_{\mathsf{sig}} \Sigma, K{:}(\Delta; T)\ \mathsf{ok}}\ \textsc{Sig\_DataCon}$$

$\boxed{\Sigma \vdash_{\mathsf{ctx}} \Gamma\ \mathsf{ok}}$ Context formation

$$\frac{\vdash_{\mathsf{sig}} \Sigma\ \mathsf{ok}}{\Sigma \vdash_{\mathsf{ctx}} \varnothing\ \mathsf{ok}}\ \textsc{Ctx\_Nil}$$

$$\frac{\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{ty}} \kappa : \mathbf{Type} \qquad a \# \Gamma \qquad \Sigma \vdash_{\mathsf{ctx}} \Gamma\ \mathsf{ok}}{\Sigma \vdash_{\mathsf{ctx}} \Gamma, a{:}_{\rho}\kappa\ \mathsf{ok}}\ \textsc{Ctx\_TyVar}$$

$$\frac{\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{prop}} \phi\ \mathsf{ok} \qquad c \# \Gamma \qquad \Sigma \vdash_{\mathsf{ctx}} \Gamma\ \mathsf{ok}}{\Sigma \vdash_{\mathsf{ctx}} \Gamma, c{:}\phi\ \mathsf{ok}}\ \textsc{Ctx\_CoVar}$$

B.6 Small-step operational semantics

$\boxed{\Sigma; \Gamma \vdash_{\mathsf{s}} \sigma \longrightarrow \sigma'}$ Small-step operational semantics

$$\frac{}{\Sigma; \Gamma \vdash_{\mathsf{s}} (\lambda a{:}_{\mathsf{Rel}}\kappa.\, \sigma_1)\_\sigma_2 \longrightarrow \sigma_1[\sigma_2/a]}\ \textsc{S\_BetaRel}$$

$$\frac{}{\Sigma; \Gamma \vdash_{\mathsf{s}} (\lambda a{:}_{\mathsf{Irrel}}\kappa.\, v_1)\_\{\sigma_2\} \longrightarrow v_1[\sigma_2/a]}\ \textsc{S\_BetaIrrel}$$

$$\frac{}{\Sigma; \Gamma \vdash_{\mathsf{s}} (\lambda c{:}\phi.\, \sigma)\_\gamma \longrightarrow \sigma[\gamma/c]}\ \textsc{S\_CBeta}$$

$$\frac{alt_i\ =\ H \to \tau_0}{\Sigma; \Gamma \vdash_{\mathsf{s}} \mathbf{case}_\kappa\, H_{\{\overline{\tau}\}}\, \overline{\psi}\, \mathbf{of}\, \overline{alt} \longrightarrow \tau_0\, \overline{\psi}\, \langle H_{\{\overline{\tau}\}}\, \overline{\psi}\rangle}\ \textsc{S\_Match}$$

$$\frac{alt_i\ =\ \_ \to \sigma \qquad \text{no alternative in } \overline{alt} \text{ matches } H}{\Sigma; \Gamma \vdash_{\mathsf{s}} \mathbf{case}_\kappa\, H_{\{\overline{\tau}\}}\, \overline{\psi}\, \mathbf{of}\, \overline{alt} \longrightarrow \sigma}\ \textsc{S\_Default}$$

$$\frac{alt_i\ =\ \_ \to \sigma \qquad \text{no alternative in } \overline{alt} \text{ matches } H}{\Sigma; \Gamma \vdash_{\mathsf{s}} \mathbf{case}_\kappa\, H_{\{\overline{\tau}\}}\, \overline{\psi} \rhd \gamma\, \mathbf{of}\, \overline{alt} \longrightarrow \sigma}\ \textsc{S\_DefaultCo}$$

$$\frac{\tau\ =\ \lambda a{:}_{\mathsf{Rel}}\kappa.\, \sigma}{\Sigma; \Gamma \vdash_{\mathsf{s}} \mathbf{fix}\, \tau \longrightarrow \sigma[\mathbf{fix}\, \tau/a]}\ \textsc{S\_Unroll}$$

$$\frac{}{\Sigma; \Gamma \vdash_{\mathsf{s}} (v \rhd \gamma_1) \rhd \gamma_2 \longrightarrow v \rhd (\gamma_1 \fatsemi \gamma_2)}\ \textsc{S\_Trans}$$

$$\frac{\Sigma; \Gamma, a{:}_{\mathsf{Irrel}}\kappa \vdash_{\mathsf{s}} \sigma \longrightarrow \sigma'}{\Sigma; \Gamma \vdash_{\mathsf{s}} \lambda a{:}_{\mathsf{Irrel}}\kappa.\, \sigma \longrightarrow \lambda a{:}_{\mathsf{Irrel}}\kappa.\, \sigma'}\ \textsc{S\_IrrelAbs\_Cong}$$

$$\frac{\Sigma; \Gamma \vdash_{\mathsf{s}} \sigma \longrightarrow \sigma'}{\Sigma; \Gamma \vdash_{\mathsf{s}} \sigma\, \psi \longrightarrow \sigma'\, \psi}\ \textsc{S\_App\_Cong}$$

$$\frac{\Sigma; \Gamma \vdash_{\mathsf{s}} \sigma \longrightarrow \sigma'}{\Sigma; \Gamma \vdash_{\mathsf{s}} \sigma \rhd \gamma \longrightarrow \sigma' \rhd \gamma} \quad \text{S\_Cast\_Cong}$$

$$\frac{\Sigma; \Gamma \vdash_{\mathsf{s}} \sigma \longrightarrow \sigma'}{\Sigma; \Gamma \vdash_{\mathsf{s}} \mathbf{case}_{\tau}\, \sigma\, \mathbf{of}\, \overline{alt} \longrightarrow \mathbf{case}_{\tau}\, \sigma'\, \mathbf{of}\, \overline{alt}} \quad \text{S\_Case\_Cong}$$

$$\frac{\Sigma; \Gamma \vdash_{\mathsf{s}} \tau \longrightarrow \tau'}{\Sigma; \Gamma \vdash_{\mathsf{s}} \mathbf{fix}\, \tau \longrightarrow \mathbf{fix}\, \tau'} \quad \text{S\_Fix\_Cong}$$

$$\frac{\begin{array}{l}\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{co}} \gamma_0 : \Pi a{:}_{\mathsf{Rel}}\kappa.\, \sigma \sim \Pi a{:}_{\mathsf{Rel}}\kappa'.\, \sigma' \\ \gamma_1 = \mathbf{sym}\,(\mathbf{argk}\,\gamma_0) \qquad \gamma_2 = \gamma_0@(\tau \rhd \gamma_1 \approx_{\mathbf{sym}\,\gamma_1} \tau)\end{array}}{\Sigma; \Gamma \vdash_{\mathsf{s}} (v \rhd \gamma_0)\, \tau \longrightarrow v\, (\tau \rhd \gamma_1) \rhd \gamma_2} \quad \text{S\_PushRel}$$

$$\frac{\begin{array}{l}\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{co}} \gamma_0 : \Pi a{:}_{\mathsf{Irrel}}\kappa.\, \sigma \sim \Pi a{:}_{\mathsf{Irrel}}\kappa'.\, \sigma' \\ \gamma_1 = \mathbf{sym}\,(\mathbf{argk}\,\gamma_0) \qquad \gamma_2 = \gamma_0@(\tau \rhd \gamma_1 \approx_{\mathbf{sym}\,\gamma_1} \tau)\end{array}}{\Sigma; \Gamma \vdash_{\mathsf{s}} (v \rhd \gamma_0)\, \{\tau\} \longrightarrow v\, \{\tau \rhd \gamma_1\} \rhd \gamma_2} \quad \text{S\_PushIrrel}$$

$$\frac{\begin{array}{l}\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{co}} \gamma_0 : \Pi c{:}\phi.\, \sigma \sim \Pi c{:}\phi'.\, \sigma' \\ \gamma_1 = \mathbf{argk}_1\, \gamma_0 \qquad \gamma_2 = \mathbf{argk}_2\, \gamma_0 \\ \eta' = \gamma_1 \fatsemi \eta \fatsemi \mathbf{sym}\, \gamma_2 \qquad \gamma_3 = \gamma_0@(\eta', \eta)\end{array}}{\Sigma; \Gamma \vdash_{\mathsf{s}} (v \rhd \gamma_0)\, \eta \longrightarrow v\, \eta' \rhd \gamma_3} \quad \text{S\_CPush}$$

$$\frac{\begin{array}{l}\gamma_1 = \tilde{\Pi} a{:}_{\mathsf{Irrel}}\langle\kappa\rangle.\, \gamma \qquad \gamma_2 = \tau_1 \approx_{\langle \mathbf{Type}\rangle} \tau_2 \\ \tau_1 = \tilde{\Pi} a{:}_{\mathsf{Irrel}}\kappa.\, (\kappa_1[a \rhd \mathbf{sym}\,\langle\kappa\rangle/a]) \qquad \tau_2 = \tilde{\Pi} a{:}_{\mathsf{Irrel}}\kappa.\, \kappa_1\end{array}}{\Sigma; \Gamma \vdash_{\mathsf{s}} \lambda a{:}_{\mathsf{Irrel}}\kappa.\, (v \rhd \gamma) \longrightarrow (\lambda a{:}_{\mathsf{Irrel}}\kappa.\, v) \rhd (\gamma_1 \fatsemi \gamma_2)} \quad \text{S\_APush}$$

$$\frac{\begin{array}{l}\gamma_1 = \gamma_0@(a \approx_{\gamma_2} a \rhd \gamma_2) \fatsemi \mathbf{sym}\, \gamma_2 \\ \gamma_2 = \mathbf{argk}\, \gamma_0\end{array}}{\Sigma; \Gamma \vdash_{\mathsf{s}} \mathbf{fix}\, ((\lambda a{:}_{\mathsf{Rel}}\kappa.\, \sigma) \rhd \gamma_0) \longrightarrow (\mathbf{fix}\, (\lambda a{:}_{\mathsf{Rel}}\kappa.\, (\sigma \rhd \gamma_1))) \rhd \gamma_2} \quad \text{S\_FPush}$$

$$\frac{\begin{array}{l}\Sigma \vdash_{\mathsf{tc}} H : \overline{a}{:}_{\mathsf{Irrel}}\overline{\kappa}; \Delta; H' \qquad \Delta = \Delta_1, \Delta_2 \qquad n = |\Delta_2| \\ \kappa = \text{'}\Pi\overline{a}{:}_{\mathsf{Irrel}}\overline{\kappa}, \Delta.\, H'\, \overline{a} \\ \sigma = \text{'}\Pi(\Delta_2[\overline{\tau}/\overline{a}][\overline{\psi}/\mathsf{dom}(\Delta_1)]).\, H'\, \overline{\tau} \\ \sigma' = \text{'}\Pi(\Delta_2[\overline{\tau}'/\overline{a}][\overline{\psi}'/\mathsf{dom}(\Delta_1)]).\, H'\, \overline{\tau}' \\ \Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{co}} \eta : \sigma \sim \sigma' \\ \Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{vec}} \overline{\tau}' : \overline{a}{:}_{\mathsf{Rel}}\overline{\kappa} \\ \forall i,\ \gamma_i = \mathsf{build\_kpush\_co}(\langle\kappa\rangle @ (\mathbf{nths}\,(\mathbf{res}^n\,\eta)); \overline{\psi}_{1\ldots i-1}) \\ \forall i,\ \psi_i' = \mathsf{cast\_kpush\_arg}(\psi_i; \gamma_i) \\ H \to \kappa' \in \overline{alt}\end{array}}{\Sigma; \Gamma \vdash_{\mathsf{s}} \mathbf{case}_{\kappa_0}\,(H_{\{\overline{\tau}\}}\,\overline{\psi}) \rhd \eta\,\mathbf{of}\,\overline{alt} \longrightarrow \mathbf{case}_{\kappa_0}\,H_{\{\overline{\tau}'\}}\,\overline{\psi}'\,\mathbf{of}\,\overline{alt}}\ \text{S\_KPush}$$

B.7 Consistency

$\boxed{\tau_1 \propto \tau_2}$ Type compatibility

$$\frac{\tau_1 \text{ is not a value}}{\tau_1 \propto \tau_2}\ \text{C\_NonValue1}$$

$$\frac{\tau_2 \text{ is not a value}}{\tau_1 \propto \tau_2}\ \text{C\_NonValue2}$$

$$\frac{}{H_{\{\overline{\tau}\}}\,\overline{\psi} \propto H_{\{\overline{\tau}'\}}\,\overline{\psi}'}\ \text{C\_TyCon}$$

$$\frac{\tau \propto \tau'}{\Pi a{:}_\rho\kappa.\,\tau \propto \Pi a{:}_\rho\kappa'.\,\tau'}\ \text{C\_PiTy}$$

$$\frac{}{\Pi c{:}\phi.\,\tau \propto \Pi c{:}\phi'.\,\tau'}\ \text{C\_PiCo}$$

$$\frac{}{\lambda\delta.\,\tau \propto \lambda\delta'.\,\tau'}\ \text{C\_Lam}$$

$\boxed{\tau \rightsquigarrow \tau'}$ Parallel reduction over erased types

$$\frac{}{\tau \rightsquigarrow \tau}\ \text{R\_Refl}$$

$$\frac{\overline{\tau} \rightsquigarrow \overline{\tau}'}{H_{\{\overline{\tau}\}} \rightsquigarrow H_{\{\overline{\tau}'\}}} \quad \text{R\_Con}$$

$$\frac{\tau \rightsquigarrow \tau' \qquad \sigma \rightsquigarrow \sigma'}{\tau\, \sigma \rightsquigarrow \tau'\, \sigma'} \quad \text{R\_AppRel}$$

$$\frac{\tau \rightsquigarrow \tau' \qquad \sigma \rightsquigarrow \sigma'}{\tau\, \{\sigma\} \rightsquigarrow \tau'\, \{\sigma'\}} \quad \text{R\_AppIrrel}$$

$$\frac{\tau \rightsquigarrow \tau'}{\tau\, \bullet \rightsquigarrow \tau'\, \bullet} \quad \text{R\_CApp}$$

$$\frac{\delta \rightsquigarrow \delta' \qquad \tau \rightsquigarrow \tau'}{\Pi \delta.\, \tau \rightsquigarrow \Pi \delta'.\, \tau'} \quad \text{R\_Pi}$$

$$\frac{\kappa \rightsquigarrow \kappa' \qquad \tau \rightsquigarrow \tau' \qquad \overline{\sigma} \rightsquigarrow \overline{\sigma}'}{\mathbf{case}_{\kappa}\, \tau\, \mathbf{of}\, \overline{\pi \to \sigma} \rightsquigarrow \mathbf{case}_{\kappa'}\, \tau'\, \mathbf{of}\, \overline{\pi \to \sigma'}} \quad \text{R\_Case}$$

$$\frac{\delta \rightsquigarrow \delta' \qquad \tau \rightsquigarrow \tau'}{\lambda \delta.\, \tau \rightsquigarrow \lambda \delta'.\, \tau'} \quad \text{R\_Lam}$$

$$\frac{\tau \rightsquigarrow \tau'}{\mathbf{fix}\, \tau \rightsquigarrow \mathbf{fix}\, \tau'} \quad \text{R\_Fix}$$

$$\frac{\tau \rightsquigarrow \tau'}{\mathbf{absurd}\, \bullet\, \tau \rightsquigarrow \mathbf{absurd}\, \bullet\, \tau'} \quad \text{R\_Absurd}$$

$$\frac{\tau_1 \rightsquigarrow \tau_1' \qquad \tau_2 \rightsquigarrow \tau_2'}{(\lambda a{:}_{\mathsf{Rel}} \kappa.\, \tau_1)\_\tau_2 \rightsquigarrow \tau_1'[\tau_2'/a]} \quad \text{R\_BetaRel}$$

$$\frac{\tau_1 \rightsquigarrow \tau_1' \qquad \tau_2 \rightsquigarrow \tau_2'}{(\lambda a{:}_{\mathsf{Irrel}} \kappa.\, \tau_1)\_\{\tau_2\} \rightsquigarrow \tau_1'[\tau_2'/a]} \quad \text{R\_BetaIrrel}$$

$$\frac{\tau \rightsquigarrow \tau'}{(\lambda \bullet{:}\phi.\, \tau)\_\bullet \rightsquigarrow \tau'} \quad \text{R\_CBeta}$$

$$\frac{alt_i \;=\; H \to \tau_0 \qquad \overline{\psi} \rightsquigarrow \overline{\psi}' \qquad \tau_0 \rightsquigarrow \tau_0'}{\mathbf{case}_\kappa\, H_{\{\overline{\tau}\}}\, \overline{\psi}\, \mathbf{of}\, \overline{alt} \rightsquigarrow \tau_0'\, \overline{\psi}'\, \bullet} \quad \text{R\_Match}$$

$$\frac{alt_i \;=\; \_ \to \sigma \qquad \text{no alternative in } \overline{alt} \text{ matches } H \qquad \sigma \rightsquigarrow \sigma'}{\mathbf{case}_\kappa\, H_{\{\overline{\tau}\}}\, \overline{\psi}\, \mathbf{of}\, \overline{alt} \rightsquigarrow \sigma'} \quad \text{R\_Default}$$

$$\frac{\sigma \rightsquigarrow \sigma' \qquad \kappa \rightsquigarrow \kappa'}{\mathbf{fix}\,(\lambda a{:}_{\mathsf{Rel}}\kappa.\,\sigma) \rightsquigarrow \sigma'[\mathbf{fix}\,(\lambda a{:}_{\mathsf{Rel}}\kappa'.\,\sigma')/a]} \quad \text{R\_Unroll}$$

$\boxed{\delta \rightsquigarrow \delta'}$ Parallel reduction of binders

$$\frac{\kappa \rightsquigarrow \kappa'}{a{:}_\rho\kappa \rightsquigarrow a{:}_\rho\kappa'} \quad \text{R\_TyBinder}$$

$$\frac{\tau \rightsquigarrow \tau' \qquad \kappa_1 \rightsquigarrow \kappa_1' \qquad \kappa_2 \rightsquigarrow \kappa_2' \qquad \sigma \rightsquigarrow \sigma'}{\bullet{:}\tau\ {}^{\kappa_1}{\sim}^{\kappa_2}\ \sigma \rightsquigarrow \bullet{:}\tau'\ {}^{\kappa_1'}{\sim}^{\kappa_2'}\ \sigma'} \quad \text{R\_CoBinder}$$

$\boxed{\gamma \rightsquigarrow \gamma'}$ "Reduction" of erased coercion

$$\frac{}{\bullet \rightsquigarrow \bullet} \quad \text{R\_ErasedCo}$$

B.8 Small-step operational semantics of erased expressions

$\boxed{e \longrightarrow e'}$ Single-step operational semantics of expressions

$$\frac{}{(\lambda a.e_1)\, e_2 \longrightarrow e_1[e_2/a]} \quad \text{E\_Beta}$$

$$\frac{}{(\lambda \bullet.e)\, \bullet \longrightarrow e} \quad \text{E\_CBeta}$$

$$\frac{ealt_i \;=\; H \to e}{\mathbf{case}\, H\, \overline{y}\, \mathbf{of}\, \overline{ealt} \longrightarrow e\, \overline{y}\, \bullet} \quad \text{E\_Match}$$

$$\frac{ealt_i \;=\; \_ \to e \qquad \text{no alternative in } \overline{ealt} \text{ matches } H}{\mathbf{case}\, H\, \overline{y}\, \mathbf{of}\, \overline{ealt} \longrightarrow e} \quad \text{E\_Default}$$

$$\frac{}{\mathbf{fix}\,(\lambda a.e) \longrightarrow e[\mathbf{fix}\,(\lambda a.e)/a]} \quad \text{E\_Unroll}$$

$$\frac{e \longrightarrow e'}{e\ y \longrightarrow e'\ y} \quad \text{E\_App\_Cong}$$

$$\frac{e \longrightarrow e'}{\mathbf{case}\ e\ \mathbf{of}\ \overline{ealt} \longrightarrow \mathbf{case}\ e'\ \mathbf{of}\ \overline{ealt}} \quad \text{E\_Case\_Cong}$$

$$\frac{e \longrightarrow e'}{\mathbf{fix}\ e \longrightarrow \mathbf{fix}\ e'} \quad \text{E\_Fix\_Cong}$$

Appendix C

Proofs about Pico

You may find the full grammar for Pico in Figure 5.1 on page 76 and its notation conventions in Figure 5.2 on page 77. The definition for values is in Section 5.7.1 and of the $\tilde{\#}$ operator in Section 5.8.5.2.

C.1 Auxiliary definitions

Definition C.1 (Free variables)**.** *Define* fv *to be a function extracting free variables, overloaded to work over types* τ*, coercions* γ*, propositions* ϕ*, vectors* ψ*, alternatives alt, and telescopes* Δ*. The definitions are entirely standard.*

Definition C.2 (Context extension)**.** *Define the relation* $\Gamma \subseteq \Gamma'$ *to mean that* Γ *is a (not necessarily contiguous) subsequence of* Γ'*.*

C.2 Structural properties

C.2.1 Relevant contexts

Lemma C.3 ($\mathsf{dom}/\mathsf{Rel}$)**.** $\mathsf{dom}(\mathsf{Rel}(\Gamma)) = \mathsf{dom}(\Gamma)$

Proof. By its definition $\mathsf{Rel}(\Gamma)$ binds the same variables as Γ. □

Lemma C.4 (Subsequence/Rel)**.** *If* $\Gamma \subseteq \Gamma'$ *then* $\mathsf{Rel}(\Gamma) \subseteq \mathsf{Rel}(\Gamma')$*.*

Proof. By the definitions of $\subseteq$ and Rel. □

Lemma C.5 (Rel is idempotent)**.** $\mathsf{Rel}(\mathsf{Rel}(\Gamma)) = \mathsf{Rel}(\Gamma)$

Proof. By the definition of Rel. □

Lemma C.6 (Increasing relevance)**.** *Let* Γ *and* Γ' *be the same except that some bindings in* Γ' *are labeled* Rel *where those same bindings in* Γ *are labeled* Irrel*.*

1. *If* $\Sigma; \Gamma \vdash_{\mathsf{ty}} \tau : \kappa$, *then* $\Sigma; \Gamma' \vdash_{\mathsf{ty}} \tau : \kappa$.
2. *If* $\Sigma; \Gamma \vdash_{\mathsf{co}} \gamma : \phi$, *then* $\Sigma; \Gamma' \vdash_{\mathsf{co}} \gamma : \phi$.
3. *If* $\Sigma; \Gamma \vdash_{\mathsf{prop}} \phi\ \mathsf{ok}$, *then* $\Sigma; \Gamma' \vdash_{\mathsf{prop}} \phi\ \mathsf{ok}$.
4. *If* $\Sigma; \Gamma; \sigma_0 \vdash^{\tau_0}_{\mathsf{alt}} alt : \kappa$, *then* $\Sigma; \Gamma'; \sigma_0 \vdash^{\tau_0}_{\mathsf{alt}} alt : \kappa$.
5. *If* $\Sigma; \Gamma \vdash_{\mathsf{vec}} \overline{\psi} : \Delta$, *then* $\Sigma; \Gamma' \vdash_{\mathsf{vec}} \overline{\psi} : \Delta$.
6. *If* $\Sigma \vdash_{\mathsf{ctx}} \Gamma\ \mathsf{ok}$, *then* $\Sigma \vdash_{\mathsf{ctx}} \Gamma'\ \mathsf{ok}$.
7. *If* $\Sigma; \Gamma \vdash_{\mathsf{s}} \tau \longrightarrow \tau'$, *then* $\Sigma; \Gamma' \vdash_{\mathsf{s}} \tau \longrightarrow \tau'$.

Proof. By straightforward mutual induction, appealing to Lemma C.5. □

C.2.2 Regularity, Part I

Lemma C.7 (Type variable kinds)**.** *If* $\Sigma \vdash_{\mathsf{ctx}} \Gamma\ \mathsf{ok}$ *and* $a{:}_\rho\kappa \in \Gamma$, *then there exists* Γ' *such that* $\Gamma' \subseteq \mathsf{Rel}(\Gamma)$ *and* $\Sigma; \Gamma' \vdash_{\mathsf{ty}} \kappa : \mathbf{Type}$. *Furthermore, the size of the derivation of* $\Sigma; \Gamma' \vdash_{\mathsf{ty}} \kappa : \mathbf{Type}$ *is smaller than that of* $\Sigma \vdash_{\mathsf{ctx}} \Gamma\ \mathsf{ok}$.

Proof. Straightforward induction on $\Sigma \vdash_{\mathsf{ctx}} \Gamma\ \mathsf{ok}$. □

Lemma C.8 (Coercion variable kinds)**.** *If* $\Sigma \vdash_{\mathsf{ctx}} \Gamma\ \mathsf{ok}$ *and* $c{:}\phi \in \Gamma$, *then there exists* Γ' *such that* $\Gamma' \subseteq \mathsf{Rel}(\Gamma)$ *and* $\Sigma; \Gamma' \vdash_{\mathsf{prop}} \phi\ \mathsf{ok}$. *Furthermore, the size of the derivation of* $\Sigma; \Gamma' \vdash_{\mathsf{prop}} \phi\ \mathsf{ok}$ *is smaller than that of* $\Sigma \vdash_{\mathsf{ctx}} \Gamma\ \mathsf{ok}$.

Proof. Straightforward induction on $\Sigma \vdash_{\mathsf{ctx}} \Gamma\ \mathsf{ok}$. □

Lemma C.9 (Context regularity)**.** *If:*

1. $\Sigma; \Gamma \vdash_{\mathsf{ty}} \tau : \kappa$, *OR*
2. $\Sigma; \Gamma \vdash_{\mathsf{co}} \gamma : \phi$, *OR*
3. $\Sigma; \Gamma \vdash_{\mathsf{prop}} \phi\ \mathsf{ok}$, *OR*
4. $\Sigma; \Gamma; \sigma_0 \vdash^{\tau_0}_{\mathsf{alt}} alt : \kappa$, *OR*
5. $\Sigma; \Gamma \vdash_{\mathsf{vec}} \overline{\psi} : \Delta$, *OR*
6. $\Sigma \vdash_{\mathsf{ctx}} \Gamma\ \mathsf{ok}$

Then $\Sigma \vdash_{\mathsf{ctx}} \mathsf{prefix}(\Gamma)\ \mathsf{ok}$ *and* $\vdash_{\mathsf{sig}} \Sigma\ \mathsf{ok}$, *where* $\mathsf{prefix}(\Gamma)$ *is an arbitrary prefix of* Γ. *Furthermore, both resulting derivations are no larger than the input derivations.*

Proof. Straightforward mutual induction. □

C.2.3 Weakening

Lemma C.10 (Weakening)**.** *Assume* $\Sigma \vdash_{\mathsf{ctx}} \Gamma' \ \mathsf{ok}$ *and* $\Gamma \subseteq \Gamma'$.

1. *If* $\Sigma; \Gamma \vdash_{\mathsf{ty}} \tau : \kappa$ *then* $\Sigma; \Gamma' \vdash_{\mathsf{ty}} \tau : \kappa$.
2. *If* $\Sigma; \Gamma \vdash_{\mathsf{co}} \gamma : \phi$, *then* $\Sigma; \Gamma' \vdash_{\mathsf{co}} \gamma : \phi$.
3. *If* $\Sigma; \Gamma \vdash_{\mathsf{prop}} \phi \ \mathsf{ok}$, *then* $\Sigma; \Gamma' \vdash_{\mathsf{prop}} \phi \ \mathsf{ok}$.
4. *If* $\Sigma; \Gamma; \sigma_0 \vdash^{\tau_0}_{\mathsf{alt}} alt : \kappa$, *then* $\Sigma; \Gamma'; \sigma_0 \vdash^{\tau_0}_{\mathsf{alt}} alt : \kappa$.
5. *If* $\Sigma; \Gamma \vdash_{\mathsf{vec}} \overline{\psi} : \Delta$, *then* $\Sigma; \Gamma' \vdash_{\mathsf{vec}} \overline{\psi} : \Delta$.
6. *If* $\Sigma \vdash_{\mathsf{ctx}} \Gamma, \Delta \ \mathsf{ok}$, *then* $\Sigma \vdash_{\mathsf{ctx}} \Gamma', \Delta \ \mathsf{ok}$.
7. *If* $\Sigma; \Gamma \vdash_{\mathsf{s}} \tau \longrightarrow \tau'$, *then* $\Sigma; \Gamma' \vdash_{\mathsf{s}} \tau \longrightarrow \tau'$.

Proof. By straightforward mutual induction, appealing to Lemma C.4, Lemma C.6 (in order to be able to use the induction hypothesis in, e.g., Ty_AppIrrel), and Lemma C.9 (in order to use the induction hypothesis in, e.g., Ty_Pi). □

Lemma C.11 (Strengthening)**.** *Assume* $\Gamma' \subseteq \Gamma$ *and the variables* $\{\mathsf{dom}(\Gamma)\} \setminus \{\mathsf{dom}(\Gamma')\}$ *are never used.*

1. *If* $\Sigma; \Gamma \vdash_{\mathsf{ty}} \tau : \kappa$ *then* $\Sigma; \Gamma' \vdash_{\mathsf{ty}} \tau : \kappa$.
2. *If* $\Sigma; \Gamma \vdash_{\mathsf{co}} \gamma : \phi$, *then* $\Sigma; \Gamma' \vdash_{\mathsf{co}} \gamma : \phi$.
3. *If* $\Sigma; \Gamma \vdash_{\mathsf{prop}} \phi \ \mathsf{ok}$, *then* $\Sigma; \Gamma' \vdash_{\mathsf{prop}} \phi \ \mathsf{ok}$.
4. *If* $\Sigma; \Gamma; \sigma_0 \vdash^{\tau_0}_{\mathsf{alt}} alt : \kappa$, *then* $\Sigma; \Gamma'; \sigma_0 \vdash^{\tau_0}_{\mathsf{alt}} alt : \kappa$.
5. *If* $\Sigma; \Gamma \vdash_{\mathsf{vec}} \overline{\psi} : \Delta$, *then* $\Sigma; \Gamma' \vdash_{\mathsf{vec}} \overline{\psi} : \Delta$.
6. *If* $\Sigma \vdash_{\mathsf{ctx}} \Gamma \ \mathsf{ok}$, *then* $\Sigma \vdash_{\mathsf{ctx}} \Gamma' \ \mathsf{ok}$.
7. *If* $\Sigma; \Gamma \vdash_{\mathsf{s}} \tau \longrightarrow \tau'$, *then* $\Sigma; \Gamma' \vdash_{\mathsf{s}} \tau \longrightarrow \tau'$.

Proof. Similar to previous proof. □

C.2.4 Scoping

Lemma C.12 (Scoping)**.**

1. *If* $\Sigma; \Gamma \vdash_{\mathsf{ty}} \tau : \kappa$, *then* $\mathsf{fv}(\tau) \subseteq \{\mathsf{dom}(\Gamma)\}$ *and* $\mathsf{fv}(\kappa) \subseteq \{\mathsf{dom}(\Gamma)\}$.
2. *If* $\Sigma; \Gamma \vdash_{\mathsf{co}} \gamma : \phi$, *then* $\mathsf{fv}(\gamma) \subseteq \{\mathsf{dom}(\Gamma)\}$ *and* $\mathsf{fv}(\phi) \subseteq \{\mathsf{dom}(\Gamma)\}$.
3. *If* $\Sigma; \Gamma \vdash_{\mathsf{prop}} \phi \ \mathsf{ok}$, *then* $\mathsf{fv}(\phi) \subseteq \{\mathsf{dom}(\Gamma)\}$.
4. *If* $\Sigma; \Gamma; \sigma_0 \vdash^{\tau_0}_{\mathsf{alt}} H \to \tau : \kappa$, *then* $\mathsf{fv}(\tau) \subseteq \{\mathsf{dom}(\Gamma)\}$.
5. *If* $\Sigma; \Gamma \vdash_{\mathsf{vec}} \overline{\psi} : \Delta$, *then* $\mathsf{fv}(\overline{\psi}) \subseteq \{\mathsf{dom}(\Gamma)\}$ *and* $\mathsf{fv}(\Delta) \subseteq \{\mathsf{dom}(\Gamma)\}$.
6. *If* $\Sigma \vdash_{\mathsf{ctx}} \Gamma \ \mathsf{ok}$, *then* $\mathsf{fv}(\Gamma) = \emptyset$.
7. *If* $\vdash_{\mathsf{sig}} \Sigma \ \mathsf{ok}$ *and* $\Sigma \vdash_{\mathsf{tc}} H : \Delta_1; \Delta_2; H'$, *then* $\mathsf{fv}(\Delta_1) = \emptyset$ *and* $\mathsf{fv}(\Delta_2) \subseteq \{\mathsf{dom}(\Delta_1)\}$.

Proof. By straightforward mutual induction, appealing to Lemma C.3, Lemma C.7, Lemma C.8, and Lemma C.9. □

C.3 Unification

We assume the following properties of our unification algorithm.

Property C.13 (Domain of match). *If* $\mathsf{match}_{\mathcal{V}}(\overline{\tau}_1; \overline{\tau}_2) = \mathsf{Just}\,\theta$, *then* $\theta = \overline{\psi}/\overline{z}$ *for some* $\overline{\psi}$ *and* $\overline{z}$ *with* $\mathcal{V} = \{\overline{z}\}$. *In other words, the domain of the substitution returned by a successful use of* match *is the variables* $\mathcal{V}$ *passed into* match.

Property C.14 (match is sound). *If* $\mathsf{match}_{\mathcal{V}}(\overline{\tau}_1; \overline{\tau}_2) = \mathsf{Just}\,\theta$, *then* $\overline{\tau}_1[\theta] = \overline{\tau}_2$.

Property C.15 (match/substitution). *If* $\mathsf{match}_{\mathcal{V}}(\overline{\tau}_1; \overline{\tau}_2) = \mathsf{Just}\,\theta$ *and* $\mathsf{dom}(\theta_0) \cap \mathcal{V} = \emptyset$, $\mathsf{match}_{\mathcal{V}}(\overline{\tau}_1[\theta_0]; \overline{\tau}_2[\theta_0]) = \mathsf{Just}\,\theta'$ *for some* θ'.

C.4 Determinacy

Lemma C.16 (Uniqueness of signatures). *Assume* $\vdash_{\mathsf{sig}} \Sigma\ \mathsf{ok}$.

1. *If* $T{:}(\overline{a}{:}\overline{\kappa}_1) \in \Sigma$ *and* $T{:}(\overline{a}{:}\overline{\kappa}_2) \in \Sigma$, *then* $\overline{\kappa}_1 = \overline{\kappa}_2$.
2. *If* $K{:}(\Delta_1; T_1) \in \Sigma$ *and* $K{:}(\Delta_2; T_2) \in \Sigma$, *then* $\Delta_1 = \Delta_2$ *and* $T_1 = T_2$.

Proof. By the freshness conditions on $\vdash_{\mathsf{sig}} \Sigma\ \mathsf{ok}$. □

Lemma C.17 (Uniqueness of contexts). *Assume* $\Sigma \vdash_{\mathsf{ctx}} \Gamma\ \mathsf{ok}$.

1. *If* $a{:}_{\rho_1}\kappa_1 \in \Gamma$ *and* $a{:}_{\rho_2}\kappa_2 \in \Gamma$, *then* $\rho_1 = \rho_2$ *and* $\kappa_1 = \kappa_2$.
2. *If* $c{:}\phi_1 \in \Gamma$ *and* $c{:}\phi_2 \in \Gamma$, *then* $\phi_1 = \phi_2$.

Proof. By the freshness conditions on $\Sigma \vdash_{\mathsf{ctx}} \Gamma\ \mathsf{ok}$. □

Lemma C.18 (Determinacy of type constants). *If* $\vdash_{\mathsf{sig}} \Sigma\ \mathsf{ok}$, $\Sigma \vdash_{\mathsf{tc}} H : \Delta_1; \Delta_1'; H_1$, *and* $\Sigma \vdash_{\mathsf{tc}} H : \Delta_2; \Delta_2'; H_2$, *then* $\Delta_1 = \Delta_2$, $\Delta_1' = \Delta_2'$, *and* $H_1 = H_2$.

Proof. From Lemma C.16. □

Lemma C.19 (Values do not step). *There exists no* τ *such that* $\Sigma; \Gamma \vdash_{\mathsf{s}} v \longrightarrow \tau$.

Proof. By induction on the structure of v. □

Lemma C.20 (Determinacy).

1. *If* $\Sigma; \Gamma \vdash_{\mathsf{ty}} \tau : \kappa_1$ *and* $\Sigma; \Gamma \vdash_{\mathsf{ty}} \tau : \kappa_2$, *then* $\kappa_1 = \kappa_2$.
2. *If* $\Sigma; \Gamma \vdash_{\mathsf{co}} \gamma : \phi_1$ *and* $\Sigma; \Gamma \vdash_{\mathsf{co}} \gamma : \phi_2$, *then* $\phi_1 = \phi_2$.
3. *If* $\Sigma; \Gamma \vdash_{\mathsf{s}} \tau \longrightarrow \sigma_1$ *and* $\Sigma; \Gamma \vdash_{\mathsf{s}} \tau \longrightarrow \sigma_2$, *then* $\sigma_1 = \sigma_2$.

Proof. By mutual induction, appealing to Lemma C.17, Lemma C.18 (which requires a use of Lemma C.9 first), and Lemma C.19. □

C.5 Vectors

Lemma C.21. *If* $\Sigma; \Gamma \vdash_{\mathsf{ty}} \tau : \kappa$ *and* $\Sigma; \Gamma \vdash_{\mathsf{cev}} \overline{\psi} : \Delta[\tau/a]$*, then* $\Sigma; \Gamma \vdash_{\mathsf{cev}} \tau, \overline{\psi} : a{:}_{\mathsf{Rel}}\kappa, \Delta$.

Proof. By induction on $\Sigma; \Gamma \vdash_{\mathsf{cev}} \overline{\psi} : \Delta[\tau/a]$.

Case Cev_Nil: In this case, $\overline{\psi}$ and Δ are both empty, and so we are done by Cev_Nil and Cev_TyRel.

Case Cev_TyRel: We now have $\overline{\psi} = \overline{\psi}', \sigma$ and $\Delta = \Delta', b{:}_{\mathsf{Rel}}\kappa_0$, with $\Sigma; \Gamma \vdash_{\mathsf{ty}} \sigma : \kappa_0[\tau/a][\overline{\psi}'/\mathsf{dom}(\Delta')]$ and $\Sigma; \Gamma \vdash_{\mathsf{cev}} \overline{\psi}' : \Delta'[\tau/a]$. The induction hypothesis gives us $\Sigma; \Gamma \vdash_{\mathsf{cev}} \tau, \overline{\psi}' : a{:}_{\mathsf{Rel}}\kappa, \Delta'$. We are done by Cev_TyRel.

Other cases: Similar.

□

Lemma C.22. *If* $\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{ty}} \tau : \kappa$ *and* $\Sigma; \Gamma \vdash_{\mathsf{cev}} \overline{\psi} : \Delta[\tau/a]$*, then* $\Sigma; \Gamma \vdash_{\mathsf{cev}} \tau, \overline{\psi} : a{:}_{\mathsf{Irrel}}\kappa, \Delta$.

Proof. Similar to previous proof. □

Lemma C.23. *If* $\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{co}} \gamma : \phi$ *and* $\Sigma; \Gamma \vdash_{\mathsf{cev}} \overline{\psi} : \Delta[\gamma/c]$*, then* $\Sigma; \Gamma \vdash_{\mathsf{cev}} \gamma, \overline{\psi} : c{:}\phi, \Delta$.

Proof. Similar to previous proof. □

Lemma C.24. *If* $\Sigma; \Gamma \vdash_{\mathsf{vec}} \overline{\psi} : \Delta$ *and* $\Sigma; \Gamma \vdash_{\mathsf{ty}} \tau : \kappa[\overline{\psi}/\mathsf{dom}(\Delta)]$*, then* $\Sigma; \Gamma \vdash_{\mathsf{vec}} \overline{\psi}, \tau : \Delta, a{:}_{\mathsf{Rel}}\kappa$.

Proof. By induction on $\Sigma; \Gamma \vdash_{\mathsf{vec}} \overline{\psi} : \Delta$.

Case Vec_Nil: In this case, $\overline{\psi}$ and Δ are both empty, and so we are done by Vec_Nil and Vec_TyRel.

Case Vec_TyRel: We now have $\overline{\psi} = \sigma, \overline{\psi}'$ and $\Delta = b{:}_{\mathsf{Rel}}\kappa_0, \Delta'$ with $\Sigma; \Gamma \vdash_{\mathsf{ty}} \sigma : \kappa_0$ and $\Sigma; \Gamma \vdash_{\mathsf{vec}} \overline{\psi}' : \Delta'[\sigma/b]$. We know, by assumption, that $\Sigma; \Gamma \vdash_{\mathsf{ty}} \tau : \kappa[\overline{\psi}/\mathsf{dom}(\Delta)]$. This expands to $\Sigma; \Gamma \vdash_{\mathsf{ty}} \tau : \kappa[\sigma/b][\overline{\psi}'/\mathsf{dom}(\Delta')]$ (noting that Lemma C.12 assures us that σ has no variables in $\mathsf{dom}(\Delta')$ free). We can thus use the induction hypothesis to get $\Sigma; \Gamma \vdash_{\mathsf{vec}} \overline{\psi}', \tau : \Delta'[\sigma/b], a{:}_{\mathsf{Rel}}\kappa[\sigma/b]$, or, equivalently, $\Sigma; \Gamma \vdash_{\mathsf{vec}} \overline{\psi}', \tau : (\Delta', a{:}_{\mathsf{Rel}}\kappa)[\sigma/b]$. We are done by Vec_TyRel.

Other cases: Similar.

□

Lemma C.25. *If* $\Sigma; \Gamma \vdash_{\mathsf{vec}} \overline{\psi} : \Delta$ *and* $\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{ty}} \tau : \kappa[\overline{\psi}/\mathsf{dom}(\Delta)]$*, then* $\Sigma; \Gamma \vdash_{\mathsf{vec}} \overline{\psi}, \tau : \Delta, a{:}_{\mathsf{Irrel}}\kappa$.

Proof. Similar to previous proof. □

Lemma C.26. *If* $\Sigma; \Gamma \vdash_{\mathsf{vec}} \overline{\psi} : \Delta$ *and* $\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{co}} \gamma : \phi[\overline{\psi}/\mathsf{dom}(\Delta)]$*, then* $\Sigma; \Gamma \vdash_{\mathsf{vec}} \overline{\psi}, \gamma : \Delta, c{:}\phi$*.*

Proof. Similar to previous proof. □

Lemma C.27 (Vec/Cev)**.** *We have* $\Sigma; \Gamma \vdash_{\mathsf{vec}} \overline{\psi} : \Delta$ *if and only if* $\Sigma; \Gamma \vdash_{\mathsf{cev}} \overline{\psi} : \Delta$*.*

Proof. We'll prove the forward direction first, by induction on the typing derivation:

Case Vec_Nil: We are done by Cev_Nil.

Case Vec_TyRel: By the induction hypothesis and Lemma C.21.

Case Vec_TyIrrel: By the induction hypothesis and Lemma C.22.

Case Vec_Co: By the induction hypothesis and Lemma C.23.

The reverse direction is similar, appealing to Lemma C.24, Lemma C.25, and Lemma C.26. □

Lemma C.28 (Vector lengths)**.** *If* $\Sigma; \Gamma \vdash_{\mathsf{vec}} \overline{\psi} : \Delta$*, then* $|\overline{\psi}| = |\Delta|$*.*

Proof. Straightforward induction on $\Sigma; \Gamma \vdash_{\mathsf{vec}} \overline{\psi} : \Delta$. □

Lemma C.29 (Vector kinds)**.** *If* $\Sigma; \Gamma \vdash_{\mathsf{vec}} \overline{\psi} : \Delta$*, then for every* $\psi \in \overline{\psi}$*, we have one of the following:*

1. $\psi = \tau$ *and* $\Sigma; \Gamma \vdash_{\mathsf{ty}} \tau : \kappa$ *for some* κ
2. $\psi = \{\tau\}$ *and* $\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{ty}} \tau : \kappa$ *for some* κ
3. $\psi = \gamma$ *and* $\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{co}} \gamma : \phi$ *for some* ϕ

The resulting derivation is smaller than the input derivation.

Proof. Straightforward induction on $\Sigma; \Gamma \vdash_{\mathsf{vec}} \overline{\psi} : \Delta$. □

Lemma C.30 (Application inversion)**.** *If* $\Sigma; \Gamma \vdash_{\mathsf{ty}} \tau\, \overline{\psi} : \kappa$ *where* $\overline{\psi} = \overline{\psi}_0, \overline{\psi}_1$*, then* $\Sigma; \Gamma \vdash_{\mathsf{ty}} \tau\, \overline{\psi}_0 : \Pi\Delta.\, \kappa_0$*,* $\Sigma; \Gamma \vdash_{\mathsf{vec}} \overline{\psi}_1 : \Delta$ *and* $\kappa = \kappa_0[\overline{\psi}_1/\mathsf{dom}(\Delta)]$*.*

Proof. Straightforward induction on $\overline{\psi}_1$. □

Lemma C.31 (Telescope application)**.** *If* $\Sigma; \Gamma \vdash_{\mathsf{ty}} \tau : \Pi\Delta.\, \sigma$ *and* $\Sigma; \Gamma \vdash_{\mathsf{vec}} \overline{\psi} : \Delta$*, then* $\Sigma; \Gamma \vdash_{\mathsf{ty}} \tau\, \overline{\psi} : \sigma[\overline{\psi}/\mathsf{dom}(\Delta)]$*.*

Proof. By straightforward induction on $\Sigma; \Gamma \vdash_{\mathsf{vec}} \overline{\psi} : \Delta$. □

Lemma C.32 (Telescope instantiation)**.** *If* $\Sigma; \Gamma \vdash_{\mathsf{co}} \eta : \Pi\Delta.\, \sigma \sim \Pi\Delta'.\, \sigma'$*,* $(\forall i, \Sigma; \Gamma \vdash_{\mathsf{co}} \gamma_i : \tau_i \sim \tau_i')$*,* $\Sigma; \Gamma \vdash_{\mathsf{vec}} \overline{\tau} : \Delta$*, and* $\Sigma; \Gamma \vdash_{\mathsf{vec}} \overline{\tau}' : \Delta'$*, then* $\Sigma; \Gamma \vdash_{\mathsf{co}} \eta @ \overline{\gamma} : \sigma[\overline{\tau}/\mathsf{dom}(\Delta)] \sim \sigma'[\overline{\tau}'/\mathsf{dom}(\Delta')]$*.*

Proof. By induction on the structure of the list $\overline{\gamma}$.

Case $\overline{\gamma} = \varnothing$**:** By Lemma C.28, we can see that Δ and Δ' must both be empty. We are done by assumption.

Case $\overline{\gamma} = \gamma_0, \overline{\gamma}_1$**:** In this case, we know $\Sigma; \Gamma \vdash_{\mathsf{vec}} \tau_0, \overline{\tau}_1 : \Delta$ and thus that $\Delta = a{:}_{\mathsf{Rel}}\kappa_0, \Delta_1$ with $\Sigma; \Gamma \vdash_{\mathsf{ty}} \tau_0 : \kappa_0$ and $\Sigma; \Gamma \vdash_{\mathsf{vec}} \overline{\tau}_1 : \Delta_1[\tau_0/a]$. Similarly, we have $\Sigma; \Gamma \vdash_{\mathsf{ty}} \tau_0' : \kappa_0'$ and $\Sigma; \Gamma \vdash_{\mathsf{vec}} \overline{\tau}_1' : \Delta_1'[\tau_0'/a]$. We must show $\Sigma; \Gamma \vdash_{\mathsf{co}} (\eta @ \gamma_0) @ \overline{\gamma}_1 : \sigma[\tau_0/a, \overline{\tau}_1/\mathsf{dom}(\Delta_1)] \sim \sigma'[\tau_0'/a, \overline{\tau}_1'/\mathsf{dom}(\Delta_1')]$. We can rewrite our assumption (expanding Δ and Δ') to be $\Sigma; \Gamma \vdash_{\mathsf{co}} \eta : \Pi a{:}_{\mathsf{Rel}}\kappa_0, \Delta_1. \sigma \sim \Pi a{:}_{\mathsf{Rel}}\kappa_0', \Delta_1'. \sigma'$ and thus derive $\Sigma; \Gamma \vdash_{\mathsf{co}} \eta @ \gamma_0 : \Pi(\Delta_1[\tau_0/a]). (\sigma[\tau_0/a]) \sim \Pi(\Delta_1'[\tau_0'/a]). (\sigma'[\tau_0'/a])$. We can then use the induction hypothesis to get $\Sigma; \Gamma \vdash_{\mathsf{co}} (\eta @ \gamma_0) @ \overline{\gamma} : \sigma[\tau_0/a][\overline{\tau}_1/\mathsf{dom}(\Delta_1)] \sim \sigma'[\tau_0'/a][\overline{\tau}_1'/\mathsf{dom}(\Delta_1')]$, which (noting that τ_0 cannot have any of the $\mathsf{dom}(\Delta_1)$ free) is what we wish to prove.

□

Remark. The above Lemma C.32 could be made more general, to work with $\tilde{\Pi}$ as well as Π. However, doing so would make the statement and proof more cluttered, and it is only ever needed with Π.

C.6 Substitution

Lemma C.33 (Value substitution)**.** *If v is a value with a free variable a, then $v[\sigma/a]$ is also a value.*

Proof. By the definition of values. □

Lemma C.34 (Substitution/erasure)**.** $\lfloor \tau \rfloor [\lfloor \sigma \rfloor / a] = \lfloor \tau[\sigma/a] \rfloor$

Proof. By induction on the structure of τ. □

Lemma C.35 (Type substitution)**.** *Assume $\Sigma; \Gamma \vdash_{\mathsf{ty}} \sigma : \kappa$.*

1. *If $\Sigma; \Gamma, a{:}_\rho\kappa, \Gamma' \vdash_{\mathsf{ty}} \tau : \kappa_0$, then $\Sigma; \Gamma, \Gamma'[\sigma/a] \vdash_{\mathsf{ty}} \tau[\sigma/a] : \kappa_0[\sigma/a]$.*
2. *If $\Sigma; \Gamma, a{:}_\rho\kappa, \Gamma' \vdash_{\mathsf{co}} \gamma : \phi$, then $\Sigma; \Gamma, \Gamma'[\sigma/a] \vdash_{\mathsf{co}} \gamma[\sigma/a] : \phi[\sigma/a]$.*
3. *If $\Sigma; \Gamma, a{:}_\rho\kappa, \Gamma' \vdash_{\mathsf{prop}} \phi\ \mathsf{ok}$, then $\Sigma; \Gamma, \Gamma'[\sigma/a] \vdash_{\mathsf{prop}} \phi[\sigma/a]\ \mathsf{ok}$.*
4. *If $\Sigma; \Gamma, a{:}_\rho\kappa, \Gamma'; \sigma_0 \vdash^{\tau_0}_{\mathsf{alt}} alt : \kappa$, then $\Sigma; \Gamma, \Gamma'[\sigma/a]; \sigma_0[\sigma/a] \vdash^{\tau_0[\sigma/a]}_{\mathsf{alt}} alt[\sigma/a] : \kappa[\sigma/a]$.*
5. *If $\Sigma; \Gamma, a{:}_\rho\kappa, \Gamma' \vdash_{\mathsf{vec}} \overline{\psi} : \Delta$, then $\Sigma; \Gamma, \Gamma'[\sigma/a] \vdash_{\mathsf{vec}} \overline{\psi}[\sigma/a] : \Delta[\sigma/a]$.*
6. *If $\Sigma \vdash_{\mathsf{ctx}} \Gamma, a{:}_\rho\kappa, \Gamma'\ \mathsf{ok}$, then $\Sigma \vdash_{\mathsf{ctx}} \Gamma, \Gamma'[\sigma/a]\ \mathsf{ok}$.*
7. *If $\Sigma; \Gamma, a{:}_\rho\kappa, \Gamma' \vdash_{\mathsf{s}} \tau \longrightarrow \tau'$, then $\Sigma; \Gamma, \Gamma'[\sigma/a] \vdash_{\mathsf{s}} \tau[\sigma/a] \longrightarrow \tau'[\sigma/a]$.*

Proof. By mutual induction. Some interesting cases are below.

Case TY_VAR: Here, we know τ is some variable b. There are three cases to consider:

Case $b{:}_{\mathsf{Rel}}\kappa_0 \in \Gamma$**:** We must derive $\Sigma; \Gamma, \Gamma'[\sigma/a] \vdash_{\mathsf{ty}} b : \kappa_0[\sigma/a]$. We will use TY_VAR. We establish $\Sigma \vdash_{\mathsf{ctx}} \Gamma, \Gamma'[\sigma/a]\ \mathsf{ok}$ by the induction hypothesis. Scoping (Lemma C.12) tells us that $a \notin \mathsf{fv}(\kappa_0)$, and so we are done by the fact that $b{:}_{\mathsf{Rel}}\kappa_0 \in \Gamma$.

Case $b = a$**:** By weakening (Lemma C.10).

Case $b{:}_{\mathsf{Rel}}\kappa_0 \in \Gamma'$**:** Once again, we get $\Sigma \vdash_{\mathsf{ctx}} \Gamma, \Gamma'[\sigma/a]\ \mathsf{ok}$ by the induction hypothesis. Furthermore, we get $b{:}_{\mathsf{Rel}}\kappa_0[\sigma/a] \in \Gamma'[\sigma/a]$ from $b{:}_{\mathsf{Rel}}\kappa_0 \in \Gamma'$.

Case TY_CON: By Lemma C.12, Lemma C.9, and induction.

Case ALT_MATCH: We adopt the metavariable names from the rule:

$$\frac{\begin{array}{l}\Sigma \vdash_{\mathsf{tc}} H : \Delta_1; \Delta_2; H' \qquad \Delta_3, \Delta_4 = \Delta_2[\overline{\sigma}/\mathsf{dom}(\Delta_1)] \\ \mathsf{dom}(\Delta_4) = \mathsf{dom}(\Delta') \\ \mathsf{match}_{\{\mathsf{dom}(\Delta_3)\}}(\mathsf{types}(\Delta_4); \mathsf{types}(\Delta')) = \mathsf{Just}\,\theta \\ \Sigma; \Gamma \vdash_{\mathsf{ty}} \tau : \Pi\Delta_3, c{:}\tau_0 \sim H_{\{\overline{\sigma}\}}\,\mathsf{dom}(\Delta_3).\,\kappa\end{array}}{\Sigma; \Gamma; \Pi\Delta'.\, H'\,\overline{\sigma} \vdash^{\tau_0}_{\mathsf{alt}} H \to \tau : \kappa}\ \textsc{Alt\_Match}$$

We will use ALT_MATCH to prove our desired conclusion. Several premises are unchanged. The remaining ones we will have to prove:

$\Delta'_3, \Delta'_4 = \Delta_2[\overline{\sigma}[\sigma/a]/\mathsf{dom}(\Delta_1)]$**:** By our choice of $\Delta'_3 = \Delta_3[\sigma/a]$ and $\Delta'_4 = \Delta_4[\sigma/a]$.

$\mathsf{match}_{\{\mathsf{dom}(\Delta_3)\}}(\mathsf{types}(\Delta_4[\sigma/a]); \mathsf{types}(\Delta'[\sigma/a])) = \mathsf{Just}\,\theta'$**:** We can freely choose θ', but we still need to make sure that the match succeeds. This is by Property C.15.

Case CO_VAR: Similar to TY_VAR.

Case CO_PITY: We adopt the metavariable names from the rule (renaming the variable to be substituted to b):

$$\frac{\begin{array}{c}\Sigma; \Gamma \vdash_{\mathsf{co}} \eta : \kappa_1\ {}^{\mathbf{Type}}{\sim}^{\mathbf{Type}}\ \kappa_2 \\ \Sigma; \Gamma, a{:}_{\mathsf{Rel}}\kappa_1 \vdash_{\mathsf{co}} \gamma : \sigma_1\ {}^{\mathbf{Type}}{\sim}^{\mathbf{Type}}\ \sigma_2\end{array}}{\Sigma; \Gamma \vdash_{\mathsf{co}} \Pi a{:}_\rho \eta.\, \gamma : (\Pi a{:}_\rho \kappa_1.\, \sigma_1) \sim (\Pi a{:}_\rho \kappa_2.\, (\sigma_2[a \rhd \mathbf{sym}\,\eta/a]))}\ \textsc{Co\_PiTy}$$

The induction hypothesis gives us:

- $\Sigma; \Gamma, \Gamma'[\sigma/b] \vdash_{\mathsf{co}} \eta[\sigma/b] : \kappa_1[\sigma/b] \sim \kappa_2[\sigma/b]$
- $\Sigma; \Gamma, \Gamma'[\sigma/b], a{:}_{\mathsf{Rel}}\kappa_1[\sigma/b] \vdash_{\mathsf{co}} \gamma[\sigma/b] : \sigma_1[\sigma/b] \sim \sigma_2[\sigma/b]$

By Co_PiTy, we get

$$\Sigma; \Gamma, \Gamma'[\sigma/b] \vdash_{\mathsf{co}} \Pi a{:}_\rho \eta[\sigma/b].\, \gamma[\sigma/b] : \\ (\Pi a{:}_\rho \kappa_1[\sigma/b].\, \sigma_1[\sigma/b]) \sim (\Pi a{:}_\rho \kappa_2[\sigma/b].\, (\sigma_2[\sigma/b][a \rhd \mathbf{sym}\, \eta[\sigma/b]/a]))$$

All that remains to show is that $\sigma_2[\sigma/b][a \rhd \mathbf{sym}\, \eta[\sigma/b]/a] = \sigma_2[a \rhd \mathbf{sym}\, \eta/a][\sigma/b]$, but this follows from the fact that $a \# \sigma$, guaranteed by the Barendregt convention. We are done with this case.

Case Co_PiCo: We adopt the metavariable names from the rule:

$$\frac{\begin{array}{l}\Sigma; \Gamma \vdash_{\mathsf{co}} \eta_1 : \tau_1 \sim \tau_2 \qquad \Sigma; \Gamma \vdash_{\mathsf{co}} \eta_2 : \sigma_1 \sim \sigma_2 \\ \Sigma; \Gamma, c{:}\tau_1 \sim \sigma_1 \vdash_{\mathsf{co}} \gamma : \kappa_1 \; {}^{\mathbf{Type}}\!\sim^{\mathbf{Type}} \kappa_2 \qquad c \mathrel{\tilde{\#}} \gamma \\ \eta_3 = \eta_1 \fatsemi c \fatsemi \mathbf{sym}\, \eta_2\end{array}}{\Sigma; \Gamma \vdash_{\mathsf{co}} \Pi c{:}(\eta_1, \eta_2).\, \gamma : (\Pi c{:}\tau_1 \sim \sigma_1.\, \kappa_1) \sim (\Pi c{:}\tau_2 \sim \sigma_2.\, (\kappa_2[\eta_3/c]))} \; \text{Co\_PiCo}$$

For the most part, this follows the pattern of case Co_PiTy, but we must make sure that $c \mathrel{\tilde{\#}} \gamma[\sigma/a]$. This fact follows from the Barendregt convention, which asserts that c cannot appear in σ.

Other cases: By the induction hypothesis, using Lemma C.33 for certain step rules, and using the Barendregt convention to rearrange substitutions (as in the Co_PiTy case).

□

Lemma C.36 (Coercion substitution)**.** *Assume* $\Sigma; \Gamma \vdash_{\mathsf{co}} \gamma : \phi$.

1. *If* $\Sigma; \Gamma, c{:}\phi, \Gamma' \vdash_{\mathsf{ty}} \tau : \kappa_0$*, then* $\Sigma; \Gamma, \Gamma'[\gamma/c] \vdash_{\mathsf{ty}} \tau[\gamma/c] : \kappa_0[\gamma/c]$.
2. *If* $\Sigma; \Gamma, c{:}\phi, \Gamma' \vdash_{\mathsf{co}} \eta : \phi'$*, then* $\Sigma; \Gamma, \Gamma'[\gamma/c] \vdash_{\mathsf{co}} \eta[\gamma/c] : \phi'[\gamma/c]$.
3. *If* $\Sigma; \Gamma, c{:}\phi, \Gamma' \vdash_{\mathsf{prop}} \phi'\ \mathsf{ok}$*, then* $\Sigma; \Gamma, \Gamma'[\gamma/c] \vdash_{\mathsf{prop}} \phi'[\gamma/c]\ \mathsf{ok}$.
4. *If* $\Sigma; \Gamma, c{:}\phi, \Gamma'; \sigma_0 \vdash^{\tau_0}_{\mathsf{alt}} alt : \kappa$*, then* $\Sigma; \Gamma, \Gamma'[\gamma/c]; \sigma_0[\gamma/c] \vdash^{\tau_0[\gamma/c]}_{\mathsf{alt}} alt[\gamma/c] : \kappa[\gamma/c]$.
5. *If* $\Sigma; \Gamma, c{:}\phi, \Gamma' \vdash_{\mathsf{vec}} \overline{\psi} : \Delta$*, then* $\Sigma; \Gamma, \Gamma'[\gamma/c] \vdash_{\mathsf{vec}} \overline{\psi}[\gamma/c] : \Delta[\gamma/c]$.
6. *If* $\Sigma \vdash_{\mathsf{ctx}} \Gamma, c{:}\phi, \Gamma'\ \mathsf{ok}$*, then* $\Sigma \vdash_{\mathsf{ctx}} \Gamma, \Gamma'[\gamma/c]\ \mathsf{ok}$.
7. *If* $\Sigma; \Gamma, c{:}\phi, \Gamma' \vdash_{\mathsf{s}} \tau \longrightarrow \tau'$*, then* $\Sigma; \Gamma, \Gamma'[\gamma/c] \vdash_{\mathsf{s}} \tau[\gamma/c] \longrightarrow \tau'[\gamma/c]$.

Proof. Similar to proof for Lemma C.35. □

Lemma C.37 (Vector substitution)**.** *If* $\Sigma; \Gamma \vdash_{\mathsf{vec}} \overline{\psi} : \Delta$ *and* $\Sigma; \Gamma, \Delta, \Gamma' \vdash_{\mathsf{ty}} \tau : \kappa$*, then* $\Sigma; \Gamma, \Gamma'[\overline{\psi}/\mathsf{dom}(\Delta)] \vdash_{\mathsf{ty}} \tau[\overline{\psi}/\mathsf{dom}(\Delta)] : \kappa[\overline{\psi}/\mathsf{dom}(\Delta)]$.

Proof. By induction on the structure of Δ.

Case $\Delta = \varnothing$: By assumption.

Case $\Delta = a_0{:}_{\mathsf{Rel}}\kappa_0, \Delta'$**:** We know $\overline{\psi} = \sigma_0, \overline{\psi}'$, $\Sigma; \Gamma \vdash_{\mathsf{ty}} \sigma_0 : \kappa_0$, and $\Sigma; \Gamma \vdash_{\mathsf{vec}} \overline{\psi}' : \Delta'[\sigma_0/a]$. Lemma C.35 tells us $\Sigma; \Gamma, \Delta'[\sigma_0/a], \Gamma'[\sigma_0/a] \vdash_{\mathsf{ty}} \tau[\sigma_0/a] : \kappa[\sigma_0/a]$. We are done by a use of the induction hypothesis.

Other cases: Similar.

□

C.7 Type constants

Lemma C.38 (Type-in-type)**.** *If* $\vdash_{\mathsf{sig}} \Sigma\ \mathsf{ok}$*, then* $\Sigma; \varnothing \vdash_{\mathsf{ty}} \mathbf{Type} : \mathbf{Type}$*.*

Proof. Working backward, use Ty_Con so that we must show the following:

$\Sigma \vdash_{\mathsf{tc}} \mathbf{Type} : \varnothing; \varnothing; \mathbf{Type}$**:** By Tc_Type.

$\Sigma \vdash_{\mathsf{ctx}} \varnothing\ \mathsf{ok}$**:** By Ctx_Nil.

$\Sigma; \varnothing \vdash_{\mathsf{vec}} \varnothing : \varnothing$**:** By Vec_Nil.

We are thus done. □

Lemma C.39 (Telescopes)**.** *If* $\Sigma \vdash_{\mathsf{ctx}} \Gamma, \Delta\ \mathsf{ok}$*, then* $\Sigma; \Gamma, \Delta \vdash_{\mathsf{vec}} \mathsf{dom}(\Delta) : \Delta$*.*

Proof. Proceed by induction on the structure of Δ.

Case $\Delta = \varnothing$**:** By Vec_Nil.

Case $\Delta = a{:}_{\mathsf{Rel}}\kappa, \Delta'$**:** We must show $\Sigma; \Gamma, a{:}_{\mathsf{Rel}}\kappa, \Delta' \vdash_{\mathsf{vec}} a, \mathsf{dom}(\Delta') : a{:}_{\mathsf{Rel}}\kappa, \Delta'$. By Vec_TyRel, we must show $\Sigma; \Gamma, a{:}_{\mathsf{Rel}}\kappa, \Delta' \vdash_{\mathsf{ty}} a : \kappa$ and $\Sigma; \Gamma, a{:}_{\mathsf{Rel}}\kappa, \Delta' \vdash_{\mathsf{vec}} \mathsf{dom}(\Delta') : \Delta'$. The first is by Ty_Var and the second is by the induction hypothesis.

Other cases: Similar.

□

Lemma C.40 (Type constant telescopes)**.** *If* $\vdash_{\mathsf{sig}} \Sigma\ \mathsf{ok}$ *and* $\Sigma \vdash_{\mathsf{tc}} H : \Delta_1; \Delta_2; H'$*, then* $\Sigma \vdash_{\mathsf{ctx}} \Delta_1, \Delta_2\ \mathsf{ok}$*.*

Proof. By case analysis on $\Sigma \vdash_{\mathsf{tc}} H : \Delta_1; \Delta_2; H'$.

Case Tc_ADT: Here $\Delta_1 = \varnothing$ and $\Delta_2 = \overline{a}{:}_{\mathsf{Rel}}\overline{\kappa}$ We see that $\Sigma \vdash_{\mathsf{ctx}} \overline{a}{:}_{\mathsf{Irrel}}\overline{\kappa}\ \mathsf{ok}$ from $\vdash_{\mathsf{sig}} \Sigma\ \mathsf{ok}$ (Sig_ADT). A use of Lemma C.6 solves our goal.

Case Tc_DataCon: Here $\Delta_1 = \overline{a}{:}_{\mathsf{Irrel}}\overline{\kappa}$. We must show $\Sigma \vdash_{\mathsf{ctx}} \overline{a}{:}_{\mathsf{Irrel}}\overline{\kappa}, \Delta_2\ \mathsf{ok}$. From $\vdash_{\mathsf{sig}} \Sigma\ \mathsf{ok}$, we see that $\Sigma \vdash_{\mathsf{ctx}} \overline{a}{:}_{\mathsf{Irrel}}\overline{\kappa}, \Delta_2\ \mathsf{ok}$ (Sig_DataCon).

Case Tc_Type: Here $\Delta_1 = \Delta_2 = \varnothing$. We are done by Ctx_Nil.

□

Lemma C.41 (Type constant kinds)**.** *If* $\vdash_{\mathsf{sig}} \Sigma\ \mathsf{ok}$ *and* $\Sigma \vdash_{\mathsf{tc}} H : \Delta_1; \Delta_2; H'$*, then* $\Sigma; \varnothing \vdash_{\mathsf{ty}} \text{'}\Pi\Delta_1, \Delta_2.\, H'\, \mathsf{dom}(\Delta_1) : \mathbf{Type}$.

Proof. To prove $\Sigma; \varnothing \vdash_{\mathsf{ty}} \text{'}\Pi\Delta_1, \Delta_2.\, H'\, \mathsf{dom}(\Delta_1) : \mathbf{Type}$, we will use Ty_Pi (repeatedly). We thus must show $\Sigma; \mathsf{Rel}(\Delta_1, \Delta_2) \vdash_{\mathsf{ty}} H'\, \mathsf{dom}(\Delta_1) : \mathbf{Type}$. This, in turn, will be by Ty_AppRel (repeatedly). We thus must show

$\Sigma; \mathsf{Rel}(\Delta_1, \Delta_2) \vdash_{\mathsf{ty}} H' : \text{'}\Pi\mathsf{Rel}(\Delta_1).\, \mathbf{Type}$ (We are being a bit more specific here than necessary.) Case analysis of $\Sigma \vdash_{\mathsf{tc}} H : \Delta_1; \Delta_2; H'$ gives us several cases:

- **Case Tc_ADT:** Here, $\Delta_1 = \varnothing$ and $H' = \mathbf{Type}$, and we must show $\Sigma; \mathsf{Rel}(\Delta_2) \vdash_{\mathsf{ty}} \mathbf{Type} : \mathbf{Type}$. According to Ty_Con we must show only that $\Sigma \vdash_{\mathsf{ctx}} \mathsf{Rel}(\Delta_2)\ \mathsf{ok}$, which follows from Lemma C.40 and Lemma C.6.
- **Case Tc_DataCon:** Here, $\Delta_1 = \overline{a}{:}_{\mathsf{Irrel}}\overline{\kappa}$ and $H' = T$. We must show $\Sigma; \overline{a}{:}_{\mathsf{Rel}}\overline{\kappa}, \mathsf{Rel}(\Delta_2) \vdash_{\mathsf{ty}} T : \text{'}\Pi\overline{a}{:}_{\mathsf{Rel}}\overline{\kappa}.\, \mathbf{Type}$. Using Ty_Con means we must show $\Sigma \vdash_{\mathsf{tc}} T : \varnothing; \overline{a}{:}_{\mathsf{Rel}}\overline{\kappa}; \mathbf{Type}$ and $\Sigma \vdash_{\mathsf{ctx}} \overline{a}{:}_{\mathsf{Rel}}\overline{\kappa}, \mathsf{Rel}(\Delta_2)\ \mathsf{ok}$. The latter comes from $\vdash_{\mathsf{sig}} \Sigma\ \mathsf{ok}$ and Lemma C.40. The former comes directly from Tc_ADT.
- **Case Tc_Type:** By Lemma C.38.

$\Sigma; \mathsf{Rel}(\Delta_1, \Delta_2) \vdash_{\mathsf{vec}} \mathsf{dom}(\Delta_1) : \mathsf{Rel}(\Delta_1)$ This last judgment expands out to be all the typing judgments we need in Ty_AppRel. See Vec_TyRel. To prove this, we use Lemma C.39, meaning that we need only show $\Sigma \vdash_{\mathsf{ctx}} \mathsf{Rel}(\Delta_1, \Delta_2)\ \mathsf{ok}$, which we get from Lemma C.40. We are done.

□

Lemma C.42 (Type constant inversion)**.** *If* $\Sigma; \Gamma \vdash_{\mathsf{ty}} H_{\{\overline{\tau}\}}\, \overline{\psi} : \kappa$*, then:*

1. $\Sigma \vdash_{\mathsf{tc}} H : \overline{a}{:}_{\mathsf{Irrel}}\overline{\kappa}; \Delta; H'$
2. $\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{vec}} \overline{\tau} : \overline{a}{:}_{\mathsf{Rel}}\overline{\kappa}$
3. $\Delta_1, \Delta_2 = \Delta[\overline{\tau}/\overline{a}]$
4. $\Sigma; \Gamma \vdash_{\mathsf{vec}} \overline{\psi} : \Delta_1$
5. $\kappa = \text{'}\Pi(\Delta_2[\overline{\psi}/\mathsf{dom}(\Delta_1)]).\, H'\, \overline{\tau}$

Proof. By Lemma C.30, Lemma C.31, and Lemma C.20, and inversion and application of typing rules. □

C.8 Regularity, Part II

Lemma C.43 (Kind regularity)**.** *If* $\Sigma; \Gamma \vdash_{\mathsf{ty}} \tau : \kappa$*, then* $\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{ty}} \kappa : \mathbf{Type}$.

Proof. By induction on the typing derivation.

Case Ty_Var: By Lemma C.7 (and Lemma C.10).

Case Ty_Con: We'll adopt the metavariable names from the rule:

$$\frac{\Sigma \vdash_{\mathsf{tc}} H : \Delta_1; \Delta_2; H' \qquad \Sigma \vdash_{\mathsf{ctx}} \Gamma\ \mathsf{ok} \qquad \Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{vec}} \overline{\tau} : \mathsf{Rel}(\Delta_1)}{\Sigma; \Gamma \vdash_{\mathsf{ty}} H_{\{\overline{\tau}\}} : \Pi(\Delta_2[\overline{\tau}/\mathsf{dom}(\Delta_1)]).\ H'\,\overline{\tau}} \quad \text{Ty\_Con}$$

Use Lemma C.9 to get $\vdash_{\mathsf{sig}} \Sigma\ \mathsf{ok}$. Then use Lemma C.41 to get $\Sigma; \varnothing \vdash_{\mathsf{ty}} \Pi\Delta_1, \Delta_2.\ H'\,\mathsf{dom}(\Delta_1) : \mathbf{Type}$. Repeated inversion on Ty_Pi gives us $\Sigma; \mathsf{Rel}(\Delta_1) \vdash_{\mathsf{ty}} \Pi\Delta_2.\ H'\,\mathsf{dom}(\Delta_1) : \mathbf{Type}$. Lemma C.10 gives us $\Sigma; \mathsf{Rel}(\Gamma), \mathsf{Rel}(\Delta_1) \vdash_{\mathsf{ty}} \Pi\Delta_2.\ H'\,\mathsf{dom}(\Delta_1) : \mathbf{Type}$. Lemma C.37 gives us $\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{ty}} \Pi(\Delta_2[\overline{\tau}/\mathsf{dom}(\Delta_1)]).\ H'\,\overline{\tau} : \mathbf{Type}$ as desired.

Case Ty_AppRel: We'll adopt the metavariable names from the rule:

$$\frac{\Sigma; \Gamma \vdash_{\mathsf{ty}} \tau_1 : \Pi a{:}_{\mathsf{Rel}}\kappa_1.\ \kappa_2 \qquad \Sigma; \Gamma \vdash_{\mathsf{ty}} \tau_2 : \kappa_1}{\Sigma; \Gamma \vdash_{\mathsf{ty}} \tau_1\,\tau_2 : \kappa_2[\tau_2/a]} \quad \text{Ty\_AppRel}$$

The induction hypothesis gives us $\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{ty}} \Pi a{:}_{\mathsf{Rel}}\kappa_1.\ \kappa_2 : \mathbf{Type}$. Inversion on Ty_Pi gives us $\Sigma; \mathsf{Rel}(\Gamma), a{:}_{\mathsf{Rel}}\kappa_1 \vdash_{\mathsf{ty}} \kappa_2 : \mathbf{Type}$. Lemma C.6 gives us $\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{ty}} \tau_2 : \kappa_1$, and then Lemma C.35 applies, giving us $\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{ty}} \kappa_2[\tau_2/a] : \mathbf{Type}$ as desired.

Case Ty_AppIrrel: Similar to last case, noting that inverting Ty_Pi converts the Irrel to a Rel and without the need for Lemma C.6.

Case Ty_CApp: Similar to previous case.

Case Ty_Pi: By Lemma C.9 and Lemma C.38.

Case Ty_Cast: By inversion.

Case Ty_Case: By inversion.

Case Ty_Lam: We'll adopt the metavariable names from the rule:

$$\frac{\Sigma; \Gamma, \delta \vdash_{\mathsf{ty}} \tau : \kappa}{\Sigma; \Gamma \vdash_{\mathsf{ty}} \lambda\delta.\ \tau : \tilde{\Pi}\delta.\ \kappa} \quad \text{Ty\_Lam}$$

We must show $\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{ty}} \tilde{\Pi}\delta.\ \kappa : \mathbf{Type}$. Working backward, use Ty_Pi so that we must show $\Sigma; \mathsf{Rel}(\Gamma, \delta) \vdash_{\mathsf{ty}} \kappa : \mathbf{Type}$, which is true by induction.

Case Ty_Fix: We'll adopt the metavariable names from the rule:

$$\frac{\Sigma; \Gamma \vdash_{\mathsf{ty}} \tau : \tilde{\Pi} a{:}_{\mathsf{Rel}}\kappa.\ \kappa}{\Sigma; \Gamma \vdash_{\mathsf{ty}} \mathbf{fix}\,\tau : \kappa} \quad \text{Ty\_Fix}$$

The induction hypothesis tells us $\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{ty}} \tilde{\Pi} a{:}_{\mathsf{Rel}}\kappa.\ \kappa : \mathbf{Type}$. Inversion on Ty_Pi tells us $\Sigma; \mathsf{Rel}(\Gamma), a{:}_{\mathsf{Rel}}\kappa \vdash_{\mathsf{ty}} \kappa : \mathbf{Type}$. Lemma C.7 gives us $\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{ty}}$

κ : **Type** as desired.

Case Ty_Absurd: Immediate.

□

Lemma C.44 (Proposition regularity)**.** *If* $\Sigma; \Gamma \vdash_{\mathsf{co}} \gamma : \phi$*, then* $\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{prop}} \phi$ ok.

Proof. By induction on the typing derivation.

Case Co_Var: By Lemma C.8, Lemma C.9, and Lemma C.10.

Case Co_Refl: Immediate.

Case Co_Sym: By induction.

Case Co_Trans: By induction.

Case Co_Coherence: Immediate.

Case Co_Con: Immediate.

Case Co_AppRel: Immediate.

Case Co_AppIrrel: Immediate.

Case Co_CApp: Immediate.

Case Co_PiTy: We adopt the metavariable names from the statement of the rule:

$$\frac{\begin{array}{l}\Sigma; \Gamma \vdash_{\mathsf{co}} \eta : \kappa_1 \ {}^{\mathbf{Type}}\!\sim^{\mathbf{Type}} \kappa_2 \\ \Sigma; \Gamma, a{:}_{\mathsf{Rel}}\kappa_1 \vdash_{\mathsf{co}} \gamma : \sigma_1 \ {}^{\mathbf{Type}}\!\sim^{\mathbf{Type}} \sigma_2\end{array}}{\Sigma; \Gamma \vdash_{\mathsf{co}} \Pi a{:}_\rho \eta.\, \gamma : (\Pi a{:}_\rho \kappa_1.\, \sigma_1) \ {}^{\mathbf{Type}}\!\sim^{\mathbf{Type}} (\Pi a{:}_\rho \kappa_2.\, (\sigma_2[a \rhd \mathbf{sym}\, \eta / a]))} \quad \textsc{Co\_PiTy}$$

The induction hypothesis (and inversion) give us the following:

- $\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{ty}} \kappa_1 : \mathbf{Type}$
- $\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{ty}} \kappa_2 : \mathbf{Type}$
- $\Sigma; \mathsf{Rel}(\Gamma), a{:}_{\mathsf{Rel}}\kappa_1 \vdash_{\mathsf{ty}} \sigma_1 : \mathbf{Type}$
- $\Sigma; \mathsf{Rel}(\Gamma), a{:}_{\mathsf{Rel}}\kappa_1 \vdash_{\mathsf{ty}} \sigma_2 : \mathbf{Type}$

We can straightforwardly use Ty_Pi to show that $\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{ty}} \Pi a{:}_\rho \kappa_1.\, \sigma_1$: **Type**. Choose a fresh b. We know $\Sigma \vdash_{\mathsf{ctx}} \mathsf{Rel}(\Gamma), a{:}_{\mathsf{Rel}}\kappa_1$ ok by Lemma C.9. We can then use Ctx_TyVar (with Lemma C.6) to show that $\Sigma \vdash_{\mathsf{ctx}} \mathsf{Rel}(\Gamma), b{:}_{\mathsf{Rel}}\kappa_2, a{:}_{\mathsf{Rel}}\kappa_1$ ok (along with a little inversion and rebuilding to re-order the variables). We established above that $\Sigma; \mathsf{Rel}(\Gamma), a{:}_{\mathsf{Rel}}\kappa_1 \vdash_{\mathsf{ty}} \sigma_2$: **Type**. Use weakening, Lemma C.10, (here and elsewhere in this case) to get $\Sigma; \mathsf{Rel}(\Gamma), b{:}_{\mathsf{Rel}}\kappa_2, a{:}_{\mathsf{Rel}}\kappa_1 \vdash_{\mathsf{ty}} \sigma_2$: **Type**. We can use Co_Sym to see that $\Sigma; \mathsf{Rel}(\Gamma), b{:}_{\mathsf{Rel}}\kappa_2 \vdash_{\mathsf{co}} \mathbf{sym}\, \eta : \kappa_2 \sim \kappa_1$ and then Ty_Cast to see that $\Sigma; \mathsf{Rel}(\Gamma), b{:}_{\mathsf{Rel}}\kappa_2 \vdash_{\mathsf{ty}} b \rhd \mathbf{sym}\, \eta : \kappa_1$. Lemma C.35 then gives us $\Sigma; \mathsf{Rel}(\Gamma), b{:}_{\mathsf{Rel}}\kappa_2 \vdash_{\mathsf{ty}} \sigma_2[b \rhd \mathbf{sym}\, \eta / a]$: **Type**. Use Ty_Pi to get $\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{ty}} \Pi b{:}_\rho \kappa_2.\, (\sigma_2[b \rhd \mathbf{sym}\, \eta / a])$: **Type** and α-equivalence to get $\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{ty}} \Pi a{:}_\rho \kappa_2.\, (\sigma_2[a \rhd \mathbf{sym}\, \eta / a])$: **Type**. We are done by Prop_Equality.

Case Co_PiCo: We adopt the metavariable names from the statement of the rule:

$$\frac{\begin{array}{l}\Sigma;\Gamma \vdash_{\mathsf{co}} \eta_1 : \tau_1 \ {}^{\kappa_3}\!\sim^{\kappa_4}\ \tau_2 \qquad \Sigma;\Gamma \vdash_{\mathsf{co}} \eta_2 : \sigma_1 \ {}^{\kappa_5}\!\sim^{\kappa_6}\ \sigma_2 \\ \Sigma;\Gamma, c{:}\tau_1 \ {}^{\kappa_3}\!\sim^{\kappa_5}\ \sigma_1 \vdash_{\mathsf{co}} \gamma : \kappa_1 \ {}^{\mathbf{Type}}\!\sim^{\mathbf{Type}}\ \kappa_2 \qquad c \mathbin{\tilde{\#}} \gamma \\ \eta_3 \ = \ \eta_1 \fatsemi c \fatsemi \mathbf{sym}\, \eta_2\end{array}}{\Sigma;\Gamma \vdash_{\mathsf{co}} \Pi c{:}(\eta_1,\eta_2).\, \gamma : (\Pi c{:}\tau_1 \ {}^{\kappa_3}\!\sim^{\kappa_5}\ \sigma_1.\, \kappa_1) \ {}^{\mathbf{Type}}\!\sim^{\mathbf{Type}}\ (\Pi c{:}\tau_2 \ {}^{\kappa_4}\!\sim^{\kappa_6}\ \sigma_2.\, (\kappa_2[\eta_3/c]))}\ \textsc{Co\_PiCo}$$

The induction hypothesis (and inversion) give us the following:

- $\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{ty}} \tau_1 : \kappa_3$
- $\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{ty}} \tau_2 : \kappa_4$
- $\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{ty}} \sigma_1 : \kappa_5$
- $\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{ty}} \sigma_2 : \kappa_6$
- $\Sigma; \mathsf{Rel}(\Gamma), c{:}\tau_1 \sim \sigma_1 \vdash_{\mathsf{ty}} \kappa_1 : \mathbf{Type}$
- $\Sigma; \mathsf{Rel}(\Gamma), c{:}\tau_1 \sim \sigma_1 \vdash_{\mathsf{ty}} \kappa_2 : \mathbf{Type}$

We can straightforwardly use Ty_Pi to show that $\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{ty}} \Pi c{:}\tau_1 \sim \sigma_1.\, \kappa_1 : \mathbf{Type}$. Choose a fresh b. We know $\Sigma \vdash_{\mathsf{ctx}} \mathsf{Rel}(\Gamma), c{:}\tau_1 \sim \sigma_1\ \mathsf{ok}$ by Lemma C.9. We can then use Ctx_CoVar (with Lemma C.6) to show that $\Sigma \vdash_{\mathsf{ctx}} \mathsf{Rel}(\Gamma), c_2{:}\tau_2 \sim \sigma_2, c{:}\tau_1 \sim \sigma_1\ \mathsf{ok}$ (along with a little inversion and rebuilding to reorder the variables). We also know $\Sigma; \mathsf{Rel}(\Gamma), c{:}\tau_1 \sim \sigma_1 \vdash_{\mathsf{ty}} \kappa_2 : \mathbf{Type}$. Use weakening, Lemma C.10, (here and elsewhere in this case) to get $\Sigma; \mathsf{Rel}(\Gamma), c_2{:}\tau_2 \sim \sigma_2, c{:}\tau_1 \sim \sigma_1 \vdash_{\mathsf{ty}} \kappa_2 : \mathbf{Type}$. We can use typing rules straightforwardly to see that $\Sigma; \mathsf{Rel}(\Gamma), c_2{:}\tau_2 \sim \sigma_2 \vdash_{\mathsf{co}} \eta_1 \fatsemi c_2 \fatsemi \mathbf{sym}\, \eta_2 : \tau_1 \sim \sigma_1$. Lemma C.36 then gives us $\Sigma; \mathsf{Rel}(\Gamma), c_2{:}\tau_2 \sim \sigma_2 \vdash_{\mathsf{ty}} \kappa_2[\eta_1 \fatsemi c_2 \fatsemi \mathbf{sym}\, \eta_2/c] : \mathbf{Type}$. Use Ty_Pi to get $\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{ty}} \Pi c_2{:}\tau_2 \sim \sigma_2.\, (\kappa_2[\eta_1 \fatsemi c_2 \fatsemi \mathbf{sym}\, \eta_2/c]) : \mathbf{Type}$ and α-equivalence to get $\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{ty}} \Pi c{:}\tau_2 \sim \sigma_2.\, (\kappa_2[\eta_1 \fatsemi c \fatsemi \mathbf{sym}\, \eta_2/c]) : \mathbf{Type}$. We are done.

Case Co_Case: Immediate.

Case Co_Lam: We adopt the metavariable names from the statement of the rule:

$$\frac{\begin{array}{l}\Sigma;\Gamma \vdash_{\mathsf{co}} \eta : \kappa_1 \ {}^{\mathbf{Type}}\!\sim^{\mathbf{Type}}\ \kappa_2 \\ \Sigma;\Gamma, a{:}_\rho\kappa_1 \vdash_{\mathsf{co}} \gamma : \tau_1 \ {}^{\sigma_1}\!\sim^{\sigma_2}\ \tau_2 \\ \Sigma;\Gamma, a{:}_\rho\kappa_1 \vdash_{\mathsf{ty}} \tau_1 : \sigma_1 \qquad \Sigma;\Gamma, a{:}_\rho\kappa_1 \vdash_{\mathsf{ty}} \tau_2 : \sigma_2\end{array}}{\Sigma;\Gamma \vdash_{\mathsf{co}} \lambda a{:}_\rho\eta.\, \gamma : \lambda a{:}_\rho\kappa_1.\, \tau_1 \ {}^{\tilde{\Pi} a:_\rho\kappa_1.\, \sigma_1}\!\sim^{\tilde{\Pi} a:_\rho\kappa_2.\, (\sigma_2[a \rhd \mathbf{sym}\, \eta/a])}\ \lambda a{:}_\rho\kappa_2.\, (\tau_2[a \rhd \mathbf{sym}\, \eta/a])}\ \textsc{Co\_Lam}$$

We can use Ty_Lam to get $\Sigma;\Gamma \vdash_{\mathsf{ty}} \lambda a{:}_\rho\kappa_1.\, \tau_1 : \tilde{\Pi} a{:}_\rho\kappa_1.\, \sigma_1$. Proceeding similarly to the case for Co_PiTy, we can get $\Sigma;\Gamma \vdash_{\mathsf{ty}} \lambda a{:}_\rho\kappa_2.\, (\tau_2[a \rhd \mathbf{sym}\, \eta/a]) : \tilde{\Pi} a{:}_\rho\kappa_2.\, (\sigma_2[a \rhd \mathbf{sym}\, \eta/a])$ and we are done by Lemma C.6.

Case Co_CLam: Similar to previous case and the case for Co_PiCo.

Case Co_Fix: Immediate.

Case Co_Absurd: By induction and Ty_Absurd.

Case Co_ArgK: By induction, inversion, Lemma C.9, and Lemma C.7.

Case Co_CArgK1: By induction, inversion, Lemma C.9, and Lemma C.8.

Case Co_CArgK2: Similar to previous case.

Case Co_ArgKLam: Similar to case for Co_ArgK.

Case Co_CArgKLam1: Similar to case for Co_CArgK1.

Case Co_CArgKLam2: Similar to previous case.

Case Co_Res: Immediate.

Case Co_ResLam: Immediate.

Case Co_InstRel: We adopt the metavariable names from the statement of the rule:

$$\frac{\begin{array}{l}\Sigma; \Gamma \vdash_{\mathsf{co}} \gamma : \Pi a{:}_{\mathsf{Rel}} \kappa_1.\, \sigma_1 \sim \Pi a{:}_{\mathsf{Rel}} \kappa_2.\, \sigma_2 \\ \Sigma; \Gamma \vdash_{\mathsf{co}} \eta : \tau_1 \ {}^{\kappa_1}\!\sim^{\kappa_2} \tau_2\end{array}}{\Sigma; \Gamma \vdash_{\mathsf{co}} \gamma @ \eta : \sigma_1[\tau_1/a] \sim \sigma_2[\tau_2/a]} \quad \text{Co\_InstRel}$$

We will prove that $\sigma_1[\tau_1/a]$ is well-typed; the proof for $\sigma_2[\tau_2/a]$ is similar. The induction hypothesis (and some inversion) tells us $\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{ty}} \Pi a{:}_{\mathsf{Rel}} \kappa_1.\, \sigma_1 : \mathbf{Type}$. Further inversion gives us $\Sigma; \Gamma, a{:}_{\mathsf{Rel}} \kappa_1 \vdash_{\mathsf{ty}} \sigma_1 : \mathbf{Type}$. The induction hypothesis and an inversion also gives us $\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{ty}} \tau_1 : \kappa_1$. Lemma C.35 gives us $\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{ty}} \sigma_1[\tau_1/a] : \mathbf{Type}$ as desired.

Case Co_InstIrrel: Similar to previous case.

Case Co_CInst: Similar to previous case.

Case Co_InstLamRel: Similar to previous case.

Case Co_InstLamIrrel: Similar to previous case.

Case Co_CInstLam: Similar to previous case.

Case Co_NthRel: Immediate.

Case Co_NthIrrel: Immediate.

Case Co_Left: Immediate.

Case Co_RightRel: We adopt the metavariable names from the statement of the rule:

$$\frac{\begin{array}{l}\Sigma; \Gamma \vdash_{\mathsf{co}} \gamma : \tau_1 \_ \sigma_1 \ {}^{\kappa_3[\sigma_1/a]}\!\sim^{\kappa_4[\sigma_2/a]} \tau_2 \_ \sigma_2 \\ \Sigma; \Gamma \vdash_{\mathsf{ty}} \sigma_1 : \kappa_1 \qquad \Sigma; \Gamma \vdash_{\mathsf{ty}} \sigma_2 : \kappa_2 \qquad \Sigma; \Gamma \vdash_{\mathsf{co}} \eta : \kappa_1 \ {}^{\mathbf{Type}}\!\sim^{\mathbf{Type}} \kappa_2\end{array}}{\Sigma; \Gamma \vdash_{\mathsf{co}} \mathbf{right}_\eta\, \gamma : \sigma_1 \ {}^{\kappa_1}\!\sim^{\kappa_2} \sigma_2} \quad \text{Co\_RightRel}$$

The induction hypothesis tells us $\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{prop}} \tau_1\, \sigma_1 \ {}^{\kappa_3[\sigma_1/a]}\!\sim^{\kappa_4[\sigma_2/a]} \tau_2\, \sigma_2\ \mathsf{ok}$, and thus inversion gives us $\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{ty}} \tau_1\, \sigma_1 : \kappa_3[\sigma_1/a]$. We know $\Sigma; \Gamma \vdash_{\mathsf{ty}} \tau_1 : \Pi a{:}_{\mathsf{Rel}} \kappa_1.\, \kappa_3$, and thus we can invert the type application to get $\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{ty}} \sigma_1 : \kappa_1$ as desired. We can similarly derive the type for σ_2, and we are thus done.

Case Co_RightIrrel: Similar to previous case.

Case Co_Kind: By Lemma C.43.

Case Co_Step: Immediate.

□

C.9 Preservation

Lemma C.45 (Correctness of build_kpush_co)**.** *Assume* $\Sigma;\Gamma \vdash_{\mathsf{cev}} \overline{\psi} : \Delta[\overline{\tau}/\overline{a}]$*, and let* $\gamma_i = \mathsf{build\_kpush\_co}(\eta; \overline{\psi}_{1\ldots i-1})$ *and* $\psi_i' = \mathsf{cast\_kpush\_arg}(\psi_i; \gamma_i)$*. If* $\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{co}} \eta : (\,\overline{\Pi}\Delta.\,\sigma)[\overline{\tau}/\overline{a}] \sim (\,\overline{\Pi}\Delta.\,\sigma)[\overline{\tau}'/\overline{a}]$*, then:*

1. $\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{co}} \mathsf{build\_kpush\_co}(\eta; \overline{\psi}) : \sigma[\overline{\tau}/\overline{a}][\overline{\psi}/\mathsf{dom}(\Delta)] \sim \sigma[\overline{\tau}'/\overline{a}][\overline{\psi}'/\mathsf{dom}(\Delta)]$
2. $\Sigma;\Gamma \vdash_{\mathsf{cev}} \overline{\psi}' : \Delta[\overline{\tau}'/\overline{a}]$

Proof. Proceed by induction on $\Sigma;\Gamma \vdash_{\mathsf{cev}} \overline{\psi} : \Delta[\overline{\tau}/\overline{a}]$.

Case Cev_Nil: In this case, both $\overline{\psi}$ and Δ are empty. We must prove $\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{co}} \mathsf{build\_kpush\_co}(\eta; \varnothing) : \sigma[\overline{\tau}/\overline{a}] \sim \sigma[\overline{\tau}'/\overline{a}]$. By definition, $\mathsf{build\_kpush\_co}(\eta; \varnothing) = \eta$. We are done by assumption and Cev_Nil.

Case Cev_TyRel: In this case, we have $\overline{\psi} = \overline{\psi}_0, \tau_0$ and $\Delta = \Delta_0, b{:}_{\mathsf{Rel}}\kappa$ with $\Sigma;\Gamma \vdash_{\mathsf{cev}} \overline{\psi}_0 : \Delta_0[\overline{\tau}/\overline{a}]$ and $\Sigma;\Gamma \vdash_{\mathsf{ty}} \tau_0 : \kappa[\overline{\tau}/\overline{a}][\overline{\psi}_0/\mathsf{dom}(\Delta_0)]$. We can see that $\mathsf{build\_kpush\_co}(\eta; \overline{\psi}_0, \tau_0) = \mathbf{let}\ c := \mathsf{build\_kpush\_co}(\eta; \overline{\psi}_0)\ \mathbf{in}\ c@(\tau_0 \approx_{\mathbf{argk}\, c} \tau_0 \rhd \mathbf{argk}\, c)$. The induction hypothesis tells us that $\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{co}} c : (\overline{\Pi} b{:}_{\mathsf{Rel}}\kappa.\,\sigma)[\overline{\tau}/\overline{a}][\overline{\psi}_0/\mathsf{dom}(\Delta_0)] \sim (\overline{\Pi} b{:}_{\mathsf{Rel}}\kappa.\,\sigma)[\overline{\tau}'/\overline{a}][\overline{\psi}_0'/\mathsf{dom}(\Delta_0)]$. We can thus deduce the following:

- $\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{co}} \mathbf{argk}\, c : \kappa[\overline{\tau}/\overline{a}][\overline{\psi}_0/\mathsf{dom}(\Delta_0)] \sim \kappa[\overline{\tau}'/\overline{a}][\overline{\psi}_0'/\mathsf{dom}(\Delta_0)]$
- $\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{ty}} \tau_0 \rhd \mathbf{argk}\, c : \kappa[\overline{\tau}'/\overline{a}][\overline{\psi}_0'/\mathsf{dom}(\Delta_0)]$
- $\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{co}} \tau_0 \approx_{\mathbf{argk}\, c} \tau_0 \rhd \mathbf{argk}\, c : \tau_0 \sim \tau_0 \rhd \mathbf{argk}\, c$
- $\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{co}} c@(\tau_0 \approx_{\mathbf{argk}\, c} \tau_0 \rhd \mathbf{argk}\, c) : \sigma[\overline{\tau}/\overline{a}][\overline{\psi}_0/\mathsf{dom}(\Delta_0)][\tau_0/b] \sim \sigma[\overline{\tau}'/\overline{a}][\overline{\psi}_0'/\mathsf{dom}(\Delta_0)][\tau_0 \rhd \mathbf{argk}\, c/b]$

Note that $\mathsf{cast\_kpush\_arg}(\tau_0; c) = \tau_0 \rhd \mathbf{argk}\, c$ and thus that we can say $\tau_0' = \tau_0 \rhd \mathbf{argk}\, c$. Noting that the $\overline{\psi}_0$ cannot have b free due to the Barendregt convention, we can rewrite the substutition $[\overline{\psi}_0/\mathsf{dom}(\Delta_0)][\tau_0/b]$ as $[\overline{\psi}/\mathsf{dom}(\Delta)]$ and rewrite the last judgment above as $\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{co}} \mathsf{build\_kpush\_co}(\eta; \overline{\psi}) : \sigma[\overline{\tau}/\overline{a}][\overline{\psi}/\mathsf{dom}(\Delta)] \sim \sigma[\overline{\tau}'/\overline{a}][\overline{\psi}'/\mathsf{dom}(\Delta)]$, which is what we are trying to prove. We are done proving result (1).

For result (2), we must prove $\Sigma;\Gamma \vdash_{\mathsf{cev}} \overline{\psi}_0', \tau_0 \rhd \mathbf{argk}\, c : \Delta_0[\overline{\tau}'/\overline{a}], b{:}_{\mathsf{Rel}}\kappa[\overline{\tau}'/\overline{a}]$. This fact comes from a straightforward use of Cev_TyRel.

Case Cev_TyIrrel: Similar to previous case.

Case Cev_Co: In this case, we have $\overline{\psi} = \overline{\psi}_0, \gamma_0$ and $\Delta = \Delta_0, c_0{:}\phi_0$ with $\Sigma;\Gamma \vdash_{\mathsf{cev}} \overline{\psi}_0 : \Delta_0[\overline{\tau}/\overline{a}]$ and $\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{co}} \gamma_0 : \phi_0[\overline{\tau}/\overline{a}][\overline{\psi}_0/\mathsf{dom}(\Delta_0)]$. We can see that $\mathsf{build\_kpush\_co}(\eta; \overline{\psi}_0, \gamma_0) = \mathbf{let}\ c :=$

$\mathsf{build\_kpush\_co}(\eta; \overline{\psi}_0)$ **in** $c@(\gamma_0, \mathbf{sym}\,(\mathbf{argk}_1\, c) \fatsemi \gamma_0 \fatsemi \mathbf{argk}_2\, c)$. The induction hypothesis tells us that $\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{co}} c : (\Pi c_0{:}\phi_0.\, \sigma)[\overline{\tau}/\overline{a}][\overline{\psi}_0/\mathsf{dom}(\Delta_0)] \sim (\Pi c_0{:}\phi_0.\, \sigma)[\overline{\tau}'/\overline{a}][\overline{\psi}_0'/\mathsf{dom}(\Delta_0)]$. Let $\phi_0 = \sigma_1 \sim \sigma_2$. We can thus deduce the following:

- $\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{co}} \mathbf{sym}\,(\mathbf{argk}_1\, c) : \sigma_1[\overline{\tau}'/\overline{a}][\overline{\psi}_0'/\mathsf{dom}(\Delta_0)] \sim \sigma_1[\overline{\tau}/\overline{a}][\overline{\psi}_0/\mathsf{dom}(\Delta_0)]$
- $\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{co}} \mathbf{argk}_2\, c : \sigma_2[\overline{\tau}/\overline{a}][\overline{\psi}_0/\mathsf{dom}(\Delta_0)] \sim \sigma_2[\overline{\tau}'/\overline{a}][\overline{\psi}_0'/\mathsf{dom}(\Delta_0)]$
- $\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{co}} \mathbf{sym}\,(\mathbf{argk}_1\, c) \fatsemi \gamma_0 \fatsemi \mathbf{argk}_2\, c : \sigma_1[\overline{\tau}'/\overline{a}][\overline{\psi}_0'/\mathsf{dom}(\Delta_0)] \sim \sigma_2[\overline{\tau}'/\overline{a}][\overline{\psi}_0'/\mathsf{dom}(\Delta_0)]$
- $\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{co}} c@(\gamma_0, \mathbf{sym}\,(\mathbf{argk}_1\, c) \fatsemi \gamma_0 \fatsemi \mathbf{argk}_2\, c) : \sigma[\overline{\tau}/\overline{a}][\overline{\psi}_0/\mathsf{dom}(\Delta_0)][\gamma_0/c_0] \sim \sigma[\overline{\tau}'/\overline{a}][\overline{\psi}_0'/\mathsf{dom}(\Delta_0)][\mathbf{sym}\,(\mathbf{argk}_1\, c) \fatsemi \gamma_0 \fatsemi \mathbf{argk}_2\, c/c_0]$

Note that $\mathsf{cast\_kpush\_arg}(\gamma_0; c) = \mathbf{sym}\,(\mathbf{argk}_1\, c) \fatsemi \gamma_0 \fatsemi \mathbf{argk}_2\, c$ and thus that we can say $\gamma_0' = \mathbf{sym}\,(\mathbf{argk}_1\, c) \fatsemi \gamma_0 \fatsemi \mathbf{argk}_2\, c$. Noting that the $\overline{\psi}_0$ cannot have c_0 free due to the Barendregt convention, we can rewrite the substitution $[\overline{\psi}_0/\mathsf{dom}(\Delta_0)][\gamma_0/c_0]$ as $[\overline{\psi}/\mathsf{dom}(\Delta)]$ and rewrite the last judgment above as $\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{co}} \mathsf{build\_kpush\_co}(\eta; \overline{\psi}) : \sigma[\overline{\tau}/\overline{a}][\overline{\psi}/\mathsf{dom}(\Delta)] \sim \sigma[\overline{\tau}'/\overline{a}][\overline{\psi}'/\mathsf{dom}(\Delta)]$, which is what we are trying to prove. We are done proving result (1).

To prove result (2), we must show $\Sigma; \Gamma \vdash_{\mathsf{cev}} \overline{\psi}_0', \mathbf{sym}\,(\mathbf{argk}_1\, c) \fatsemi \gamma_0 \fatsemi \mathbf{argk}_2\, c : \Delta_0[\overline{\tau}'/\overline{a}], c_0{:}\phi_0[\overline{\tau}'/\overline{a}]$, which we get from a straightforward use of Cev_Co.

□

Remark. Lemma C.45 could also be rewritten to work with $\underset{\sim}{\Pi}$, but with no need.

Theorem C.46 (Preservation)**.** *If* $\Sigma; \Gamma \vdash_{\mathsf{ty}} \tau : \kappa$ *and* $\Sigma; \Gamma \vdash_{\mathsf{s}} \tau \longrightarrow \tau'$*, then* $\Sigma; \Gamma \vdash_{\mathsf{ty}} \tau' : \kappa$*.*

Proof. By induction on the typing derivation.

Case Ty_Var: Impossible, as variables do not step.

Case Ty_Con: Impossible, as constants do not step.

Case Ty_AppRel: We now have several cases, depending on how the expression has stepped:

Case S_BetaRel: By Lemma C.35.

Case S_App_Cong: By induction.

Case S_PushRel: We adopt the metavariable names from the statement of the rule:

$$\frac{\begin{array}{l}\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{co}} \gamma_0 : \Pi a{:}_{\mathsf{Rel}}\kappa.\, \sigma \sim \Pi a{:}_{\mathsf{Rel}}\kappa'.\, \sigma' \\ \gamma_1 = \mathbf{sym}\,(\mathbf{argk}\,\gamma_0) \qquad \gamma_2 = \gamma_0@(\tau \rhd \gamma_1 \approx_{\mathbf{sym}\,\gamma_1} \tau)\end{array}}{\Sigma; \Gamma \vdash_{\mathsf{s}} (v \rhd \gamma_0)\,\tau \longrightarrow v\,(\tau \rhd \gamma_1) \rhd \gamma_2}\ \text{S\_PushRel}$$

Inversion on $\Sigma; \Gamma \vdash_{\mathsf{ty}} (v \rhd \gamma_0)\,\tau : \kappa_0$ gives us $\Sigma; \Gamma \vdash_{\mathsf{ty}} \tau : \kappa'$ and $\Sigma; \Gamma \vdash_{\mathsf{ty}} v : \Pi a{:}_\rho \kappa.\, \sigma$. Straightforward application of typing rules gives us $\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{co}} \gamma_1 : \kappa' \sim \kappa$ and $\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{co}} \gamma_2 : \sigma[\tau \rhd \gamma_1/a] \sim \sigma'[\tau/a]$. We can then derive $\Sigma; \Gamma \vdash_{\mathsf{ty}} \tau \rhd \gamma_1 : \kappa$ and thus $\Sigma; \Gamma \vdash_{\mathsf{ty}} v\,(\tau \rhd \gamma_1) : \sigma[\tau \rhd \gamma_1/a]$ and $\Sigma; \Gamma \vdash_{\mathsf{ty}} v\,(\tau \rhd \gamma_1) \rhd \gamma_2 : \sigma'[\tau/a]$ as desired.

Case Ty_AppIrrel: We now have several cases:

Case S_BetaIrrel: By Lemma C.35.

Case S_App_Cong: By induction.

Case S_PushIrrel: Similar to the case for S_PushRel.

Case Ty_CApp: We now have several cases:

Case S_CBeta: By Lemma C.36.

Case S_App_Cong: By induction.

Case S_CPush: We adopt the metavariable names of the rule:

$$\frac{\begin{array}{l}\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{co}} \gamma_0 : \Pi c{:}\phi.\, \sigma \sim \Pi c{:}\phi'.\, \sigma' \\ \gamma_1 = \mathbf{argk}_1\,\gamma_0 \qquad \gamma_2 = \mathbf{argk}_2\,\gamma_0 \\ \eta' = \gamma_1 \fatsemi \eta \fatsemi \mathbf{sym}\,\gamma_2 \qquad \gamma_3 = \gamma_0@(\eta', \eta)\end{array}}{\Sigma; \Gamma \vdash_{\mathsf{s}} (v \rhd \gamma_0)\,\eta \longrightarrow v\,\eta' \rhd \gamma_3}\ \text{S\_CPush}$$

We can see that $\Sigma; \Gamma \vdash_{\mathsf{ty}} (v \rhd \gamma_0)\,\eta : \sigma'[\eta/c]$. Let $\phi = \tau_1 \sim \tau_2$ and $\phi' = \tau_3 \sim \tau_4$. Inversion and application of typing rules tells us the following:

- $\Sigma; \Gamma \vdash_{\mathsf{ty}} v : \Pi c{:}\phi.\, \sigma$
- $\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{co}} \eta : \tau_3 \sim \tau_4$
- $\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{co}} \gamma_1 : \tau_1 \sim \tau_3$
- $\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{co}} \gamma_2 : \tau_2 \sim \tau_4$
- $\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{co}} \eta' : \tau_1 \sim \tau_2$
- $\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{co}} \gamma_3 : \sigma[\eta'/c] \sim \sigma'[\eta/c]$
- $\Sigma; \Gamma \vdash_{\mathsf{ty}} v\,\eta' : \sigma[\eta'/c]$
- $\Sigma; \Gamma \vdash_{\mathsf{ty}} v\,\eta' \rhd \gamma_3 : \sigma'[\eta/c]$

Note that the last fact proves this case.

Case Ty_Pi: Impossible, as Π-types do not step.

Case Ty_Cast: We now have several cases:

Case S_Trans: We adopt the metavariable names of the rule:

$$\frac{}{\Sigma; \Gamma \vdash_{\mathsf{s}} (v \rhd \gamma_1) \rhd \gamma_2 \longrightarrow v \rhd (\gamma_1 \fatsemi \gamma_2)} \quad \text{S\_Trans}$$

We know $\Sigma; \Gamma \vdash_{\mathsf{ty}} (v \rhd \gamma_1) \rhd \gamma_2 : \kappa$. Inversion and typing rules give us the following:

- $\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{co}} \gamma_2 : \kappa_2 \sim \kappa$
- $\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{co}} \gamma_1 : \kappa_3 \sim \kappa_2$
- $\Sigma; \Gamma \vdash_{\mathsf{ty}} v : \kappa_3$
- $\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{co}} \gamma_1 \fatsemi \gamma_2 : \kappa_3 \sim \kappa$
- $\Sigma; \Gamma \vdash_{\mathsf{ty}} v \rhd (\gamma_1 \fatsemi \gamma_2) : \kappa$

Note that the last fact proves this case.

Case S_Cast_Cong: By induction.

Case Ty_Case: We now have several cases:

Case S_Match: We adopt the metavariable names of the rule:

$$\frac{alt_i \;=\; H \to \tau_0}{\Sigma; \Gamma \vdash_{\mathsf{s}} \mathbf{case}_\kappa\, H_{\{\overline{\tau}\}}\, \overline{\psi}\, \mathbf{of}\, \overline{alt} \longrightarrow \tau_0\, \overline{\psi}\, \langle H_{\{\overline{\tau}\}}\, \overline{\psi} \rangle} \quad \text{S\_Match}$$

Inversion and typing rules tell us the following:

- $\Sigma; \Gamma \vdash_{\mathsf{ty}} H_{\{\overline{\tau}\}}\, \overline{\psi} : \Pi \Delta'.\, H'\, \overline{\sigma}$ (premise of Ty_Case)
- Using Lemma C.42:
  - $\Sigma \vdash_{\mathsf{tc}} H : \overline{a}{:}_{\mathsf{Irrel}} \overline{\kappa}; \Delta_2; H'$
  - $\Delta_0, \Delta_1 \;=\; \Delta_2[\overline{\tau}/\overline{a}]$
  - $\Sigma; \Gamma \vdash_{\mathsf{vec}} \overline{\psi} : \Delta_0$
  - $\Delta' \;=\; \Delta_1[\overline{\psi}/\mathsf{dom}(\Delta_0)]$ and $\overline{\sigma} \;=\; \overline{\tau}$ (Lemma C.20)
- The premises of Alt_Match (also using Lemma C.18):
  - $\Delta_3, \Delta_4 \;=\; \Delta_2[\overline{\tau}/\overline{a}]$
  - $|\Delta_4| \;=\; |\Delta_1|$
  - $\Sigma; \Gamma \vdash_{\mathsf{ty}} \tau_0 : \Pi \Delta_3, c{:}H_{\{\overline{\tau}\}}\, \overline{\psi} \sim H_{\{\overline{\tau}\}}\, \mathsf{dom}(\Delta_3).\, \kappa$
- $\Delta_3 \;=\; \Delta_0$ and $\Delta_4 \;=\; \Delta_1$ (from $|\Delta_4| \;=\; |\Delta_1|$ and the definitions of Δ_0, Δ_1, Δ_3, and Δ_4)
- $\Sigma; \Gamma \vdash_{\mathsf{ty}} \tau_0 : \Pi \Delta_0, c{:}H_{\{\overline{\tau}\}}\, \overline{\psi} \sim H_{\{\overline{\tau}\}}\, \mathsf{dom}(\Delta_0).\, \kappa$ (rewriting)
- $\Sigma; \Gamma \vdash_{\mathsf{ty}} \tau_0\, \overline{\psi} : \Pi c{:}H_{\{\overline{\tau}\}}\, \overline{\psi} \sim H_{\{\overline{\tau}\}}\, \overline{\psi}.\, \kappa$ (Lemma C.31, where the κ needs no substitution by Lemma C.12)
- $\Sigma; \Gamma \vdash_{\mathsf{ty}} \tau_0\, \overline{\psi}\, \langle H_{\{\overline{\tau}\}}\, \overline{\psi} \rangle : \kappa$ (Co_Refl and Ty_CApp, where the κ needs no substitution by Lemma C.12)

Note that this last fact proves this case.

Case S_DEFAULT: We adopt the metavariable names of the rule:

$$\frac{alt_i \;=\; \_ \to \sigma \qquad \text{no alternative in } \overline{alt} \text{ matches } H}{\Sigma; \Gamma \vdash_{\mathsf{s}} \mathbf{case}_{\kappa}\, H_{\{\overline{\tau}\}}\, \overline{\psi}\, \mathbf{of}\, \overline{alt} \longrightarrow \sigma} \quad \text{S\_DEFAULT}$$

By TY_CASE, the redex has kind κ; inversion also gives us $\Sigma; \Gamma; \sigma_0 \vdash^{\tau}_{\mathsf{alt}} \_ \to \sigma : \kappa$. Inverting ALT_DEFAULT gives us our goal.

Case S_DEFAULTCO: Similar to previous case.

Case S_CASE_CONG: By induction.

Case S_KPUSH: We adopt the metavariable names of the rule:

$$\frac{\begin{array}{l}\Sigma \vdash_{\mathsf{tc}} H : \overline{a}{:}_{\mathsf{Irrel}}\overline{\kappa}; \Delta; H' \qquad \Delta \;=\; \Delta_1, \Delta_2 \qquad n \;=\; |\Delta_2| \\ \kappa \;=\; \Pi \overline{a}{:}_{\mathsf{Irrel}}\overline{\kappa}, \Delta.\, H'\, \overline{a} \\ \sigma \;=\; \Pi(\Delta_2[\overline{\tau}/\overline{a}][\overline{\psi}/\mathsf{dom}(\Delta_1)]).\, H'\, \overline{\tau} \\ \sigma' \;=\; \Pi(\Delta_2[\overline{\tau}'/\overline{a}][\overline{\psi}'/\mathsf{dom}(\Delta_1)]).\, H'\, \overline{\tau}' \\ \Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{co}} \eta : \sigma \sim \sigma' \\ \Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{vec}} \overline{\tau}' : \overline{a}{:}_{\mathsf{Rel}}\overline{\kappa} \\ \forall i,\; \gamma_i \;=\; \mathsf{build\_kpush\_co}(\langle\kappa\rangle @ (\mathbf{nths}\,(\mathbf{res}^n\,\eta)); \overline{\psi}_{1\ldots i-1}) \\ \forall i,\; \psi'_i \;=\; \mathsf{cast\_kpush\_arg}(\psi_i; \gamma_i) \\ H \to \kappa' \in \overline{alt}\end{array}}{\Sigma; \Gamma \vdash_{\mathsf{s}} \mathbf{case}_{\kappa_0}\, (H_{\{\overline{\tau}\}}\, \overline{\psi}) \rhd \eta\, \mathbf{of}\, \overline{alt} \longrightarrow \mathbf{case}_{\kappa_0}\, H_{\{\overline{\tau}'\}}\, \overline{\psi}'\, \mathbf{of}\, \overline{alt}} \quad \text{S\_KPUSH}$$

Note that we need to prove only that the type of $(H_{\{\overline{\tau}\}}\, \overline{\psi}) \rhd \eta$ matches that of $H_{\{\overline{\tau}'\}}\, \overline{\psi}'$, namely $\Pi(\Delta_2[\overline{\tau}'/\overline{a}][\overline{\psi}'/\mathsf{dom}(\Delta_1)]).\, H'\, \overline{\tau}'$. We can derive these facts:

- $\Sigma; \Gamma \vdash_{\mathsf{ty}} H_{\{\overline{\tau}\}}\, \overline{\psi} : \Pi(\Delta_2[\overline{\tau}/\overline{a}][\overline{\psi}/\mathsf{dom}(\Delta_1)]).\, H'\, \overline{\tau}$ (by inversion of the typing judgment on the redex)
- $\Sigma; \Gamma \vdash_{\mathsf{ty}} H_{\{\overline{\tau}\}} : \Pi(\Delta_1[\overline{\tau}/\overline{a}], \Delta_2[\overline{\tau}/\overline{a}]).\, H'\, \overline{\tau}$ (by Lemma C.30 followed by inverting TY_CON)
- $\Sigma; \Gamma \vdash_{\mathsf{vec}} \overline{\psi} : \Delta_1[\overline{\tau}/\overline{a}]$ (also from Lemma C.30)
- $\Sigma; \Gamma \vdash_{\mathsf{vec}} \overline{\tau} : \overline{a}{:}_{\mathsf{Rel}}\overline{\kappa}$ (by inversion)
- $\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{vec}} \overline{\tau}' : \overline{a}{:}_{\mathsf{Rel}}\overline{\kappa}$ (from S_KPUSH)
- $\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{co}} \mathbf{res}^n\, \eta : H'\, \overline{\tau} \sim H'\, \overline{\tau}'$ (with the well-formedness of $\overline{\tau}$ and $\overline{\tau}'$ telling us that the $\overline{\tau}$ and $\overline{\tau}'$ do not have any variables in $\mathsf{dom}(\Delta_2)$ free)
- $\forall i, \exists \kappa_i,\, \Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{ty}} \tau_i : \kappa_i$ (by Lemma C.29)
- $\forall i, \exists \kappa'_i,\, \Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{ty}} \tau'_i : \kappa'_i$ (by Lemma C.29)
- $\forall i,\, \Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{co}} \mathbf{nth}_i\, (\mathbf{res}^n\, \eta) : \tau_i \sim \tau'_i$ (from CO_NTHREL)
- $\Sigma; \Gamma \vdash_{\mathsf{ty}} \kappa : \mathbf{Type}$, recalling that $\kappa \;=\; \Pi \overline{a}{:}_{\mathsf{Irrel}}\overline{\kappa}, \Delta.\, H'\, \overline{a}$ (by Lemma C.9, Lemma C.41, and Lemma C.10)
- $\Sigma; \Gamma \vdash_{\mathsf{co}} \langle\kappa\rangle : \kappa \sim \kappa$ (by CO_REFL)
- $\Sigma; \Gamma \quad \vdash_{\mathsf{co}} \quad \langle\kappa\rangle @ (\mathbf{nths}\,(\mathbf{res}^n\,\eta)) \quad : \quad (\Pi \Delta_1, \Delta_2.\, H'\, \overline{a})[\overline{\tau}/\overline{a}] \quad \sim$

$(\Pi\Delta_1, \Delta_2.\, H'\,\overline{a})[\overline{\tau}'/\overline{a}]$ (by Lemma C.32)
- $\Sigma; \Gamma \vdash_{\mathsf{cev}} \overline{\psi} : \Delta_1[\overline{\tau}/\overline{a}]$ (by Lemma C.27)
- $\Sigma; \Gamma \vdash_{\mathsf{cev}} \overline{\psi}' : \Delta_1[\overline{\tau}'/\overline{a}]$ (by Lemma C.45)
- $\Sigma; \Gamma \vdash_{\mathsf{vec}} \overline{\psi}' : \Delta_1[\overline{\tau}'/\overline{a}]$ (by Lemma C.27)
- $\Sigma; \Gamma \vdash_{\mathsf{ty}} H_{\{\overline{\tau}'\}} : \Pi(\Delta_1[\overline{\tau}'/\overline{a}], \Delta_2[\overline{\tau}'/\overline{a}]).\, H'\,\overline{\tau}'$ (by a use of Ty_Con, along with Lemma C.9)
- $\Sigma; \Gamma \vdash_{\mathsf{ty}} H_{\{\overline{\tau}'\}}\,\overline{\psi}' : \Pi(\Delta_2[\overline{\tau}'/\overline{a}][\overline{\psi}'/\mathsf{dom}(\Delta_1)]).\, H'\,\overline{\tau}'$

This last fact is what we are trying to prove, and so we are done.

Case Ty_Lam: We now have several cases:

Case S_IrrelAbs_Cong: By induction.

Case S_APush: We adopt the metavariable names from the rule:

$$\frac{\gamma_1 = \mathop{\Pi}\limits_{\sim} a{:}_{\mathsf{Irrel}}\langle\kappa\rangle.\, \gamma \qquad \gamma_2 = \tau_1 \approx_{\langle\mathbf{Type}\rangle} \tau_2 \\ \tau_1 = \mathop{\Pi}\limits_{\sim} a{:}_{\mathsf{Irrel}}\kappa.\, (\kappa_1[a \rhd \mathbf{sym}\,\langle\kappa\rangle/a]) \qquad \tau_2 = \mathop{\Pi}\limits_{\sim} a{:}_{\mathsf{Irrel}}\kappa.\, \kappa_1}{\Sigma; \Gamma \vdash_{\mathsf{s}} \lambda a{:}_{\mathsf{Irrel}}\kappa.\, (v \rhd \gamma) \longrightarrow (\lambda a{:}_{\mathsf{Irrel}}\kappa.\, v) \rhd (\gamma_1 \fatsemi \gamma_2)}\ \text{S\_APush}$$

Inversion and typing rules then give us the following facts:

- $\Sigma; \Gamma \vdash_{\mathsf{ty}} \lambda a{:}_{\mathsf{Irrel}}\kappa.\, (v \rhd \gamma) : \mathop{\Pi}\limits_{\sim} a{:}_{\mathsf{Irrel}}\kappa.\, \kappa_1$
- $\Sigma; \Gamma, a{:}_{\mathsf{Irrel}}\kappa \vdash_{\mathsf{ty}} v \rhd \gamma : \kappa_1$
- $\Sigma; \mathsf{Rel}(\Gamma), a{:}_{\mathsf{Rel}}\kappa \vdash_{\mathsf{co}} \gamma : \kappa_0 \sim \kappa_1$
- $\Sigma; \Gamma, a{:}_{\mathsf{Irrel}}\kappa \vdash_{\mathsf{ty}} v : \kappa_0$
- $\Sigma; \Gamma \vdash_{\mathsf{ty}} \lambda a{:}_{\mathsf{Irrel}}\kappa.\, v : \mathop{\Pi}\limits_{\sim} a{:}_{\mathsf{Irrel}}\kappa.\, \kappa_0$
- $\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{ty}} \kappa : \mathbf{Type}$ (by Lemma C.9 and Lemma C.7)
- $\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{co}} \langle\kappa\rangle : \kappa \sim \kappa$ (by Co_Refl)
- $\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{co}} \mathop{\Pi}\limits_{\sim} a{:}_{\mathsf{Irrel}}\langle\kappa\rangle.\, \gamma : \mathop{\Pi}\limits_{\sim} a{:}_{\mathsf{Irrel}}\kappa.\, \kappa_0 \sim \mathop{\Pi}\limits_{\sim} a{:}_{\mathsf{Irrel}}\kappa.\, (\kappa_1[a \rhd \mathbf{sym}\,\langle\kappa\rangle/a])$ (by Co_PiTy)
- $\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{ty}} \mathop{\Pi}\limits_{\sim} a{:}_{\mathsf{Irrel}}\kappa.\, (\kappa_1[a \rhd \mathbf{sym}\,\langle\kappa\rangle/a]) : \mathbf{Type}$ (by Lemma C.44)
- $\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{ty}} \mathop{\Pi}\limits_{\sim} a{:}_{\mathsf{Irrel}}\kappa.\, \kappa_1 : \mathbf{Type}$ (by Lemma C.43)
- $\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{co}} (\mathop{\Pi}\limits_{\sim} a{:}_{\mathsf{Irrel}}\kappa.\, (\kappa_1[a \rhd \mathbf{sym}\,\langle\kappa\rangle/a])) \approx_{\langle\mathbf{Type}\rangle} (\mathop{\Pi}\limits_{\sim} a{:}_{\mathsf{Irrel}}\kappa.\, \kappa_1) : (\mathop{\Pi}\limits_{\sim} a{:}_{\mathsf{Irrel}}\kappa.\, (\kappa_1[a \rhd \mathbf{sym}\,\langle\kappa\rangle/a])) \sim (\mathop{\Pi}\limits_{\sim} a{:}_{\mathsf{Irrel}}\kappa.\, \kappa_1)$ (by Co_Coherence)

We can then conclude, by Co_Trans and Ty_Cast, that the result has the same type, $\mathop{\Pi}\limits_{\sim} a{:}_{\mathsf{Irrel}}\kappa.\, \kappa_1$ as the redex.

Case Ty_Fix: We now have several cases:

Case S_Unroll: We adopt the variable names from the rule:

$$\frac{\tau = \lambda a{:}_{\mathsf{Rel}}\kappa.\, \sigma}{\Sigma; \Gamma \vdash_{\mathsf{s}} \mathbf{fix}\,\tau \longrightarrow \sigma[\mathbf{fix}\,\tau/a]}\ \text{S\_Unroll}$$

We can then derive the following:

- $\Sigma; \Gamma \vdash_{\mathsf{ty}} \lambda a{:}_{\mathsf{Rel}}\kappa.\, \sigma : \tilde{\Pi} a{:}_{\mathsf{Rel}}\kappa.\, \kappa$ (by inversion)
- $\Sigma; \Gamma, a{:}_{\mathsf{Rel}}\kappa \vdash_{\mathsf{ty}} \sigma : \kappa$ (by inversion)
- $\Sigma; \Gamma \vdash_{\mathsf{ty}} \mathbf{fix}\,(\lambda a{:}_{\mathsf{Rel}}\kappa.\, \sigma) : \kappa$ (by Ty_Fix)
- $\Sigma; \Gamma \vdash_{\mathsf{ty}} \sigma[\mathbf{fix}\,(\lambda a{:}_{\mathsf{Rel}}\kappa.\, \sigma)/a] : \kappa$ (by Lemma C.35)

This last judgment is what we are trying to prove; we are done.

Case S_Fix_Cong: By induction.

Case S_FPush: We adopt the metavariable names from the rule:

$$\frac{\begin{array}{l}\gamma_1 \;=\; \gamma_0 @ (a \approx_{\gamma_2} a \rhd \gamma_2) \fatsemi \mathbf{sym}\,\gamma_2 \\ \gamma_2 \;=\; \mathbf{argk}\,\gamma_0\end{array}}{\Sigma; \Gamma \vdash_{\mathsf{s}} \mathbf{fix}\,((\lambda a{:}_{\mathsf{Rel}}\kappa.\, \sigma) \rhd \gamma_0) \longrightarrow (\mathbf{fix}\,(\lambda a{:}_{\mathsf{Rel}}\kappa.\, (\sigma \rhd \gamma_1))) \rhd \gamma_2}\ \text{S\_FPush}$$

We can derive the following facts:

- $\Sigma; \Gamma \vdash_{\mathsf{ty}} \mathbf{fix}\,((\lambda a{:}_{\mathsf{Rel}}\kappa.\, \sigma) \rhd \gamma_0) : \kappa_1$ (conclusion of Ty_Fix)
- $\Sigma; \Gamma \vdash_{\mathsf{ty}} (\lambda a{:}_{\mathsf{Rel}}\kappa.\, \sigma) \rhd \gamma_0 : \tilde{\Pi} a{:}_{\mathsf{Rel}}\kappa_1.\, \kappa_1$ (premise of Ty_Fix)
- $\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{co}} \gamma_0 : \kappa_0 \sim \tilde{\Pi} a{:}_{\mathsf{Rel}}\kappa_1.\, \kappa_1$ (inversion on Ty_Cast)
- $\Sigma; \Gamma \vdash_{\mathsf{ty}} \lambda a{:}_{\mathsf{Rel}}\kappa.\, \sigma : \kappa_0$ (same inversion)
- $\Sigma; \Gamma \vdash_{\mathsf{ty}} \lambda a{:}_{\mathsf{Rel}}\kappa.\, \sigma : \tilde{\Pi} a{:}_{\mathsf{Rel}}\kappa.\, \kappa_2$ (inversion by Ty_Lam)
- $\kappa_0 = \tilde{\Pi} a{:}_{\mathsf{Rel}}\kappa.\, \kappa_2$ (Lemma C.20)
- $\Sigma; \Gamma, a{:}_{\mathsf{Rel}}\kappa \vdash_{\mathsf{ty}} \sigma : \kappa_2$ (inversion by Ty_Lam)
- $\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{co}} \gamma_0 : (\tilde{\Pi} a{:}_{\mathsf{Rel}}\kappa.\, \kappa_2) \sim (\tilde{\Pi} a{:}_{\mathsf{Rel}}\kappa_1.\, \kappa_1)$ (substitution)
- $\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{co}} \mathbf{argk}\,\gamma_0 : \kappa \sim \kappa_1$ (Co_ArgK)
- $\gamma_2 = \mathbf{argk}\,\gamma_0$ (premise of S_FPush)
- $\Sigma; \Gamma, a{:}_{\mathsf{Rel}}\kappa \vdash_{\mathsf{ty}} a \rhd \gamma_2 : \kappa_1$ (Ty_Cast)
- $\Sigma; \Gamma, a{:}_{\mathsf{Rel}}\kappa \vdash_{\mathsf{co}} a \approx_{\gamma_2} a \rhd \gamma_2 : a \sim a \rhd \gamma_2$ (Co_Coherence)
- $\Sigma; \Gamma, a{:}_{\mathsf{Rel}}\kappa \vdash_{\mathsf{co}} \gamma_0 @ (a \approx_{\gamma_2} a \rhd \gamma_2) : \kappa_2 \sim \kappa_1[a \rhd \gamma_2/a]$ (Co_InstRel)
- $\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{ty}} \tilde{\Pi} a{:}_{\mathsf{Rel}}\kappa_1.\, \kappa_1 : \mathbf{Type}$ (Lemma C.43)
- $\Sigma; \mathsf{Rel}(\Gamma), a{:}_{\mathsf{Rel}}\kappa_1 \vdash_{\mathsf{ty}} \kappa_1 : \mathbf{Type}$ (inversion on Ty_Pi)
- $\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{ty}} \kappa_1 : \mathbf{Type}$ (Lemma C.7 and Lemma C.10)
- $\kappa_1[a \rhd \gamma_2/a] = \kappa_1$ (Lemma C.12, noting that $a \# \kappa_1$)
- $\Sigma; \Gamma, a{:}_{\mathsf{Rel}}\kappa \vdash_{\mathsf{co}} \gamma_0 @ (a \approx_{\gamma_2} a \rhd \gamma_2) : \kappa_2 \sim \kappa_1$ (substitution)
- $\Sigma; \Gamma, a{:}_{\mathsf{Rel}}\kappa \vdash_{\mathsf{co}} \mathbf{sym}\,\gamma_2 : \kappa_1 \sim \kappa$ (Co_Sym with Lemma C.10)
- $\gamma_1 = \gamma_0 @ (a \approx_{\gamma_2} a \rhd \gamma_2) \fatsemi \mathbf{sym}\,\gamma_2$ (premise of S_FPush)
- $\Sigma; \Gamma, a{:}_{\mathsf{Rel}}\kappa \vdash_{\mathsf{co}} \gamma_1 : \kappa_2 \sim \kappa$ (Co_Trans)
- $\Sigma; \Gamma, a{:}_{\mathsf{Rel}}\kappa \vdash_{\mathsf{ty}} \sigma \rhd \gamma_1 : \kappa$ (Ty_Cast and Lemma C.6)
- $\Sigma; \Gamma \vdash_{\mathsf{ty}} \lambda a{:}_{\mathsf{Rel}}\kappa.\, (\sigma \rhd \gamma_1) : \tilde{\Pi} a{:}_{\mathsf{Rel}}\kappa.\, \kappa$ (Ty_Lam)
- $\Sigma; \Gamma \vdash_{\mathsf{ty}} \mathbf{fix}\,(\lambda a{:}_{\mathsf{Rel}}\kappa.\, (\sigma \rhd \gamma_1)) : \kappa$ (Ty_Fix)
- $\Sigma; \Gamma \vdash_{\mathsf{ty}} (\mathbf{fix}\,(\lambda a{:}_{\mathsf{Rel}}\kappa.\, (\sigma \rhd \gamma_1))) \rhd \gamma_2 : \kappa_1$ (Ty_Cast)

The last item proves this case.

Case Ty_Absurd: Impossible, as $\mathbf{absurd}\,\gamma\,\tau$ does not step.

□

C.10 Consistency

Definition C.47 (Coercion erasure)**.** *Define the erasure of a type $\epsilon = \lfloor \tau \rfloor$ by the following function (including auxiliary functions):*

$$\begin{aligned}
\lfloor a \rfloor &= a \\
\lfloor H_{\{\overline{\tau}\}} \rfloor &= H_{\{\lfloor \overline{\tau} \rfloor\}} \\
\lfloor \tau_1\, \tau_2 \rfloor &= \lfloor \tau_1 \rfloor\, \lfloor \tau_2 \rfloor \\
\lfloor \tau_1\, \{\tau_2\} \rfloor &= \lfloor \tau_1 \rfloor\, \{\lfloor \tau_2 \rfloor\} \\
\lfloor \tau\, \gamma \rfloor &= \lfloor \tau \rfloor \bullet \\
\lfloor \Pi \delta.\, \tau \rfloor &= \Pi \lfloor \delta \rfloor.\, \lfloor \tau \rfloor \\
\lfloor \tau \rhd \gamma \rfloor &= \lfloor \tau \rfloor \\
\lfloor \mathbf{case}_\kappa\, \tau\, \mathbf{of}\, \overline{alt} \rfloor &= \mathbf{case}_{\lfloor \kappa \rfloor}\, \lfloor \tau \rfloor\, \mathbf{of}\, \lfloor \overline{alt} \rfloor \\
\lfloor \lambda \delta.\, \tau \rfloor &= \lambda \lfloor \delta \rfloor.\, \lfloor \tau \rfloor \\
\lfloor \mathbf{fix}\, \tau \rfloor &= \mathbf{fix}\, \lfloor \tau \rfloor \\
\lfloor \mathbf{absurd}\, \gamma\, \tau \rfloor &= \mathbf{absurd} \bullet \lfloor \tau \rfloor
\end{aligned}$$

$$\begin{aligned}
\lfloor a{:}_\rho \kappa \rfloor &= a{:}_\rho \lfloor \kappa \rfloor \\
\lfloor c{:}\phi \rfloor &= \bullet{:}\lfloor \phi \rfloor
\end{aligned}$$

$$\lfloor \tau_1\ {}^{\kappa_1}{\sim}^{\kappa_2}\ \tau_2 \rfloor = \lfloor \tau_1 \rfloor\ {}^{\lfloor \kappa_1 \rfloor}{\sim}^{\lfloor \kappa_2 \rfloor}\ \lfloor \tau_2 \rfloor$$

$$\lfloor \pi \to \tau \rfloor = \pi \to \lfloor \tau \rfloor$$

Notation C.48 (Erased types in consistency proof). The rewrite relation $\rightsquigarrow$ is defined only over *erased* types. We use a convention that the occurrence of a metavariable in a mention of the $\rightsquigarrow$ relation indicates that the metavariable represents an erased element.

Notation C.49 (Reduction).

- We write $\overline{\psi} \rightsquigarrow \overline{\psi}'$ to mean $\forall i, \psi_i \rightsquigarrow \psi'_i$.
- We write $\tau_1 \rightsquigarrow \tau_3 \leftsquigarrow \tau_2$ to mean $\tau_1 \rightsquigarrow \tau_3$ and $\tau_2 \rightsquigarrow \tau_3$.
- We write $\rightsquigarrow^*$ to mean the reflexive, transitive closure of $\rightsquigarrow$.
- We write $\tau_1 \rightsquigarrow^* \tau_3\ {}^*\!\leftsquigarrow \tau_2$ to mean $\tau_1 \rightsquigarrow^* \tau_3$ and $\tau_2 \rightsquigarrow^* \tau_3$.

Lemma C.50 (Parallel reduction substitution)**.** *Assume $\overline{\psi} \rightsquigarrow \overline{\psi}'$. We can then conclude:*

1. $\tau[\overline{\psi}/\overline{z}] \rightsquigarrow \tau[\overline{\psi}'/\overline{z}]$
2. $\delta[\overline{\psi}/\overline{z}] \rightsquigarrow \delta[\overline{\psi}'/\overline{z}]$

Proof. By straightforward mutual induction on the structure of τ/δ. □

Lemma C.51 (Parallel reduction substitution in parallel)**.** *Assume* $\overline{\psi} \rightsquigarrow \overline{\psi}'$.

1. *If* $\tau_1 \rightsquigarrow \tau_2$, *then* $\tau_1[\overline{\psi}/\overline{z}] \rightsquigarrow \tau_2[\overline{\psi}'/\overline{z}]$.
2. *If* $\delta_1 \rightsquigarrow \delta_2$, *then* $\delta_1[\overline{\psi}/\overline{z}] \rightsquigarrow \delta_2[\overline{\psi}'/\overline{z}]$.

Proof. By induction on $\tau_1 \rightsquigarrow \tau_2/\delta_1 \rightsquigarrow \delta_2$.

Case R_Refl: By Lemma C.50.

Congruence rules: By induction.

Case R_BetaRel: It must be that $\tau_1 = (\lambda b{:}_{\mathsf{Rel}}\kappa_1.\, \tau_3)\, \tau_4$ and $\tau_2 = \tau_3'[\tau_4'/b]$ where $\tau_3 \rightsquigarrow \tau_3'$ and $\tau_4 \rightsquigarrow \tau_4'$. We must show that $(\lambda b{:}_{\mathsf{Rel}}\kappa_1[\overline{\psi}/\overline{z}].\, \tau_3[\overline{\psi}/\overline{z}])\, \tau_4[\overline{\psi}/\overline{z}] \rightsquigarrow \tau_3'[\tau_4'/b][\overline{\psi}'/\overline{z}]$. Proceeding by R_BetaRel, the left-hand-side steps to $\tau_5[\tau_6/b]$ where $\tau_3[\overline{\psi}/\overline{z}] \rightsquigarrow \tau_5$ and $\tau_4[\overline{\psi}/\overline{z}] \rightsquigarrow \tau_6$. (We can choose τ_5 and τ_6.) We must thus show that $\tau_5[\tau_6/b] = \tau_3'[\tau_4'/b][\overline{\psi}'/\overline{z}]$. First, we reorder substitutions to get $\tau_3'[\tau_4'/b][\overline{\psi}'/\overline{z}] = \tau_3'[\overline{\psi}'/\overline{z}][\tau_4'[\overline{\psi}'/\overline{z}]/b]$, noting that $b \# \overline{\psi}'$ by the Barendregt convention. Choose $\tau_5 = \tau_3'[\overline{\psi}'/\overline{z}]$ and $\tau_6 = \tau_4'[\overline{\psi}'/\overline{z}]$. We must show that $\tau_3[\overline{\psi}/\overline{z}] \rightsquigarrow \tau_5$ and $\tau_4[\overline{\psi}/\overline{z}] \rightsquigarrow \tau_6$; expanding gives us that we must show $\tau_3[\overline{\psi}/\overline{z}] \rightsquigarrow \tau_3'[\overline{\psi}'/\overline{z}]$ and $\tau_4[\overline{\psi}/\overline{z}] \rightsquigarrow \tau_4'[\overline{\psi}'/\overline{z}]$. Both of these follow directly from the induction hypothesis, and so we are done.

Case R_BetaIrrel: Similar to previous case.

Case R_CBeta: By induction.

Case R_Match: It must be that:

- $\tau_1 = \mathbf{case}_\kappa\, H_{\{\overline{\sigma}\}}\, \overline{\psi}_0\, \mathbf{of}\, \overline{alt}$
- $\tau_2 = \tau_4\, \overline{\psi}_0'\, \bullet$ where $H \to \tau_3 \in \overline{alt}$, $\tau_3 \rightsquigarrow \tau_4$, and $\overline{\psi}_0 \rightsquigarrow \overline{\psi}_0'$.

We must show that $\mathbf{case}_{\kappa[\overline{\psi}/\overline{z}]}\, H_{\{\overline{\sigma}[\overline{\psi}/\overline{z}]\}}\, \overline{\psi}_0[\overline{\psi}/\overline{z}]\, \mathbf{of}\, \overline{alt}[\overline{\psi}/\overline{z}] \rightsquigarrow \tau_4[\overline{\psi}'/\overline{z}]\, \overline{\psi}_0'[\overline{\psi}'/\overline{z}]\, \bullet$. Proceeding by R_Match, the left-hand side steps to $\tau_5\, \overline{\psi}_0''\, \bullet$ where $\tau_3[\overline{\psi}/\overline{z}] \rightsquigarrow \tau_5$ and $\overline{\psi}_0[\overline{\psi}/\overline{z}] \rightsquigarrow \overline{\psi}_0''$, and we get to choose τ_5 and $\overline{\psi}_0''$. We must show that $\tau_5\, \overline{\psi}_0''\, \bullet = \tau_4[\overline{\psi}'/\overline{z}]\, \overline{\psi}_0'[\overline{\psi}'/\overline{z}]\, \bullet$. Choose $\tau_5 = \tau_4[\overline{\psi}'/\overline{z}]$ and $\overline{\psi}_0'' = \overline{\psi}_0'[\overline{\psi}'/\overline{z}]$. We must show that $\tau_3[\overline{\psi}/\overline{z}] \rightsquigarrow \tau_4[\overline{\psi}'/\overline{z}]$ and $\overline{\psi}_0[\overline{\psi}/\overline{z}] \rightsquigarrow \overline{\psi}_0'[\overline{\psi}'/\overline{z}]$. Both of these follow from the induction hypothesis, and so we are done.

Case R_Default: It must be that:

- $\tau_1 = \mathbf{case}_\kappa\, H_{\{\overline{\sigma}\}}\, \overline{\psi}_0\, \mathbf{of}\, \_ \to \sigma_0; \overline{alt}$
- $\tau_2 = \sigma_0'$ where $\sigma_0 \rightsquigarrow \sigma_0'$

We are done by the induction hypothesis.

Case R_Unroll: It must be that:

- $\tau_1 = \mathbf{fix}\,(\lambda a{:}_{\mathsf{Rel}}\kappa_1.\,\tau_3)$
- $\tau_2 = \tau_4[\mathbf{fix}\,(\lambda a{:}_{\mathsf{Rel}}\kappa_2.\,\tau_4)/a]$ where $\kappa_1 \rightsquigarrow \kappa_2$ and $\tau_3 \rightsquigarrow \tau_4$.

We must show that $\mathbf{fix}\,(\lambda a{:}_{\mathsf{Rel}}\kappa_1[\overline{\psi}/\overline{z}].\,\tau_3[\overline{\psi}/\overline{z}]) \rightsquigarrow \tau_4[\mathbf{fix}\,(\lambda a{:}_{\mathsf{Rel}}\kappa_2.\,\tau_4)/a][\overline{\psi}'/\overline{z}]$. Proceeding by R_UNROLL, the left-hand side steps to $\tau_5[\mathbf{fix}\,(\lambda a{:}_{\mathsf{Rel}}\kappa_3.\,\tau_5)/a]$ where $\tau_3[\overline{\psi}/\overline{z}] \rightsquigarrow \tau_5$ and $\kappa_1[\overline{\psi}/\overline{z}] \rightsquigarrow \kappa_3$. We must show that $\tau_5[\mathbf{fix}\,(\lambda a{:}_{\mathsf{Rel}}\kappa_3.\,\tau_5)/a] = \tau_4[\mathbf{fix}\,(\lambda a{:}_{\mathsf{Rel}}\kappa_2.\,\tau_4)/a][\overline{\psi}'/\overline{z}]$. Reorder substitutions on the right to get

$$\tau_4[\mathbf{fix}\,(\lambda a{:}_{\mathsf{Rel}}\kappa_2.\,\tau_4)/a][\overline{\psi}'/\overline{z}] = \tau_4[\overline{\psi}'/\overline{z}][\mathbf{fix}\,(\lambda a{:}_{\mathsf{Rel}}\kappa_2[\overline{\psi}'/\overline{z}].\,\tau_4[\overline{\psi}'/\overline{z}])/a],$$

where $a \# \overline{\psi}'$ by the Barendregt convention. Choose $\tau_5 = \tau_4[\overline{\psi}'/\overline{z}]$ and $\kappa_3 = \kappa_2[\overline{\psi}'/\overline{z}]$. It remains only to show that $\tau_3[\overline{\psi}/\overline{z}] \rightsquigarrow \tau_4[\overline{\psi}'/\overline{z}]$ and $\kappa_1[\overline{\psi}/\overline{z}] \rightsquigarrow \kappa_2[\overline{\psi}'/\overline{z}]$, both of which follow from the induction hypothesis. We are done.

□

Lemma C.52 (Parallel repeated reduction substitution)**.** *If $\tau_1 \rightsquigarrow^* \tau_2$ and $\overline{\psi} \rightsquigarrow^* \overline{\psi}'$, then $\tau_1[\overline{\psi}/\overline{z}] \rightsquigarrow^* \tau_2[\overline{\psi}'/\overline{z}]$.*

Proof. By iterated induction on the lengths of the reduction chains. □

Lemma C.53 (Application reduction)**.** *If $H_{\{\overline{\tau}\}}\,\overline{\psi} \rightsquigarrow \sigma$, then $\sigma = H_{\{\overline{\tau}'\}}\,\overline{\psi}'$ where $\overline{\tau} \rightsquigarrow \overline{\tau}'$ and $\overline{\psi} \rightsquigarrow \overline{\psi}'$.*

Proof. Straightforward induction on the structure of $\sigma_0 = H_{\{\overline{\tau}\}}\,\overline{\psi}$. □

Lemma C.54 (Local diamond)**.** *Let τ_i denote an erased type and δ_i an erased binder.*

1. *If $\tau_0 \rightsquigarrow \tau_1$ and $\tau_0 \rightsquigarrow \tau_2$, then there exists τ_3 such that $\tau_1 \rightsquigarrow \tau_3 \leftsquigarrow \tau_2$.*
2. *If $\delta_0 \rightsquigarrow \delta_1$ and $\delta_0 \rightsquigarrow \delta_2$, then there exists δ_3 such that $\delta_1 \rightsquigarrow \delta_3 \leftsquigarrow \delta_2$.*

Proof. By induction on the structure of τ_0/δ_0 followed by case analysis on the reduction of τ_0/δ_0. We ignore overlap with the R_REFL rule, as this is always trivially handled.

Case $\tau_0 = a$: $\tau_1 = \tau_2 = \tau_3 = a$.

Case $\tau_0 = H_{\{\overline{\tau}\}}$: By induction.

Case $\tau_0 = \sigma_1\,\sigma_2$: We now have several cases:

Case R_APPREL/R_APPREL: By induction.

Case R_APPREL/R_BETAREL: It must be that:

- $\tau_0 = (\lambda a{:}_\rho \kappa_1.\,\sigma_3)\,\sigma_4$
- $\tau_1 = (\lambda a{:}_\rho \kappa_2.\,\sigma_5)\,\sigma_6$, where $\kappa_1 \rightsquigarrow \kappa_2$, $\sigma_3 \rightsquigarrow \sigma_5$, and $\sigma_4 \rightsquigarrow \sigma_6$, and
- $\tau_2 = \sigma_3[\sigma_4/a]$.

Choose $\tau_3 = \sigma_5[\sigma_6/a]$. We must show $\tau_1 \rightsquigarrow \tau_3$ and $\tau_2 \rightsquigarrow \tau_3$. The first is by R_BetaRel. The second is by Lemma C.51.

Case R_BetaRel/R_BetaRel: It must be that:

- $\tau_0 = (\lambda a{:}_\rho \kappa.\, \sigma_3)\, \sigma_4$
- $\tau_1 = \sigma_3'[\sigma_4'/a]$, where $\sigma_3 \rightsquigarrow \sigma_3'$ and $\sigma_4 \rightsquigarrow \sigma_4'$
- $\tau_2 = \sigma_3''[\sigma_4''/a]$, where $\sigma_3 \rightsquigarrow \sigma_3''$ and $\sigma_4 \rightsquigarrow \sigma_4''$.

Using the induction hypothesis, we can get σ_5 and σ_6 such that

- $\sigma_3' \rightsquigarrow \sigma_5 \leftsquigarrow \sigma_3''$
- $\sigma_4' \rightsquigarrow \sigma_6 \leftsquigarrow \sigma_4''$.

Choose $\tau_3 = \sigma_5[\sigma_6/a]$. We must show $\sigma_3'[\sigma_4'/a] \rightsquigarrow \sigma_5[\sigma_6/a]$ and $\sigma_3''[\sigma_4''/a] \rightsquigarrow \sigma_5[\sigma_6/a]$. Both of these follow from Lemma C.51.

Case $\tau_0 = \sigma_1\, \{\sigma_2\}$: Similar to $\tau_0 = \sigma_1\, \sigma_2$.

Case $\tau_0 = \sigma\, \bullet$: We now have several cases:

Case R_CApp/R_CApp: By induction.

Case R_CApp/R_CBeta: It must be that:

- $\tau_0 = (\lambda \bullet{:}\kappa_1 \sim \kappa_2.\, \sigma_3)\, \bullet$
- $\tau_1 = (\lambda \bullet{:}\kappa_3 \sim \kappa_4.\, \sigma_4)\, \bullet$ where $\kappa_1 \rightsquigarrow \kappa_3$, $\kappa_2 \rightsquigarrow \kappa_4$, and $\sigma_3 \rightsquigarrow \sigma_4$.
- $\tau_2 = \sigma_5$ where $\sigma_3 \rightsquigarrow \sigma_5$

The induction hypothesis gives us σ_6 such that $\sigma_4 \rightsquigarrow \sigma_6 \leftsquigarrow \sigma_5$. Choose $\tau_3 = \sigma_6$. We must show $\tau_1 \rightsquigarrow \tau_3$ and $\tau_2 \rightsquigarrow \tau_3$. The first is by R_CBeta. The second is immediate.

Case R_CBeta/R_CBeta: By induction.

Case $\tau_0 = \Pi\delta.\, \sigma_0$: By induction and R_Pi.

Case $\tau_0 = \mathbf{case}_\kappa\, \sigma_0\, \mathbf{of}\, \overline{alt}$: We now have several cases:

Case R_Case/R_Case: By induction and R_Case.

Case R_Case/R_Match: It must be that:

- $\tau_0 = \mathbf{case}_\kappa\, H_{\{\overline{\sigma}\}}\, \overline{\psi}\, \mathbf{of}\, \overline{H \to \epsilon}$
- $\tau_1 = \mathbf{case}_{\kappa'}\, H_{\{\overline{\sigma}'\}}\, \overline{\psi}'\, \mathbf{of}\, \overline{H \to \epsilon'}$ where $\kappa \rightsquigarrow \kappa'$, $\overline{\sigma} \rightsquigarrow \overline{\sigma}'$, $\overline{\psi} \rightsquigarrow \overline{\psi}'$, and $\overline{\epsilon} \rightsquigarrow \overline{\epsilon}'$ (appealing to Lemma C.53)
- $\tau_2 = \epsilon_i''\, \overline{\psi}''\, \bullet$, where $H_i = H$, $\epsilon_i \rightsquigarrow \epsilon_i''$, and $\overline{\psi} \rightsquigarrow \overline{\psi}''$.

Using the induction hypothesis, we can get ϵ_i''' such that $\epsilon_i' \rightsquigarrow \epsilon_i''' \leftsquigarrow \epsilon_i''$ and $\overline{\psi}'''$ such that $\overline{\psi}' \rightsquigarrow \overline{\psi}''' \leftsquigarrow \overline{\psi}''$. Choose $\tau_3 = \epsilon_i'''\, \overline{\psi}'''\, \bullet$. We must show both $\tau_1 \rightsquigarrow \tau_3$ and $\tau_2 \rightsquigarrow \tau_3$. The first is by R_Match. The second is by repeated use of R_AppRel/R_AppIrrel/R_CApp.

Case R_Case/R_Default: It must be that:

- $\tau_0 = \mathbf{case}_\kappa\, H_{\{\overline{\sigma}\}}\, \overline{\psi}\, \mathbf{of}\, \_ \to \sigma_0; \overline{alt}$
- $\tau_1 = \mathbf{case}_{\kappa'}\, H_{\{\overline{\sigma}'\}}\, \overline{\psi}'\, \mathbf{of}\, \_ \to \sigma_0'; \overline{alt}'$ where $\kappa \rightsquigarrow \kappa'$, $\overline{\sigma} \rightsquigarrow \overline{\sigma}'$, $\overline{\psi} \rightsquigarrow \overline{\psi}'$, $\sigma_0 \rightsquigarrow \sigma_0'$, and $\overline{alt} \rightsquigarrow \overline{alt}'$
- $\tau_2 = \sigma_0''$ where $\sigma_0 \rightsquigarrow \sigma_0''$

The induction hypothesis gives us ϵ such that $\sigma_0' \rightsquigarrow \epsilon \leftsquigarrow \sigma_0''$. We can see that τ_1 can step by R_Default (as the type constant H does not change), and thus that $\tau_1 \rightsquigarrow \epsilon \leftsquigarrow \tau_2$. We are done.

Case R_Match/R_Match: It must be that:

- $\tau_0 = \mathbf{case}_\kappa\, H_{\{\overline{\sigma}\}}\, \overline{\psi}\, \mathbf{of}\, \overline{alt}$
- $alt_i = H \to \kappa_1$
- $\tau_1 = \kappa_1'\, \overline{\psi}'\, \bullet$ where $\kappa_1 \rightsquigarrow \kappa_1'$ and $\overline{\psi} \rightsquigarrow \overline{\psi}'$.
- $\tau_2 = \kappa_1''\, \overline{\psi}''\, \bullet$ where $\kappa_1 \rightsquigarrow \kappa_1''$ and $\overline{\psi} \rightsquigarrow \overline{\psi}''$.

The induction hypothesis gives us κ_1''' and $\overline{\psi}'''$ such that:

- $\kappa_1' \rightsquigarrow \kappa_1''' \leftsquigarrow \kappa_1''$
- $\overline{\psi}' \rightsquigarrow \overline{\psi}''' \leftsquigarrow \overline{\psi}''$

Choose $\tau_3 = \kappa_1'''[\overline{\psi}'''/\overline{z}]$ and we are done by Lemma C.51.

Case R_Match/R_Default: Impossible, as the premises contradict each other.

Case R_Default/R_Default: By induction.

Case $\tau_0 = \lambda\delta_0.\, \sigma_0$**:** By induction and R_Lam.

Case $\tau_0 = \mathbf{fix}\, \sigma_0$**:** We have several cases:

Case R_Fix/R_Fix: By induction.

Case R_Fix/R_Unroll: It must be that:

- $\tau_0 = \mathbf{fix}\, (\lambda a{:}_{\mathsf{Rel}}\kappa_1.\, \sigma_1)$
- $\tau_1 = \mathbf{fix}\, (\lambda a{:}_{\mathsf{Rel}}\kappa_2.\, \sigma_2)$ where $\kappa_1 \rightsquigarrow \kappa_2$ and $\sigma_1 \rightsquigarrow \sigma_2$
- $\tau_2 = \sigma_3[\mathbf{fix}\, (\lambda a{:}_{\mathsf{Rel}}\kappa_3.\, \sigma_3)/a]$ where $\kappa_1 \rightsquigarrow \kappa_3$ and $\sigma_1 \rightsquigarrow \sigma_3$

The induction hypothesis gives us κ_4 and σ_4 such that $\kappa_2 \rightsquigarrow \kappa_4 \leftsquigarrow \kappa_3$ and $\sigma_2 \rightsquigarrow \sigma_4 \leftsquigarrow \sigma_3$. Choose $\tau_3 = \sigma_4[\mathbf{fix}\, (\lambda a{:}_{\mathsf{Rel}}\kappa_4.\, \sigma_4)/a]$. We must show $\tau_1 \rightsquigarrow \tau_3$ and $\tau_2 \rightsquigarrow \tau_3$. The first is by R_Unroll, and the second is by Lemma C.51.

Case R_Unroll/R_Unroll: It must be that:

- $\tau_0 = \mathbf{fix}\, (\lambda a{:}_{\mathsf{Rel}}\kappa_1.\, \sigma_1)$
- $\tau_1 = \sigma_2[\mathbf{fix}\, (\lambda a{:}_{\mathsf{Rel}}\kappa_2.\, \sigma_2)/a]$ where $\kappa_1 \rightsquigarrow \kappa_2$ and $\sigma_1 \rightsquigarrow \sigma_2$
- $\tau_2 = \sigma_3[\mathbf{fix}\, (\lambda a{:}_{\mathsf{Rel}}\kappa_3.\, \sigma_3)/a]$ where $\kappa_1 \rightsquigarrow \kappa_3$ and $\sigma_1 \rightsquigarrow \sigma_3$

The induction hypothesis gives us κ_4 and σ_4 such that $\kappa_2 \rightsquigarrow \kappa_4 \leftsquigarrow \kappa_3$ and $\sigma_2 \rightsquigarrow \sigma_4 \leftsquigarrow \sigma_3$. Choose $\tau_3 = \sigma_4[\mathbf{fix}\, (\lambda a{:}_{\mathsf{Rel}}\kappa_4.\, \sigma_4)/a]$ and we are done by Lemma C.51.

Case $\tau_0 = \mathbf{absurd}\,\gamma\,\sigma_0$**:** By induction and R_ABSURD.

Case $\delta_0 = a{:}_\rho\kappa_0$**:** By induction and R_TYBINDER.

Case $\delta_0 = \bullet{:}\tau_1 \sim \tau_1$**:** By induction and R_COBINDER.

□

Lemma C.55 (Confluence)**.** *Let τ_i denote an erased type. If $\tau_1 \rightsquigarrow^* \tau_2$ and $\tau_1 \rightsquigarrow^* \tau_3$, then there exists τ_4 such that $\tau_2 \rightsquigarrow^* \tau_4\ {}^*\!\!\leftsquigarrow \tau_3$.*

Proof. Consequence of Lemma C.54. □

Lemma C.56 (Π-reduction)**.** *If $\Pi\delta.\,\tau \rightsquigarrow \sigma$, then there exist δ' and τ' such that $\sigma = \Pi\delta'.\,\tau'$, $\delta \rightsquigarrow \delta'$, and $\tau \rightsquigarrow \tau'$.*

Proof. Case anlysis on $\Pi\delta.\,\tau \rightsquigarrow \sigma$. □

Lemma C.57 (λ-reduction)**.** *If $\lambda\delta.\,\tau \rightsquigarrow \sigma$, then there exist δ' and τ' such that $\sigma = \lambda\delta'.\,\tau'$, $\delta \rightsquigarrow \delta'$, and $\tau \rightsquigarrow \tau'$.*

Proof. Case anlysis on $\lambda\delta.\,\tau \rightsquigarrow \sigma$. □

Lemma C.58 (Matchable application reduction)**.** *If $\tau\_\psi \rightsquigarrow \sigma$, then there exist τ' and ψ' such that $\sigma = \tau'\_\psi'$, $\tau \rightsquigarrow \tau'$, and $\psi \rightsquigarrow \psi'$.*

Proof. Case analysis on $\tau\_\psi \rightsquigarrow \sigma$. □

Lemma C.59 (Coercion substitution/erasure)**.** $\lfloor\tau[\gamma/c]\rfloor = \lfloor\tau\rfloor$

Proof. By induction on the structure of τ. □

Lemma C.60 (Type constant kinds shape)**.** *If $\Sigma;\Gamma \vdash_{\mathsf{ty}} H_{\{\overline{\tau}_1\}}\,\overline{\psi}_1 : \kappa_1$ and $\Sigma;\Gamma \vdash_{\mathsf{ty}} H_{\{\overline{\tau}_2\}}\,\overline{\psi}_2 : \kappa_2$ (where the lengths of $\overline{\psi}_1$ and $\overline{\psi}_2$ are the same), then there exists a κ such that $\mathsf{fv}(\kappa) = \{\overline{a}\} \cup \{\overline{z}\}$, $\kappa_1 = \kappa[\overline{\tau}_1/\overline{a}, \overline{\psi}_1/\overline{z}]$, and $\kappa_2 = \kappa[\overline{\tau}_2/\overline{a}, \overline{\psi}_2/\overline{z}]$.*

Proof. Lemma C.30 tells us that there exist κ_3 and κ_4 such that $\Sigma;\Gamma \vdash_{\mathsf{ty}} H_{\{\overline{\tau}_1\}} : \kappa_3$ and $\Sigma;\Gamma \vdash_{\mathsf{ty}} H_{\{\overline{\tau}_2\}} : \kappa_4$. Inversion (via the only applicable rule, TY_CON) then tells us that $\Sigma \vdash_{\mathsf{tc}} H : \Delta_1;\Delta_2;H'$, $\kappa_3 = \text{'}\Pi(\Delta_2[\overline{\tau}_1/\mathsf{dom}(\Delta_1)]).\,H'\,\overline{\tau}_1$, and $\kappa_4 = \text{'}\Pi(\Delta_2[\overline{\tau}_2/\mathsf{dom}(\Delta_1)]).\,H'\,\overline{\tau}_2$. Lemma C.30 also tells us that $\Sigma;\Gamma \vdash_{\mathsf{vec}} \overline{\psi}_1 : \mathsf{prefix}(\Delta_2[\overline{\tau}_1/\mathsf{dom}(\Delta_1)])$ and $\Sigma;\Gamma \vdash_{\mathsf{vec}} \overline{\psi}_2 : \mathsf{prefix}(\Delta_2[\overline{\tau}_2/\mathsf{dom}(\Delta_1)])$. Let $\Delta_3, \Delta_4 = \Delta_2$, where the length of Δ_3 matches that of $\overline{\psi}_1$. Thus Lemma C.31 tells us that $\kappa_1 = \text{'}\Pi(\Delta_4[\overline{\tau}_1/\mathsf{dom}(\Delta_1), \overline{\psi}_1/\mathsf{dom}(\Delta_3)]).\,H'\,\overline{\tau}_1$ and $\kappa_2 = \text{'}\Pi(\Delta_4[\overline{\tau}_2/\mathsf{dom}(\Delta_1), \overline{\psi}_2/\mathsf{dom}(\Delta_3)]).\,H'\,\overline{\tau}_2$. Thus, we are done, with $\overline{a} = \mathsf{dom}(\Delta_1)$, $\overline{z} = \mathsf{dom}(\Delta_2)$, and $\kappa = \text{'}\Pi\Delta_4.\,H'\,\overline{a}$. □

Definition C.61 (Joinability)**.** *We say that two types τ_1 and τ_2 are* joinable *if there exists an erased type ϵ such that $\lfloor\tau_1\rfloor \rightsquigarrow^* \epsilon\ {}^*\!\!\leftsquigarrow \lfloor\tau_2\rfloor$.*

Lemma C.62 (Completeness of type reduction). *If* $\Sigma; \Gamma \vdash_{\mathsf{co}} \gamma : \tau_1 \ ^{\kappa_1}\!\sim^{\kappa_2} \tau_2$ *and* $c \tilde{\#} \gamma$ *for every* $c{:}\phi \in \Gamma$*, then:*

1. *There exists some erased type* ϵ *such that* $\lfloor \tau_1 \rfloor \rightsquigarrow^* \epsilon \ ^*\!\!\leftsquigarrow \lfloor \tau_2 \rfloor$.
2. *There exists some erased type* ϵ *such that* $\lfloor \kappa_1 \rfloor \rightsquigarrow^* \epsilon \ ^*\!\!\leftsquigarrow \lfloor \kappa_2 \rfloor$.

Proof. By induction on the typing derivation. For the purposes of exposition, we present the types cases separately from the kinds cases, but in a formal proof, they would be interleaved. First, the types cases:

Case Co_Var: Impossible.

Case Co_Refl: Choose $\epsilon = \lfloor \tau_1 \rfloor$ and we are done.

Case Co_Sym: By induction.

Case Co_Trans: Use the metavariable names from the rule:

$$\frac{\Sigma; \Gamma \vdash_{\mathsf{co}} \gamma_1 : \tau_1 \sim \tau_2 \qquad \Sigma; \Gamma \vdash_{\mathsf{co}} \gamma_2 : \tau_2 \sim \tau_3}{\Sigma; \Gamma \vdash_{\mathsf{co}} \gamma_1 \fatsemi \gamma_2 : \tau_1 \sim \tau_3} \ \text{Co\_Trans}$$

The induction hypothesis gives us ϵ_1 such that $\lfloor \tau_1 \rfloor \rightsquigarrow^* \epsilon_1 \ ^*\!\!\leftsquigarrow \lfloor \tau_2 \rfloor$. It also gives us ϵ_2 such that $\lfloor \tau_2 \rfloor \rightsquigarrow^* \epsilon_2 \ ^*\!\!\leftsquigarrow \lfloor \tau_3 \rfloor$. Lemma C.55 gives us ϵ_3 such that $\epsilon_1 \rightsquigarrow^* \epsilon_3 \ ^*\!\!\leftsquigarrow \epsilon_2$. Thus, ϵ_3 is a common reduct of $\lfloor \tau_1 \rfloor$ and $\lfloor \tau_3 \rfloor$ as desired.

Case Co_Coherence: We know that $\lfloor \tau_1 \rfloor = \lfloor \tau_2 \rfloor$ and thus either can be the common reduct.

Case Co_Con: By induction and repeated use of R_Con.

Case Co_AppRel: By induction and repeated use of R_AppRel.

Case Co_AppIrrel: By induction and repeated use of R_AppIrrel.

Case Co_CApp: By induction.

Case Co_PiTy: By induction. Note that the substitution in the conclusion is erased by coercion erasure and so poses no complications.

Case Co_PiCo: By induction. Note that we need the $c \tilde{\#} \gamma$ premise of Co_PiCo in order to use the induction hypothesis here. Once again, the substitution in the conclusion causes no bother.

Case Co_Case: By induction and R_Case.

Case Co_Lam: Similar to Co_PiTy, noting that the substitution in the result of Co_Lam is erased by coercion erasure and so poses no complications.

Case Co_CLam: Similar to Co_PiCo. Once again, the premise of $c \tilde{\#} \gamma$ is critical.

Case Co_Fix: By induction and repeated use of R_Fix.

Case Co_Absurd: By induction.

Case Co_ArgK: The induction hypothesis gives us ϵ_0 such that $\lfloor \Pi a{:}_\rho \kappa_1.\, \sigma_1 \rfloor \rightsquigarrow^* \epsilon_0 \ {}^*\!\!\leftsquigarrow \lfloor \Pi a{:}_\rho \kappa_2.\, \sigma_2 \rfloor$. By repeated use of Lemma C.56, we see that $\epsilon_0 = \Pi a{:}_\rho \kappa_3.\, \sigma_3$ such that $\lfloor \kappa_1 \rfloor \rightsquigarrow^* \kappa_3 \ {}^*\!\!\leftsquigarrow \lfloor \kappa_2 \rfloor$ and $\lfloor \sigma_1 \rfloor \rightsquigarrow^* \sigma_3 \ {}^*\!\!\leftsquigarrow \lfloor \sigma_2 \rfloor$. Thus κ_3 is a reduct of $\lfloor \kappa_1 \rfloor$ and $\lfloor \kappa_2 \rfloor$ as desired.

Case Co_CArgK1: Like previous case.

Case Co_CArgK2: Like previous case.

Case Co_ArgKLam: Like case Co_ArgK, but appealing to Lemma C.57.

Case Co_CArgKLam1: Like previous case.

Case Co_CArgKLam2: Like previous case.

Case Co_Res: By induction and Lemma C.56.

Case Co_ResLam: By induction and Lemma C.57.

Case Co_InstRel: We use the metavariable names from the rule:

$$\frac{\begin{array}{l}\Sigma; \Gamma \vdash_{\mathsf{co}} \gamma : \Pi a{:}_{\mathsf{Rel}} \kappa_1.\, \sigma_1 \sim \Pi a{:}_{\mathsf{Rel}} \kappa_2.\, \sigma_2 \\ \Sigma; \Gamma \vdash_{\mathsf{co}} \eta : \tau_1 \ {}^{\kappa_1}\!\!\sim^{\kappa_2} \tau_2\end{array}}{\Sigma; \Gamma \vdash_{\mathsf{co}} \gamma @ \eta : \sigma_1[\tau_1/a] \sim \sigma_2[\tau_2/a]} \quad \text{Co\_InstRel}$$

The induction hypothesis (along with Lemma C.56) gives us ϵ_0 and ϵ_1 such that $\lfloor \sigma_1 \rfloor \rightsquigarrow^* \epsilon_0 \ {}^*\!\!\leftsquigarrow \lfloor \sigma_2 \rfloor$ and $\lfloor \tau_1 \rfloor \rightsquigarrow^* \epsilon_1 \ {}^*\!\!\leftsquigarrow \lfloor \tau_2 \rfloor$. Lemma C.52 (with Lemma C.34) then tells us that $\lfloor \sigma_1[\tau_1/a] \rfloor \rightsquigarrow^* \epsilon_0[\epsilon_1/a] \ {}^*\!\!\leftsquigarrow \lfloor \sigma_2[\tau_2/a] \rfloor$ as desired.

Case Co_InstIrrel: Similar to previous case.

Case Co_CInst: By induction, Lemma C.56, and Lemma C.59.

Case Co_InstLamRel: Like case Co_Inst, but appealing to Lemma C.57.

Case Co_InstLamIrrel: Like previous case.

Case Co_CInstLam: Like case Co_CInst, but appealing to Lemma C.57.

Case Co_NthRel: By induction and Lemma C.53.

Case Co_NthIrrel: By induction and Lemma C.53.

Case Co_Left: By induction and Lemma C.58.

Case Co_RightRel: By induction and Lemma C.58.

Case Co_RightIrrel: By induction and Lemma C.58.

Case Co_Kind: By induction.

Case Co_Step: We now must consider the different step rules:

Case S_BetaRel: By R_BetaRel.

Case S_BetaIrrel: By R_BetaIrrel.

Case S_CBeta: By R_CBeta and Lemma C.59.

Case S_Match: By R_Match.

Case S_Default: By R_Default.

Case S_DefaultCo: By R_Default.

Case S_Unroll: By R_Unroll.

Case S_Trans: $\lfloor \tau_1 \rfloor = \lfloor \tau_2 \rfloor$ in this case.

Congruence rules: By induction.

Case S_KPush: We adopt the metavariable names from the statement of the rule:

$$\frac{\begin{array}{l}\Sigma \vdash_{\mathsf{tc}} H : \overline{a}{:}_{\mathsf{Irrel}}\overline{\kappa}; \Delta; H' \qquad \Delta = \Delta_1, \Delta_2 \qquad n = |\Delta_2| \\ \kappa = \Pi \overline{a}{:}_{\mathsf{Irrel}}\overline{\kappa}, \Delta.\, H'\, \overline{a} \\ \sigma = \Pi(\Delta_2[\overline{\tau}/\overline{a}][\overline{\psi}/\mathsf{dom}(\Delta_1)]).\, H'\, \overline{\tau} \\ \sigma' = \Pi(\Delta_2[\overline{\tau}'/\overline{a}][\overline{\psi}'/\mathsf{dom}(\Delta_1)]).\, H'\, \overline{\tau}' \\ \Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{co}} \eta : \sigma \sim \sigma' \\ \Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{vec}} \overline{\tau}' : \overline{a}{:}_{\mathsf{Rel}}\overline{\kappa} \\ \forall i,\ \gamma_i = \mathsf{build\_kpush\_co}(\langle \kappa \rangle @ (\mathbf{nths}\, (\mathbf{res}^n\, \eta)); \overline{\psi}_{1 \ldots i-1}) \\ \forall i,\ \psi_i' = \mathsf{cast\_kpush\_arg}(\psi_i; \gamma_i) \\ H \to \kappa' \in \overline{alt}\end{array}}{\Sigma; \Gamma \vdash_{\mathsf{s}} \mathbf{case}_{\kappa_0}\, (H_{\{\overline{\tau}\}}\, \overline{\psi}) \rhd \eta\, \mathbf{of}\, \overline{alt} \longrightarrow \mathbf{case}_{\kappa_0}\, H_{\{\overline{\tau}'\}}\, \overline{\psi}'\, \mathbf{of}\, \overline{alt}}\ \text{S\_KPush}$$

The only differences between τ_1 (the redex) and τ_2 (the reduct) are the $\overline{\tau}$ becoming the $\overline{\tau}'$ and the $\overline{\psi}$ becoming $\overline{\psi}'$, along with the dropped cast by η. Casting is erased, so losing η is inconsequential. By the definition of $\mathsf{cast\_kpush\_arg}$, we can see that $\lfloor \mathsf{cast\_kpush\_arg}(\psi; \gamma) \rfloor = \lfloor \psi \rfloor$ for any ψ, so $\lfloor \overline{\psi} \rfloor = \lfloor \overline{\psi}' \rfloor$. This leaves us only the $\overline{\tau}$, but we can see that $\lfloor \overline{\tau} \rfloor \rightsquigarrow^* \overline{\epsilon}\ {}^*\!\!\leftsquigarrow \lfloor \overline{\tau}' \rfloor$ (for some $\overline{\epsilon}$) by the induction hypothesis. We are done by Lemma C.52.

Other push rules: $\lfloor \tau_1 \rfloor = \lfloor \tau_2 \rfloor$ in these cases.

We now proceed to the kinds cases.

Case Co_Var: Impossible.

Case Co_Refl: Choose $\epsilon = \lfloor \kappa_1 \rfloor$ and we are done.

Case Co_Sym: By induction.

Case Co_Trans: Similar to the Co_Trans case for types, above.

Case Co_Coherence: By induction.

Case Co_Con: We adopt the metavariable names from the rule:

$$\frac{\begin{array}{l}\forall i,\ \Sigma; \Gamma \vdash_{\mathsf{co}} \gamma_i : \sigma_i \sim \sigma_i' \\ \Sigma; \Gamma \vdash_{\mathsf{ty}} H_{\{\overline{\sigma}\}} : \kappa_1 \qquad \Sigma; \Gamma \vdash_{\mathsf{ty}} H_{\{\overline{\sigma}'\}} : \kappa_2\end{array}}{\Sigma; \Gamma \vdash_{\mathsf{co}} H_{\{\overline{\gamma}\}} : H_{\{\overline{\sigma}\}} \sim H_{\{\overline{\sigma}'\}}}\ \text{Co\_Con}$$

We invert $\Sigma; \Gamma \vdash_{\mathsf{ty}} H_{\{\overline{\sigma}\}} : \kappa_1$ and $\Sigma; \Gamma \vdash_{\mathsf{ty}} H_{\{\overline{\sigma}'\}} : \kappa_2$. These both can be proved only by Ty_Con. The H in both judgments is the same, and so by Lemma C.9 and Lemma C.18, we have unique Δ_1, Δ_2, and H' such that $\Sigma \vdash_{\mathsf{tc}} H : \Delta_1; \Delta_2; H'$. We can thus see that $\kappa_1 = \Pi(\Delta_2[\overline{\sigma}/\mathsf{dom}(\Delta_1)]).\, H'\,\overline{\sigma}$ and $\kappa_2 = \Pi(\Delta_2[\overline{\sigma}'/\mathsf{dom}(\Delta_1)]).\, H'\,\overline{\sigma}'$. The induction hypothesis gives us $\overline{\epsilon}'$ such that, $\forall i$, $\lfloor \sigma_i \rfloor \rightsquigarrow^* \epsilon_i' \ {}^*\!\!\leftsquigarrow \lfloor \sigma_i' \rfloor$. Choose $\epsilon = \Pi(\lfloor \Delta_2 \rfloor[\overline{\epsilon}'/\mathsf{dom}(\Delta_1)]).\, H'\,\overline{\epsilon}'$. We must show the following:

- $\Pi(\lfloor \Delta_2 \rfloor[\lfloor \overline{\sigma} \rfloor/\mathsf{dom}(\Delta_1)]).\, H'\,\lfloor \overline{\sigma} \rfloor \rightsquigarrow^* \epsilon$
- $\Pi(\lfloor \Delta_2 \rfloor[\lfloor \overline{\sigma}' \rfloor/\mathsf{dom}(\Delta_1)]).\, H'\,\lfloor \overline{\sigma}' \rfloor \rightsquigarrow^* \epsilon$

Both of these follow from Lemma C.52.

Case Co_AppRel: We adopt the metavariable names from the rule:

$$\frac{\begin{array}{l}\Sigma; \Gamma \vdash_{\mathsf{co}} \gamma_1 : \tau_1 \sim \tau_2 \\ \Sigma; \Gamma \vdash_{\mathsf{co}} \gamma_2 : \sigma_1 \sim \sigma_2 \\ \Sigma; \Gamma \vdash_{\mathsf{ty}} \tau_1\,\sigma_1 : \kappa_1 \qquad \Sigma; \Gamma \vdash_{\mathsf{ty}} \tau_2\,\sigma_2 : \kappa_2\end{array}}{\Sigma; \Gamma \vdash_{\mathsf{co}} \gamma_1\,\gamma_2 : \tau_1\,\sigma_1 \sim \tau_2\,\sigma_2}\ \text{Co\_AppRel}$$

We invert both $\Sigma; \Gamma \vdash_{\mathsf{ty}} \tau_1\,\sigma_1 : \kappa_1$ and $\Sigma; \Gamma \vdash_{\mathsf{ty}} \tau_2\,\sigma_2 : \kappa_2$. Both must be proved by Ty_AppRel. We thus get all of the following:

- $\Sigma; \Gamma \vdash_{\mathsf{ty}} \tau_1 : \Pi_1\, a{:}_{\mathsf{Rel}}\kappa_3.\, \kappa_4$
- $\Sigma; \Gamma \vdash_{\mathsf{ty}} \sigma_1 : \kappa_3$
- $\kappa_1 = \kappa_4[\sigma_1/a]$
- $\Sigma; \Gamma \vdash_{\mathsf{ty}} \tau_2 : \Pi_2\, a{:}_{\mathsf{Rel}}\kappa_5.\, \kappa_6$
- $\Sigma; \Gamma \vdash_{\mathsf{ty}} \sigma_2 : \kappa_5$
- $\kappa_2 = \kappa_6[\sigma_2/a]$.

The (kind) induction hypothesis gives us ϵ_1 such that $\Pi_1\, a{:}_{\mathsf{Rel}}\lfloor \kappa_3 \rfloor.\, \lfloor \kappa_4 \rfloor \rightsquigarrow^* \epsilon_1 \ {}^*\!\!\leftsquigarrow \Pi_2\, a{:}_{\mathsf{Rel}}\lfloor \kappa_5 \rfloor.\, \lfloor \kappa_6 \rfloor$. Lemma C.56 tells us $\Pi_1 = \Pi_2$ and gives us ϵ_3 and ϵ_4 such that $\epsilon_1 = \Pi_1\, a{:}_{\mathsf{Rel}}\epsilon_3.\, \epsilon_4$. The (type) induction hypothesis also gives us ϵ_2 such that $\lfloor \sigma_1 \rfloor \rightsquigarrow^* \epsilon_2 \ {}^*\!\!\leftsquigarrow \lfloor \sigma_2 \rfloor$. Choose $\epsilon = \epsilon_4[\epsilon_2/a]$. We must show $\lfloor \kappa_4[\sigma_1/a] \rfloor \rightsquigarrow^* \epsilon_4[\epsilon_2/a] \ {}^*\!\!\leftsquigarrow \lfloor \kappa_6[\sigma_2/a] \rfloor$. Lemma C.34 reduces this to $\lfloor \kappa_4 \rfloor[\lfloor \sigma_1 \rfloor/a] \rightsquigarrow^* \epsilon_4[\epsilon_2/a] \ {}^*\!\!\leftsquigarrow \lfloor \kappa_6 \rfloor[\lfloor \sigma_2 \rfloor/a]$. We are done by two uses of Lemma C.52.

Case Co_AppIrrel: Similar to previous case.

Case Co_CApp: Similar to (but easier than—no argument to worry about) previous case.

Case Co_PiTy: Immediate. Both kinds are **Type**.

Case Co_PiCo: Immediate. Both kinds are **Type**.

Case Co_Case: By induction.

Case Co_Lam: We adopt the metavariable names from the rule:

$$\frac{\begin{array}{l}\Sigma;\Gamma \vdash_{\mathsf{co}} \eta : \kappa_1 \ {}^{\mathbf{Type}}\!\sim^{\mathbf{Type}} \kappa_2 \\ \Sigma;\Gamma, a{:}_\rho \kappa_1 \vdash_{\mathsf{co}} \gamma : \tau_1 \ {}^{\sigma_1}\!\sim^{\sigma_2} \tau_2 \\ \Sigma;\Gamma, a{:}_\rho \kappa_1 \vdash_{\mathsf{ty}} \tau_1 : \sigma_1 \qquad \Sigma;\Gamma, a{:}_\rho \kappa_1 \vdash_{\mathsf{ty}} \tau_2 : \sigma_2\end{array}}{\Sigma;\Gamma \vdash_{\mathsf{co}} \lambda a{:}_\rho \eta.\, \gamma : \lambda a{:}_\rho \kappa_1.\, \tau_1 \ {}^{\Pi a:_\rho \kappa_1.\, \sigma_1}\!\sim^{\Pi a:_\rho \kappa_2.\, (\sigma_2[a \rhd \mathbf{sym}\,\eta/a])} \lambda a{:}_\rho \kappa_2.\, (\tau_2[a \rhd \mathbf{sym}\,\eta/a])} \ \textsc{Co\_Lam}$$

The induction hypothesis tells us both that κ_1 and κ_2 are joinable and also that σ_1 and σ_2 are joinable. We are done by R_Pi.

Case Co_CLam: Similar to previous case, again requiring the $c \mathbin{\tilde{\#}} \gamma$ condition in order to use the induction hypothesis.

Case Co_Fix: We adopt the metavariable names from the rule:

$$\frac{\begin{array}{l}\Sigma;\Gamma \vdash_{\mathsf{co}} \gamma : \tau_1 \sim \tau_2 \\ \Sigma;\Gamma \vdash_{\mathsf{ty}} \mathbf{fix}\,\tau_1 : \kappa_1 \qquad \Sigma;\Gamma \vdash_{\mathsf{ty}} \mathbf{fix}\,\tau_2 : \kappa_2\end{array}}{\Sigma;\Gamma \vdash_{\mathsf{co}} \mathbf{fix}\,\gamma : \mathbf{fix}\,\tau_1 \sim \mathbf{fix}\,\tau_2} \ \textsc{Co\_Fix}$$

Inversion on $\Sigma;\Gamma \vdash_{\mathsf{ty}} \mathbf{fix}\,\tau_1 : \kappa_1$ tells us that $\Sigma;\Gamma \vdash_{\mathsf{ty}} \tau_1 : \Pi_1 a{:}_{\mathsf{Rel}} \kappa_1.\, \kappa_1$. Similarly, we can see that $\Sigma;\Gamma \vdash_{\mathsf{ty}} \tau_2 : \Pi_2 a{:}_{\mathsf{Rel}} \kappa_2.\, \kappa_2$. The induction hypothesis gives us ϵ_0 such that $\lfloor \Pi_1 a{:}_{\mathsf{Rel}} \kappa_1.\, \kappa_1 \rfloor \rightsquigarrow^* \epsilon_0 \ {}^*\!\!\leftsquigarrow \lfloor \Pi_2 a{:}_{\mathsf{Rel}} \kappa_2.\, \kappa_2 \rfloor$. Use of Lemma C.56 gives us ϵ_1 such that $\lfloor \kappa_1 \rfloor \rightsquigarrow^* \epsilon_1 \ {}^*\!\!\leftsquigarrow \lfloor \kappa_2 \rfloor$ and we are done.

Case Co_Absurd: By induction.

Case Co_ArgK: Here is the rule with all kinds included:

$$\frac{\Sigma;\Gamma \vdash_{\mathsf{co}} \gamma : (\Pi a{:}_\rho \kappa_1.\, \sigma_1) \ {}^{\mathbf{Type}}\!\sim^{\mathbf{Type}} (\Pi a{:}_\rho \kappa_2.\, \sigma_2)}{\Sigma;\Gamma \vdash_{\mathsf{co}} \mathbf{argk}\,\gamma : \kappa_1 \ {}^{\mathbf{Type}}\!\sim^{\mathbf{Type}} \kappa_2} \ \textsc{Co\_ArgK}$$

Both kinds are **Type** and so we are done.

Case Co_CArgK1: Examine the typing rule with kinds included:

$$\frac{\Sigma;\Gamma \vdash_{\mathsf{co}} \gamma : (\Pi c{:}(\tau_1 \ {}^{\kappa_1}\!\sim^{\kappa_2} \tau_1').\, \sigma_1) \ {}^{\mathbf{Type}}\!\sim^{\mathbf{Type}} (\Pi c{:}(\tau_2 \ {}^{\kappa_3}\!\sim^{\kappa_4} \tau_2').\, \sigma_2)}{\Sigma;\Gamma \vdash_{\mathsf{co}} \mathbf{argk}_1\,\gamma : \tau_1 \ {}^{\kappa_1}\!\sim^{\kappa_3} \tau_2} \ \textsc{Co\_CArgK1}$$

The induction hypothesis (with Lemma C.56) gives us our result.

Case Co_CArgK2: Similar to previous caes.

Case Co_ArgKLam: Immediate. Both kinds are **Type**.

Case Co_CArgKLam1: Similar to case Co_CArgK1.

Case Co_CArgKLam2: Similar to previous case.

Case Co_Res: Immediate. Both kinds are **Type**.

Case Co_ResLam: Examine the typing rule with kinds included:

$$\frac{\begin{array}{l}\Sigma;\Gamma \vdash_{\mathsf{co}} \gamma : \lambda\Delta_1.\,\tau_1 \;\;\underset{\sim}{\Pi}\Delta_1.\,\kappa_1 \sim \underset{\sim}{\Pi}\Delta_2.\,\kappa_2\;\; \lambda\Delta_2.\,\tau_2 \qquad |\Delta_1| = |\Delta_2| = n \\ \Sigma;\Gamma \vdash_{\mathsf{ty}} \tau_1 : \kappa_1 \qquad \Sigma;\Gamma \vdash_{\mathsf{ty}} \tau_2 : \kappa_2\end{array}}{\Sigma;\Gamma \vdash_{\mathsf{co}} \mathbf{res}^n\,\gamma : \tau_1 \;^{\kappa_1}\!\sim^{\kappa_2}\; \tau_2}\;\textsc{Co\_ResLam}$$

We are done by the induction hypothesis and Lemma C.56.

Case Co_InstRel: Immediate. Both kinds are **Type**.

Case Co_InstIrrel: Immediate. Both kinds are **Type**.

Case Co_CInst: Immediate. Both kinds are **Type**.

Case Co_InstLamRel: Here is the rule with kinds shown:

$$\frac{\begin{array}{l}\Sigma;\Gamma \vdash_{\mathsf{co}} \gamma : \lambda a{:}_{\mathsf{Rel}}\kappa_1.\,\tau_1 \;\;\underset{\sim}{\Pi}a{:}_{\mathsf{Rel}}\kappa_1.\,\kappa_3 \sim \underset{\sim}{\Pi}a{:}_{\mathsf{Rel}}\kappa_2.\,\kappa_4\;\; \lambda a{:}_{\mathsf{Rel}}\kappa_2.\,\tau_2 \\ \Sigma;\Gamma \vdash_{\mathsf{co}} \eta : \sigma_1 \;^{\kappa_1}\!\sim^{\kappa_2}\; \sigma_2\end{array}}{\Sigma;\Gamma \vdash_{\mathsf{co}} \gamma@\eta : \tau_1[\sigma_1/a] \;^{\kappa_3[\sigma_1/a]}\!\sim^{\kappa_4[\sigma_2/a]}\; \tau_2[\sigma_2/a]}\;\textsc{Co\_InstLamRel}$$

Our desired result follows from the induction hypothesis and Lemma C.52.

Case Co_InstLamIrrel: Similar to previous case.

Case Co_CInstLam: Here is the rule with kinds shown:

$$\frac{\begin{array}{l}\Sigma;\Gamma \vdash_{\mathsf{co}} \gamma : \lambda c{:}\phi_1.\,\sigma_1 \;\;\underset{\sim}{\Pi}c{:}\phi_1.\,\kappa_1 \sim \underset{\sim}{\Pi}c{:}\phi_2.\,\kappa_2\;\; \lambda c{:}\phi_2.\,\sigma_2 \\ \Sigma;\Gamma \vdash_{\mathsf{co}} \eta_1 : \phi_1 \qquad \Sigma;\Gamma \vdash_{\mathsf{co}} \eta_2 : \phi_2\end{array}}{\Sigma;\Gamma \vdash_{\mathsf{co}} \gamma@(\eta_1,\eta_2) : \sigma_1[\eta_1/c] \;^{\kappa_1[\eta_1/c]}\!\sim^{\kappa_2[\eta_2/c]}\; \sigma_2[\eta_2/c]}\;\textsc{Co\_CInstLam}$$

Our desired result follows by the induction hypothesis and Lemma C.59.

Case Co_NthRel: We adopt metavariable names from the statement of the rule:

$$\frac{\begin{array}{l}\Sigma;\Gamma \vdash_{\mathsf{co}} \gamma : H_{\{\overline{\kappa}\}}\,\overline{\psi} \;^{\sigma_1}\!\sim^{\sigma_2}\; H_{\{\overline{\kappa}'\}}\,\overline{\psi}' \\ \psi_i = \tau \qquad \psi_i' = \sigma \\ \Sigma;\Gamma \vdash_{\mathsf{ty}} \tau : \kappa_1 \qquad \Sigma;\Gamma \vdash_{\mathsf{ty}} \sigma : \kappa_2\end{array}}{\Sigma;\Gamma \vdash_{\mathsf{co}} \mathbf{nth}_i\,\gamma : \tau \;^{\kappa_1}\!\sim^{\kappa_2}\; \sigma}\;\textsc{Co\_NthRel}$$

The induction hypothesis gives us ϵ' such that $H_{\{\lfloor\overline{\kappa}\rfloor\}}\,\lfloor\overline{\psi}\rfloor \rightsquigarrow^* \epsilon' \;^*\!\!\leftsquigarrow H_{\{\lfloor\overline{\kappa}'\rfloor\}}\,\lfloor\overline{\psi}'\rfloor$. Furthermore, we know that the number of $\overline{\psi}$ is non-zero. The reductions must thus be combinations of R_AppRel, R_AppIrrel, and R_CApp, and we can thus consider the reduction of prefixes of the original types. Specifically, we can deduce $H_{\{\lfloor\overline{\kappa}\rfloor\}}\,\lfloor\overline{\psi}_0\rfloor\,\lfloor\tau\rfloor \rightsquigarrow^* \epsilon_0 \;^*\!\!\leftsquigarrow H_{\{\lfloor\overline{\kappa}'\rfloor\}}\,\lfloor\overline{\psi}_0'\rfloor\,\lfloor\sigma\rfloor$, where $\lfloor\overline{\psi}_0\rfloor$ is a prefix of $\overline{\psi}$ and $\overline{\psi}_0'$ is a prefix of $\overline{\psi}'$ (and τ and σ are as in the statement of the rule). Let $\tau_3 = H_{\{\overline{\kappa}\}}\,\overline{\psi}_0$ and $\tau_4 = H_{\{\overline{\kappa}'\}}\,\overline{\psi}_0'$. Lemma C.44 (and inversion) tell us that $\Sigma;\mathsf{Rel}(\Gamma) \vdash_{\mathsf{ty}} H_{\{\overline{\kappa}\}}\,\overline{\psi} : \sigma_1$ and $\Sigma;\mathsf{Rel}(\Gamma) \vdash_{\mathsf{ty}} H_{\{\overline{\kappa}'\}}\,\overline{\psi}' : \sigma_2$. By Lemma C.30, there must be σ_3 and σ_4 such that $\Sigma;\mathsf{Rel}(\Gamma) \vdash_{\mathsf{ty}} \tau_3 : \sigma_3$ and $\Sigma;\mathsf{Rel}(\Gamma) \vdash_{\mathsf{ty}} \tau_4 : \sigma_4$. Lemma

C.60 tells us that $\sigma_3 = \sigma_5[\overline{\kappa}/\overline{a}, \overline{\psi}/\overline{z}]$ and $\sigma_4 = \sigma_5[\overline{\kappa}'/\overline{a}, \overline{\psi}'/\overline{z}]$ for some σ_5, $\overline{a}$, and $\overline{z}$. Lemma C.53 tells us that $\overline{\kappa}$ and $\overline{\kappa}'$ are joinable, as are $\overline{\psi}$ and $\overline{\psi}'$. We thus have, by Lemma C.52 that σ_3 and σ_4 are joinable. Inversion on $\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{ty}} \tau_3\, \tau : \sigma_6$ and $\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{ty}} \tau_4\, \sigma : \sigma_7$ tell us that σ_3 and σ_4 must have the form $\Pi_1\, a{:}_\rho \kappa_1.\, \sigma_8$ and $\Pi_2\, a{:}_\rho \kappa_2.\, \sigma_9$, where $\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{ty}} \tau : \kappa_1$ and $\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{ty}} \sigma : \kappa_2$. By Lemma C.56, we can see that the joinability of σ_3 and σ_4 imply the joinability of κ_1 and κ_2, as desired.

Case Co_NthIrrel: Similar to previous case.

Case Co_Left: By induction.

Case Co_RightRel: By induction.

Case Co_RightIrrel: By induction.

Case Co_Kind: Immediate, as both kinds are **Type**.

Case Co_Step: With kinds shown, the rule is as follows:

$$\frac{\Sigma; \Gamma \vdash_{\mathsf{ty}} \tau : \kappa \qquad \Sigma; \Gamma \vdash_{\mathsf{ty}} \tau' : \kappa \qquad \Sigma; \Gamma \vdash_{\mathsf{s}} \tau \longrightarrow \tau'}{\Sigma; \Gamma \vdash_{\mathsf{co}} \mathbf{step}\, \tau : \tau \sim^{\kappa} \tau'} \text{ Co\_Step}$$

We can see that the desired result is immediate, as both types have the same kind κ.

□

Definition C.63 (Erased values)**.** *An* erased value *is an erased type ϵ such that there exists a value v with $\lfloor v \rfloor = \epsilon$.*

Definition C.64 (Consistency over erased types)**.** *We overload the notation $\tau_1 \propto \tau_2$ to include relating erased types, where the rules are the same except that all types are erased.*

Lemma C.65 (Consistency is reflexive)**.** $\epsilon \propto \epsilon$

Proof. By induction on the structure of ϵ. □

Lemma C.66 (Consistency is symmetric)**.** *If $\tau_1 \propto \tau_2$, then $\tau_2 \propto \tau_1$.*

Proof. By induction on $\tau_1 \propto \tau_2$. □

Lemma C.67 (Reduction preserves values)**.** *If $\epsilon_1 \rightsquigarrow \epsilon_2$ and ϵ_1 is an erased value, then ϵ_2 is an erased value.*

Proof. By induction. The induction hypothesis in needed only in the $\epsilon_1 = \lambda a{:}_{\mathsf{Irrel}} \kappa.\, \sigma$ case. □

Lemma C.68 (Consistency of reduction)**.** *If $\epsilon_1 \rightsquigarrow \epsilon_2$, then $\epsilon_1 \propto \epsilon_2$.*

Proof. If ϵ_1 is not an erased value, the result is immediate. We thus assume ϵ_1 is an erased value. By induction over $\epsilon_1 \rightsquigarrow \epsilon_2$.

Case R_Refl: By Lemma C.65.

Case R_Con: Immediate.

Case R_AppRel: Since ϵ_1 is an erased value, it must be $H_{\{\overline{\tau}\}}\,\overline{\psi}$. We are done by Lemma C.53.

Case R_AppIrrel: Similar to previous case.

Case R_CApp: Similar to previous case.

Case R_Pi: By induction.

Case R_Case: Impossible.

Case R_Lam: Immediate.

Case R_Fix: Impossible.

Case R_Absurd: Impossible.

Case R_BetaRel: Impossible.

Case R_BetaIrrel: Impossible.

Case R_CBeta: Impossible.

Case R_Match: Impossible.

Case R_Default: Impossible.

Case R_Unroll: Impossible.

□

Lemma C.69 (Consistency of reductions)**.** *If $\epsilon_1 \rightsquigarrow^* \epsilon_2$, then $\epsilon_1 \propto \epsilon_2$.*

Proof. By induction on the length of the reduction chain, appealing to Lemma C.68. □

Lemma C.70 (Π-expansion)**.** *If ϵ_1 is an erased value and $\epsilon_1 \rightsquigarrow \Pi\delta.\,\tau$, then there exist δ' and τ' such that $\epsilon_1 = \Pi\delta'.\,\tau'$ where $\delta \rightsquigarrow \delta'$ and $\tau \rightsquigarrow \tau'$.*

Proof. By case analysis on $\epsilon_1 \rightsquigarrow \Pi\delta.\,\tau$. □

Lemma C.71 (Π-expansions)**.** *If ϵ_1 is an erased value and $\epsilon_1 \rightsquigarrow^* \Pi\delta.\,\tau$, then there exist δ' and τ' such that $\epsilon_1 = \Pi\delta'.\,\tau'$ where $\delta \rightsquigarrow^* \delta'$ and $\tau \rightsquigarrow^* \tau'$.*

Proof. By induction on the length of the reduction chain, using Lemma C.67 to establish the value condition and appealing to Lemma C.70. □

Lemma C.72 (Joinable types are consistent)**.** *If* $\epsilon_1 \rightsquigarrow^* \epsilon_3 \ {}^*\!\!\leftsquigarrow \epsilon_2$*, then* $\epsilon_1 \propto \epsilon_2$*.*

Proof. By induction on the structure of ϵ_1. In all cases: If either ϵ_1 or ϵ_2 is not an erased value, the result is immediate. We thus assume both are values. We know (from Lemma C.69) that $\epsilon_1 \propto \epsilon_3$ and $\epsilon_2 \propto \epsilon_3$ and (from Lemma C.67) that ϵ_3 is a value.

Now, suppose ϵ_1 is not a Π-type or is a Π-type over a proposition. We can see from inversion on $\epsilon_1 \propto \epsilon_3$ that ϵ_3 must have the same head. We can further see from inversion on $\epsilon_2 \propto \epsilon_3$ that ϵ_2 must have the same shape, and thus that $\epsilon_1 \propto \epsilon_2$ as desired.

Finally, we consider $\epsilon_1 = \Pi a{:}_\rho \kappa.\, \tau$. We see (from Lemma C.56) that $\epsilon_3 = \Pi a{:}_\rho \kappa'.\, \tau'$ with $\kappa \rightsquigarrow^* \kappa'$ and $\tau \rightsquigarrow^* \tau'$. Now we can use Lemma C.71 to see that $\epsilon_2 = \Pi a{:}_\rho \kappa''.\, \tau''$ with $\kappa'' \rightsquigarrow^* \kappa'$ and $\tau'' \rightsquigarrow^* \tau'$. The induction hypothesis tells us $\tau \propto \tau''$, which gives us $\epsilon_1 \propto \epsilon_2$ by C_PiTy. □

Lemma C.73 (Erasure/consistency)**.** *If* $\lfloor \tau_1 \rfloor \propto \lfloor \tau_2 \rfloor$*, then* $\tau_1 \propto \tau_2$*.*

Proof. If either τ_1 or τ_2 is not a value, the result is immediate. We thus assume both are values. Proceed by induction on the structure of τ_1.

Case $\tau_1 = a$**:** Impossible.

Case $\tau_1 = H_{\{\overline{\tau}\}}$**:** We have $\lfloor \tau_1 \rfloor = H_{\{\lfloor \overline{\tau} \rfloor\}}$, and thus $\lfloor \tau_2 \rfloor = H_{\{\overline{\tau'}\}}\, \overline{\psi}$. From the definition of $\lfloor \tau_2 \rfloor$, we can see that τ_2 must be headed by H or be a cast. The latter is impossible, as a cast is not a value. Thus τ_2 is headed by H and we are done.

Case $\tau_1 = \sigma_1\, \sigma_2$**:** For τ_1 to be a value, it must be headed by some constant H. Proceed as in the previous case.

Case $\tau_1 = \Pi a{:}_\rho \kappa.\, \tau$**:** Similar to case for $H_{\{\overline{\tau}\}}$, but also using the induction hypothesis.

Case $\tau_1 = \Pi c{:}\phi.\, \tau$**:** Similar to case for $H_{\{\overline{\tau}\}}$.

Case $\tau_1 = \tau \rhd \gamma$**:** Impossible.

Case $\tau_1 = \gamma$**:** Impossible.

Case $\tau_1 = \mathbf{case}_\kappa\, \tau\, \mathbf{of}\, \overline{alt}$**:** Impossible.

Case $\tau_1 = \lambda \delta.\, \sigma$**:** Similar to case for $H_{\{\overline{\tau}\}}$.

Case $\tau_1 = \mathbf{fix}\, \sigma$**:** Impossible.

Case $\tau_1 = \mathbf{absurd}\, \gamma\, \tau_0$**:** Impossible.

□

Lemma C.74 (Consistency)**.** *If* Γ *contains only irrelevant type variable bindings and* $\Sigma; \Gamma \vdash_{\mathsf{co}} \gamma : \tau_1 \sim \tau_2$ *then* $\tau_1 \propto \tau_2$*.*

Proof. If either τ_1 or τ_2 is not a value, then we are done. So, we assume that both are values. Lemma C.62 gives us ϵ such that $\lfloor \tau_1 \rfloor \rightsquigarrow^* \epsilon \ {}^*\!\!\leftsquigarrow \lfloor \tau_2 \rfloor$. (This lemma is applicable because there are no coercion bindings in Γ.) Lemma C.72 then tell us that $\lfloor \tau_1 \rfloor \propto \lfloor \tau_2 \rfloor$. Finally, Lemma C.73 gives us $\tau_1 \propto \tau_2$ as desired. □

C.11 Progress

Lemma C.75 (Canonical forms)**.**

1. *If* $\Sigma; \Gamma \vdash_{\mathsf{ty}} v : \tilde{\Pi}\delta.\,\kappa$*, then* $v = \lambda\delta.\,\sigma$*.*
2. *If* $\Sigma; \Gamma \vdash_{\mathsf{ty}} v : \dot{\Pi}\delta.\,\kappa$*, then* $v = H_{\{\overline{\tau}\}}\,\overline{\psi}$*.*
3. *If* $\Sigma; \Gamma \vdash_{\mathsf{ty}} v : H\,\overline{\sigma}$*, then* $v = H'_{\{\overline{\sigma}\}}\,\overline{\psi}$*.*

Proof. By case analysis on the shape of values (along with Lemma C.20). □

Lemma C.76 (Value types)**.** *If* $\Sigma; \Gamma \vdash_{\mathsf{ty}} v : \kappa$*, then* κ *is a value.*

Proof. By case analysis on the possible shapes of values. □

Lemma C.77 (Type constant parents)**.** *If* $\vdash_{\mathsf{sig}} \Sigma\ \mathsf{ok}$ *and* $\Sigma \vdash_{\mathsf{tc}} H : \Delta_1; \Delta_2; H'$*, then* $\Sigma \vdash_{\mathsf{tc}} H' : \varnothing; \mathsf{Rel}(\Delta_1); \mathbf{Type}$*.*

Proof. By case analysis on $\Sigma \vdash_{\mathsf{tc}} H : \Delta_1; \Delta_2; H'$ □

Theorem C.78 (Progress)**.** *Assume* Γ *has only irrelevant variable bindings. If* $\Sigma; \Gamma \vdash_{\mathsf{ty}} \tau : \kappa$*, then either* τ *is a value* v*,* τ *is a coerced value* $v \rhd \gamma$*, or there exists* τ' *such that* $\Sigma; \Gamma \vdash_{\mathsf{s}} \tau \longrightarrow \tau'$*.*

Proof. By induction on the typing judgment.

Case Ty_Var: Impossible.

Case Ty_Con: τ is a value.

Case Ty_AppRel: We adopt the metavariable names from the rule:

$$\frac{\Sigma; \Gamma \vdash_{\mathsf{ty}} \tau_1 : \Pi a{:}_{\mathsf{Rel}}\kappa_1.\,\kappa_2 \qquad \Sigma; \Gamma \vdash_{\mathsf{ty}} \tau_2 : \kappa_1}{\Sigma; \Gamma \vdash_{\mathsf{ty}} \tau_1\,\tau_2 : \kappa_2[\tau_2/a]}\ \textsc{Ty\_AppRel}$$

Use the induction hypothesis on τ_1, giving us several cases:

Case $\tau_1 = v$**:** We now use Lemma C.75 to give us two cases:

Case $\tau_1 = H_{\{\overline{\tau}\}}\,\overline{\psi}$**:** Then $\tau = H_{\{\overline{\tau}\}}\,\overline{\psi}\,\tau_2$ is a value and we are done.
Case $\tau_1 = \lambda a{:}_{\mathsf{Rel}}\kappa_1.\,\sigma$**:** We are done by S_BetaRel.

Case $\tau_1 = v \rhd \gamma$**:** We wish to use S_PushRel but we must prove $\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{co}} \gamma : \Pi a{:}_{\mathsf{Rel}}\kappa.\,\sigma \sim \Pi a{:}_{\mathsf{Rel}}\kappa'.\,\sigma'$ (for some Π, a, κ, σ, κ', and σ'). We know by inversion that $\Sigma; \Gamma \vdash_{\mathsf{ty}} v \rhd \gamma : \Pi a{:}_{\mathsf{Rel}}\kappa_1.\,\kappa_2$. Further inversion gives us $\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{co}} \gamma : \kappa_0 \sim \Pi a{:}_{\mathsf{Rel}}\kappa_1.\,\kappa_2$ and $\Sigma; \Gamma \vdash_{\mathsf{ty}} v : \kappa_0$. Lemma C.74 tells us that $\kappa_0 \propto \Pi a{:}_{\mathsf{Rel}}\kappa_1.\,\kappa_2$. Lemma C.76 tells us that κ_0 is a value. Inversion on $\kappa_0 \propto \Pi a{:}_{\mathsf{Rel}}\kappa_1.\,\kappa_2$ must happen via C_PiTy, telling us that $\kappa_0 = \Pi a{:}_{\mathsf{Rel}}\kappa_1'.\,\kappa_2'$ for some κ_1' and κ_2'. We can thus use S_PushRel and are done with this case.

Case $\Sigma; \Gamma \vdash_{\mathsf{s}} \tau_1 \longrightarrow \tau_1'$**:** We are done by S_AppRel_Cong.

Case Ty_AppIrrel: We adopt the metavariable names from the rule:

$$\frac{\Sigma; \Gamma \vdash_{\mathsf{ty}} \tau_1 : \Pi a{:}_{\mathsf{Irrel}} \kappa_1.\, \kappa_2 \qquad \Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{ty}} \tau_2 : \kappa_1}{\Sigma; \Gamma \vdash_{\mathsf{ty}} \tau_1\, \{\tau_2\} : \kappa_2[\tau_2/a]} \quad \text{Ty\_AppIrrel}$$

Use the induction hypothesis on τ_1, giving us several cases:

Case $\tau_1 = v$**:** We now use Lemma C.75 to give us two cases, which are handled like the Ty_AppRel case, but using S_BetaIrrel in place of S_BetaRel.

Case $\tau_1 = v \rhd \gamma$**:** As in Ty_AppRel, but using S_PushIrrel.

Case $\Sigma; \Gamma \vdash_{\mathsf{s}} \tau_1 \longrightarrow \tau_1'$**:** By S_AppIrrel_Cong.

Case Ty_CApp: Like previous application cases, but using S_CBeta, S_CPush, and S_CApp_Cong. (The S_CPush rule looks a bit different than S_PushRel, but the typing premise of that rule has the identical structure as the previous case.)

Case Ty_Pi: Immediate, as all Π-types are values.

Case Ty_Cast: In this case, we know $\tau = \tau_0 \rhd \gamma$. Using the induction hypothesis on τ_0 gives us several cases:

Case $\tau_0 = v$**:** $v \rhd \gamma$ is a coerced value and so we are done.

Case $\tau_0 = v \rhd \eta$**:** We have $\tau = (v \rhd \eta) \rhd \gamma$. We are done by S_Trans.

Case $\Sigma; \Gamma \vdash_{\mathsf{s}} \tau_0 \longrightarrow \tau_0'$**:** We are done by Cast_Cong.

Case Ty_Case: We know here that $\tau = \mathbf{case}_\kappa\, \tau_0\, \mathbf{of}\, \overline{alt}$. Using the induction hypothesis on τ_0 gives us several cases:

Case $\tau_0 = v$**:** We can derive the following:

- $\Sigma; \Gamma \vdash_{\mathsf{ty}} v : \Pi\Delta.\, H'\, \overline{\sigma}$ (from a premise of Ty_Case)
- $v = \tau_0 = H_{\{\overline{\tau}\}}\, \overline{\psi}$ (by Lemma C.75). Note that it does not matter whether $|\Delta| = 0$ when using Lemma C.75.
- $\Sigma; \Gamma \vdash_{\mathsf{ty}} H_{\{\overline{\tau}\}}\, \overline{\psi} : \Pi\Delta'.\, H''\, \overline{\tau}$ (Lemma C.42)
- $\Sigma \vdash_{\mathsf{tc}} H : \Delta_1; \Delta_2; H''$ (same invocation of Lemma C.42)
- $\Delta' = \Delta$, $H' = H''$, and $\overline{\tau} = \overline{\sigma}$ (Lemma C.20)

Since we have $\Sigma \vdash_{\mathsf{tc}} H : \Delta_1; \Delta_2; H'$ and $\overline{alt}$ are exhaustive and distinct for H', (w.r.t. Σ), we can conclude that either there exists $H \to \tau_1 \in \overline{alt}$ or there exists $\_ \to \tau_1 \in \overline{alt}$. In the former case, we use S_Match and we are done; in the latter case, we use S_Default.

Case $\tau_0 = v \rhd \gamma$**:** We can derive the following:

- $\Sigma; \Gamma \vdash_{\mathsf{ty}} v \rhd \gamma : \text{'}\Pi\Delta.\, H'\,\overline{\sigma}$ (from a premise of Ty_Case)
- $\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{co}} \gamma : \kappa_0 \sim \text{'}\Pi\Delta.\, H'\,\overline{\sigma}$ (inversion of Ty_Cast)
- $\Sigma; \Gamma \vdash_{\mathsf{ty}} v : \kappa_0$ (same inversion)
- $\kappa_0 \propto \text{'}\Pi\Delta.\, H'\,\overline{\sigma}$ (Lemma C.74)
- κ_0 is a value (Lemma C.76)
- $\kappa_0 = \text{'}\Pi\delta_1.\, \kappa_1$ (inversion on $\kappa_0 \propto \text{'}\Pi\Delta.\, H'\,\overline{\sigma}$)
- $v = H_{\{\overline{\tau}\}}\,\overline{\psi}$ (Lemma C.75)
- $\Sigma \vdash_{\mathsf{tc}} H : \overline{a}{:}_{\mathsf{Irrel}}\overline{\kappa}; \Delta_2; H''$ (Lemma C.42)
- $\Sigma; \Gamma \vdash_{\mathsf{ty}} H_{\{\overline{\tau}\}}\,\overline{\psi} : \text{'}\Pi(\Delta_4[\overline{\psi}/\mathsf{dom}(\Delta_3)]).\, H''\,\overline{\tau}$ where $\Delta_3, \Delta_4 = \Delta_2[\overline{\tau}/\overline{a}]$ (same invocation of Lemma C.42)
- $\kappa_0 = \text{'}\Pi(\Delta_4[\overline{\psi}/\mathsf{dom}(\Delta_3)]).\, H''\,\overline{\tau}$ (Lemma C.20)
- $H'' = H'$ and $|\Delta| = |\Delta_4|$ (repeated inversion on $\kappa_0 \propto \text{'}\Pi\Delta.\, H'\,\overline{\sigma}$)

There are now two possibilities: either $H \to \sigma_0 \in \overline{alt}$ or there is a default case that matches. In the latter case, we are done by S_DefaultCo. We thus assume the former.

- $\Sigma; \Gamma; \text{'}\Pi\Delta.\, H'\,\overline{\sigma} \vdash^{v \rhd \gamma}_{\mathsf{alt}} H \to \sigma_0 : \kappa$ (a premise of Ty_Case)
- From the premises of Alt_Match:
  - $\Delta_0, \Delta_1 = \Delta_2[\overline{\sigma}/\overline{a}]$
  - $\mathsf{dom}(\Delta_1) = \mathsf{dom}(\Delta)$
  - $\mathsf{match}_{\{\mathsf{dom}(\Delta_0)\}}(\mathsf{types}(\Delta_1); \mathsf{types}(\Delta)) = \mathsf{Just}\,(\overline{\psi}'/\mathsf{dom}(\Delta_0))$ (also using Property C.13)
- $|\Delta_1| = |\Delta|$ (from the fact that their domains are the same)
- $|\Delta_1| = |\Delta_4|$ (transitivity of $=$)
- $\mathsf{dom}(\Delta_0) = \mathsf{dom}(\Delta_3)$ (from the definitions of Δ_0, Δ_1, Δ_3, and Δ_4 and the fact that $|\Delta_1| = |\Delta_4|$)
- Let $n = |\Delta_1|$ and Δ_5 be the suffix of Δ_2 of length n.
- $\Delta = \Delta_5[\overline{\sigma}/\overline{a}][\overline{\psi}'/\mathsf{dom}(\Delta_0)]$ (Property C.14)
- $\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{co}} \gamma : \text{'}\Pi(\Delta_5[\overline{\tau}/\overline{a}][\overline{\psi}/\mathsf{dom}(\Delta_0)]).\, H'\,\overline{\tau} \sim \text{'}\Pi(\Delta_5[\overline{\sigma}/\overline{a}][\overline{\psi}'/\mathsf{dom}(\Delta_0)]).\, H'\,\overline{\sigma}$ (substitution in the kind of γ as stated above)
- $\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{ty}} H'\,\overline{\sigma} : \mathbf{Type}$ (premise of Ty_Case)
- $\Sigma \vdash_{\mathsf{tc}} H' : \varnothing; \overline{a}{:}_{\mathsf{Rel}}\overline{\kappa}; \mathbf{Type}$ (Lemma C.77)
- $\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{vec}} \overline{\sigma} : \overline{a}{:}_{\mathsf{Rel}}\overline{\kappa}$ (Lemma C.42 with Lemma C.18)

We have now proved the premises of S_KPush and so stepping is possible. We are done with this case.

Case $\Sigma; \Gamma \vdash_{\mathsf{s}} \tau_0 \longrightarrow \tau_0'$**:** We are done by S_Case_Cong.

Case Ty_Lam: We know that $\tau = \lambda\delta.\, \tau_0$. If δ is anything but an irrelevant-type-variable binder, we are done. So we assume that we have $\tau = \lambda a{:}_{\mathsf{Irrel}}\kappa_0.\, \tau_0$. Using the induction hypothesis on τ_0 gives us several cases:

Case $\tau_0 = v$**:** We are done, as $\lambda a{:}_{\mathsf{Irrel}}\kappa_0.\, v$ is a value.

Case $\tau_0 = v \rhd \gamma$**:** We are done by S_APush.

Case $\Sigma; \Gamma, a{:}_{\mathsf{Irrel}}\kappa_0 \vdash_{\mathsf{s}} \tau_0 \longrightarrow \tau_0'$**:** We are done by S_IrrelAbs_Cong.

Case Ty_Fix: We know that $\tau = \mathbf{fix}\,\tau_0$. Using the induction hypothesis on τ_0 gives us several cases:

Case $\tau_0 = v$**:** We know $\Sigma; \Gamma \vdash_{\mathsf{ty}} v : \tilde{\Pi} a{:}_{\mathsf{Rel}}\kappa.\, \kappa$. Lemma C.75 tells us $v = \lambda a{:}_{\mathsf{Rel}}\kappa.\, \sigma_0$ and we are done by S_Unroll.

Case $\tau_0 = v \rhd \gamma$**:** We can derive the following facts:

- $\Sigma; \Gamma \vdash_{\mathsf{ty}} v \rhd \gamma : \tilde{\Pi} a{:}_{\mathsf{Rel}}\kappa.\, \kappa$ (premise of Ty_Fix)
- $\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{co}} \gamma : \kappa_0 \sim \tilde{\Pi} a{:}_{\mathsf{Rel}}\kappa.\, \kappa$ (inversion on Ty_Cast)
- $\Sigma; \Gamma \vdash_{\mathsf{ty}} v : \kappa_0$ (same inversion)
- $\kappa_0 \propto \tilde{\Pi} a{:}_{\mathsf{Rel}}\kappa.\, \kappa$ (Lemma C.74)
- κ_0 is a value (Lemma C.76)
- $\kappa_0 = \tilde{\Pi} a{:}_{\mathsf{Rel}}\kappa_1.\, \kappa_2$ (inversion on C_PiTy)
- $v = \lambda a{:}_{\mathsf{Rel}}\kappa_1.\, \sigma$ (Lemma C.75)

We are done by S_FPush.

Case $\Sigma; \Gamma \vdash_{\mathsf{s}} \tau_0 \longrightarrow \tau_0'$**:** We are done by S_Fix_Cong.

Case Ty_Absurd: We know here that $\tau = \mathbf{absurd}\,\gamma\,\tau_0$ where $\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{co}} \gamma : H_{1\{\overline{\tau}_1\}}\,\overline{\psi}_1 \sim H_{2\{\overline{\tau}_2\}}\,\overline{\psi}_2$. By Lemma C.74, we also know that $H_{1\{\overline{\tau}_1\}}\,\overline{\psi}_1 \propto H_{2\{\overline{\tau}_2\}}\,\overline{\psi}_2$. Both of these types are values, so this could only be by C_TyCon, but that rule requires $H_1 = H_2$, which is a contradiction. This case cannot happen.

□

C.12 Type erasure

The type erasure operation $e = \llbracket \tau \rrbracket$ is defined in Figure 5.19 on page 131.

Definition C.79 (Expression values)**.** *Let values w be defined by the following subgrammar of e:*

$$w ::= H\,\overline{y} \mid \Pi \mid \lambda a.e \mid \lambda \bullet.e$$

Lemma C.80 (Expression substitution)**.** $\llbracket \tau[\sigma/a] \rrbracket = \llbracket \tau \rrbracket[\llbracket \sigma \rrbracket/a]$

Proof. By induction on the structure of τ. □

Lemma C.81 (Irrelevant expression substitution)**.** *If $\Sigma; \Gamma \vdash_{\mathsf{ty}} \tau : \kappa$ and $a{:}_{\mathsf{Irrel}}\kappa' \in \Gamma$, then $\llbracket \tau[\sigma/a] \rrbracket = \llbracket \tau \rrbracket$.*

Proof. By induction on the typing derivation.

Case Ty_Var: Here is the rule:

$$\frac{\Sigma \vdash_{\mathsf{ctx}} \Gamma\ \mathsf{ok} \qquad a{:}_{\mathsf{Rel}}\kappa \in \Gamma}{\Sigma; \Gamma \vdash_{\mathsf{ty}} a : \kappa}\ \textsc{Ty\_Var}$$

We see that $\tau \neq a$, because the rule would require a to be relevant. Thus $\tau = b$ (for some $b \neq a$) and thus the substitution causes no change.

Case Ty_Con: Immediate from the definition of $[\![\cdot]\!]$.

Case Ty_AppRel: By induction.

Case Ty_AppIrrel: By induction. Note that we do not need to use the induction hypothesis on the argument; we would not be able to because of the use of the $\mathsf{Rel}(\Gamma)$ context.

Case Ty_CApp: By induction, not looking at the coercion.

Case Ty_Pi: Immediate from the definition of $[\![\cdot]\!]$.

Case Ty_Cast: By induction, not looking at the coercion.

Case Ty_Case: By induction, not looking at the kind.

Case Ty_Lam: By induction, not looking at the classifier of the binder.

Case Ty_Fix: By induction.

Case Ty_Absurd: Immediate from the definition of $[\![\cdot]\!]$.

□

Lemma C.82 (Expression substitution of coercions)**.** $[\![\tau[\gamma/c]]\!] = [\![\tau]\!]$

Proof. By induction on the structure of τ. □

Theorem C.83 (Type erasure)**.** *If* $\Sigma; \Gamma \vdash_{\mathsf{s}} \tau \longrightarrow \tau'$*, then either* $[\![\tau]\!] \longrightarrow [\![\tau']\!]$ *or* $[\![\tau]\!] = [\![\tau']\!]$.

Proof. By induction on $\Sigma; \Gamma \vdash_{\mathsf{s}} \tau \longrightarrow \tau'$.

Case S_BetaRel: By E_Beta and Lemma C.80.

Case S_BetaIrrel: Both expressions are equal by Lemma C.81.

Case S_CBeta: By E_CBeta and Lemma C.82.

Case S_Match: By E_Match.

Case S_Default: By E_Default.

Case S_DefaultCo: By E_Default.

Case S_Unroll: By E_Unroll.

Case S_Trans: Both expressions are equal by the definition of $[\![\cdot]\!]$.

Case S_IrrelAbs_Cong: By the induction hypothesis.

Case S_App_Cong: By the induction hypothesis and E_App_Cong.

Case S_Cast_Cong: By the induction hypothesis.

Case S_Case_Cong: By the induction hypothesis and E_Case_Cong.

Case S_Fix_Cong: By the induction hypothesis and E_Fix_Cong.

Push rules: Both expressions are equal by the definition of $[\![\cdot]\!]$.

□

Lemma C.84 (Expression redexes)**.** *If $[\![\tau]\!]$ is not an expression value, then τ is neither a value nor a coerced value.*

Proof. By induction on the structure of τ.

Case $\tau = a$: Immediate.

Case $\tau = H_{\{\overline{\tau}\}}$: Impossible.

Case $\tau = \tau_0\,\psi_0$: We have two cases here:

> **Case $\tau_1 = H_{\{\overline{\tau}\}}\,\overline{\psi}$:** Impossible, as $[\![\tau]\!]$ is an expression value.
>
> **Otherwise:** Immediate, as τ is neither a value nor a coerced value.

Case $\tau = \Pi\delta.\,\tau_0$: Impossible.

Case $\tau = \tau_0 \rhd \gamma$: Since $[\![\tau_0 \rhd \gamma]\!]$ is not an expression value, we know that $[\![\tau_0]\!]$ is not an expression value, because these expressions are the same. We thus use the induction hypothesis to discover that τ_0 is not a value or a coerced value. We thus know that $\tau_0 \rhd \gamma$ is not a coerced value (and is obviously not a value).

Case $\tau = \mathbf{case}_\kappa\,\tau_0\,\mathbf{of}\,\overline{alt}$: Immediate.

Case $\tau = \lambda a{:}_{\mathsf{Rel}}\kappa_0.\,\tau_0$: Impossible.

Case $\tau = \lambda a{:}_{\mathsf{Irrel}}\kappa_0.\,\tau_0$: We have two cases:

> **Case $[\![\tau_0]\!]$ is an expression value:** In this case $[\![\lambda a{:}_{\mathsf{Irrel}}\kappa_0.\,\tau_0]\!]$ is also an expression value, a contradiction.
>
> **Case $[\![\tau_0]\!]$ is not an expression value:** By induction, τ_0 is neither a value nor a coerced value. Thus, $\tau = \lambda a{:}_{\mathsf{Irrel}}\kappa_0.\,\tau_0$ must also not be a value. (It is clearly not a coerced value.)

Case $\tau = \lambda c{:}\phi.\,\tau_0$: Impossible.

Case $\tau = \mathbf{fix}\,\tau_0$: Immediate.

Case $\tau = \mathbf{absurd}\,\gamma\,\sigma$: Impossible.

□

Lemma C.85 (Expression values do not step)**.** *There is no e' such that $w \longrightarrow e'$.*

Proof. Straightforward case analysis on w. □

Theorem C.86 (Types do not prevent evaluation)**.** *Suppose $\Sigma; \Gamma \vdash_{\mathsf{ty}} \tau : \kappa$ and Γ has only irrelevant variable bindings. If $[\![\tau]\!] \longrightarrow e'$, then $\Sigma; \Gamma \vdash_{\mathsf{s}} \tau \longrightarrow \tau'$ and either $[\![\tau']\!] = e'$ or $[\![\tau']\!] = [\![\tau]\!]$.*

Proof. We know that $[\![\tau]\!]$ is not an expression value via the contrapositive of Lemma C.85. We thus know that τ is neither a value nor a coerced value by Lemma C.84. We can now use Theorem C.78 to get τ' such that $\Sigma; \Gamma \vdash_{\mathsf{s}} \tau \longrightarrow \tau'$. Finally, we use Theorem C.83 to see that $[\![\tau']\!] = e'$ or $[\![\tau']\!] = [\![\tau]\!]$ as desired. □

Remark. Note in the statement of Theorem C.86 that the context Γ must have only irrelevant variable bindings. This means that the expression $[\![\tau]\!]$ is closed, as one would expect of a program that we wish to evaluate.

C.13 Congruence

Definition C.87 (Unrestricted coercion variables)**.** *Define a new judgment $\vdash^*_{\mathsf{co}}$ to be identical to $\vdash_{\mathsf{co}}$, except with the $c \mathrel{\tilde{\#}} \gamma$ premises removed from rules* Co_PiCo *and* Co_CLam *and all recursive uses of $\vdash_{\mathsf{co}}$ replaced with $\vdash^*_{\mathsf{co}}$.*

Remark. It is not necessary to introduce a $\vdash^*_{\mathsf{ty}}$ judgment that uses $\vdash^*_{\mathsf{co}}$. Thus, for example, the Co_Refl rule of $\vdash^*_{\mathsf{co}}$ has a $\vdash_{\mathsf{ty}}$ premise that may contain proofs of $\vdash_{\mathsf{co}}$.

Lemma C.88 (Subsumption of coercion typing)**.** *If $\Sigma; \Gamma \vdash_{\mathsf{co}} \gamma : \phi$, then $\Sigma; \Gamma \vdash^*_{\mathsf{co}} \gamma : \phi$.*

Proof. Straightforward induction. □

Lemma C.89 (Unrestricted proposition regularity)**.** *If $\Sigma; \Gamma \vdash^*_{\mathsf{co}} \gamma : \phi$, then $\Sigma; \Gamma \vdash_{\mathsf{prop}} \phi$ ok.*

Proof. Identical to the proof for Lemma C.44. □

Theorem C.90 ((Almost) Congruence)**.** *If $\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{co}} \gamma : \sigma_1 \mathrel{^\kappa\!\sim^\kappa} \sigma_2$ and $\Sigma; \Gamma, a{:}_\rho\kappa, \Gamma' \vdash_{\mathsf{ty}} \tau : \kappa_0$ where none of τ, κ_0, κ and the types in Γ and Γ' bind any coercion variables, then there exists η such that $\Sigma; \mathsf{Rel}(\Gamma, \Gamma'[\sigma_1/a]) \vdash^*_{\mathsf{co}} \eta : \tau[\sigma_1/a] \mathrel{^{\kappa_0[\sigma_1/a]}\!\sim^{\kappa_0[\sigma_2/a]}} \tau[\sigma_2/a]$.*

Proof. By induction on the size of the derivation of $\Sigma; \Gamma, a{:}_\rho\kappa, \Gamma' \vdash_{\mathsf{ty}} \tau : \kappa_0$, using Lemma C.88 frequently to convert between the coercion typing relations.

Case Ty_Var: Here $\tau = b$. We have several cases:

> **Case** $b \in \mathsf{dom}(\Gamma)$**:** By Lemma C.12, $a \# \kappa_0$. We are done, choosing $\eta = \langle b \rangle$.

Case $b = a$**:** By Lemma C.12, $a \# \kappa_0$. We are done, choosing $\eta = \gamma$.

Case $b \in \mathsf{dom}(\Gamma')$**:** We know $\Gamma' = \Gamma_1, b{:}_{\mathsf{Rel}}\kappa_0, \Gamma_2$. Lemma C.9 and Lemma C.7 give us $\Sigma; \mathsf{Rel}(\Gamma, a{:}_\rho\kappa, \Gamma_1) \vdash_{\mathsf{ty}} \kappa_0 : \mathbf{Type}$ with a derivation smaller than that with which we started. Use the induction hypothesis to get $\Sigma; \mathsf{Rel}(\Gamma, \Gamma_1[\sigma_1/a]) \vdash^*_{\mathsf{co}} \eta_0 : \kappa_0[\sigma_1/a] \sim \kappa_0[\sigma_2/a]$. Choose $\eta = b \approx_{\eta_0} b \rhd \eta_0$ and we are done.

Case Ty_Con**:** By Lemma C.29, repeated use of the induction hypothesis, Lemma C.35, and Co_Con.

Case Ty_AppRel**:** By the induction hypothesis, Lemma C.35, and Co_AppRel.

Case Ty_AppIrrel**:** By the induction hypothesis, Lemma C.35, and Co_AppIrrel.

Case Ty_CApp**:** We adopt the metavariable names from the rule (changing the name of the coercion used to γ'):

$$\frac{\Sigma; \Gamma \vdash_{\mathsf{ty}} \tau : \Pi c{:}\phi.\, \kappa \qquad \Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{co}} \gamma : \phi}{\Sigma; \Gamma \vdash_{\mathsf{ty}} \tau\, \gamma : \kappa[\gamma/c]}\ \text{Ty\_CApp}$$

The induction hypothesis gives us η_1 such that $\Sigma; \mathsf{Rel}(\Gamma, \Gamma'[\sigma_1/a]) \vdash^*_{\mathsf{co}} \eta_1 : \tau[\sigma_1/a] \sim \tau[\sigma_2/a]$. Choose $\eta = \eta_1\,(\gamma'[\sigma_1/a], \gamma'[\sigma_2/a])$. We are done by Lemma C.35 and Co_CApp.

Case Ty_Pi**:** We have several cases, depending on the shape of the binder:

Type variable binder: In this case, we know that $\tau = \Pi b{:}_{\rho'}\kappa_1.\, \tau_0$ and $\kappa_0 = \mathbf{Type}$. The induction hypothesis gives us η_1 such that $\Sigma; \mathsf{Rel}(\Gamma, \Gamma'[\sigma_1/a]), b{:}_{\mathsf{Rel}}\kappa_1[\sigma_1/a] \vdash^*_{\mathsf{co}} \eta_1 : \tau_0[\sigma_1/a] \sim \tau_0[\sigma_2/a]$. We can also use Lemma C.9 and Lemma C.7 to see that $\Sigma; \mathsf{Rel}(\Gamma, a{:}_\rho\kappa, \Gamma') \vdash_{\mathsf{ty}} \kappa_1 : \mathbf{Type}$, with a smaller derivation height than $\Sigma; \Gamma, a{:}_\rho\kappa, \Gamma' \vdash_{\mathsf{ty}} \Pi b{:}_{\rho'}\kappa_1.\, \tau_0 : \mathbf{Type}$. We can thus use the induction hypothesis again to get η_2 such that $\Sigma; \mathsf{Rel}(\Gamma, \Gamma'[\sigma_1/a]) \vdash^*_{\mathsf{co}} \eta_2 : \kappa_1[\sigma_1/a] \sim \kappa_1[\sigma_2/a]$. Choose $\eta = (\Pi b{:}_{\rho'}\eta_2.\, \eta_1) \fatsemi \eta_3$, where

- $\eta_3 = \sigma_3 \approx_{\langle \mathbf{Type} \rangle} \sigma_4$
- $\sigma_3 = \Pi b{:}_{\rho'}\kappa_1[\sigma_2/a].\, (\tau_0[\sigma_2/a][b \rhd \mathbf{sym}\, \eta_2/b])$
- $\sigma_4 = \Pi b{:}_{\rho'}\kappa_1[\sigma_2/a].\, \tau_0[\sigma_2/a]$

We must show $\Sigma; \mathsf{Rel}(\Gamma, \Gamma'[\sigma_1/a]) \vdash^*_{\mathsf{co}} \eta : (\Pi b{:}_{\rho'}\kappa_1.\, \tau_0)[\sigma_1/a] \sim (\Pi b{:}_{\rho'}\kappa_1.\, \tau_0)[\sigma_2/a]$. We will do this by proving both of these:

$\Sigma; \mathsf{Rel}(\Gamma, \Gamma'[\sigma_1/a]) \vdash^*_{\mathsf{co}} \Pi b{:}_{\rho'}\eta_2.\, \eta_1 : (\Pi b{:}_{\rho'}\kappa_1[\sigma_1/a].\, \tau_0[\sigma_1/a]) \sim \sigma_3$ This is straightforward from Co_PiTy.

$\Sigma; \mathsf{Rel}(\Gamma, \Gamma'[\sigma_1/a]) \vdash^*_{\mathsf{co}} \sigma_3 \approx_{\langle \mathbf{Type} \rangle} \sigma_4 : \sigma_3 \sim \sigma_4$ We must prove that both the left-hand type and right-hand type have kind **Type**. The left-hand result comes from Lemma C.89 on the result of the previous branch

of this list of things to prove. The right-hand result comes from Lemma C.44 on our assumption about γ and Lemma C.35 (using Lemma C.6 in the $\rho = \mathsf{Irrel}$ case). Now we must prove that the erasure of the two types equal, which boils down to proving $\lfloor \tau_0[\sigma_2/a][b \rhd \mathbf{sym}\,\eta_2/b] \rfloor = \lfloor \tau_0[\sigma_2/a] \rfloor$. By Lemma C.34, the LHS becomes $\lfloor \tau_0[\sigma_2/a] \rfloor [\lfloor b \rhd \mathbf{sym}\,\eta_2 \rfloor / b]$. We can see that $\lfloor b \rhd \mathbf{sym}\,\eta_2 \rfloor = b$ and thus the two sides of the equation are equal.

Coercion variable binder: In this case, we know that $\tau = \Pi c{:}\phi.\,\tau_0$ and $\kappa_0 = \mathbf{Type}$. The induction hypothesis gives us η_1 such that $\Sigma; \mathsf{Rel}(\Gamma, \Gamma'[\sigma_1/a]), c{:}\phi[\sigma_1/a] \vdash^*_{\mathsf{co}} \eta_1 : \tau_0[\sigma_1/a] \sim \tau_0[\sigma_2/a]$. Let $\phi = \kappa_1 \;{}^{\kappa_1'}\!\sim^{\kappa_2'}\; \kappa_2$. We can also use Lemma C.9, Lemma C.8, and inversion on Prop_Equality to see that $\Sigma; \mathsf{Rel}(\Gamma, a{:}_\rho \kappa, \Gamma') \vdash_{\mathsf{ty}} \kappa_1 : \kappa_1'$ and $\Sigma; \mathsf{Rel}(\Gamma, a{:}_\rho \kappa, \Gamma') \vdash_{\mathsf{ty}} \kappa_2 : \kappa_2'$, both with a smaller derivation height than $\Sigma; \Gamma, a{:}_\rho \kappa, \Gamma' \vdash_{\mathsf{ty}} \Pi c{:}\phi.\,\tau_0 : \mathbf{Type}$. We can thus use the induction hypothesis again to get η_2 and η_3 such that $\Sigma; \mathsf{Rel}(\Gamma, \Gamma'[\sigma_1/a]) \vdash^*_{\mathsf{co}} \eta_2 : \kappa_1[\sigma_1/a] \sim \kappa_1[\sigma_2/a]$ and $\Sigma; \mathsf{Rel}(\Gamma, \Gamma'[\sigma_1/a]) \vdash^*_{\mathsf{co}} \eta_3 : \kappa_2[\sigma_1/a] \sim \kappa_2[\sigma_2/a]$. Choose $\eta = (\Pi c{:}(\eta_2, \eta_3).\,\eta_1) \fatsemi \eta_4$, where

- $\eta_4 = \sigma_3 \approx_{\langle \mathbf{Type} \rangle} \sigma_4$
- $\sigma_3 = \Pi c{:}\phi[\sigma_2/a].\,(\tau_0[\sigma_2/a][\eta_5/c])$
- $\sigma_4 = \Pi c{:}\phi[\sigma_2/a].\,\tau_0[\sigma_2/a]$
- $\eta_5 = \eta_2 \fatsemi c \fatsemi \mathbf{sym}\,\eta_3$

We must show $\Sigma; \mathsf{Rel}(\Gamma, \Gamma'[\sigma_1/a]) \vdash^*_{\mathsf{co}} \eta : (\Pi c{:}\phi.\,\tau_0)[\sigma_1/a] \sim (\Pi c{:}\phi.\,\tau_0)[\sigma_2/a]$. We will do this by proving both of these:

$\Sigma; \mathsf{Rel}(\Gamma, \Gamma'[\sigma_1/a]) \vdash^*_{\mathsf{co}} \Pi c{:}(\eta_2, \eta_3).\,\eta_1 : (\Pi c{:}\phi[\sigma_1/a].\,\tau_0[\sigma_1/a]) \sim \sigma_3$ This is straightforward from Co_PiCo. Note that we cannot guarantee the $c \,\tilde{\#}\, \eta_1$ condition here, necessitating the use of $\vdash^*_{\mathsf{co}}$ instead of $\vdash_{\mathsf{co}}$.

$\Sigma; \mathsf{Rel}(\Gamma, \Gamma'[\sigma_1/a]) \vdash^*_{\mathsf{co}} \sigma_3 \approx_{\langle \mathbf{Type} \rangle} \sigma_4 : \sigma_3 \sim \sigma_4$ We must prove that both the left-hand type and right-hand type have kind **Type**. The left-hand result comes from Lemma C.89 on the result of the previous branch of this list of things to prove. The right-hand result comes from Lemma C.44 on our assumption about γ and Lemma C.35 (using Lemma C.6 in the $\rho = \mathsf{Irrel}$ case). Now we must prove that the erasure of the two types equal, which boils down to proving $\lfloor \tau_0[\sigma_2/a][\eta_5/c] \rfloor = \lfloor \tau_0[\sigma_2/a] \rfloor$. This holds by Lemma C.59.

Case Ty_Cast: We adopt the metavariable names from the rule (but renaming the coercion used in the cast to γ'):

$$\frac{\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{co}} \gamma : \kappa_1 \sim \kappa_2 \quad \Sigma; \Gamma \vdash_{\mathsf{ty}} \tau : \kappa_1 \qquad \Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{ty}} \kappa_2 : \mathbf{Type}}{\Sigma; \Gamma \vdash_{\mathsf{ty}} \tau \rhd \gamma : \kappa_2} \;\text{Ty\_Cast}$$

The induction hypothesis gives us η_1 such that $\Sigma; \mathsf{Rel}(\Gamma, \Gamma'[\sigma_1/a]) \vdash^*_{\mathsf{co}} \eta_1 : \tau[\sigma_1/a] \ {}^{\kappa_1[\sigma_1/a]}\!\sim^{\kappa_1[\sigma_2/a]} \tau[\sigma_2/a]$. Let $\eta_2 = ((\tau[\sigma_1/a] \rhd \gamma'[\sigma_1/a]) \approx_{\mathbf{sym}\,\gamma'[\sigma_1/a]} \tau[\sigma_1/a])$ and $\eta_3 = (\tau[\sigma_2/a] \approx_{\gamma'[\sigma_2/a]} (\tau[\sigma_2/a] \rhd \gamma'[\sigma_2/a]))$. It is easy to see (using Lemma C.89 and Lemma C.35) that η_2 and η_3 are well-typed. Choose $\eta = \eta_2 \fatsemi \eta_1 \fatsemi \eta_3$, and we are done.

Case TY_CASE: By repeated use of the induction hypothesis, Lemma C.35, and CO_CASE.

Case TY_LAM: Like the case for TY_PI.

Case TY_FIX: By the induction hypothesis, Lemma C.35, and CO_FIX.

Case TY_ABSURD: We adopt the metavariable names from the rule (but renaming the coercion used to γ'):

$$\frac{\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{co}} \gamma : H_{1\{\overline{\tau}_1\}}\,\overline{\psi}_1 \sim H_{2\{\overline{\tau}_2\}}\,\overline{\psi}_2 \qquad H_1 \neq H_2 \qquad \Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{ty}} \tau : \mathbf{Type}}{\Sigma; \Gamma \vdash_{\mathsf{ty}} \mathbf{absurd}\,\gamma\,\tau : \tau}\ \text{TY\_ABSURD}$$

The induction hypothesis gives us η_1 such that $\Sigma; \mathsf{Rel}(\Gamma, \Gamma'[\sigma_1/a]) \vdash^*_{\mathsf{co}} \eta_1 : \tau[\sigma_1/a] \sim \tau[\sigma_2/a]$. Choose $\eta = \mathbf{absurd}\,(\gamma'[\sigma_1/a], \gamma'[\sigma_2/a])\,\eta_1$. We know $\gamma'[\sigma_i/a]$ (for $i \in \{1, 2\}$) is well-typed by Lemma C.35. We are thus done.

□

Appendix D

Type inference rules, in full

D.1 Closing substitution validity

$\boxed{\Sigma; \Gamma \vdash_{\mathsf{subst}} \theta : \Delta}$ "θ substitutes the variables in Δ away."

$$\frac{}{\Sigma; \Gamma \vdash_{\mathsf{subst}} \theta : \varnothing} \textsc{Subst\_Nil}$$

$$\frac{\Sigma; \Gamma \vdash_{\mathsf{ty}} a[\theta] : \kappa \quad \Sigma; \Gamma \vdash_{\mathsf{subst}} \theta : \Delta[\theta|_a]}{\Sigma; \Gamma \vdash_{\mathsf{subst}} \theta : a{:}_{\mathsf{Rel}}\kappa, \Delta} \textsc{Subst\_TyRel}$$

$$\frac{\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{ty}} a[\theta] : \kappa \quad \Sigma; \Gamma \vdash_{\mathsf{subst}} \theta : \Delta[\theta|_a]}{\Sigma; \Gamma \vdash_{\mathsf{subst}} \theta : a{:}_{\mathsf{Irrel}}\kappa, \Delta} \textsc{Subst\_TyIrrel}$$

$$\frac{\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{co}} c[\theta] : \phi \quad \Sigma; \Gamma \vdash_{\mathsf{subst}} \theta : \Delta[\theta|_c]}{\Sigma; \Gamma \vdash_{\mathsf{subst}} \theta : c{:}\phi, \Delta} \textsc{Subst\_Co}$$

D.2 Additions to Pico judgments

$\boxed{\Sigma; \Psi \models_{\mathsf{ty}} \tau : \kappa}$ Extra rule to support unification variables in types

$$\frac{\alpha :_{\mathsf{Rel}} \forall\, \Delta.\kappa \in \Psi \qquad \Sigma \models_{\mathsf{ctx}} \Psi\ \mathsf{ok} \quad \Sigma; \Psi \models_{\mathsf{vec}} \overline{\psi} : \Delta}{\Sigma; \Psi \models_{\mathsf{ty}} \alpha_{\overline{\psi}} : \kappa[\overline{\psi}/\mathsf{dom}(\Delta)]} \textsc{Ty\_UVar}$$

$\boxed{\Sigma; \Psi \vDash_{\mathsf{co}} \gamma : \phi}$ Extra rule to support unification variables in coercions

$$\frac{\iota : \forall\, \Delta.\phi \in \Psi \qquad \Sigma \vDash_{\mathsf{ctx}} \Psi\ \mathsf{ok} \qquad \Sigma; \Psi \vDash_{\mathsf{vec}} \overline{\psi} : \Delta}{\Sigma; \Psi \vDash_{\mathsf{co}} \iota_{\overline{\psi}} : \phi[\overline{\psi}/\mathsf{dom}(\Delta)]}\ \textsc{Co\_UVar}$$

$\boxed{\Sigma \vDash_{\mathsf{ctx}} \Psi\ \mathsf{ok}}$ Extra rules to support binding unification variables

$$\frac{\Sigma; \mathsf{Rel}(\Psi, \Delta) \vDash_{\mathsf{ty}} \kappa : \mathbf{Type} \qquad \Sigma \vDash_{\mathsf{ctx}} \Psi\ \mathsf{ok}}{\Sigma \vDash_{\mathsf{ctx}} \Psi, \alpha :_{\rho} \forall\, \Delta.\kappa\ \mathsf{ok}}\ \textsc{Ctx\_UTyVar}$$

$$\frac{\Sigma; \mathsf{Rel}(\Psi, \Delta) \vDash_{\mathsf{prop}} \phi\ \mathsf{ok} \qquad \Sigma \vDash_{\mathsf{ctx}} \Psi\ \mathsf{ok}}{\Sigma \vDash_{\mathsf{ctx}} \Psi, \iota : \forall\, \Delta.\phi\ \mathsf{ok}}\ \textsc{Ctx\_UCoVar}$$

D.3 Zonker validity

$\boxed{\Sigma; \Psi \vDash_{\mathsf{z}} \Theta : \Omega}$ "Θ zonks all the unification variables in Ω."

$$\frac{}{\Sigma; \Psi \vDash_{\mathsf{z}} \varnothing : \varnothing}\ \textsc{Zonk\_Nil}$$

$$\frac{\Sigma; \Psi, \Delta \vDash_{\mathsf{ty}} \tau : \kappa \qquad \Sigma; \Psi \vDash_{\mathsf{z}} \Theta : \Omega[\forall\, \mathsf{dom}(\Delta).\tau/\alpha]}{\Sigma; \Psi \vDash_{\mathsf{z}} \forall\, \mathsf{dom}(\Delta).\tau/\alpha, \Theta : \alpha :_{\mathsf{Rel}} \forall\, \Delta.\kappa, \Omega}\ \textsc{Zonk\_TyVarRel}$$

$$\frac{\Sigma; \mathsf{Rel}(\Psi, \Delta) \vDash_{\mathsf{ty}} \tau : \kappa \qquad \Sigma; \Psi \vDash_{\mathsf{z}} \Theta : \Omega[\forall\, \mathsf{dom}(\Delta).\tau/\alpha]}{\Sigma; \Psi \vDash_{\mathsf{z}} \forall\, \mathsf{dom}(\Delta).\tau/\alpha, \Theta : \alpha :_{\mathsf{Irrel}} \forall\, \Delta.\kappa, \Omega}\ \textsc{Zonk\_TyVarIrrel}$$

$$\frac{\Sigma; \Psi, \Delta \vDash_{\mathsf{co}} \gamma : \phi \qquad \Sigma; \Psi \vDash_{\mathsf{z}} \Theta : \Omega[\forall\, \mathsf{dom}(\Delta).\gamma/\iota]}{\Sigma; \Psi \vDash_{\mathsf{z}} \forall\, \mathsf{dom}(\Delta).\gamma/\iota, \Theta : \iota : \forall\, \Delta.\phi, \Omega}\ \textsc{Zonk\_CoVar}$$

D.4 Synthesis

$\boxed{\Sigma; \Psi \Rightarrow_{\mathsf{ty}} \mathrm{t} \rightsquigarrow \tau : \kappa \dashv \Omega}$ Synthesize a type with no invisible binders.

$$\frac{\begin{array}{l}\Sigma;\Psi \Rightarrow^{*}_{\mathsf{ty}} \mathrm{t} \rightsquigarrow \tau : \kappa \dashv \Omega_1 \\ \Rightarrow^{\mathsf{Spec}}_{\mathsf{inst}} \kappa \rightsquigarrow \overline{\psi}; \kappa' \dashv \Omega_2\end{array}}{\Sigma;\Psi \Rightarrow_{\mathsf{ty}} \mathrm{t} \rightsquigarrow \tau\,\overline{\psi} : \kappa' \dashv \Omega_1, \Omega_2} \quad \textsc{ITy\_Inst}$$

$\boxed{\Sigma;\Psi \Rightarrow^{*}_{\mathsf{ty}} \mathrm{t} \rightsquigarrow \tau : \kappa \dashv \Omega}$ Synthesize a type, perhaps with specified binders.

$$\frac{a{:}_{\mathsf{Rel}}\kappa \in \Psi \qquad \Rightarrow^{\mathsf{Inf}}_{\mathsf{inst}} \kappa \rightsquigarrow \overline{\psi}; \kappa' \dashv \Omega}{\Sigma;\Psi \Rightarrow^{*}_{\mathsf{ty}} a \rightsquigarrow a\,\overline{\psi} : \kappa' \dashv \Omega} \quad \textsc{ITy\_Var}$$

$$\frac{\begin{array}{l}\Sigma;\Psi \Rightarrow_{\mathsf{ty}} \mathrm{t}_1 \rightsquigarrow \tau_1 : \kappa_0 \dashv \Omega_1 \\ \Rightarrow_{\mathsf{fun}} \kappa_0; \mathsf{Rel} \rightsquigarrow \gamma; \Pi; a; \rho; \kappa_1; \kappa_2 \dashv \Omega_2 \\ \Sigma;\Psi,\Omega_1,\Omega_2;\rho \Rightarrow^{*}_{\mathsf{arg}} \mathrm{t}_2 : \kappa_1 \rightsquigarrow \psi_2; \tau_2 \dashv \Omega_3\end{array}}{\Sigma;\Psi \Rightarrow^{*}_{\mathsf{ty}} \mathrm{t}_1\,\mathrm{t}_2 \rightsquigarrow (\tau_1 \rhd \gamma)\,\psi_2 : \kappa_2[\tau_2/a] \dashv \Omega_1,\Omega_2,\Omega_3} \quad \textsc{ITy\_App}$$

$$\frac{\begin{array}{l}\Sigma;\Psi \Rightarrow^{*}_{\mathsf{ty}} \mathrm{t}_1 \rightsquigarrow \tau_1 : \Pi_{\mathsf{Spec}} a{:}_{\rho}\kappa_1.\,\kappa_2 \dashv \Omega_1 \\ \Sigma;\Psi,\Omega_1;\rho \Rightarrow^{*}_{\mathsf{arg}} \mathrm{t}_2 : \kappa_1 \rightsquigarrow \psi_2; \tau_2 \dashv \Omega_2\end{array}}{\Sigma;\Psi \Rightarrow^{*}_{\mathsf{ty}} \mathrm{t}_1\,@\mathrm{t}_2 \rightsquigarrow \tau_1\,\psi_2 : \kappa_2[\tau_2/a] \dashv \Omega_1,\Omega_2} \quad \textsc{ITy\_AppSpec}$$

$$\frac{\begin{array}{l}\Sigma;\mathsf{Rel}(\Psi) \Rightarrow_{\mathsf{pt}} \mathrm{s} \rightsquigarrow \sigma \dashv \Omega_1 \\ \Sigma;\Psi,\Omega_1 \Rightarrow^{*}_{\mathsf{ty}} \mathrm{t} : \sigma \rightsquigarrow \tau \dashv \Omega_2\end{array}}{\Sigma;\Psi \Rightarrow^{*}_{\mathsf{ty}} (\mathrm{t} :: \mathrm{s}) \rightsquigarrow \tau : \sigma \dashv \Omega_1,\Omega_2} \quad \textsc{ITy\_Annot}$$

$$\frac{\begin{array}{l}\Sigma;\Psi \Rightarrow_{\mathsf{ty}} \mathrm{t}_0 \rightsquigarrow \tau_0 : \kappa_0 \dashv \Omega_0 \\ \Sigma;\Psi,\Omega_0 \Rightarrow_{\mathsf{scrut}} \overline{\mathrm{alt}}; \kappa_0 \rightsquigarrow \gamma; \Delta; H'; \overline{\tau} \dashv \Omega_0' \\ \mathsf{fresh}\,\alpha \qquad \Omega' = \Omega_0, \Omega_0', \alpha{:}_{\mathsf{Irrel}}\mathbf{Type} \\ \forall i,\ \Sigma;\Psi,\Omega';'\Pi\Delta.\,H'\,\overline{\tau};\tau_0 \rhd \gamma \Rightarrow_{\mathsf{alt}} \mathrm{alt}_i : \alpha \rightsquigarrow alt_i \dashv \Omega_i \\ \overline{alt}' = \mathsf{make\_exhaustive}(\overline{alt}; \kappa)\end{array}}{\Sigma;\Psi \Rightarrow^{*}_{\mathsf{ty}} \mathbf{case}\,\mathrm{t}_0\,\mathbf{of}\,\overline{\mathrm{alt}} \rightsquigarrow \mathbf{case}_{\alpha}\,(\tau_0 \rhd \gamma)\,\mathbf{of}\,\overline{alt}' : \alpha \dashv \Omega', \overline{\Omega}} \quad \textsc{ITy\_Case}$$

$$\frac{\begin{array}{l}\Sigma;\Psi \Rightarrow_{\mathsf{q}} \mathrm{qvar} \rightsquigarrow a : \kappa_1; \nu \dashv \Omega_1 \\ \Sigma;\Psi,\Omega_1,a{:}_{\mathsf{Rel}}\kappa_1 \Rightarrow^{*}_{\mathsf{ty}} \mathrm{t} \rightsquigarrow \tau : \kappa_2 \dashv \Omega_2 \\ \Omega_2 \hookrightarrow a{:}_{\mathsf{Rel}}\kappa_1 \rightsquigarrow \Omega_2'; \xi\end{array}}{\Sigma;\Psi \Rightarrow^{*}_{\mathsf{ty}} \lambda\mathrm{qvar}.\,\mathrm{t} \rightsquigarrow \lambda a{:}_{\mathsf{Rel}}\kappa_1.\,(\tau[\xi]) : \underset{\sim}{\Pi}_{\nu} a{:}_{\mathsf{Rel}}\kappa_1.\,(\kappa_2[\xi]) \dashv \Omega_1,\Omega_2'} \quad \textsc{ITy\_Lam}$$

$$\frac{\begin{array}{l}\Sigma;\Psi \Rightarrow_{\mathsf{q}} \mathrm{qvar} \rightsquigarrow a : \kappa_1; \nu \dashv \Omega_1 \\ \Sigma;\Psi,\Omega_1,a{:}_{\mathsf{Irrel}}\kappa_1 \Rightarrow^{*}_{\mathsf{ty}} \mathrm{t} \rightsquigarrow \tau : \kappa_2 \dashv \Omega_2 \\ \Omega_2 \hookrightarrow a{:}_{\mathsf{Irrel}}\kappa_1 \rightsquigarrow \Omega_2'; \xi\end{array}}{\Sigma;\Psi \Rightarrow^{*}_{\mathsf{ty}} \Lambda\mathrm{qvar}.\,\mathrm{t} \rightsquigarrow \lambda a{:}_{\mathsf{Irrel}}\kappa_1.\,(\tau[\xi]) : \underset{\sim}{\Pi}_{\nu} a{:}_{\mathsf{Rel}}\kappa_1.\,(\kappa_2[\xi]) \dashv \Omega_1,\Omega_2'} \quad \textsc{ITy\_LamIrrel}$$

$$\frac{\begin{array}{l}\Sigma; \Psi \overset{}{\underset{\mathsf{ty}}{\mapsto}} \mathrm{t}_1 : \mathbf{Type} \rightsquigarrow \tau_1 \dashv \Omega_1 \\ \Sigma; \Psi \overset{}{\underset{\mathsf{ty}}{\mapsto}} \mathrm{t}_2 : \mathbf{Type} \rightsquigarrow \tau_2 \dashv \Omega_2 \\ a \# \tau_2\end{array}}{\Sigma; \Psi \overset{*}{\underset{\mathsf{ty}}{\mapsto}} \mathrm{t}_1 \to \mathrm{t}_2 \rightsquigarrow \underset{\sim}{\Pi}_{\mathsf{Req}}\, a{:}_{\mathsf{Rel}}\tau_1.\, \tau_2 : \mathbf{Type} \dashv \Omega_1, \Omega_2} \quad \textsc{Ity\_Arrow}$$

$$\frac{\begin{array}{l}\Sigma; \Psi \overset{}{\underset{\mathsf{ty}}{\mapsto}} \mathrm{t}_1 : \mathbf{Type} \rightsquigarrow \tau_1 \dashv \Omega_1 \\ \Sigma; \Psi \overset{}{\underset{\mathsf{ty}}{\mapsto}} \mathrm{t}_2 : \mathbf{Type} \rightsquigarrow \tau_2 \dashv \Omega_2 \\ a \# \tau_2\end{array}}{\Sigma; \Psi \overset{*}{\underset{\mathsf{ty}}{\mapsto}} \mathrm{t}_1 \,'\!\to \mathrm{t}_2 \rightsquigarrow \text{'}\underset{\sim}{\Pi}_{\mathsf{Req}}\, a{:}_{\mathsf{Rel}}\tau_1.\, \tau_2 : \mathbf{Type} \dashv \Omega_1, \Omega_2} \quad \textsc{Ity\_MArrow}$$

$$\frac{\begin{array}{l}\Sigma; \Psi \overset{}{\underset{\mathsf{ty}}{\mapsto}} \mathrm{t} \rightsquigarrow \tau : \kappa \dashv \Omega_1 \\ \overset{}{\underset{\mathsf{fun}}{\mapsto}}\, \kappa; \mathsf{Rel} \rightsquigarrow \gamma; \underset{\sim}{\Pi}; a; \mathsf{Rel}; \kappa_1; \kappa_2 \dashv \Omega_2 \\ \Sigma; \mathsf{Rel}(\Psi, \Omega_1, \Omega_2) \overset{}{\underset{\mathsf{ty}}{\mapsto}} \kappa_2 : \mathbf{Type} \\ \mathsf{fresh}\, \iota \qquad \Omega \;=\; \Omega_1, \Omega_2, \iota{:}\kappa_2 \sim \kappa_1\end{array}}{\Sigma; \Psi \overset{*}{\underset{\mathsf{ty}}{\mapsto}} \mathbf{fix}\, \mathrm{t} \rightsquigarrow \mathbf{fix}\, (\tau \rhd (\gamma \fatsemi \underset{\sim}{\Pi} a{:}_{\mathsf{Rel}}\langle \kappa_1 \rangle.\, \iota)) : \kappa_1 \dashv \Omega} \quad \textsc{Ity\_Fix}$$

$$\frac{\begin{array}{l}\Sigma; \Psi \overset{*}{\underset{\mathsf{ty}}{\mapsto}} \mathrm{t}_1 \rightsquigarrow \tau_1 : \kappa_1 \dashv \Omega \\ \Sigma; \Psi, \Omega, x{:}_{\mathsf{Rel}}\kappa_1 \overset{*}{\underset{\mathsf{ty}}{\mapsto}} \mathrm{t}_2 \rightsquigarrow \tau_2 : \kappa_2 \dashv \Omega_2 \\ \Omega_2 \hookrightarrow x{:}_{\mathsf{Rel}}\kappa_1 \rightsquigarrow \Omega_2'; \xi\end{array}}{\Sigma; \Psi \overset{*}{\underset{\mathsf{ty}}{\mapsto}} \mathbf{let}\, x := \mathrm{t}_1 \,\mathbf{in}\, \mathrm{t}_2 \rightsquigarrow (\lambda x{:}_{\mathsf{Rel}}\kappa_1.\, (\tau_2[\xi]))\, \tau_1 : \kappa_2[\xi][\tau_1/x] \dashv \Omega, \Omega_2'} \quad \textsc{Ity\_Let}$$

D.5 Checking

$\boxed{\Sigma; \Psi \overset{}{\underset{\mathsf{ty}}{\mapsto}} \mathrm{t} : \kappa \rightsquigarrow \tau \dashv \Omega}$ Check against a type with no invisible binders.

$$\frac{\begin{array}{l}\Sigma; \Psi \overset{}{\underset{\mathsf{ty}}{\mapsto}} \mathrm{t}_0 \rightsquigarrow \tau_0 : \kappa_0 \dashv \Omega_0 \\ \Sigma; \Psi, \Omega_0 \overset{}{\underset{\mathsf{scrut}}{\mapsto}} \overline{\mathrm{alt}}; \kappa_0 \rightsquigarrow \gamma; \Delta; H'; \overline{\tau} \dashv \Omega_0' \\ \Omega' \;=\; \Omega_0, \Omega_0' \\ \forall i,\ \Sigma; \Psi, \Omega'; \text{'}\Pi\Delta.\, H'\, \overline{\tau}; \tau_0 \rhd \gamma \overset{}{\underset{\mathsf{altc}}{\mapsto}} \mathrm{alt}_i : \kappa \rightsquigarrow alt_i \dashv \Omega_i \\ \overline{alt}' \;=\; \mathsf{make\_exhaustive}(\overline{alt}; \kappa)\end{array}}{\Sigma; \Psi \overset{}{\underset{\mathsf{ty}}{\mapsto}} \mathbf{case}\, \mathrm{t}_0\, \mathbf{of}\, \overline{\mathrm{alt}} : \kappa \rightsquigarrow \mathbf{case}_{\kappa}\, (\tau_0 \rhd \gamma)\, \mathbf{of}\, \overline{alt}' \dashv \Omega', \overline{\Omega}} \quad \textsc{ItyC\_Case}$$

$$\frac{\begin{array}{l}\vdash^{\to}_{\mathsf{fun}}\ \kappa; \mathsf{Rel} \rightsquigarrow \gamma; \tilde{\Pi}; a; \mathsf{Rel}; \kappa_1; \kappa_2 \dashv \Omega_0 \\ \neg(a \# \kappa_2) \\ \Sigma; \mathsf{Rel}(\Psi) \vdash^{\to}_{\mathsf{pt}} \mathrm{s} \rightsquigarrow \kappa_1' \dashv \Omega_1 \\ \Omega = \Omega_0, \Omega_1, \iota{:}\kappa_1 \sim \kappa_1' \\ \Sigma; \Psi, \Omega, b{:}_{\mathsf{Rel}}\kappa_1' \vdash^{*}_{\mathsf{ty}} \mathrm{t} : \kappa_2[b \rhd \mathbf{sym}\,\iota/a] \rightsquigarrow \tau \dashv \Omega_2 \\ \Omega_2 \hookrightarrow b{:}_{\mathsf{Rel}}\kappa_1' \rightsquigarrow \Omega_2'; \xi \\ \eta = \kappa_2[(a \rhd \iota) \rhd \mathbf{sym}\,\iota/a] \approx_{\langle \mathbf{Type} \rangle} \kappa_2 \\ \tau_0 = (\lambda a{:}_{\mathsf{Rel}}\kappa_1.\, (\tau[\xi][a \rhd \iota/b] \rhd \eta)) \rhd \mathbf{sym}\,\gamma \end{array}}{\Sigma; \Psi \vdash^{\to}_{\mathsf{ty}} \lambda(a :: \mathrm{s}).\, \mathrm{t} : \kappa \rightsquigarrow \tau_0 \dashv \Omega, \Omega_2'} \quad \textsc{ITyC\_LamDep}$$

$$\frac{\begin{array}{l}\vdash^{\to}_{\mathsf{fun}}\ \kappa; \mathsf{Rel} \rightsquigarrow \gamma; \tilde{\Pi}; a; \mathsf{Rel}; \kappa_1; \kappa_2 \dashv \Omega_0 \\ \Sigma; \Psi \vdash^{\to}_{\mathsf{aq}} \mathrm{aqvar} : \kappa_1 \rightsquigarrow b : \kappa_1'; x.\tau_1 \dashv \Omega_1 \\ \Sigma; \Psi, \Omega_0, \Omega_1, b{:}_{\mathsf{Rel}}\kappa_1' \vdash^{*}_{\mathsf{ty}} \mathrm{t} : \kappa_2 \rightsquigarrow \tau \dashv \Omega_2 \\ \Omega_2 \hookrightarrow b{:}_{\mathsf{Rel}}\kappa_1' \rightsquigarrow \Omega_2'; \xi \\ \Omega' = \Omega_0, \Omega_1, \Omega_2' \end{array}}{\Sigma; \Psi \vdash^{\to}_{\mathsf{ty}} \lambda \mathrm{aqvar}.\, \mathrm{t} : \kappa \rightsquigarrow (\lambda a{:}_{\mathsf{Rel}}\kappa_1.\, \tau[\xi][\tau_1[a/x]/b]) \rhd \mathbf{sym}\,\gamma \dashv \Omega'} \quad \textsc{ITyC\_Lam}$$

$$\frac{\begin{array}{l}\vdash^{\to}_{\mathsf{fun}}\ \kappa; \mathsf{Irrel} \rightsquigarrow \gamma; \tilde{\Pi}; a; \mathsf{Irrel}; \kappa_1; \kappa_2 \dashv \Omega_0 \\ \neg(a \# \kappa_2) \\ \Sigma; \mathsf{Rel}(\Psi) \vdash^{\to}_{\mathsf{pt}} \mathrm{s} \rightsquigarrow \kappa_1' \dashv \Omega_1 \\ \Omega = \Omega_0, \Omega_1, \iota{:}\kappa_1 \sim \kappa_1' \\ \Sigma; \Psi, \Omega, b{:}_{\mathsf{Irrel}}\kappa_1' \vdash^{*}_{\mathsf{ty}} \mathrm{t} : \kappa_2[b \rhd \mathbf{sym}\,\iota/a] \rightsquigarrow \tau \dashv \Omega_2 \\ \Omega_2 \hookrightarrow b{:}_{\mathsf{Irrel}}\kappa_1' \rightsquigarrow \Omega_2'; \xi \\ \eta = \kappa_2[(a \rhd \iota) \rhd \mathbf{sym}\,\iota/a] \approx_{\langle \mathbf{Type} \rangle} \kappa_2 \\ \tau_0 = (\lambda a{:}_{\mathsf{Irrel}}\kappa_1.\, (\tau[\xi][a \rhd \iota/b] \rhd \eta)) \rhd \mathbf{sym}\,\gamma \end{array}}{\Sigma; \Psi \vdash^{\to}_{\mathsf{ty}} \Lambda(a :: \mathrm{s}).\, \mathrm{t} : \kappa \rightsquigarrow \tau_0 \dashv \Omega, \Omega_2'} \quad \textsc{ITyC\_LamIrrelDep}$$

$$\frac{\begin{array}{l}\vdash^{\to}_{\mathsf{fun}}\ \kappa; \mathsf{Irrel} \rightsquigarrow \gamma; \tilde{\Pi}; a; \mathsf{Irrel}; \kappa_1; \kappa_2 \dashv \Omega_0 \\ \Sigma; \Psi \vdash^{\to}_{\mathsf{aq}} \mathrm{aqvar} : \kappa_1 \rightsquigarrow b : \kappa_1'; x.\tau_1 \dashv \Omega_1 \\ \Sigma; \Psi, \Omega_0, \Omega_1, b{:}_{\mathsf{Irrel}}\kappa_1' \vdash^{*}_{\mathsf{ty}} \mathrm{t} : \kappa_2 \rightsquigarrow \tau \dashv \Omega_2 \\ \Omega_2 \hookrightarrow b{:}_{\mathsf{Irrel}}\kappa_1' \rightsquigarrow \Omega_2'; \xi \\ \tau_0 = (\lambda a{:}_{\mathsf{Irrel}}\kappa_1.\, \tau[\xi][\tau_1[a/x]/b]) \rhd \mathbf{sym}\,\gamma \end{array}}{\Sigma; \Psi \vdash^{\to}_{\mathsf{ty}} \Lambda \mathrm{aqvar}.\, \mathrm{t} : \kappa \rightsquigarrow \tau_0 \dashv \Omega_0, \Omega_1, \Omega_2'} \quad \textsc{ITyC\_LamIrrel}$$

$$\frac{\Sigma; \Psi \vdash^{\to}_{\mathsf{ty}} \mathrm{t} : \tilde{\Pi}_{\mathsf{Req}}\, a{:}_{\mathsf{Rel}}\kappa.\, \kappa \rightsquigarrow \tau \dashv \Omega}{\Sigma; \Psi \vdash^{\to}_{\mathsf{ty}} \mathbf{fix}\,\mathrm{t} : \kappa \rightsquigarrow \mathbf{fix}\,\tau \dashv \Omega} \quad \textsc{ITyC\_Fix}$$

$$\frac{\begin{array}{l}\Sigma;\Psi \overset{*}{\underset{\mathsf{ty}}{\Rightarrow}} \mathrm{t} \rightsquigarrow \tau : \kappa_1 \dashv \Omega \\ \underset{\mathsf{pre}}{\Rightarrow} \kappa_2 \rightsquigarrow \Delta; \kappa_2'; \tau_2 \\ \Omega \hookrightarrow \Delta \rightsquigarrow \Omega'; \xi_1 \\ \kappa_1[\xi_1] \leq^* \kappa_2' \rightsquigarrow \tau_2' \dashv \Omega_2 \\ \Omega_2 \hookrightarrow \Delta \rightsquigarrow \Omega_2'; \xi_2\end{array}}{\Sigma;\Psi \underset{\mathsf{ty}}{\Rightarrow} \mathrm{t} : \kappa_2 \rightsquigarrow \tau_2\,(\lambda\Delta.\,\tau_2'[\xi_2]\,\tau[\xi_1]) \dashv \Omega', \Omega_2'} \quad \textsc{ITyC\_Infer}$$

$\boxed{\Sigma;\Psi \overset{*}{\underset{\mathsf{ty}}{\Leftarrow}} \mathrm{t} : \kappa \rightsquigarrow \tau \dashv \Omega}$ Check against a type that may have specified binders.

$$\frac{\begin{array}{l}\neg(a \# \kappa_2) \\ \Sigma; \mathsf{Rel}(\Psi) \underset{\mathsf{pt}}{\Rightarrow} \mathrm{s} \rightsquigarrow \kappa_1' \dashv \Omega_1 \\ \Omega = \Omega_1, \iota{:}\kappa_1 \sim \kappa_1' \\ \Sigma;\Psi,\Omega,b{:}_{\mathsf{Rel}}\kappa_1' \overset{*}{\underset{\mathsf{ty}}{\Leftarrow}} \mathrm{t} : \kappa_2[b \rhd \mathbf{sym}\,\iota/a] \rightsquigarrow \tau \dashv \Omega_2 \\ \Omega_2 \hookrightarrow b{:}_{\mathsf{Rel}}\kappa_1' \rightsquigarrow \Omega_2'; \xi \\ \eta = \kappa_2[(a \rhd \iota) \rhd \mathbf{sym}\,\iota/a] \approx_{\langle \mathbf{Type} \rangle} \kappa_2 \\ \tau_0 = \lambda a{:}_{\mathsf{Rel}}\kappa_1.\,(\tau[\xi][a \rhd \iota/b] \rhd \eta)\end{array}}{\Sigma;\Psi \overset{*}{\underset{\mathsf{ty}}{\Leftarrow}} \lambda@(a :: \mathrm{s}).\,\mathrm{t} : \underset{\sim}{\Pi}_{\mathsf{Spec}}\, a{:}_{\mathsf{Rel}}\kappa_1.\,\kappa_2 \rightsquigarrow \tau_0 \dashv \Omega, \Omega_2'} \quad \textsc{ITyC\_LamInvisDep}$$

$$\frac{\begin{array}{l}\Sigma;\Psi \underset{\mathsf{aq}}{\Rightarrow} \mathrm{aqvar} : \kappa_1 \rightsquigarrow b : \kappa_1'; x.\tau_1 \dashv \Omega_1 \\ \Sigma;\Psi,\Omega_1,b{:}_{\mathsf{Rel}}\kappa_1' \overset{*}{\underset{\mathsf{ty}}{\Leftarrow}} \mathrm{t} : \kappa_2 \rightsquigarrow \tau \dashv \Omega_2 \\ \Omega_2 \hookrightarrow b{:}_{\mathsf{Rel}}\kappa_1' \rightsquigarrow \Omega_2'; \xi \\ \tau_0 = \lambda a{:}_{\mathsf{Rel}}\kappa_1.\,\tau[\xi][\tau_1[a/x]/b]\end{array}}{\Sigma;\Psi \overset{*}{\underset{\mathsf{ty}}{\Leftarrow}} \lambda@\mathrm{aqvar}.\,\mathrm{t} : \underset{\sim}{\Pi}_{\mathsf{Spec}}\, a{:}_{\mathsf{Rel}}\kappa_1.\,\kappa_2 \rightsquigarrow \tau_0 \dashv \Omega_1, \Omega_2'} \quad \textsc{ITyC\_LamInvis}$$

$$\frac{\begin{array}{l}\neg(a \# \kappa_2) \\ \Sigma; \mathsf{Rel}(\Psi) \underset{\mathsf{pt}}{\Rightarrow} \mathrm{s} \rightsquigarrow \kappa_1' \dashv \Omega_1 \\ \Omega = \Omega_1, \iota{:}\kappa_1 \sim \kappa_1' \\ \Sigma;\Psi,\Omega,b{:}_{\mathsf{Irrel}}\kappa_1' \overset{*}{\underset{\mathsf{ty}}{\Leftarrow}} \mathrm{t} : \kappa_2[b \rhd \mathbf{sym}\,\iota/a] \rightsquigarrow \tau \dashv \Omega_2 \\ \Omega_2 \hookrightarrow b{:}_{\mathsf{Irrel}}\kappa_1' \rightsquigarrow \Omega_2'; \xi \\ \eta = \kappa_2[(a \rhd \iota) \rhd \mathbf{sym}\,\iota/a] \approx_{\langle \mathbf{Type} \rangle} \kappa_2 \\ \tau_0 = \lambda a{:}_{\mathsf{Irrel}}\kappa_1.\,(\tau[\xi][a \rhd \iota/b] \rhd \eta)\end{array}}{\Sigma;\Psi \overset{*}{\underset{\mathsf{ty}}{\Leftarrow}} \Lambda@(a :: \mathrm{s}).\,\mathrm{t} : \underset{\sim}{\Pi}_{\mathsf{Spec}}\, a{:}_{\mathsf{Irrel}}\kappa_1.\,\kappa_2 \rightsquigarrow \tau_0 \dashv \Omega, \Omega_2'} \quad \textsc{ITyC\_LamInvisIrrelDep}$$

$$\frac{\begin{array}{l}\Sigma;\Psi \underset{\mathsf{aq}}{\Rightarrow} \mathrm{aqvar} : \kappa_1 \rightsquigarrow b : \kappa_1'; x.\tau_1 \dashv \Omega_1 \\ \Sigma;\Psi,\Omega_1,b{:}_{\mathsf{Irrel}}\kappa_1' \overset{*}{\underset{\mathsf{ty}}{\Leftarrow}} \mathrm{t} : \kappa_2 \rightsquigarrow \tau \dashv \Omega_2 \\ \Omega_2 \hookrightarrow b{:}_{\mathsf{Irrel}}\kappa_1' \rightsquigarrow \Omega_2'; \xi \\ \tau_0 = \lambda a{:}_{\mathsf{Irrel}}\kappa_1.\,\tau[\xi][\tau_1[a/x]/b]\end{array}}{\Sigma;\Psi \overset{*}{\underset{\mathsf{ty}}{\Leftarrow}} \Lambda@\mathrm{aqvar}.\,\mathrm{t} : \underset{\sim}{\Pi}_{\mathsf{Spec}}\, a{:}_{\mathsf{Irrel}}\kappa_1.\,\kappa_2 \rightsquigarrow \tau_0 \dashv \Omega_1, \Omega_2'} \quad \textsc{ITyC\_LamInvisIrrel}$$

$$\frac{\begin{array}{l}\Sigma;\Psi \overset{*}{\underset{\mathsf{ty}}{\Rightarrow}} \mathrm{t}_1 \rightsquigarrow \tau_1 : \kappa_1 \dashv \Omega \\ \Sigma;\Psi,\Omega,x{:}_{\mathsf{Rel}}\kappa_1 \overset{*}{\underset{\mathsf{ty}}{\Rightarrow}} \mathrm{t}_2 : \kappa \rightsquigarrow \tau_2 \dashv \Omega_2 \\ \Omega_2 \hookrightarrow x{:}_{\mathsf{Rel}}\kappa_1 \rightsquigarrow \Omega_2';\xi\end{array}}{\Sigma;\Psi \overset{*}{\underset{\mathsf{ty}}{\Rightarrow}} \mathbf{let}\, x := \mathrm{t}_1 \,\mathbf{in}\, \mathrm{t}_2 : \kappa \rightsquigarrow (\lambda x{:}_{\mathsf{Rel}}\kappa_1.\,(\tau_2[\xi]))\,\tau_1 \dashv \Omega,\Omega_2'} \quad \textsc{ITyC\_Let}$$

$$\frac{\begin{array}{l}\nu \leq \mathsf{Spec} \\ \Sigma;\Psi,\$a{:}_\rho\kappa_1 \overset{*}{\underset{\mathsf{ty}}{\Rightarrow}} \mathrm{t} : \kappa_2 \rightsquigarrow \tau \dashv \Omega \\ \Omega \hookrightarrow \$a{:}_\rho\kappa_1 \rightsquigarrow \Omega';\xi\end{array}}{\Sigma;\Psi \overset{*}{\underset{\mathsf{ty}}{\Rightarrow}} \mathrm{t} : \underset{\sim}{\Pi}_\nu \$a{:}_\rho\kappa_1.\,\kappa_2 \rightsquigarrow \lambda\$a{:}_\rho\kappa_1.\,\tau[\xi] \dashv \Omega'} \quad \textsc{ITyC\_Skol}$$

$$\frac{\Sigma;\Psi \underset{\mathsf{ty}}{\Rightarrow} \mathrm{t} : \kappa \rightsquigarrow \tau \dashv \Omega}{\Sigma;\Psi \overset{*}{\underset{\mathsf{ty}}{\Rightarrow}} \mathrm{t} : \kappa \rightsquigarrow \tau \dashv \Omega} \quad \textsc{ITyC\_Otherwise}$$

$\boxed{\Sigma;\Psi \underset{\mathsf{pt}}{\Rightarrow} \mathrm{s} \rightsquigarrow \tau \dashv \Omega}$ Check a poly-type (which always has type **Type**).

$$\frac{\begin{array}{l}\underset{\mathsf{pi}}{\Rightarrow} \mathrm{quant} \rightsquigarrow \Pi;\rho \\ \Sigma;\Psi \underset{\mathsf{q}}{\Rightarrow} \mathrm{qvar} \rightsquigarrow a : \kappa;\nu \dashv \Omega \\ \Sigma;\Psi,\Omega,a{:}_\rho\kappa \underset{\mathsf{pt}}{\Rightarrow} \mathrm{s} \rightsquigarrow \sigma \dashv \Omega_2 \\ \Omega_2 \hookrightarrow a{:}_\rho\kappa \rightsquigarrow \Omega_2';\xi\end{array}}{\Sigma;\Psi \underset{\mathsf{pt}}{\Rightarrow} \forall\,\mathrm{qvar}.\,\mathrm{s} \rightsquigarrow \Pi_\nu a{:}_\rho\kappa.\,(\sigma[\xi]) \dashv \Omega,\Omega_2'} \quad \textsc{IPtC\_Pi}$$

$$\frac{\begin{array}{l}\Sigma;\Psi \underset{\mathsf{ty}}{\Rightarrow} \mathrm{t} : \mathbf{Type} \rightsquigarrow \tau \dashv \Omega_1 \\ \Sigma;\Psi,\Omega_1,\$a{:}_{\mathsf{Rel}}\tau \underset{\mathsf{pt}}{\Rightarrow} \mathrm{s} \rightsquigarrow \sigma \dashv \Omega_2 \\ \Omega_2 \hookrightarrow \$a{:}_{\mathsf{Rel}}\tau \rightsquigarrow \Omega_2';\xi\end{array}}{\Sigma;\Psi \underset{\mathsf{pt}}{\Rightarrow} \mathrm{t} \Rightarrow \mathrm{s} \rightsquigarrow \underset{\sim}{\Pi}_{\mathsf{Inf}} \$a{:}_{\mathsf{Rel}}\tau.\,(\sigma[\xi]) \dashv \Omega_1,\Omega_2'} \quad \textsc{IPtC\_Constrained}$$

$$\frac{\Sigma;\Psi \underset{\mathsf{ty}}{\Rightarrow} \mathrm{t} : \mathbf{Type} \rightsquigarrow \tau \dashv \Omega}{\Sigma;\Psi \underset{\mathsf{pt}}{\Rightarrow} \mathrm{t} \rightsquigarrow \tau \dashv \Omega} \quad \textsc{IPtC\_Mono}$$

D.6 Inference for auxiliary syntactic elements

$\boxed{\Sigma;\Psi;\rho \overset{*}{\underset{\mathsf{arg}}{\Rightarrow}} \mathrm{t} : \kappa \rightsquigarrow \psi;\tau \dashv \Omega}$ Check a function argument against its known type.

$$\frac{\Sigma;\Psi \overset{*}{\underset{\mathsf{ty}}{\Rightarrow}} \mathrm{t} : \kappa \rightsquigarrow \tau \dashv \Omega}{\Sigma;\Psi;\mathsf{Rel} \overset{*}{\underset{\mathsf{arg}}{\Rightarrow}} \mathrm{t} : \kappa \rightsquigarrow \tau;\tau \dashv \Omega} \quad \textsc{IArg\_Rel}$$

$$\frac{\Sigma;\mathsf{Rel}(\Psi) \overset{*}{\underset{\mathsf{ty}}{\Rightarrow}} \mathrm{t} : \kappa \rightsquigarrow \tau \dashv \Omega}{\Sigma;\Psi;\mathsf{Irrel} \overset{*}{\underset{\mathsf{arg}}{\Rightarrow}} \mathrm{t} : \kappa \rightsquigarrow \{\tau\};\tau \dashv \Omega} \quad \textsc{IArg\_Irrel}$$

$\boxed{\Sigma; \Psi; \kappa_0; \tau_0 \mathrel{\overset{\rightarrow}{\mathsf{alt}}} \mathrm{alt} : \kappa \rightsquigarrow \mathit{alt} \dashv \Omega}$ Synth. a case alt. against a unification variable.

$$\frac{\begin{array}{l}\Sigma \mathrel{\overset{}{\vdash_{\mathsf{tc}}}} H : \Delta_1; \Delta_2; H' \qquad \Delta_3, \Delta_4 = \Delta_2[\overline{\tau}/\mathsf{dom}(\Delta_1)] \\ \mathsf{dom}(\Delta_3) = \overline{x} \qquad \mathsf{dom}(\Delta_4) = \mathsf{dom}(\Delta') \\ \mathsf{match}_{\{\mathsf{dom}(\Delta_3)\}}(\mathsf{types}(\Delta_4); \mathsf{types}(\Delta')) = \mathsf{Just}\,\theta \\ \Sigma; \Psi, \Delta_3 \mathrel{\overset{\rightarrow}{\mathsf{ty}}} \mathrm{t} : \kappa \rightsquigarrow \tau \dashv \Omega \\ \Omega \hookrightarrow \Delta_3 \rightsquigarrow \Omega'; \xi \\ \Delta_3' = \Delta_3, c{:}\tau_0 \sim H_{\{\overline{\tau}\}}\,\overline{x}\end{array}}{\Sigma; \Psi; \Pi\Delta'.\, H'\,\overline{\tau}; \tau_0 \mathrel{\overset{\rightarrow}{\mathsf{alt}}} H\,\overline{x} \to \mathrm{t} : \kappa \rightsquigarrow H \to \lambda\Delta_3'.\,(\tau[\xi]) \dashv \Omega'}\ \textsc{IAlt\_Con}$$

$$\frac{\Sigma; \Psi \mathrel{\overset{\rightarrow}{\mathsf{ty}}} \mathrm{t} : \kappa \rightsquigarrow \tau \dashv \Omega}{\Sigma; \Psi; \kappa_0; \tau_0 \mathrel{\overset{\rightarrow}{\mathsf{alt}}} \_ \to \mathrm{t} : \kappa \rightsquigarrow \_ \to \tau \dashv \Omega}\ \textsc{IAlt\_Default}$$

$\boxed{\Sigma; \Psi; \kappa_0; \tau_0 \mathrel{\overset{\rightarrow}{\mathsf{altc}}} \mathrm{alt} : \kappa \rightsquigarrow \mathit{alt} \dashv \Omega}$ Check a case alt. against a known result type.

$$\frac{\begin{array}{l}\Sigma \mathrel{\overset{}{\vdash_{\mathsf{tc}}}} H : \Delta_1; \Delta_2; H' \qquad \Delta_3, \Delta_4 = \Delta_2[\overline{\tau}/\mathsf{dom}(\Delta_1)] \\ \mathsf{dom}(\Delta_3) = \overline{x} \qquad \mathsf{dom}(\Delta_4) = \mathsf{dom}(\Delta') \\ \mathsf{match}_{\{\mathsf{dom}(\Delta_3)\}}(\mathsf{types}(\Delta_4); \mathsf{types}(\Delta')) = \mathsf{Just}\,\theta_0 \\ \Delta_3' = \Delta_3, c{:}\tau_0 \sim H_{\{\overline{\tau}\}}\,\overline{x} \\ \Sigma; \Psi, \Delta_3' \mathrel{\overset{\rightarrow}{\mathsf{ty}}} \mathrm{t} : \kappa \rightsquigarrow \tau \dashv \Omega \\ \Omega \hookrightarrow \Delta_3' \rightsquigarrow \Omega'; \xi\end{array}}{\Sigma; \Psi; \Pi\Delta'.\, H'\,\overline{\tau}; \tau_0 \mathrel{\overset{\rightarrow}{\mathsf{altc}}} H\,\overline{x} \to \mathrm{t} : \kappa \rightsquigarrow H \to \lambda\Delta_3'.\,(\tau[\xi]) \dashv \Omega'}\ \textsc{IAltC\_Con}$$

$$\frac{\Sigma; \Psi \mathrel{\overset{\rightarrow}{\mathsf{ty}}} \mathrm{t} : \kappa \rightsquigarrow \tau \dashv \Omega}{\Sigma; \Psi; \kappa_0; \tau_0 \mathrel{\overset{\rightarrow}{\mathsf{altc}}} \_ \to \mathrm{t} : \kappa \rightsquigarrow \_ \to \tau \dashv \Omega}\ \textsc{IAltC\_Default}$$

$\boxed{\Sigma; \Psi \mathrel{\overset{\rightarrow}{\mathsf{q}}} \mathrm{qvar} \rightsquigarrow a : \kappa; \nu \dashv \Omega}$ Synthesize a bound variable.

$$\frac{\Sigma; \Psi \mathrel{\overset{\rightarrow}{\mathsf{aq}}} \mathrm{aqvar} \rightsquigarrow a : \kappa \dashv \Omega}{\Sigma; \Psi \mathrel{\overset{\rightarrow}{\mathsf{q}}} \mathrm{aqvar} \rightsquigarrow a : \kappa; \mathsf{Req} \dashv \Omega}\ \textsc{IQVar\_Req}$$

$$\frac{\Sigma; \Psi \mathrel{\overset{\rightarrow}{\mathsf{aq}}} \mathrm{aqvar} \rightsquigarrow a : \kappa \dashv \Omega}{\Sigma; \Psi \mathrel{\overset{\rightarrow}{\mathsf{q}}} @\mathrm{aqvar} \rightsquigarrow a : \kappa; \mathsf{Spec} \dashv \Omega}\ \textsc{IQVar\_Spec}$$

$\boxed{\Sigma; \Psi \mathrel{\overset{\rightarrow}{\mathsf{aq}}} \mathrm{aqvar} \rightsquigarrow a : \kappa \dashv \Omega}$ Synthesize a bound variable (w/o vis. marker).

$$\frac{\mathsf{fresh}\ \beta}{\Sigma; \Psi \mathrel{\overset{\rightarrow}{\mathsf{aq}}} a \rightsquigarrow a : \beta \dashv \beta{:}_{\mathsf{Irrel}}\mathbf{Type}}\ \textsc{IAQVar\_Var}$$

$$\frac{\Sigma; \mathsf{Rel}(\Psi) \mathrel{\overset{\rightarrow}{\mathsf{pt}}} \mathrm{s} \rightsquigarrow \sigma \dashv \Omega}{\Sigma; \Psi \mathrel{\overset{\rightarrow}{\mathsf{aq}}} (a :: \mathrm{s}) \rightsquigarrow a : \sigma \dashv \Omega}\ \textsc{IAQVar\_Annot}$$

$\boxed{\Sigma; \Psi \Rightarrow_{\mathsf{aq}} \text{aqvar} : \kappa \rightsquigarrow a : \kappa'; x.\tau \dashv \Omega}$ Check a bound variable (w/o vis. marker).

$$\frac{}{\Sigma; \Psi \Rightarrow_{\mathsf{aq}} a : \kappa \rightsquigarrow a : \kappa; x.x \dashv \varnothing} \quad \textsc{IAQVarC\_Var}$$

$$\frac{\begin{array}{l}\Sigma; \mathsf{Rel}(\Psi) \Rightarrow_{\mathsf{pt}} \mathrm{s} \rightsquigarrow \sigma \dashv \Omega_1 \\ \kappa \leq \sigma \rightsquigarrow \tau \dashv \Omega_2\end{array}}{\Sigma; \Psi \Rightarrow_{\mathsf{aq}} (a :: \mathrm{s}) : \kappa \rightsquigarrow a : \sigma; x.\tau\, x \dashv \Omega_1, \Omega_2} \quad \textsc{IAQVarC\_Annot}$$

$\boxed{\Rightarrow_{\mathsf{pi}} \text{quant} \rightsquigarrow \Pi; \rho}$ Interpret a quantifier.

$$\frac{}{\Rightarrow_{\mathsf{pi}} \forall \rightsquigarrow \underset{\sim}{\Pi}; \mathsf{Irrel}} \quad \textsc{IQu\_ForAll}$$

$$\frac{}{\Rightarrow_{\mathsf{pi}} {}'\forall \rightsquigarrow {}'\Pi; \mathsf{Irrel}} \quad \textsc{IQu\_MForAll}$$

$$\frac{}{\Rightarrow_{\mathsf{pi}} \Pi \rightsquigarrow \underset{\sim}{\Pi}; \mathsf{Rel}} \quad \textsc{IQu\_Pi}$$

$$\frac{}{\Rightarrow_{\mathsf{pi}} {}'\Pi \rightsquigarrow {}'\Pi; \mathsf{Rel}} \quad \textsc{IQu\_MPi}$$

D.7 Kind conversions

$\boxed{\Rightarrow_{\mathsf{fun}} \kappa; \rho_1 \rightsquigarrow \gamma; \Pi; a; \rho_2; \kappa_1; \kappa_2 \dashv \Omega}$ Extract out the parts of a function kind.

$$\frac{}{\Rightarrow_{\mathsf{fun}} \Pi_{\mathsf{Req}}\, a{:}_\rho \kappa_1.\, \kappa_2; \rho_0 \rightsquigarrow \langle \Pi_{\mathsf{Req}}\, a{:}_\rho \kappa_1.\, \kappa_2 \rangle; \Pi; a; \rho; \kappa_1; \kappa_2 \dashv \varnothing} \quad \textsc{IFun\_Id}$$

$$\frac{\begin{array}{l}\mathsf{fresh}\, \iota \qquad \mathsf{fresh}\, \beta_1, \beta_2 \\ \Omega \;=\; \beta_1{:}_{\mathsf{Irrel}} \mathbf{Type}, \beta_2{:}_{\mathsf{Irrel}} \mathbf{Type}, \iota{:}\kappa_0 \sim \underset{\sim}{\Pi}_{\mathsf{Req}}\, a{:}_\rho \beta_1.\, \beta_2\end{array}}{\Rightarrow_{\mathsf{fun}} \kappa_0; \rho \rightsquigarrow \iota; \underset{\sim}{\Pi}; a; \rho; \beta_1; \beta_2 \dashv \Omega} \quad \textsc{IFun\_Cast}$$

$\boxed{\Sigma; \Psi \Rightarrow_{\mathsf{scrut}} \overline{\text{alt}}; \kappa \rightsquigarrow \gamma; \Delta; H; \overline{\tau} \dashv \Omega}$ Extract out the parts of a scrutinee's kind.

$$\frac{\Sigma; \mathsf{Rel}(\Psi) \vDash_{\mathsf{ty}} H\, \overline{\tau} : \mathbf{Type}}{\Sigma; \Psi \Rightarrow_{\mathsf{scrut}} \overline{\text{alt}}; {}'\Pi\Delta.\, H\, \overline{\tau} \rightsquigarrow \langle {}'\Pi\Delta.\, H\, \overline{\tau} \rangle; \Delta; H; \overline{\tau} \dashv \varnothing} \quad \textsc{IScrut\_Id}$$

$$\frac{\begin{array}{c}\Sigma \vdash_{\mathsf{tc}} H : \overline{a}{:}_{\mathsf{Irrel}}\overline{\kappa}; \Delta_2; H' \\ \mathsf{fresh}\,\overline{\alpha} \qquad \mathsf{fresh}\,\iota \\ \Omega \;=\; \overline{\alpha}{:}_{\mathsf{Irrel}}\overline{\kappa}[\overline{\alpha}/\overline{a}], \iota{:}\kappa \sim H'\,\overline{\alpha}\end{array}}{\Sigma; \Psi \vdash^{\mathsf{scrut}}_{\mathsf{scrut}} (H\,\overline{x} \to \mathrm{t}; \overline{\mathrm{alt}}); \kappa \rightsquigarrow \iota; \varnothing; H'; \overline{\alpha} \dashv \Omega} \quad \textsc{IScrut\_Cast}$$

D.8 Instantiation

$\boxed{\vdash^{\nu_1}_{\mathsf{inst}} \kappa \rightsquigarrow \overline{\psi}; \kappa' \dashv \Omega}$ Instantiate so that a type's first binder is more visible than ν.

$$\frac{\begin{array}{c}\mathsf{fresh}\,\alpha \qquad \nu_2 \leq \nu_1 \\ \vdash^{\nu_1}_{\mathsf{inst}} \kappa_2[\alpha/a] \rightsquigarrow \overline{\psi}; \kappa_2' \dashv \Omega\end{array}}{\vdash^{\nu_1}_{\mathsf{inst}} \Pi_{\nu_2} a{:}_{\mathsf{Rel}}\kappa_1.\, \kappa_2 \rightsquigarrow \alpha, \overline{\psi}; \kappa_2' \dashv \alpha{:}_{\mathsf{Rel}}\kappa_1, \Omega} \quad \textsc{IInst\_Rel}$$

$$\frac{\begin{array}{c}\mathsf{fresh}\,\alpha \qquad \nu_2 \leq \nu_1 \\ \vdash^{\nu_1}_{\mathsf{inst}} \kappa_2[\alpha/a] \rightsquigarrow \overline{\psi}; \kappa_2' \dashv \Omega\end{array}}{\vdash^{\nu_1}_{\mathsf{inst}} \Pi_{\nu_2} a{:}_{\mathsf{Rel}}\kappa_1.\, \kappa_2 \rightsquigarrow \{\alpha\}, \overline{\psi}; \kappa_2' \dashv \alpha{:}_{\mathsf{Irrel}}\kappa_1, \Omega} \quad \textsc{IInst\_Irrel}$$

$$\frac{\begin{array}{c}\mathsf{fresh}\,\iota \\ \vdash^{\nu_1}_{\mathsf{inst}} \kappa_2[\iota/c] \rightsquigarrow \overline{\psi}; \kappa_2' \dashv \Omega\end{array}}{\vdash^{\nu_1}_{\mathsf{inst}} \Pi_{\mathsf{Inf}} c{:}\phi.\, \kappa \rightsquigarrow \iota, \overline{\psi}; \kappa_2' \dashv \iota{:}\phi, \Omega} \quad \textsc{IInst\_Co}$$

$$\frac{}{\vdash^{\nu_1}_{\mathsf{inst}} \kappa \rightsquigarrow \varnothing; \kappa \dashv \varnothing} \quad \textsc{IInst\_Done}$$

$\boxed{\nu_1 \leq \nu_2}$ "Less-visible-than" relation

$$\frac{}{\nu \leq \nu} \quad \textsc{IVis\_Refl}$$

$$\frac{\nu_1 \leq \nu_2 \qquad \nu_2 \leq \nu_3}{\nu_1 \leq \nu_3} \quad \textsc{IVis\_Trans}$$

$$\frac{}{\mathsf{Inf} \leq \mathsf{Spec}} \quad \textsc{IVis\_InfSpec}$$

$$\frac{}{\mathsf{Spec} \leq \mathsf{Req}} \quad \textsc{IVis\_SpecReq}$$

D.9 Subsumption

$\boxed{\Rightarrow_{\mathsf{pre}} \kappa \rightsquigarrow \Delta; \kappa'; \tau}$ Convert a kind into prenex form.

$$\frac{\begin{array}{c}\nu \leq \mathsf{Spec} \\ \Rightarrow_{\mathsf{pre}} \kappa_2 \rightsquigarrow \Delta; \kappa_2'; \tau\end{array}}{\Rightarrow_{\mathsf{pre}} \underset{\sim}{\Pi}_\nu \delta.\, \kappa_2 \rightsquigarrow \delta, \Delta; \kappa_2'; \lambda(x{:}_{\mathsf{Rel}} \underset{\sim}{\Pi} \delta, \Delta.\, \kappa_2'), \delta.\, \tau\,(x\,\mathsf{dom}(\delta))} \quad \textsc{IPrenex\_Invis}$$

$$\frac{\begin{array}{l}\Rightarrow_{\mathsf{pre}} \kappa_2 \rightsquigarrow \Delta; \kappa_2'; \tau \\ \tau_0 \;=\; \lambda(x{:}_{\mathsf{Rel}} \underset{\sim}{\Pi} \Delta, \delta.\, \kappa_2'), \delta.\, \tau\,(\lambda \Delta.\, x\,\mathsf{dom}(\Delta)\,\mathsf{dom}(\delta))\end{array}}{\Rightarrow_{\mathsf{pre}} \underset{\sim}{\Pi}_{\mathsf{Req}} \delta.\, \kappa_2 \rightsquigarrow \Delta; \underset{\sim}{\Pi}_{\mathsf{Req}} \delta.\, \kappa_2'; \tau_0} \quad \textsc{IPrenex\_Vis}$$

$$\frac{}{\Rightarrow_{\mathsf{pre}} \kappa \rightsquigarrow \varnothing; \kappa; \lambda x{:}_{\mathsf{Rel}} \kappa.\, x} \quad \textsc{IPrenex\_NoPi}$$

$\boxed{\kappa_1 \leq^* \kappa_2 \rightsquigarrow \tau \dashv \Omega}$ "κ_1 subsumes κ_2." (κ_2 is in prenex form)

$$\frac{\begin{array}{l}\kappa_3 \leq \kappa_1 \rightsquigarrow \tau_1 \dashv \Omega_1 \qquad \kappa_2[\tau_1\ b/a] \leq \kappa_4 \rightsquigarrow \tau_2 \dashv \Omega_2 \\ \Omega_2 \hookrightarrow b{:}_{\mathsf{Rel}} \kappa_3 \rightsquigarrow \Omega_2'; \xi \\ \tau_0 \;=\; \lambda x{:}_{\mathsf{Rel}} (\Pi a{:}_{\mathsf{Rel}} \kappa_1.\, \kappa_2), b{:}_{\mathsf{Rel}} \kappa_3.\, \tau_2[\xi]\,(x\,(\tau_1\ b))\end{array}}{\Pi_{\mathsf{Req}} a{:}_{\mathsf{Rel}} \kappa_1.\, \kappa_2 \leq^* \underset{\sim}{\Pi}_{\mathsf{Req}} b{:}_{\mathsf{Rel}} \kappa_3.\, \kappa_4 \rightsquigarrow \tau_0 \dashv \Omega_1, \Omega_2'} \quad \textsc{ISub\_FunRel}$$

$$\frac{\begin{array}{l}\kappa_3 \leq \kappa_1 \rightsquigarrow \tau_1 \dashv \Omega_1 \qquad \kappa_2[\tau_1\ b/a] \leq \kappa_4 \rightsquigarrow \tau_2 \dashv \Omega_2 \\ \Omega_2 \hookrightarrow b{:}_{\mathsf{Rel}} \kappa_3 \rightsquigarrow \Omega_2'; \xi \\ \tau_0 \;=\; \lambda x{:}_{\mathsf{Rel}} (\Pi a{:}_{\mathsf{Irrel}} \kappa_1.\, \kappa_2), b{:}_{\mathsf{Rel}} \kappa_3.\, \tau_2[\xi]\,(x\,\{\tau_1\ b\})\end{array}}{\Pi_{\mathsf{Req}} a{:}_{\mathsf{Irrel}} \kappa_1.\, \kappa_2 \leq^* \underset{\sim}{\Pi}_{\mathsf{Req}} b{:}_{\mathsf{Rel}} \kappa_3.\, \kappa_4 \rightsquigarrow \tau_0 \dashv \Omega_1, \Omega_2'} \quad \textsc{ISub\_FunIrrelRel}$$

$$\frac{\begin{array}{l}\kappa_3 \leq \kappa_1 \rightsquigarrow \tau_1 \dashv \Omega_1 \qquad \kappa_2[\tau_1\ b/a] \leq \kappa_4 \rightsquigarrow \tau_2 \dashv \Omega_2 \\ \Omega_2 \hookrightarrow b{:}_{\mathsf{Irrel}} \kappa_3 \rightsquigarrow \Omega_2'; \xi \\ \tau_0 \;=\; \lambda x{:}_{\mathsf{Rel}} (\Pi a{:}_{\mathsf{Irrel}} \kappa_1.\, \kappa_2), b{:}_{\mathsf{Irrel}} \kappa_3.\, \tau_2[\xi]\,(x\,\{\tau_1\ b\})\end{array}}{\Pi_{\mathsf{Req}} a{:}_{\mathsf{Irrel}} \kappa_1.\, \kappa_2 \leq^* \underset{\sim}{\Pi}_{\mathsf{Req}} b{:}_{\mathsf{Irrel}} \kappa_3.\, \kappa_4 \rightsquigarrow \tau_0 \dashv \Omega_1, \Omega_2'} \quad \textsc{ISub\_FunIrrel}$$

$$\frac{\mathsf{fresh}\ \iota}{\tau_1 \leq^* \tau_2 \rightsquigarrow \lambda x{:}_{\mathsf{Rel}} \tau_1.\, (x \rhd \iota) \dashv \iota{:}\tau_1 \sim \tau_2} \quad \textsc{ISub\_Unify}$$

$\boxed{\kappa_1 \leq \kappa_2 \rightsquigarrow \tau \dashv \Omega}$ "κ_1 subsumes κ_2."

$$\frac{\begin{array}{l}\Rightarrow_{\mathsf{pre}} \kappa_2 \rightsquigarrow \Delta; \kappa_2'; \tau_1 \\ \overset{\mathsf{Spec}}{\Rightarrow_{\mathsf{inst}}} \kappa_1 \rightsquigarrow \overline{\psi}; \kappa_1' \dashv \Omega_1 \\ \kappa_1' \leq^* \kappa_2' \rightsquigarrow \tau_2 \dashv \Omega_2 \\ \Omega_1, \Omega_2 \hookrightarrow \Delta \rightsquigarrow \Omega'; \xi\end{array}}{\kappa_1 \leq \kappa_2 \rightsquigarrow \lambda x{:}_{\mathsf{Rel}} \kappa_1.\, \tau_1\,(\lambda \Delta.\, \tau_2[\xi]\,(x\,\overline{\psi}[\xi])) \dashv \Omega'} \quad \textsc{ISub\_DeepSkol}$$

D.10 Generalization

$\boxed{\Omega \hookrightarrow \Delta \rightsquigarrow \Omega'; \xi}$ Generalize Ω over Δ.

$$\frac{}{\varnothing \hookrightarrow \Delta \rightsquigarrow \varnothing; \varnothing} \quad \textsc{IGen\_Nil}$$

$$\frac{\xi_0 \;=\; \alpha \mapsto \mathsf{dom}(\Delta) \qquad \Omega[\xi_0] \hookrightarrow \Delta \rightsquigarrow \Omega'; \xi}{\alpha :_\rho \forall\, \Delta'.\kappa, \Omega \hookrightarrow \Delta \rightsquigarrow \alpha :_\rho \forall\, \Delta, \Delta'.\kappa, \Omega'; \xi_0, \xi} \quad \textsc{IGen\_TyVar}$$

$$\frac{\xi_0 \;=\; \iota \mapsto \mathsf{dom}(\Delta) \qquad \Omega[\xi_0] \hookrightarrow \Delta \rightsquigarrow \Omega'; \xi}{\iota : \forall\, \Delta'.\phi, \Omega \hookrightarrow \Delta \rightsquigarrow \iota : \forall\, \Delta, \Delta'.\phi, \Omega'; \xi_0, \xi} \quad \textsc{IGen\_CoVar}$$

D.11 Programs

$\boxed{\Sigma; \Gamma \vdash^{\rightarrow}_{\mathsf{decl}} \mathrm{decl} \rightsquigarrow x : \kappa := \tau}$ Check a Haskell declaration.

$$\frac{\begin{array}{l}\Sigma; \Gamma \vdash^{\rightarrow}_{\mathsf{ty}} \mathrm{t} \rightsquigarrow \tau : \kappa \dashv \Omega \\ \Sigma; \Gamma \vdash^{\rightarrow}_{\mathsf{solv}} \Omega \rightsquigarrow \Delta; \Theta \\ \tau' \;=\; \lambda\Delta.\,(\tau[\Theta]) \qquad \kappa' \;=\; \prod\!\!\!\!\sim_{\mathsf{Inf}} \Delta.\,(\kappa[\Theta])\end{array}}{\Sigma; \Gamma \vdash^{\rightarrow}_{\mathsf{decl}} x := \mathrm{t} \rightsquigarrow x : \kappa' := \tau'} \quad \textsc{IDecl\_Synthesize}$$

$$\frac{\begin{array}{l}\Sigma; \Gamma \vdash^{\rightarrow}_{\mathsf{pt}} \mathrm{s} \rightsquigarrow \sigma \dashv \Omega_1 \\ \Sigma; \mathsf{Rel}(\Gamma) \vdash^{\rightarrow}_{\mathsf{solv}} \mathsf{Rel}(\Omega_1) \rightsquigarrow \Delta_1; \Theta_1 \\ \sigma' \;=\; \prod\!\!\!\!\sim_{\mathsf{Inf}} \Delta_1.\,(\sigma[\Theta_1]) \\ \Sigma; \Gamma \vdash^{\rightarrow^*}_{\mathsf{ty}} \mathrm{t} : \sigma' \rightsquigarrow \tau \dashv \Omega_2 \\ \Sigma; \Gamma \vdash^{\rightarrow}_{\mathsf{solv}} \Omega_2 \rightsquigarrow \varnothing; \Theta_2 \\ \tau' \;=\; \tau[\Theta_2]\end{array}}{\Sigma; \Gamma \vdash^{\rightarrow}_{\mathsf{decl}} x :: \mathrm{s} := \mathrm{t} \rightsquigarrow x : \sigma' := \tau'} \quad \textsc{IDecl\_Check}$$

$\boxed{\Sigma; \Gamma \vdash^{\rightarrow}_{\mathsf{prog}} \mathrm{prog} \rightsquigarrow \Gamma'; \theta}$ Check a Haskell program.

$$\frac{}{\Sigma; \Gamma \vdash^{\rightarrow}_{\mathsf{prog}} \varnothing \rightsquigarrow \varnothing; \varnothing} \quad \textsc{IProg\_Nil}$$

$$\frac{\begin{array}{l}\Sigma; \Gamma \mathrel{\mathsf{\Rightarrow_{decl}}} \mathrm{decl} \rightsquigarrow x : \kappa := \tau \\ \Sigma; \Gamma, x{:}_{\mathsf{Rel}}\kappa,\, c{:}x \sim \tau \mathrel{\mathsf{\Rightarrow_{prog}}} \mathrm{prog} \rightsquigarrow \Gamma'; \theta\end{array}}{\Sigma; \Gamma \mathrel{\mathsf{\Rightarrow_{prog}}} \mathrm{decl}; \mathrm{prog} \rightsquigarrow x{:}_{\mathsf{Rel}}\kappa,\, c{:}x \sim \tau, \Gamma'; (\tau/x, \langle\tau\rangle/c) \circ \theta} \quad \textsc{IProg\_Decl}$$

Appendix E

Proofs about the BAKE algorithm

Throughout this appendix, I use a convention whereby in any case where the rule under consideration is printed, any metavariable names in the rule shadow any metavariable names in the lemma or theorem statement.

E.1 Type inference judgment properties

Definition E.1 (Judgments with unification variables)**.** *I write judgments with a new turnstile* $\models$*; these judgments are identical to the corresponding judgments written with a* $\vdash$ *except with the new rules as given in Appendix D. All lemmas proved over the old judgments hold over the new ones, noting that the new* UVAR *rules are unaffected by context extension.*

Definition E.2 (Generalized judgments)**.** *I sometimes write* $\Sigma; \Psi \models \mathcal{J}$*, where* $\mathcal{J}$ *stands for a* judgment*, one of the judgments headed by* $\models_{\mathsf{ty}}$*,* $\models_{\mathsf{co}}$*,* $\models_{\mathsf{prop}}$*,* $\models_{\mathsf{alt}}$*,* $\models_{\mathsf{vec}}$*,* $\models_{\mathsf{ctx}}$*, or* $\models_{\mathsf{s}}$*. Similarly, I write* $\mathcal{J}[\theta]$ *to denote substitution in the component parts of the judgment* $\mathcal{J}$*.*

Lemma E.3 (Extension)**.**

1. *If* $\Sigma; \Gamma \vdash \mathcal{J}$*, then* $\Sigma; \Gamma \models \mathcal{J}$*.*
2. *If* $\Sigma; \Gamma \models \mathcal{J}$ *and* $\mathcal{J}$ *mentions no unification variables, then* $\Sigma; \Gamma \vdash \mathcal{J}$*.*

Proof. The difference between the $\vdash$ judgments and the $\models$ judgments is only the addition of new rules for new forms. No previously valid derivations are affected. Note that, although we can't prove it now, the "mentions no unification variables" is redundant, as shown by Lemma E.11, below. □

E.2 Properties adopted from Appendix C

Remark. By the straightforward extension of the $\mathsf{Rel}(\cdot)$ operation, all previous lemmas (Lemma C.3, Lemma C.4, Lemma C.5, Lemma C.6) dealing with contexts and relevance

remain true under the $\vDash$ judgments.

Lemma E.4 (Type variable kinds [Lemma C.7]). *(as stated previously, but with reference to $\vDash$ judgments)*

Proof. As before; the new forms do not pose any problems. □

Lemma E.5 (Unification type variable kinds). *If $\Sigma \vDash_{\mathsf{ctx}} \Psi\ \mathsf{ok}$ and $\alpha :_{\rho} \forall\, \Delta.\kappa \in \Psi$, then there exists Ψ' such that $\Psi' \subseteq \mathsf{Rel}(\Psi)$ and $\Sigma; \Psi', \mathsf{Rel}(\Delta) \vDash_{\mathsf{ty}} \kappa : \mathbf{Type}$. Furthermore, the size of the derivation of $\Sigma; \Psi', \mathsf{Rel}(\Delta) \vDash_{\mathsf{ty}} \kappa : \mathbf{Type}$ is smaller than that of $\Sigma \vDash_{\mathsf{ctx}} \Psi\ \mathsf{ok}$.*

Proof. Straightforward induction on $\Sigma \vDash_{\mathsf{ctx}} \Psi\ \mathsf{ok}$. □

Lemma E.6 (Coercion variable kinds [Lemma C.8]). *(as stated previously, but with reference to $\vDash$ judgments)*

Proof. As before; the new forms do not pose any problems. □

Lemma E.7 (Unification coercion variable kinds). *If $\Sigma \vDash_{\mathsf{ctx}} \Psi\ \mathsf{ok}$ and $\iota : \forall\, \Delta.\phi \in \Psi$, then there exists Ψ' such that $\Psi' \subseteq \mathsf{Rel}(\Psi)$ and $\Sigma; \Psi', \mathsf{Rel}(\Delta) \vDash_{\mathsf{prop}} \phi\ \mathsf{ok}$. Furthermore, the size of the derivation of $\Sigma; \Psi', \mathsf{Rel}(\Delta) \vDash_{\mathsf{prop}} \phi\ \mathsf{ok}$ is smaller than that of $\Sigma \vDash_{\mathsf{ctx}} \Psi\ \mathsf{ok}$.*

Proof. Straightforward induction on $\Sigma \vDash_{\mathsf{ctx}} \Psi\ \mathsf{ok}$. □

Lemma E.8 (Context regularity [Lemma C.9]). *(as stated previously, but with reference to $\vDash$ judgments)*

Proof. As before; the new forms do not pose any problems. □

Lemma E.9 (Weakening [Lemma C.10]). *Assume $\Sigma \vDash_{\mathsf{ctx}} \Psi'\ \mathsf{ok}$ and $\Psi \subseteq \Psi'$. If $\Sigma; \Psi \vDash \mathcal{J}$, then $\Sigma; \Psi' \vDash \mathcal{J}$.*

Proof. As before; the new forms do not pose any problems. □

Lemma E.10 (Strengthening [Lemma C.11]). *(as stated previously, but with reference to $\vDash$ judgments)*

Proof. As before; the new forms do not pose any problems. □

Lemma E.11 (Scoping [Lemma C.12]). *(as stated previously, but with reference to $\vDash$ judgments)*

Proof. We must consider now Ty_UVar and Co_UVar. These cases are similar; let's focus on Ty_UVar:

$$\frac{\alpha :_{\mathsf{Rel}} \forall\, \Delta.\kappa \in \Psi \qquad \Sigma \vDash_{\mathsf{ctx}} \Psi\ \mathsf{ok} \qquad \Sigma; \Psi \vDash_{\mathsf{vec}} \overline{\psi} : \Delta}{\Sigma; \Psi \vDash_{\mathsf{ty}} \alpha_{\overline{\psi}} : \kappa[\overline{\psi}/\mathsf{dom}(\Delta)]}\ \textsc{Ty\_UVar}$$

We see that $\alpha \in \{\mathsf{dom}(\Psi)\}$, and the induction hypothesis tells us that the scoping requirement holds for $\overline{\psi}$. Lemma E.5 tells us that $\Sigma; \Psi', \mathsf{Rel}(\Delta) \vDash_{\mathsf{ty}} \kappa : \mathbf{Type}$ for some $\Psi' \subseteq \mathsf{Rel}(\Psi)$. This derivation is smaller than the one ending in TY_UVAR, and so we can use the induction hypothesis to see that $\mathsf{fv}(\kappa) \subseteq (\{\mathsf{dom}(\Psi)\} \cup \{\mathsf{dom}(\Delta)\})$. The substitution in the conclusion removes all use of variables in $\mathsf{dom}(\Delta)$, and so $\mathsf{fv}(\kappa) \subseteq \{\mathsf{dom}(\Psi)\}$ as desired. □

Lemma E.12 (Determinacy [Lemma C.20])**.** *(as stated previously, but with reference to $\vDash$ judgments)*

Proof. As before. □

Lemma E.13 (Type substitution [Lemma C.35])**.** *If $\Sigma; \Psi \vDash_{\mathsf{ty}} \sigma : \kappa$ and $\Sigma; \Psi, a{:}_\rho\kappa, \Psi' \vDash \mathcal{J}$, then $\Sigma; \Psi, \Psi'[\sigma/a] \vDash \mathcal{J}[\sigma/a]$.*

Proof. By induction on $\Sigma; \Psi, a{:}_\rho\kappa, \Psi' \vDash \mathcal{J}$. We consider only the new cases.

Case TY_UVAR:

$$\frac{\alpha :_{\mathsf{Rel}} \forall\, \Delta.\kappa \in \Psi \qquad \Sigma \vDash_{\mathsf{ctx}} \Psi\ \mathsf{ok} \qquad \Sigma; \Psi \vDash_{\mathsf{vec}} \overline{\psi} : \Delta}{\Sigma; \Psi \vDash_{\mathsf{ty}} \alpha_{\overline{\psi}} : \kappa[\overline{\psi}/\mathsf{dom}(\Delta)]}\ \textsc{Ty\_UVar}$$

We must prove $\Sigma; \Psi, \Psi'[\sigma/a] \vDash_{\mathsf{ty}} \alpha_{\overline{\psi}[\sigma/a]} : \kappa[\overline{\psi}/\mathsf{dom}(\Delta)][\sigma/a]$. (Recall that normal substitutions θ do not map unification variables.) We know $\alpha :_{\mathsf{Rel}} \forall\, \Delta.\kappa \in \Psi, a{:}_\rho\kappa, \Psi'$, $\Sigma \vDash_{\mathsf{ctx}} \Psi, a{:}_\rho\kappa, \Psi'\ \mathsf{ok}$ and $\Sigma; \Psi, a{:}_\rho\kappa, \Psi' \vDash_{\mathsf{vec}} \overline{\psi} : \Delta$. By the induction hypothesis, we can conclude $\Sigma \vDash_{\mathsf{ctx}} \Psi, \Psi'[\sigma/a]\ \mathsf{ok}$ and $\Sigma; \Psi, \Psi'[\sigma/a] \vDash_{\mathsf{vec}} \overline{\psi}[\sigma/a] : \Delta[\sigma/a]$. We now have two cases, depending on the location of α:

Case $\alpha :_{\mathsf{Rel}} \forall\, \Delta.\kappa \in \Psi$: In this case, Lemma E.11 tells us that Δ cannot mention a, and thus $\Delta[\sigma/a] = \Delta$. We can thus use $\alpha :_{\mathsf{Rel}} \forall\, \Delta.\kappa \in \Psi$ to complete the premises for TY_UVAR, showing that $\Sigma; \Psi, \Psi'[\sigma/a] \vDash_{\mathsf{ty}} \alpha_{\overline{\psi}[\sigma/a]} : \kappa[\overline{\psi}[\sigma/a]/\mathsf{dom}(\Delta)]$. The kind can be rewritten as $\kappa[\sigma/a][\overline{\psi}[\sigma/a]/\mathsf{dom}(\Delta)]$ as we know $a \notin \mathsf{fv}(\kappa)$. It can then further be rewritten to $\kappa[\overline{\psi}/\mathsf{dom}(\Delta)][\sigma/a]$ as desired.

Case $\alpha :_{\mathsf{Rel}} \forall\, \Delta.\kappa \in \Psi'$: It must be the case that $\alpha :_{\mathsf{Rel}} \forall\, (\Delta[\sigma/a]).(\kappa[\sigma/a]) \in \Psi'[\sigma/a]$. Rule TY_UVAR then gives us $\Sigma; \Psi, \Psi'[\sigma/a] \vDash_{\mathsf{ty}} \alpha_{\overline{\psi}[\sigma/a]} : \kappa[\sigma/a][\overline{\psi}[\sigma/a]/\mathsf{dom}(\Delta)]$ which can be (see above) rewritten as $\Sigma; \Psi, \Psi'[\sigma/a] \vDash_{\mathsf{ty}} \alpha_{\overline{\psi}[\sigma/a]} : \kappa[\overline{\psi}/\mathsf{dom}(\Delta)][\sigma/a]$ as desired.

Case CO_UVAR: Similar to previous case.

□

Lemma E.14 (Coercion substitution [Lemma C.36])**.** *If $\Sigma; \Psi \vDash_{\mathsf{co}} \gamma : \phi$ and $\Sigma; \Psi, c{:}\phi, \Psi' \vDash \mathcal{J}$, then $\Sigma; \Psi, \Psi'[\gamma/c] \vDash \mathcal{J}[\gamma/c]$.*

Proof. Similar to previous proof. □

Lemma E.15 (Vector substitution [Lemma C.37])**.** *If* $\Sigma; \Psi \vDash_{\mathsf{vec}} \overline{\psi} : \Delta$ *and* $\Sigma; \Psi, \Delta, \Psi' \vDash \mathcal{J}$*, then* $\Sigma; \Psi, \Psi'[\overline{\psi}/\mathsf{dom}(\Delta)] \vDash \mathcal{J}[\overline{\psi}/\mathsf{dom}(\Delta)]$.

Proof. As before, referring to Lemma E.13 and Lemma E.14. Note that this version is generalized to work over any judgment $\mathcal{J}$ while the previous proof lemma works only over $\vDash_{\mathsf{ty}}$. This generalization poses no trouble. □

E.3 Regularity

Lemma E.16 (Increasing relevance in vectors)**.** *If* $\Sigma; \Psi \vDash_{\mathsf{vec}} \overline{\psi} : \Delta$*, then* $\Sigma; \mathsf{Rel}(\Psi) \vDash_{\mathsf{vec}} \overline{\psi} : \mathsf{Rel}(\Delta)$.

Proof. Straightforward induction on the typing derivation, appealing to Lemma C.6. □

Lemma E.17 (Kind regularity [Lemma C.43])**.** *If* $\Sigma; \Psi \vDash_{\mathsf{ty}} \tau : \kappa$*, then* $\Sigma; \mathsf{Rel}(\Psi) \vDash_{\mathsf{ty}} \kappa :$ **Type**.

Proof. By induction on the typing derivation. We consider only the new case:

Case Ty_UVar:

$$\frac{\alpha :_{\mathsf{Rel}} \forall\, \Delta.\kappa \in \Psi \qquad \Sigma \vDash_{\mathsf{ctx}} \Psi\ \mathsf{ok} \qquad \Sigma; \Psi \vDash_{\mathsf{vec}} \overline{\psi} : \Delta}{\Sigma; \Psi \vDash_{\mathsf{ty}} \alpha_{\overline{\psi}} : \kappa[\overline{\psi}/\mathsf{dom}(\Delta)]}\ \textsc{Ty\_UVar}$$

We must prove $\Sigma; \mathsf{Rel}(\Psi) \vDash_{\mathsf{ty}} \kappa[\overline{\psi}/\mathsf{dom}(\Delta)] :$ **Type**. By Lemma E.8 and Lemma E.5, there exists Ψ' such that $\Psi' \subseteq \mathsf{Rel}(\Psi)$ and $\Sigma; \Psi', \mathsf{Rel}(\Delta) \vDash_{\mathsf{ty}} \kappa :$ **Type**. Lemma E.9 then gives us $\Sigma; \mathsf{Rel}(\Psi, \Delta) \vDash_{\mathsf{ty}} \kappa :$ **Type**. Lemma E.16 tells us that $\Sigma; \mathsf{Rel}(\Psi) \vDash_{\mathsf{vec}} \overline{\psi} : \mathsf{Rel}(\Delta)$. We can thus use Lemma E.15 to get $\Sigma; \mathsf{Rel}(\Psi) \vDash_{\mathsf{ty}} \kappa[\overline{\psi}/\mathsf{dom}(\Delta)] :$ **Type** as desired.

□

Lemma E.18 (Proposition regularity [Lemma C.44])**.** *If* $\Sigma; \Psi \vDash_{\mathsf{co}} \gamma : \phi$*, then* $\Sigma; \mathsf{Rel}(\Psi) \vDash_{\mathsf{prop}} \phi\ \mathsf{ok}$.

Proof. The proof for the Co_UVar case is similar to the proof above for Ty_UVar. Other cases are as before. □

E.4 Zonking

Definition E.19 (Zonker)**.** *A* zonker Θ *is a substitution from unification variables* α *and* ι *to types and coercions, respectively. Each mapping also includes a list of type and coercion variables under which it is quantified.*

$$\Theta ::= \varnothing \mid \Theta, \forall\, \overline{z}.\tau/\alpha \mid \Theta, \forall\, \overline{z}.\gamma/\iota$$

Lemma E.20 (Zonker domains)**.** *If* $\Sigma; \Psi \models_{\overline{z}} \Theta : \Omega$*, then* $\mathsf{dom}(\Theta) = \mathsf{dom}(\Omega)$*.*

Proof. By straightforward induction. □

Lemma E.21 (Zonking a relevant type variable)**.** *If* $\alpha :_{\mathsf{Rel}} \forall\, \Delta.\kappa \in \Omega$*,* $\Sigma; \Psi \models_{\overline{z}} \Theta : \Omega$*, no binding in* Ω *refers to a later one, and the range of* Θ *is disjoint from its domain, then there exists* τ *such that* $\forall\, \mathsf{dom}(\Delta).\tau/\alpha \in \Theta$ *and* $\Sigma; \Psi, \Delta[\Theta] \models_{\overline{\mathsf{ty}}} \tau : \kappa[\Theta]$*.*

Proof. By induction on $\Sigma; \Psi \models_{\overline{z}} \Theta : \Omega$.

Case Zonk_Nil: Impossible, as Ω is empty.

Case Zonk_TyVarRel: We have two cases here:

- **Case** $\Omega = \alpha :_{\mathsf{Rel}} \forall\, \Delta.\kappa, \Omega'$**:** We see that $\Theta = \forall\, \mathsf{dom}(\Delta).\tau/\alpha, \Theta'$, satisfying the first conclusion. The premise of TyVarRel tells us $\Sigma; \Psi, \Delta \models_{\overline{\mathsf{ty}}} \tau : \kappa$. By assumption, we know that Δ and κ cannot refer to α nor any variables in Ω'. Thus $\Delta = \Delta[\Theta]$ and $\kappa = \kappa[\Theta]$, and thus we can conclude $\Sigma; \Psi, \Delta[\Theta] \models_{\overline{\mathsf{ty}}} \tau : \kappa[\Theta]$ as desired.
- **Case** $\Omega = \alpha' :_{\rho} \forall\, \Delta'.\kappa', \Omega'$**, with** $\alpha \neq \alpha'$**:** We see that $\Theta = \forall\, \mathsf{dom}(\Delta').\tau'/\alpha', \Theta'$. Let $\Theta_0 = \forall\, \mathsf{dom}(\Delta').\tau'/\alpha'$. We can further see that $\alpha :_{\mathsf{Rel}} \forall\, (\Delta[\Theta_0]).(\kappa[\Theta_0]) \in \Omega'[\Theta_0]$ and $\Sigma; \Psi \models_{\overline{z}} \Theta' : \Omega'[\Theta_0]$. Because the range of Θ is disjoint from its domain and the fact that Ω is well-scoped, we know $\Omega'[\Theta_0]$ must be well-scoped. We can thus use the induction hypothesis to get τ such that $\forall\, \mathsf{dom}(\Delta).\tau/\alpha \in \Theta'$ and $\Sigma; \Psi, \Delta[\Theta_0][\Theta'] \models_{\overline{\mathsf{ty}}} \tau : \kappa[\Theta_0][\Theta']$. Because Θ is idempotent, we can rewrite this as $\Sigma; \Psi, \Delta[\Theta] \models_{\overline{\mathsf{ty}}} \tau : \kappa[\Theta]$ as desired.

Case Zonk_TyVarIrrel: Like second half of previous case.

Case Zonk_CoVar: Like previous case.

□

Lemma E.22 (Zonking a coercion variable)**.** *If* $\iota : \forall\, \Delta.\phi \in \Omega$*,* $\Sigma; \Psi \models_{\overline{z}} \Theta : \Omega$*, no binding in* Ω *refers to a later one, and the range of* Θ *is disjoint from its domain, then there exists* γ *such that* $\forall\, \mathsf{dom}(\Delta).\gamma/\iota \in \Theta$ *and* $\Sigma; \Psi, \Delta[\Theta] \models_{\overline{\mathsf{co}}} \gamma : \phi[\Theta]$*.*

Proof. Similar to previous proof. □

Lemma E.23 (Zonking)**.** *If* Θ *is idempotent,* $\Sigma; \Psi \models_{\mathsf{z}} \Theta : \Omega$ *and* $\Sigma; \Psi, \Omega, \Delta_2 \vDash \mathcal{J}$*, then* $\Sigma; \Psi, \Delta_2[\Theta] \vDash \mathcal{J}[\Theta]$.

Proof. By induction on the derivation $\Sigma; \Psi, \Omega, \Delta_2 \vDash \mathcal{J}$.

Case TY_VAR:

$$\frac{\Sigma \vdash_{\mathsf{ctx}} \Gamma \ \mathsf{ok} \qquad a{:}_{\mathsf{Rel}}\kappa \in \Gamma}{\Sigma; \Gamma \vdash_{\mathsf{ty}} a : \kappa} \ \textsc{Ty\_Var}$$

We know $\Sigma \models_{\mathsf{ctx}} \Psi, \Omega, \Delta_2 \ \mathsf{ok}$ and $a{:}_{\mathsf{Rel}}\kappa \in \Psi, \Omega, \Delta_2$. We must prove $\Sigma; \Psi, \Delta_2[\Theta] \models_{\mathsf{ty}} a[\Theta] : \kappa[\Theta]$. Zonking a non-unification variable (like a) has no effect, so we must prove $\Sigma; \Psi, \Delta_2[\Theta] \models_{\mathsf{ty}} a : \kappa[\Theta]$. We will use TY_VAR, so we must prove the following:

$\Sigma \models_{\mathsf{ctx}} \Psi, \Delta_2[\Theta] \ \mathsf{ok}$**:** By the induction hypothesis.

$a{:}_{\mathsf{Rel}}\kappa[\Theta] \in \Psi, \Delta_2[\Theta]$**:** From $a{:}_{\mathsf{Rel}}\kappa \in \Psi, \Omega, \Delta_2$, we know that a must appear either in Ψ or in Δ_2. If a is in Ψ, we are done, using Lemma E.11 to show that zonking κ has no effect. If a is in Δ_2, then $a{:}_{\mathsf{Rel}}\kappa[\Theta]$ must be in $\Delta_2[\Theta]$, and so we are done with this case.

Case CO_VAR: Similar to previous case.

Case TY_UVAR:

$$\frac{\alpha :_{\mathsf{Rel}} \forall\, \Delta.\kappa \in \Psi \qquad \Sigma \models_{\mathsf{ctx}} \Psi \ \mathsf{ok} \qquad \Sigma; \Psi \models_{\mathsf{vec}} \overline{\psi} : \Delta}{\Sigma; \Psi \models_{\mathsf{ty}} \alpha_{\overline{\psi}} : \kappa[\overline{\psi}/\mathsf{dom}(\Delta)]} \ \textsc{Ty\_UVar}$$

We know $\Sigma; \Psi, \Omega, \Delta_2 \models_{\mathsf{ty}} \alpha_{\overline{\psi}} : \kappa[\overline{\psi}/\mathsf{dom}(\Delta)]$ and must prove $\Sigma; \Psi, \Delta_2[\Theta] \models_{\mathsf{ty}} \alpha_{\overline{\psi}}[\Theta] : \kappa[\overline{\psi}/\mathsf{dom}(\Delta)][\Theta]$. We further know that $\Sigma; \Psi, \Omega, \Delta_2 \models_{\mathsf{vec}} \overline{\psi} : \Delta$ By the induction hypothesis, $\Sigma; \Psi, \Delta_2[\Theta] \models_{\mathsf{vec}} \overline{\psi}[\Theta] : \Delta[\Theta]$ and $\Sigma \models_{\mathsf{ctx}} \Psi, \Delta_2[\Theta] \ \mathsf{ok}$. There are then several possibilities:

Case $\alpha :_{\mathsf{Rel}} \forall\, \Delta.\kappa \in \Psi$**:** By Lemma E.20, we know that $\mathsf{dom}(\Theta) = \mathsf{dom}(\Omega)$. From $\Sigma \models_{\mathsf{ctx}} \Psi, \Omega, \Delta_2 \ \mathsf{ok}$ and Lemma E.11 we know that nothing in Ψ can mention any variable bound in Ω. We also know that $\alpha_{\overline{\psi}}[\Theta] = \alpha_{\overline{\psi}[\Theta]}$ and $\kappa[\Theta] = \kappa$. The telescope Δ is mentioned in Ψ and therefore is unaffected by the zonking substitution Θ. We can thus conclude that $\alpha :_{\mathsf{Rel}} \forall\, \Delta.\kappa \in \Psi, \Delta_2[\Theta]$ and $\Sigma; \Psi, \Delta_2[\Theta] \models_{\mathsf{vec}} \overline{\psi}[\Theta] : \Delta$. We can thus use TY_UVAR to conclude $\Sigma; \Psi, \Delta_2[\Theta] \models_{\mathsf{ty}} \alpha_{\overline{\psi}[\Theta]} : \kappa[\overline{\psi}[\Theta]/\mathsf{dom}(\Delta)]$. We can rewrite this kind to be $\kappa[\overline{\psi}/\mathsf{dom}(\Delta)][\Theta]$ as desired because $\kappa[\Theta] = \kappa$.

Case $\alpha :_{\mathsf{Rel}} \forall\, \Delta.\kappa \in \Omega$**:** We then use Lemma E.21 to get $\Sigma; \Psi, \Delta[\Theta] \models_{\mathsf{ty}} \tau : \kappa[\Theta]$ and $\forall\, \mathsf{dom}(\Delta).\tau/\alpha \in \Theta$. Thus (by Definition E.19) $\alpha_{\overline{\psi}}[\Theta] = \tau[\overline{\psi}[\Theta]/\mathsf{dom}(\Delta)]$. Lemma E.9 gives us $\Sigma; \Psi, \Delta_2[\Theta], \Delta[\Theta] \models_{\mathsf{ty}} \tau : \kappa[\Theta]$.

The induction hypothesis tells us that $\Sigma; \Psi, \Delta_2[\Theta] \models_{\mathsf{vec}} \overline{\psi}[\Theta] : \Delta[\Theta]$. Now, we apply Lemma E.15 to get $\Sigma; \Psi, \Delta_2[\Theta] \models_{\mathsf{ty}} \tau[\overline{\psi}[\Theta]/\mathsf{dom}(\Delta)] : \kappa[\Theta][\overline{\psi}[\Theta]/\mathsf{dom}(\Delta)]$, which can easily be rewritten to $\Sigma; \Psi, \Delta_2[\Theta] \models_{\mathsf{ty}} \tau[\overline{\psi}[\Theta]/\mathsf{dom}(\Delta)] : \kappa[\overline{\psi}/\mathsf{dom}(\Delta)][\Theta]$ as desired.

Case Co_UVar: Similar to previous case, but using Lemma E.22.

Other cases: Similar to proof for Lemma C.35.

□

E.5 Solver

The solver ($\mapsto_{\mathsf{solv}}$) must have the following properties.

Property E.24 (Solver is sound)**.** *If* $\Sigma \models_{\mathsf{ctx}} \Psi, \Omega\ \mathsf{ok}$ *and* $\Sigma; \Psi \mapsto_{\mathsf{solv}} \Omega \rightsquigarrow \Delta; \Theta$*, then* Θ *is idempotent,* $\Sigma \models_{\mathsf{ctx}} \Psi, \Delta\ \mathsf{ok}$*, and* $\Sigma; \Psi, \Delta \models_{\mathsf{z}} \Theta : \Omega$*.*

E.6 Supporting functions

Definition E.25 (make_exhaustive)**.** *Define* $\mathsf{make\_exhaustive}(\overline{alt}; \kappa)$ *as follows:*

$$\mathsf{make\_exhaustive}(\overline{alt}; \kappa) = \overline{alt} \qquad ((\_ \to \tau) \in \overline{alt})$$

$$\mathsf{make\_exhaustive}(\overline{alt}; \kappa) = \overline{alt}; \_ \to \mathit{error}\,\kappa\ \texttt{"failed match"} \qquad \textit{(otherwise)}$$

E.7 Supporting lemmas

Lemma E.26 (Vector extension)**.** *If* $\Sigma; \Psi, \Delta, \Psi' \models_{\mathsf{vec}} \overline{\psi} : \Delta'$*, then* $\Sigma; \Psi, \Delta, \Psi' \models_{\mathsf{vec}} \mathsf{dom}(\Delta), \overline{\psi} : \Delta, \Delta'$*.*

Proof. We know $\Sigma \models_{\mathsf{ctx}} \Psi, \Delta, \Psi'\ \mathsf{ok}$ by Lemma E.8. Proceed by induction on the structure of Δ.

Case $\Delta = \varnothing$**:** Trivial.

Case $\Delta = a{:}_\rho\kappa, \Delta_1$**:** To use Vec_TyRel, we must show $\Sigma; \Psi, \Delta, \Psi' \models_{\mathsf{ty}} a : \kappa$ (which is by Ty_Var) and $\Sigma; \Psi, \Delta, \Psi' \models_{\mathsf{vec}} \mathsf{dom}(\Delta_1), \overline{\psi} : (\Delta_1, \Delta')[a/a]$. The substitution clearly has no effect, so we are done by the induction hypothesis.

Other cases: Similar.

□

Lemma E.27 (Type variables instantiation)**.** *If* $\Sigma \vdash_{\mathsf{ctx}} \overline{a}{:}_{\mathsf{Irrel}}\overline{\kappa}\ \mathsf{ok}$*, then* $\Sigma \vdash_{\mathsf{ctx}} \overline{b}{:}_{\mathsf{Irrel}}\overline{\kappa}[\overline{b}/\overline{a}]\ \mathsf{ok}$*.*

Proof. By induction on the length of $\overline{\kappa}$.

Case $\overline{\kappa} = \varnothing$**:** Trivial.

Case $\overline{\kappa} = \overline{\kappa}', \kappa_0$**:** Here, we know $\overline{a} = \overline{a}', a_0$ and $\overline{b} = \overline{b}', b_0$. Our assumption is that $\Sigma \vdash_{\mathsf{ctx}} \overline{a}' :_{\mathsf{Irrel}} \overline{\kappa}', a_0 :_{\mathsf{Irrel}} \kappa_0$ ok. Inversion (of Ctx_TyVar) gives us $\Sigma; \overline{a}' :_{\mathsf{Rel}} \overline{\kappa}' \vdash_{\mathsf{ty}} \kappa_0 : \mathbf{Type}$ and $\Sigma \vdash_{\mathsf{ctx}} \overline{a}' :_{\mathsf{Irrel}} \overline{\kappa}'$ ok. The induction hypothesis tells us $\Sigma \vdash_{\mathsf{ctx}} \overline{b}' :_{\mathsf{Irrel}} \overline{\kappa}'[\overline{b}'/\overline{a}']$ ok. We must show $\Sigma; \overline{b}' :_{\mathsf{Rel}} \overline{\kappa}'[\overline{b}'/\overline{a}'] \vdash_{\mathsf{ty}} \kappa_0[\overline{b}'/\overline{a}'] : \mathbf{Type}$. Use Lemma C.10 (Weakening) to get $\Sigma; \overline{b}' :_{\mathsf{Rel}} \overline{\kappa}'[\overline{b}'/\overline{a}'], \overline{a}' :_{\mathsf{Rel}} \overline{\kappa}' \vdash_{\mathsf{ty}} \kappa_0 : \mathbf{Type}$. Lemma C.39 gives us $\Sigma; \overline{b}' :_{\mathsf{Rel}} \overline{\kappa}'[\overline{b}'/\overline{a}'] \vdash_{\mathsf{vec}} \overline{b}' : (\overline{b}' :_{\mathsf{Rel}} \overline{\kappa}'[\overline{b}'/\overline{a}'])$. We can thus use Lemma C.37 to get $\Sigma; \overline{b}' :_{\mathsf{Rel}} \overline{\kappa}'[\overline{b}'/\overline{a}'] \vdash_{\mathsf{ty}} \kappa_0[\overline{b}'/\overline{a}'] : \mathbf{Type}$ as desired. We then use Ctx_TyVar and we are done.

□

Lemma E.28 (Decreasing relevance)**.** *If* $\Sigma \models_{\mathsf{ctx}} \mathsf{Rel}(\Psi)$ ok*, then* $\Sigma \models_{\mathsf{ctx}} \Psi$ ok.

Proof. Straightforward induction on $\Sigma \models_{\mathsf{ctx}} \mathsf{Rel}(\Psi)$ ok. □

Lemma E.29 (Closing substitution substitution)**.**

1. *If* $\Sigma; \Gamma, a :_{\mathsf{Rel}} \kappa, \Gamma' \vdash_{\mathsf{subst}} \theta : \Delta$ *and* $\Sigma; \Gamma \vdash_{\mathsf{ty}} \sigma : \kappa$*, then* $\Sigma; \Gamma, \Gamma'[\sigma/a] \vdash_{\mathsf{subst}} \sigma/a \circ \theta : \Delta[\sigma/a]$.
2. *If* $\Sigma; \Gamma, a :_{\mathsf{Irrel}} \kappa, \Gamma' \vdash_{\mathsf{subst}} \theta : \Delta$ *and* $\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{ty}} \sigma : \kappa$*, then* $\Sigma; \Gamma, \Gamma'[\sigma/a] \vdash_{\mathsf{subst}} \sigma/a \circ \theta : \Delta[\sigma/a]$.
3. *If* $\Sigma; \Gamma, c :\phi, \Gamma' \vdash_{\mathsf{subst}} \theta : \Delta$ *and* $\Sigma; \Gamma \vdash_{\mathsf{co}} \gamma : \phi$*, then* $\Sigma; \Gamma, \Gamma'[\gamma/c] \vdash_{\mathsf{subst}} \gamma/c \circ \theta : \Delta[\gamma/c]$

Proof. By induction on the $\vdash_{\mathsf{subst}}$ derivation. We will consider the type substitution case; the others are similar.

Case Subst_Nil: Trivial.

Case Subst_TyRel: In this case, we know $\Sigma; \Gamma, a :_{\rho} \kappa, \Gamma' \vdash_{\mathsf{subst}} \theta : b :_{\mathsf{Rel}} \kappa_0, \Delta$ and must show $\Sigma; \Gamma, \Gamma'[\sigma/a] \vdash_{\mathsf{subst}} \sigma/a \circ \theta : b :_{\mathsf{Rel}} \kappa_0[\sigma/a], \Delta[\sigma/a]$. Inverting gives us $\Sigma; \Gamma, a :_{\rho} \kappa, \Gamma' \vdash_{\mathsf{ty}} b[\theta] : \kappa_0$ and $\Sigma; \Gamma, a :_{\rho} \kappa, \Gamma' \vdash_{\mathsf{subst}} \theta : \Delta[\theta|_b]$. To use Subst_TyRel, we must show $\Sigma; \Gamma, \Gamma'[\sigma/a] \vdash_{\mathsf{ty}} b[\sigma/a \circ \theta] : \kappa_0[\sigma/a]$ and $\Sigma; \Gamma, \Gamma'[\sigma/a] \vdash_{\mathsf{subst}} \sigma/a \circ \theta : \Delta[\sigma/a][(\sigma/a \circ \theta)|_b]$. The first of these is directly from the induction hypothesis. The induction hypothesis also gives us $\Sigma; \Gamma, \Gamma'[\sigma/a] \vdash_{\mathsf{subst}} \sigma/a \circ \theta : \Delta[\theta|_b][\sigma/a]$. We are left only to show that $\Delta[\theta|_b][\sigma/a] = \Delta[\sigma/a][(\sigma/a \circ \theta)|_b]$. On the right, we care only about θ's action on b, so we can rewrite to $\Delta[\sigma/a][\sigma/a \circ (\theta|_b)]$, which can then be rewritten to $\Delta[\theta|_b][\sigma/a]$ as desired.

Case Subst_TyIrrel: Similar to previous case.

Case Subst_Co: Similar to previous case.

□

Lemma E.30 (Closing substitution)**.** *Assume* $\Sigma; \Gamma \vdash_{\mathsf{subst}} \theta : \Delta$*. Let* $\theta' = \theta|_{\mathsf{dom}(\Delta)}$*.*

1. *If* $\Sigma; \Gamma, \Delta, \Gamma' \vdash_{\mathsf{ty}} \tau : \kappa$*, then* $\Sigma; \Gamma, \Gamma'[\theta'] \vdash_{\mathsf{ty}} \tau[\theta'] : \kappa[\theta']$*.*
2. *If* $\Sigma; \Gamma, \Delta, \Gamma' \vdash_{\mathsf{co}} \gamma : \phi$*, then* $\Sigma; \Gamma, \Gamma'[\theta'] \vdash_{\mathsf{co}} \gamma[\theta'] : \phi[\theta']$*.*
3. *If* $\Sigma; \Gamma, \Delta, \Gamma' \vdash_{\mathsf{prop}} \phi\ \mathsf{ok}$*, then* $\Sigma; \Gamma, \Gamma'[\theta'] \vdash_{\mathsf{prop}} \phi[\theta']\ \mathsf{ok}$*.*
4. *If* $\Sigma; \Gamma, \Delta, \Gamma'; \sigma_0 \vdash^{\tau_0}_{\mathsf{alt}} alt : \kappa$*, then* $\Sigma; \Gamma, \Gamma'[\theta']; \sigma_0[\theta'] \vdash^{\tau_0[\theta']}_{\mathsf{alt}} alt[\theta'] : \kappa[\theta']$*.*
5. *If* $\Sigma; \Gamma, \Delta, \Gamma' \vdash_{\mathsf{vec}} \overline{\psi} : \Delta$*, then* $\Sigma; \Gamma, \Gamma'[\theta'] \vdash_{\mathsf{vec}} \overline{\psi}[\theta'] : \Delta[\theta']$*.*
6. *If* $\Sigma \vdash_{\mathsf{ctx}} \Gamma, \Delta, \Gamma'\ \mathsf{ok}$*, then* $\Sigma \vdash_{\mathsf{ctx}} \Gamma, \Gamma'[\theta']\ \mathsf{ok}$*.*
7. *If* $\Sigma; \Gamma, \Delta, \Gamma' \vdash_{\mathsf{s}} \tau \longrightarrow \tau'$*, then* $\Sigma; \Gamma, \Gamma'[\theta'] \vdash_{\mathsf{s}} \tau[\theta'] \longrightarrow \tau'[\theta']$*.*

Proof. By induction on $\Sigma; \Gamma \vdash_{\mathsf{subst}} \theta : \Delta$. By analogy with the $\Sigma; \Psi \vDash \mathcal{J}$ notation, I will use $\Sigma; \Gamma \vdash \mathcal{J}$ to refer collectively to the judgments over which this lemma is defined.

Case Subst_Nil: In this case, $\Delta = \varnothing$ and we are done by assumption.

Case Subst_TyRel:

$$\frac{\begin{array}{c}\Sigma; \Gamma \vdash_{\mathsf{ty}} a[\theta] : \kappa \\ \Sigma; \Gamma \vdash_{\mathsf{subst}} \theta : \Delta[\theta|_a]\end{array}}{\Sigma; \Gamma \vdash_{\mathsf{subst}} \theta : a{:}_{\mathsf{Rel}}\kappa, \Delta}\ \textsc{Subst\_TyRel}$$

We know $\Sigma; \Gamma \vdash_{\mathsf{subst}} \theta : a{:}_{\mathsf{Rel}}\kappa, \Delta$ and $\Sigma; \Gamma, a{:}_{\mathsf{Rel}}\kappa, \Delta, \Gamma' \vdash \mathcal{J}$. We must prove $\Sigma; \Gamma \vdash \mathcal{J}[\theta|_{a,\mathsf{dom}(\Delta)}]$. We know $\Sigma; \Gamma \vdash_{\mathsf{ty}} a[\theta] : \kappa$ and thus we can use Lemma C.35 to get $\Sigma; \Gamma, \Delta[\theta|_a], \Gamma'[\theta|_a] \vdash \mathcal{J}[\theta|_a]$. We then use the induction hypothesis to get $\Sigma; \Gamma \vdash \mathcal{J}[\theta|_a][\theta|_{\mathsf{dom}(\Delta)}]$. It remains only to show that $\theta|_a \circ \theta|_{\mathsf{dom}(\Delta)} = \theta|_{a,\mathsf{dom}(\Delta)}$. This amounts to showing that $\mathsf{dom}(\Delta) \# a[\theta]$. We have this by Lemma C.12, and so we are done.

Case Subst_TyIrrel: Similar to previous case.

Case Subst_Co: Similar to previous case, referring to Lemma C.36.

□

E.8 Generalization

Definition E.31 (Generalizer)**.** *A* generalizer ξ *is a mapping from unification variables to vectors:*

$$\xi ::= \varnothing \mid \xi, \alpha \mapsto \overline{\psi} \mid \xi, \iota \mapsto \overline{\psi}$$

A generalizer can be applied postfix as a function. It operates only on occurrences of unification variables, acting homomorphically on all other forms:

$$\begin{array}{rcl} \alpha \mapsto \overline{\psi}_1 \in \xi & \Rightarrow & \alpha_{\overline{\psi}_2}[\xi] = \alpha_{\overline{\psi}_1, \overline{\psi}_2} \\ \textit{otherwise} & & \alpha_{\overline{\psi}}[\xi] = \alpha_{\overline{\psi}[\xi]} \\ \iota \mapsto \overline{\psi}_1 \in \xi & \Rightarrow & \iota_{\overline{\psi}_2}[\xi] = \iota_{\overline{\psi}_1, \overline{\psi}_2} \\ \textit{otherwise} & & \iota_{\overline{\psi}}[\xi] = \iota_{\overline{\psi}[\xi]} \end{array}$$

Lemma E.32 (Generalization by type variable)**.** *If* $\Sigma; \Psi, \Delta, \alpha :_\rho \forall\, \Delta'.\kappa, \Psi' \vDash \mathcal{J}$*, then* $\Sigma; \Psi, \alpha :_\rho \forall\, \Delta, \Delta'.\kappa, \Delta, \Psi'[\alpha \mapsto \mathsf{dom}(\Delta)] \vDash \mathcal{J}[\alpha \mapsto \mathsf{dom}(\Delta)]$.

Proof. Let $\xi = \alpha \mapsto \mathsf{dom}(\Delta)$. Proceed by induction on the typing derivation. The only interesting case is for unification variables:

Case Ty_UVar: Here, we know $\Sigma; \Psi, \Delta, \alpha :_\rho \forall\, \Delta'.\kappa, \Psi' \vDash_{\mathsf{ty}} \beta_{\overline{\psi}} : \kappa_0$ and must show $\Sigma; \Psi, \alpha :_\rho \forall\, \Delta, \Delta'.\kappa, \Delta, \Psi'[\xi] \vDash_{\mathsf{ty}} \beta_{\overline{\psi}}[\xi] : \kappa_0[\xi]$. We have two cases:

- **Case** $\alpha = \beta$**:** In this case, we know $\rho = \mathsf{Rel}$ and $\kappa_0 = \kappa[\overline{\psi}/\mathsf{dom}(\Delta')]$. In order to use Ty_UVar, we must show $\Sigma \vDash_{\mathsf{ctx}} \Psi, \alpha :_{\mathsf{Rel}} \forall\, \Delta, \Delta'.\kappa, \Delta, \Psi'[\xi]\ \mathsf{ok}$ (which we get from the induction hypothesis) and $\Sigma; \Psi, \alpha :_{\mathsf{Rel}} \forall\, \Delta, \Delta'.\kappa, \Delta, \Psi'[\xi] \vDash_{\overline{\mathsf{vec}}} \mathsf{dom}(\Delta), \overline{\psi} : \Delta, \Delta'$. We know $\Sigma; \Psi, \Delta, \alpha :_\rho \forall\, \Delta'.\kappa, \Psi' \vDash_{\overline{\mathsf{vec}}} \overline{\psi} : \Delta'$. The induction hypothesis tells us that $\Sigma; \Psi, \alpha :_\rho \forall\, \Delta, \Delta'.\kappa, \Delta, \Psi'[\xi] \vDash_{\overline{\mathsf{vec}}} \overline{\psi}[\xi] : \Delta'[\xi]$. However, we can see (Lemma E.11) that $\Delta'[\xi] = \Delta$. Then, Lemma E.26 tells us $\Sigma; \Psi, \alpha :_\rho \forall\, \Delta, \Delta'.\kappa, \Delta, \Psi'[\xi] \vDash_{\overline{\mathsf{vec}}} \mathsf{dom}(\Delta), \overline{\psi}[\xi] : \Delta, \Delta'$ as desired. Rule Ty_UVar gives us

  $$\Sigma; \Psi, \alpha :_\rho \forall\, \Delta, \Delta'.\kappa, \Delta, \Psi'[\xi] \vDash_{\mathsf{ty}} \alpha_{\mathsf{dom}(\Delta), \overline{\psi}[\xi]} : \kappa[\mathsf{dom}(\Delta), \overline{\psi}/\mathsf{dom}(\Delta, \Delta')].$$

  Indeed we can rewrite the kind as $\kappa[\overline{\psi}/\mathsf{dom}(\Delta')]$ and we are done.
- **Case** $\alpha \neq \beta$**:** As with other substitution properties, we must break into cases depending on where β is, but all cases are straightforwardly shown by the induction hypothesis.

Case Co_UVar: Similar to non-matching sub-case of previous case.

□

Lemma E.33 (Generalization by coercion variable)**.** *If* $\Sigma; \Psi, \Delta, \iota : \forall\, \Delta'.\phi, \Psi' \vDash \mathcal{J}$*, then* $\Sigma; \Psi, \iota : \forall\, \Delta, \Delta'.\phi, \Delta, \Psi'[\iota \mapsto \mathsf{dom}(\Delta)] \vDash \mathcal{J}[\iota \mapsto \mathsf{dom}(\Delta)]$.

Proof. Similar to previous proof. □

Lemma E.34 (Generalizer scope)**.** *If* $\Omega \hookrightarrow \Delta \rightsquigarrow \Omega'; \xi$*, then* $\mathsf{dom}(\xi) = \mathsf{dom}(\Omega)$.

Proof. Straightforward induction on $\Omega \hookrightarrow \Delta \rightsquigarrow \Omega'; \xi$. □

Lemma E.35 (Generalization). *If* $\Omega \hookrightarrow \Delta \rightsquigarrow \Omega'; \xi$ *and* $\Sigma; \Psi, \Delta, \Omega \vDash \mathcal{J}$*, then* $\Sigma; \Psi, \Omega', \Delta \vDash \mathcal{J}[\xi]$*.*

Proof. By induction on $\Omega \hookrightarrow \Delta \rightsquigarrow \Omega'; \xi$.

Case IGen_Nil: By assumption.

Case IGen_TyVar: Here, we know $\Omega = \alpha :_\rho \forall\, \Delta'.\kappa, \Omega_1$ and $\Omega' = \alpha :_\rho \forall\, \Delta, \Delta'.\kappa, \Omega_1'$. Let $\xi_0 = \alpha \mapsto \mathsf{dom}(\Delta)$. The first step is to show $\Sigma; \Psi, \alpha :_\rho \forall\, \Delta, \Delta'.\kappa, \Delta, \Omega_1[\xi_0] \vDash \mathcal{J}[\xi_0]$. This is true by Lemma E.32. We know $\Omega_1[\xi_0] \hookrightarrow \Delta \rightsquigarrow \Omega_1'; \xi_1$. We then use the induction hypothesis to get $\Sigma; \Psi, \alpha :_\rho \forall\, \Delta, \Delta'.\kappa, \Omega_1', \Delta \vDash \mathcal{J}[\xi_0][\xi_1]$. However, because the domains of ξ_0 and ξ_1 are distinct (by the well-formedness of Ω), we can rewrite as $\Sigma; \Psi, \Omega', \Delta \vDash \mathcal{J}[\xi]$ as desired.

Case IGen_CoVar: Similar to previous case, appealing to Lemma E.33.

□

E.9 Soundness

Lemma E.36 (Instantiation). *If* $\Sigma; \Psi \vDash_{\mathsf{ty}} \tau : \kappa$ *and* $\vDash^{\nu}_{\mathsf{inst}} \kappa \rightsquigarrow \overline{\psi}; \kappa' \dashv \Omega$*, then* $\Sigma; \Psi, \Omega \vDash_{\mathsf{ty}} \tau\, \overline{\psi} : \kappa'$ *and* κ' *is not a* Π*-type with a binder (with visibility* ν_2*) such that* $\nu_2 \leq \nu$*.*

Proof. Let's call the condition on the visibility of the binder (if any) of the result kind the *visibility condition*. Proceed by induction on the derivation of the $\vdash^{\nu}_{\mathsf{inst}}$ judgment.

Case IInst_Rel:

$$\frac{\mathsf{fresh}\,\alpha \qquad \nu_2 \leq \nu_1 \qquad \vdash^{\nu_1}_{\mathsf{inst}} \kappa_2[\alpha/a] \rightsquigarrow \overline{\psi}; \kappa_2' \dashv \Omega}{\vdash^{\nu_1}_{\mathsf{inst}} \Pi_{\nu_2} a{:}_{\mathsf{Rel}}\kappa_1.\, \kappa_2 \rightsquigarrow \alpha, \overline{\psi}; \kappa_2' \dashv \alpha{:}_{\mathsf{Rel}}\kappa_1, \Omega}\ \textsc{IInst\_Rel}$$

We must show that $\Sigma; \Psi, \alpha{:}_{\mathsf{Rel}}\kappa_1, \Omega \vDash_{\mathsf{ty}} \tau\, \alpha\, \overline{\psi} : \kappa_2'$ and that κ_2' satisfies the visibility condition. We can assume that $\Sigma; \Psi \vDash_{\mathsf{ty}} \tau : \Pi_{\nu_2} a{:}_{\mathsf{Rel}}\kappa_1.\, \kappa_2$. By inversion by Ty_Pi, Lemma E.8, and Lemma E.4, we can see that $\Sigma; \mathsf{Rel}(\Psi) \vDash_{\mathsf{ty}} \kappa_1 : \mathbf{Type}$. Thus $\Sigma \vDash_{\mathsf{ctx}} \Psi, \alpha{:}_{\mathsf{Rel}}\kappa_1\ \mathsf{ok}$ and Lemma E.9 gives us $\Sigma; \Psi, \alpha{:}_{\mathsf{Rel}}\kappa_1 \vDash_{\mathsf{ty}} \tau : \Pi_{\nu_2} a{:}_{\mathsf{Rel}}\kappa_1.\, \kappa_2$. Thus, Ty_AppRel gives us $\Sigma; \Psi, \alpha{:}_{\mathsf{Rel}}\kappa_1 \vDash_{\mathsf{ty}} \tau\, \alpha : \kappa_2[\alpha/a]$. The induction hypothesis then tells us that $\Sigma; \Psi, \alpha{:}_{\mathsf{Rel}}\kappa_1, \Omega \vDash_{\mathsf{ty}} \tau\, \alpha\, \overline{\psi} : \kappa_2'$ and gives us the visibility condition, as desired.

Case IInst_Irrel: Like previous case.

Case IInst_Co: Like previous cases, but appealing to Lemma E.6 instead of Lemma E.4.

Case IINST_DONE: The typing rule is by assumption. The visibility condition is by the fact that no previous rule in the judgment applied.

□

Lemma E.37 (Function position)**.** *If* $\Sigma; \Psi \vDash_{\mathsf{ty}} \kappa : \mathbf{Type}$ *and* $\vdash\!\!\!\rightarrow_{\mathsf{fun}} \kappa; \rho_1 \rightsquigarrow \gamma; \Pi; a; \rho_2; \kappa_1; \kappa_2 \dashv \Omega$*, then* $\Sigma; \Psi, \Omega \vDash_{\mathsf{co}} \gamma : \kappa \sim \Pi_{\mathsf{Req}} a{:}_{\rho_2} \kappa_1.\, \kappa_2$.

Proof. By case analysis on the derivation of $\vdash\!\!\!\rightarrow_{\mathsf{fun}}$.

Case IFUN_ID:

$$\frac{}{\vdash\!\!\!\rightarrow_{\mathsf{fun}} \Pi_{\mathsf{Req}} a{:}_{\rho} \kappa_1.\, \kappa_2; \rho_0 \rightsquigarrow \langle \Pi_{\mathsf{Req}} a{:}_{\rho} \kappa_1.\, \kappa_2 \rangle; \Pi; a; \rho; \kappa_1; \kappa_2 \dashv \varnothing} \quad \textsc{IFun\_Id}$$

Let $\kappa = \Pi_{\mathsf{Req}} a{:}_{\rho} \kappa_1.\, \kappa_2$. We know $\Sigma; \Psi \vDash_{\mathsf{ty}} \kappa : \mathbf{Type}$ and thus $\Sigma; \Psi \vDash_{\mathsf{co}} \langle \kappa \rangle : \kappa \sim \kappa$ as desired.

Case IFUN_CAST:

$$\frac{\begin{array}{ll}\mathsf{fresh}\ \iota & \mathsf{fresh}\ \beta_1, \beta_2 \\ \Omega = \beta_1{:}_{\mathsf{Irrel}} \mathbf{Type}, \beta_2{:}_{\mathsf{Irrel}} \mathbf{Type}, \iota{:}\kappa_0 \sim \underset{\sim}{\Pi}_{\mathsf{Req}} a{:}_{\rho} \beta_1.\, \beta_2 & \end{array}}{\vdash\!\!\!\rightarrow_{\mathsf{fun}} \kappa_0; \rho \rightsquigarrow \iota; \underset{\sim}{\Pi}; a; \rho; \beta_1; \beta_2 \dashv \Omega} \quad \textsc{IFun\_Cast}$$

Let $\Psi_0 = \Psi, \beta_1{:}_{\mathsf{Irrel}} \mathbf{Type}, \beta_2{:}_{\mathsf{Irrel}} \mathbf{Type}$ and $\Psi_1 = \Psi_0, \iota{:}\kappa_0 \sim \underset{\sim}{\Pi}_{\mathsf{Req}} a{:}_{\rho} \beta_1.\, \beta_2$. We first must show $\Sigma \vDash_{\mathsf{ctx}} \Psi'\ \mathsf{ok}$. We know $\Sigma \vDash_{\mathsf{ctx}} \Psi\ \mathsf{ok}$ by Lemma E.8. Adding β_1 and β_2 to Ψ maintains well-formedness; thus $\Sigma \vDash_{\mathsf{ctx}} \Psi_0\ \mathsf{ok}$. In order to add the binding for ι, we must show that $\Sigma; \mathsf{Rel}(\Psi_0) \vDash_{\mathsf{ty}} \kappa_0 : \mathbf{Type}$ and $\Sigma; \mathsf{Rel}(\Psi_0) \vDash_{\mathsf{ty}} \underset{\sim}{\Pi}_{\mathsf{Req}} a{:}_{\rho} \beta_1.\, \beta_2 : \mathbf{Type}$. The former is by assumption. The latter comes from $\Sigma \vDash_{\mathsf{ctx}} \Psi_0\ \mathsf{ok}$, two uses of TY_UVAR, and a use of TY_PI. Thus $\Sigma \vDash_{\mathsf{ctx}} \Psi_1\ \mathsf{ok}$ and $\Sigma; \Psi_1 \vDash_{\mathsf{co}} \iota : \kappa_0 \sim \underset{\sim}{\Pi}_{\mathsf{Req}} a{:}_{\rho} \beta_1.\, \beta_2$ as desired.

□

Lemma E.38 (Scrutinee position)**.** *If* $\Sigma; \Psi \vDash_{\mathsf{ty}} \tau : \kappa$ *and* $\Sigma; \Psi \vdash\!\!\!\rightarrow_{\mathsf{scrut}} \overline{\mathrm{alt}}; \kappa \rightsquigarrow \gamma; \Delta; H'; \overline{\tau} \dashv \Omega$*, then* $\Sigma; \Psi, \Omega \vDash_{\mathsf{ty}} \tau \rhd \gamma : \text{'}\Pi\Delta.\, H'\, \overline{\tau}$ *and* $\Sigma; \mathsf{Rel}(\Psi, \Omega) \vDash_{\mathsf{ty}} H'\, \overline{\tau} : \mathbf{Type}$.

Proof. By case analysis on the derivation for the $\vdash\!\!\!\rightarrow_{\mathsf{scrut}}$ judgment.

Case ISCRUT_ID:

$$\frac{\Sigma; \mathsf{Rel}(\Psi) \vDash_{\mathsf{ty}} H\, \overline{\tau} : \mathbf{Type}}{\Sigma; \Psi \vdash\!\!\!\rightarrow_{\mathsf{scrut}} \overline{\mathrm{alt}}; \text{'}\Pi\Delta.\, H\, \overline{\tau} \rightsquigarrow \langle \text{'}\Pi\Delta.\, H\, \overline{\tau} \rangle; \Delta; H; \overline{\tau} \dashv \varnothing} \quad \textsc{IScrut\_Id}$$

Let $\kappa = \text{'}\Pi\Delta.\, H\, \overline{\tau}$. Working backwards from a use of TY_CAST, we need to show that $\Sigma; \mathsf{Rel}(\Psi) \vDash_{\mathsf{co}} \langle \kappa \rangle : \kappa \sim \kappa$, and thus that $\Sigma; \mathsf{Rel}(\Psi) \vDash_{\mathsf{ty}} \kappa : \mathbf{Type}$. This comes directly from Lemma E.17. The second conclusion is assumed as a premise of ISCRUT_ID.

Case ISCRUT_CAST:

$$\frac{\begin{array}{l}\Sigma \vdash_{\mathsf{tc}} H : \overline{a}{:}_{\mathsf{Irrel}}\overline{\kappa}; \Delta_2; H' \\ \mathsf{fresh}\,\overline{\alpha} \qquad \mathsf{fresh}\,\iota \\ \Omega = \overline{\alpha}{:}_{\mathsf{Irrel}}\overline{\kappa}[\overline{\alpha}/\overline{a}], \iota{:}\kappa \sim H'\,\overline{\alpha}\end{array}}{\Sigma; \Psi \vdash_{\mathsf{scrut}} (H\,\overline{x} \to \mathrm{t}; \overline{\mathrm{alt}}); \kappa \rightsquigarrow \iota; \varnothing; H'; \overline{\alpha} \dashv \Omega} \quad \text{ISCRUT\_CAST}$$

Let $\Psi_0 = \Psi, \overline{\alpha}{:}_{\mathsf{Irrel}}\overline{\kappa}[\overline{\alpha}/\overline{a}]$ and $\Psi_1 = \Psi_0, \iota{:}\kappa \sim H'\,\overline{\alpha}$. We must first show that $\Sigma \vDash_{\mathsf{ctx}} \Psi_0$ ok. We know $\vdash_{\mathsf{sig}} \Sigma$ ok (by Lemma E.8). Lemma C.40 tells us $\Sigma \vdash_{\mathsf{ctx}} \overline{a}{:}_{\mathsf{Irrel}}\overline{\kappa}$ ok. Lemma E.27 and Lemma E.3 then tell us $\Sigma \vDash_{\mathsf{ctx}} \overline{\alpha}{:}_{\mathsf{Irrel}}\overline{\kappa}[\overline{\alpha}/\overline{a}]$ ok. We have $\Sigma \vDash_{\mathsf{ctx}} \Psi$ ok by Lemma E.8 and thus can use Lemma E.9 $\Sigma \vDash_{\mathsf{ctx}} \Psi_0$ ok as desired. To show $\Sigma \vDash_{\mathsf{ctx}} \Psi_1$ ok, we must now show that $\Sigma; \Psi_0 \vDash_{\mathsf{ty}} \kappa : \mathbf{Type}$ and $\Sigma; \Psi_0 \vDash_{\mathsf{ty}} H'\,\overline{\alpha} : \mathbf{Type}$. The former is by Lemma E.17 and Lemma E.9. For the latter: use Lemma C.41 and Lemma E.9 to see that $\Sigma; \Psi_0 \vDash_{\mathsf{ty}} \Pi\overline{a}{:}_{\mathsf{Irrel}}\overline{\kappa}, \Delta_2.\, H'\,\overline{a} : \mathbf{Type}$. Repeated inversion on TY_PI tells us $\Sigma; \Psi_0, \overline{a}{:}_{\mathsf{Irrel}}\overline{\kappa}, \Delta_2 \vDash_{\mathsf{ty}} H'\,\overline{a} : \mathbf{Type}$. Lemma E.10 gives us $\Sigma; \Psi_0, \overline{a}{:}_{\mathsf{Irrel}}\overline{\kappa} \vDash_{\mathsf{ty}} H'\,\overline{a} : \mathbf{Type}$. Lemma C.39 tells us that $\Sigma; \Psi_0 \vDash_{\mathsf{vec}} \overline{\alpha} : (\overline{\alpha}{:}_{\mathsf{Irrel}}\overline{\kappa}[\overline{\alpha}/\overline{a}])$. We thus use Lemma E.15 to see that $\Sigma; \Psi_0 \vDash_{\mathsf{ty}} H'\,\overline{\alpha} : \mathbf{Type}$ as desired. We can thus conclude $\Sigma \vDash_{\mathsf{ctx}} \Psi_1$ ok by CTX_UCOVAR. We are done with the first conclusion by TY_CAST and CO_VAR. We get the second conclusion easily by noting that $\Delta = \varnothing$ and by Lemma E.17.

□

Lemma E.39 (make_exhaustive)**.** *Assume that,* $\forall i$, $\Sigma; \Psi; \Pi\Delta.\, H\,\overline{\sigma} \vDash^{\mathsf{r}}_{\mathsf{alt}} alt_i : \kappa$ *and* $\overline{alt}' = \mathsf{make\_exhaustive}(\overline{alt}; \kappa)$*. Furthermore, assume no pattern appears twice in* $\overline{alt}$*. Then* $\forall j$, $\Sigma; \Psi; \Pi\Delta.\, H\,\overline{\sigma} \vDash^{\mathsf{r}}_{\mathsf{alt}} alt'_j : \kappa$ *and* $\overline{alt}'$ *are exhaustive and distinct for* H*, (w.r.t.* Σ*).*

Proof. If there is a default pattern in $\overline{alt}$, then make_exhaustive does nothing. In this case, the default pattern makes the $\overline{alt}$ exhaustive. We have already assumed they are unique.

Otherwise, make_exhaustive adds a default. Assuming $error{:}_{\mathsf{Rel}}\tilde{\Pi}(a{:}_{\mathsf{Irrel}}\mathbf{Type}), (b{:}_{\mathsf{Rel}}String).\, a$, we have $\forall j$, $\Sigma; \Psi; \Pi\Delta.\, H\,\overline{\sigma} \vDash^{\mathsf{r}}_{\mathsf{alt}} alt'_j : \kappa$, and indeed the alternatives are now exhaustive. □

Lemma E.40 (Prenex)**.** *If* $\Sigma; \mathsf{Rel}(\Psi) \vDash_{\mathsf{ty}} \kappa : \mathbf{Type}$ *and* $\vdash_{\mathsf{pre}} \kappa \rightsquigarrow \Delta; \kappa'; \tau$*, then* $\Sigma; \Psi \vDash_{\mathsf{ty}} \tau : \tilde{\Pi}x{:}_{\mathsf{Rel}}(\tilde{\Pi}\Delta.\, \kappa').\, \kappa$.

Proof. By induction on the $\vdash_{\mathsf{pre}}$ judgment.

Case IPRENEX_INVIS:

$$\frac{\begin{array}{c}\nu \leq \mathsf{Spec} \\ \vdash_{\mathsf{pre}} \kappa_2 \rightsquigarrow \Delta; \kappa'_2; \tau\end{array}}{\vdash_{\mathsf{pre}} \tilde{\Pi}_\nu \delta.\, \kappa_2 \rightsquigarrow \delta, \Delta; \kappa'_2; \lambda(x{:}_{\mathsf{Rel}}\tilde{\Pi}\delta, \Delta.\, \kappa'_2), \delta.\, \tau\,(x\,\mathsf{dom}(\delta))} \quad \text{IPRENEX\_INVIS}$$

We know $\Sigma; \mathsf{Rel}(\Psi) \models_{\mathsf{ty}} \tilde{\Pi}_\nu \delta.\, \kappa_2 : \mathbf{Type}$. Inversion gives us $\Sigma; \mathsf{Rel}(\Psi, \delta) \models_{\mathsf{ty}} \kappa_2 : \mathbf{Type}$. The induction hypothesis thus tells us that $\Sigma; \Psi, \delta \models_{\mathsf{ty}} \tau : \tilde{\Pi} x_2{:}_{\mathsf{Rel}} (\tilde{\Pi} \Delta.\, \kappa_2').\, \kappa_2$. Let $\Psi' = \Psi, x{:}_{\mathsf{Rel}} (\tilde{\Pi} \delta, \Delta.\, \kappa_2'), \delta$. We need $\Sigma \models_{\mathsf{ctx}} \Psi'\ \mathsf{ok}$, for which we need $\Sigma; \mathsf{Rel}(\Psi) \models_{\mathsf{ty}} \tilde{\Pi} \delta, \Delta.\, \kappa_2' : \mathbf{Type}$, which can be proved by inversions and Ty_Pi. We thus have $\Sigma \models_{\mathsf{ctx}} \Psi'\ \mathsf{ok}$. We now show that $\Sigma; \Psi' \models_{\mathsf{ty}} \tau\, (x\, \mathsf{dom}(\delta)) : \kappa_2$. First, we note that $\Sigma; \Psi' \models_{\mathsf{ty}} x\, \mathsf{dom}(\delta) : \tilde{\Pi} \Delta.\, \kappa_2'$ by the appropriate application rule. (It depends on the relevance of δ.) There is no substitution in the kind, because we are applying to $\mathsf{dom}(\delta)$. Thus $\Sigma; \Psi' \models_{\mathsf{ty}} \tau\, (x\, \mathsf{dom}(\delta)) : \kappa_2[x\, \mathsf{dom}(\delta)/x_2]$ by Ty_AppRel. However, we know $x_2 \# \kappa_2$ by Lemma E.11 and so we are done by two uses of Ty_Lam.

Case IPrenex_Vis:

$$\frac{\begin{array}{l} \Rightarrow_{\mathsf{pre}} \kappa_2 \rightsquigarrow \Delta; \kappa_2'; \tau \\ \tau_0 = \lambda (x{:}_{\mathsf{Rel}} \tilde{\Pi} \Delta, \delta.\, \kappa_2'), \delta.\, \tau\, (\lambda \Delta.\, x\, \mathsf{dom}(\Delta)\, \mathsf{dom}(\delta)) \end{array}}{\Rightarrow_{\mathsf{pre}} \tilde{\Pi}_{\mathsf{Req}} \delta.\, \kappa_2 \rightsquigarrow \Delta; \tilde{\Pi}_{\mathsf{Req}} \delta.\, \kappa_2'; \tau_0} \quad \text{IPrenex\_Vis}$$

We know $\Sigma; \mathsf{Rel}(\Psi) \models_{\mathsf{ty}} \tilde{\Pi}_{\mathsf{Req}} \delta.\, \kappa_2 : \mathbf{Type}$. Inversion gives us $\Sigma; \mathsf{Rel}(\Psi, \delta) \models_{\mathsf{ty}} \kappa_2 : \mathbf{Type}$. The induction hypothesis then gives us $\Sigma; \Psi, \delta \models_{\mathsf{ty}} \tau : \tilde{\Pi} x_2{:}_{\mathsf{Rel}} (\tilde{\Pi} \Delta.\, \kappa_2').\, \kappa_2$. Let $\Psi' = \Psi, x{:}_{\mathsf{Rel}} (\tilde{\Pi} \Delta, \delta.\, \kappa_2'), \delta$. We need $\Sigma \models_{\mathsf{ctx}} \Psi'\ \mathsf{ok}$, for which we need $\Sigma; \mathsf{Rel}(\Psi) \models_{\mathsf{ty}} \tilde{\Pi} \Delta, \delta.\, \kappa_2' : \mathbf{Type}$. This can be proved by inversions and Ty_Pi. We thus have $\Sigma \models_{\mathsf{ctx}} \Psi'\ \mathsf{ok}$. We now show that $\Sigma; \Psi' \models_{\mathsf{ty}} \tau\, (\lambda \Delta.\, x\, \mathsf{dom}(\Delta)\, \mathsf{dom}(\delta)) : \kappa_2$. First, we show that $\Sigma; \Psi', \Delta \models_{\mathsf{ty}} x\, \mathsf{dom}(\Delta)\, \mathsf{dom}(\delta) : \kappa_2'$. Once we show that $\Sigma \models_{\mathsf{ctx}} \Psi', \Delta\ \mathsf{ok}$ (as can be shown by inversions, Lemma E.8, and Lemma E.9), then this comes directly from the type of x. Thus, we can conclude, by repeated use ofTy_Lam, that $\Sigma; \Psi' \models_{\mathsf{ty}} \lambda \Delta.\, x\, \mathsf{dom}(\Delta)\, \mathsf{dom}(\delta) : \tilde{\Pi} \Delta.\, \kappa_2'$. Accordingly, $\Sigma; \Psi' \models_{\mathsf{ty}} \tau\, (\lambda \Delta.\, x\, \mathsf{dom}(\Delta)\, \mathsf{dom}(\delta)) : \kappa_2[(\lambda \Delta.\, x\, \mathsf{dom}(\Delta)\, \mathsf{dom}(\delta))/x_2]$, but the substitution in the kind has no effect by Lemma E.11. We thus have $\Sigma; \Psi' \models_{\mathsf{ty}} \tau\, (\lambda \Delta.\, x\, \mathsf{dom}(\Delta)\, \mathsf{dom}(\delta)) : \kappa_2$. We are done by several uses of Ty_Lam.

Case IPrenex_NoPi:

$$\frac{}{\Rightarrow_{\mathsf{pre}} \kappa \rightsquigarrow \varnothing; \kappa; \lambda x{:}_{\mathsf{Rel}} \kappa.\, x} \quad \text{IPrenex\_NoPi}$$

Assuming $\Sigma; \mathsf{Rel}(\Psi) \models_{\mathsf{ty}} \kappa : \mathbf{Type}$, we must show $\Sigma; \Psi \models_{\mathsf{ty}} \lambda x{:}_{\mathsf{Rel}} \kappa.\, x : \tilde{\Pi} x{:}_{\mathsf{Rel}} \kappa.\, \kappa$. This is true by straightforward application of typing rules.

□

Lemma E.41 (Subsumption)**.** *Assume* $\Sigma; \mathsf{Rel}(\Psi) \models_{\mathsf{ty}} \kappa_1 : \mathbf{Type}$ *and* $\Sigma; \mathsf{Rel}(\Psi) \models_{\mathsf{ty}} \kappa_2 : \mathbf{Type}$. *If either*

1. $\kappa_1 \leq^* \kappa_2 \rightsquigarrow \tau \dashv \Omega$, *OR*
2. $\kappa_1 \leq \kappa_2 \rightsquigarrow \tau \dashv \Omega$

Then $\Sigma; \Psi, \Omega \vDash_{\mathsf{ty}} \tau : \tilde{\Pi} x{:}_{\mathsf{Rel}} \kappa_1.\, \kappa_2$.

Proof. By mutual induction on the subsumption judgments.

Case ISub_FunRel:

$$\frac{\begin{array}{l}\kappa_3 \leq \kappa_1 \rightsquigarrow \tau_1 \dashv \Omega_1 \qquad \kappa_2[\tau_1\ b/a] \leq \kappa_4 \rightsquigarrow \tau_2 \dashv \Omega_2 \\ \Omega_2 \hookrightarrow b{:}_{\mathsf{Rel}} \kappa_3 \rightsquigarrow \Omega_2'; \xi \\ \tau_0 \;=\; \lambda x{:}_{\mathsf{Rel}} (\Pi a{:}_{\mathsf{Rel}} \kappa_1.\, \kappa_2), b{:}_{\mathsf{Rel}} \kappa_3.\, \tau_2[\xi]\ (x\ (\tau_1\ b))\end{array}}{\Pi_{\mathsf{Req}}\, a{:}_{\mathsf{Rel}} \kappa_1.\, \kappa_2 \leq^* \tilde{\Pi}_{\mathsf{Req}}\, b{:}_{\mathsf{Rel}} \kappa_3.\, \kappa_4 \rightsquigarrow \tau_0 \dashv \Omega_1, \Omega_2'} \quad \textsc{ISub\_FunRel}$$

Our assumption says that $\Sigma; \mathsf{Rel}(\Psi) \vDash_{\mathsf{ty}} \Pi a{:}_{\mathsf{Rel}} \kappa_1.\, \kappa_2 : \mathbf{Type}$ and $\Sigma; \mathsf{Rel}(\Psi) \vDash_{\mathsf{ty}} \tilde{\Pi} a{:}_{\mathsf{Rel}} \kappa_3.\, \kappa_4 : \mathbf{Type}$. Inversion of Ty_Pi tells us the following:

- $\Sigma; \mathsf{Rel}(\Psi) \vDash_{\mathsf{ty}} \kappa_1 : \mathbf{Type}$
- $\Sigma; \mathsf{Rel}(\Psi), a{:}_{\mathsf{Rel}} \kappa_1 \vDash_{\mathsf{ty}} \kappa_2 : \mathbf{Type}$
- $\Sigma; \mathsf{Rel}(\Psi) \vDash_{\mathsf{ty}} \kappa_3 : \mathbf{Type}$
- $\Sigma; \mathsf{Rel}(\Psi), b{:}_{\mathsf{Rel}} \kappa_3 \vDash_{\mathsf{ty}} \kappa_4 : \mathbf{Type}$

The induction hypothesis then tells us $\Sigma; \Psi, \Omega_1 \vDash_{\mathsf{ty}} \tau_1 : \tilde{\Pi} x_1{:}_{\mathsf{Rel}} \kappa_3.\, \kappa_1$. Lemma E.9 gives us $\Sigma; \mathsf{Rel}(\Psi, \Omega_1), b{:}_{\mathsf{Rel}} \kappa_3, a{:}_{\mathsf{Rel}} \kappa_1 \vDash_{\mathsf{ty}} \kappa_2 : \mathbf{Type}$. Rule Ty_AppRel tells us $\Sigma; \Psi, \Omega_1, b{:}_{\mathsf{Rel}} \kappa_3 \vDash_{\mathsf{ty}} \tau_1\ b : \kappa_1[b/x]$, but Lemma E.11 tells us that the substitution in the kind has no effect. We can thus use Lemma E.13 to get $\Sigma; \mathsf{Rel}(\Psi, \Omega_1), b{:}_{\mathsf{Rel}} \kappa_3 \vDash_{\mathsf{ty}} \kappa_2[\tau_1\ b/a] : \mathbf{Type}$. Now, we can use the induction hypothesis again to get $\Sigma; \Psi, \Omega_1, b{:}_{\mathsf{Rel}} \kappa_3, \Omega_2 \vDash_{\mathsf{ty}} \tau_2 : \tilde{\Pi} x_2{:}_{\mathsf{Rel}} \kappa_2[\tau_1\ b/a].\, \kappa_4$. Lemma E.35 tells us now that $\Sigma; \Psi, \Omega_1, \Omega_2', b{:}_{\mathsf{Rel}} \kappa_3 \vDash_{\mathsf{ty}} \tau_2[\xi] : (\tilde{\Pi} x_2{:}_{\mathsf{Rel}} \kappa_2[\tau_1\ b/a].\, \kappa_4)[\xi]$, but Lemma E.34 tells us the $[\xi]$ in the kind has no effect. Let

$$\Psi' \;=\; \Psi, \Omega_1, \Omega_2', x{:}_{\mathsf{Rel}} (\Pi a{:}_{\mathsf{Rel}} \kappa_1.\, \kappa_2), b{:}_{\mathsf{Rel}} \kappa_3.$$

To show $\Sigma \vDash_{\mathsf{ctx}} \Psi'\ \mathsf{ok}$, we need only show that $\Sigma; \mathsf{Rel}(\Psi, \Omega_1), \Omega_2' \vDash_{\mathsf{ty}} \Pi a{:}_{\mathsf{Rel}} \kappa_1.\, \kappa_2 : \mathbf{Type}$ (noting that Lemma E.35 and Lemma E.8 imply $\Sigma \vDash_{\mathsf{ctx}} \Psi, \Omega_1, \Omega_2'\ \mathsf{ok}$), but this is true by Lemma E.9. We must now show $\Sigma; \Psi' \vDash_{\mathsf{ty}} \tau_2[\xi]\ (x\ (\tau_1\ b)) : \kappa_4$. We've already ascertained that $\Sigma; \Psi' \vDash_{\mathsf{ty}} \tau_1\ b : \kappa_1$. We see that $\Sigma; \Psi' \vDash_{\mathsf{ty}} x\ (\tau_1\ b) : \kappa_2[\tau_1\ b/a]$. Thus $\Sigma; \Psi' \vDash_{\mathsf{ty}} \tau_2[\xi]\ (x\ (\tau_1\ b)) : \kappa_4[x\ (\tau_1\ b)/x_2]$, but Lemma E.11 tells us that the substitution in the kind has no effect. We are thus done by two uses of Ty_Lam.

Case ISub_FunIrrelRel: Similar to previous case. Note that b can be used irrelevantly even though it is bound relevantly. The opposite way would not work.

Case ISub_FunIrrel: Similar to previous case.

Case ISUB_UNIFY:

$$\frac{\mathsf{fresh}\ \iota}{\tau_1 \leq^* \tau_2 \rightsquigarrow \lambda x{:}_{\mathsf{Rel}}\tau_1.\,(x \rhd \iota) \dashv \iota{:}\tau_1 \sim \tau_2}\ \textsc{ISub\_Unify}$$

We must show that $\Sigma; \Psi, \iota{:}\tau_1 \sim \tau_2 \vDash_{\mathsf{ty}} \lambda x{:}_{\mathsf{Rel}}\tau_1.\,(x \rhd \iota) : \tilde{\Pi} x{:}_{\mathsf{Rel}}\tau_1.\,\tau_2$. Our last step will be TY_LAM and thus we must show $\Sigma; \Psi, \iota{:}\tau_1 \sim \tau_2, x{:}_{\mathsf{Rel}}\tau_1 \vDash_{\mathsf{ty}} x \rhd \iota : \tau_2$, for which we only need show that $\Sigma \vDash_{\mathsf{ctx}} \Psi, \iota{:}\tau_1 \sim \tau_2\ \mathsf{ok}$, for which we only need show that $\Sigma; \mathsf{Rel}(\Psi) \vDash_{\mathsf{ty}} \tau_1 : \mathbf{Type}$ and $\Sigma; \mathsf{Rel}(\Psi) \vDash_{\mathsf{ty}} \tau_2 : \mathbf{Type}$, which we know by assumption. We are done.

Case ISUB_DEEPSKOL:

$$\frac{\begin{array}{l}\vDash^{\rightarrow}_{\mathsf{pre}} \kappa_2 \rightsquigarrow \Delta; \kappa_2'; \tau_1 \\ \vDash^{\mathsf{Spec}}_{\mathsf{inst}} \kappa_1 \rightsquigarrow \overline{\psi}; \kappa_1' \dashv \Omega_1 \\ \kappa_1' \leq^* \kappa_2' \rightsquigarrow \tau_2 \dashv \Omega_2 \\ \Omega_1, \Omega_2 \hookrightarrow \Delta \rightsquigarrow \Omega'; \xi\end{array}}{\kappa_1 \leq \kappa_2 \rightsquigarrow \lambda x{:}_{\mathsf{Rel}}\kappa_1.\,\tau_1\,(\lambda\Delta.\,\tau_2[\xi]\,(x\,\overline{\psi}[\xi])) \dashv \Omega'}\ \textsc{ISub\_DeepSkol}$$

We must show $\Sigma; \Psi, \Omega' \vDash_{\mathsf{ty}} \lambda x{:}_{\mathsf{Rel}}\kappa_1.\,\tau_1\,(\lambda\Delta.\,\tau_2[\xi]\,(x\,\overline{\psi}[\xi])) : \tilde{\Pi} x{:}_{\mathsf{Rel}}\kappa_1.\,\kappa_2$. The last step will be TY_LAM, so we must show $\Sigma; \Psi, \Omega', x{:}_{\mathsf{Rel}}\kappa_1 \vDash_{\mathsf{ty}} \tau_1\,(\lambda\Delta.\,\tau_2[\xi]\,(x\,\overline{\psi}[\xi])) : \kappa_2$. From $\Sigma; \mathsf{Rel}(\Psi) \vDash_{\mathsf{ty}} \kappa_1 : \mathbf{Type}$, we can use CTX_TYVAR to see $\Sigma \vDash_{\mathsf{ctx}} \Psi, x{:}_{\mathsf{Rel}}\kappa_1\ \mathsf{ok}$. Thus $\Sigma; \Psi, x{:}_{\mathsf{Rel}}\kappa_1 \vDash_{\mathsf{ty}} x : \kappa_1$. Lemma E.36 then tells us that $\Sigma; \Psi, x{:}_{\mathsf{Rel}}\kappa_1, \Omega_1 \vDash_{\mathsf{ty}} x\,\overline{\psi} : \kappa_1'$. We then know (by Lemma E.17) that $\Sigma; \mathsf{Rel}(\Psi, x{:}_{\mathsf{Rel}}\kappa_1, \Omega_1) \vDash_{\mathsf{ty}} \kappa_1' : \mathbf{Type}$. Lemma E.40 tells us that $\Sigma; \Psi \vDash_{\mathsf{ty}} \tau_1 : \tilde{\Pi} x_1{:}_{\mathsf{Rel}}(\tilde{\Pi}\Delta.\,\kappa_2').\,\kappa_2$. Lemma E.17 and inversion gives $\Sigma; \mathsf{Rel}(\Psi, \Delta) \vDash_{\mathsf{ty}} \kappa_2' : \mathbf{Type}$. We can then use the induction hypothesis with context $\Psi, x{:}_{\mathsf{Rel}}\kappa_1, \Delta, \Omega_1$ (known well-formed by Lemma E.9) to get $\Sigma; \Psi, x{:}_{\mathsf{Rel}}\kappa_1, \Delta, \Omega_1, \Omega_2 \vDash_{\mathsf{ty}} \tau_2 : \tilde{\Pi} x_2{:}_{\mathsf{Rel}}\kappa_1'.\,\kappa_2'$. Lemma E.35 shows that $\Sigma; \Psi, x{:}_{\mathsf{Rel}}\kappa_1, \Omega', \Delta \vDash_{\mathsf{ty}} \tau_2[\xi] : (\tilde{\Pi} x_2{:}_{\mathsf{Rel}}\kappa_1'.\,\kappa_2')[\xi]$. The kind can be rewritten to $\tilde{\Pi} x_2{:}_{\mathsf{Rel}}(\kappa_1'[\xi]).\,\kappa_2'[\xi]$ but Lemma E.34 tells us that $\kappa_2'[\xi] = \kappa_2'$. We established earlier that $\Sigma; \Psi, x{:}_{\mathsf{Rel}}\kappa_1, \Omega_1 \vDash_{\mathsf{ty}} x\,\overline{\psi} : \kappa_1'$. We can weaken this to $\Sigma; \Psi, x{:}_{\mathsf{Rel}}\kappa_1, \Delta, \Omega_1, \Omega_2 \vDash_{\mathsf{ty}} x\,\overline{\psi} : \kappa_1'$ and then use Lemma E.35 to get $\Sigma; \Psi, x{:}_{\mathsf{Rel}}\kappa_1, \Omega', \Delta \vDash_{\mathsf{ty}} (x\,\overline{\psi})[\xi] : \kappa_1'[\xi]$. We know that $x[\xi] = x$ because x is just a non-unification variable. Rule TY_APPREL thus gives us $\Sigma; \Psi, x{:}_{\mathsf{Rel}}\kappa_1, \Omega', \Delta \vDash_{\mathsf{ty}} \tau_2[\xi]\,(x\,\overline{\psi}[\xi]) : \kappa_2'[x\,\overline{\psi}[\xi]/x_2]$ but Lemma E.11 tells us that the substitution in the kind has no effect. We now use TY_LAM (repeatedly) to see $\Sigma; \Psi, x{:}_{\mathsf{Rel}}\kappa_1, \Omega' \vDash_{\mathsf{ty}} \lambda\Delta.\,\tau_2[\xi]\,(x\,\overline{\psi}[\xi]) : \tilde{\Pi}\Delta.\,\kappa_2'$. Thus TY_APPREL tells us $\Sigma; \Psi, x{:}_{\mathsf{Rel}}\kappa_1, \Omega' \vDash_{\mathsf{ty}} \tau_1\,(\lambda\Delta.\,\tau_2[\xi]\,(x\,\overline{\psi}[\xi])) : \kappa_2[(\lambda\Delta.\,\tau_2[\xi]\,(x\,\overline{\psi}[\xi]))/x_1]$, but Lemma E.11 tells us that the substitution in the kind has no effect. We only need to reshuffle the context; in other words, we must now show $\Sigma \vDash_{\mathsf{ctx}} \Psi, \Omega', x{:}_{\mathsf{Rel}}\kappa_1\ \mathsf{ok}$ to be done. For this to hold, we need to know that none of Ω' depend on x. First, note that x is local to rule ISUB_DEEPSKOL. We see that Δ is produced by $\vDash^{\rightarrow}_{\mathsf{pre}}$ with no mention of x, Ω_1 is produced by $\vDash^{\rightarrow}_{\mathsf{inst}}$ with no mention of x, and Ω_2 is

produced by $\leq^*$ with no mention of x. Therefore, x is not mentioned in any of these, and we are done.

□

Lemma E.42 (Type elaboration is sound)**.**

1. *If any of the following:*
   (a) $\Sigma \vDash_{\mathsf{ctx}} \Psi\ \mathsf{ok}$ *and* $\Sigma; \Psi \vdash_{\mathsf{ty}} \mathrm{t} \rightsquigarrow \tau : \kappa \dashv \Omega$*, OR*
   (b) $\Sigma \vDash_{\mathsf{ctx}} \Psi\ \mathsf{ok}$ *and* $\Sigma; \Psi \vdash^{*}_{\mathsf{ty}} \mathrm{t} \rightsquigarrow \tau : \kappa \dashv \Omega$*, OR*
   (c) $\Sigma; \mathsf{Rel}(\Psi) \vDash_{\mathsf{ty}} \kappa : \mathbf{Type}$ *and* $\Sigma; \Psi \vdash_{\mathsf{ty}} \mathrm{t} : \kappa \rightsquigarrow \tau \dashv \Omega$*, OR*
   (d) $\Sigma; \mathsf{Rel}(\Psi) \vDash_{\mathsf{ty}} \kappa : \mathbf{Type}$ *and* $\Sigma; \Psi \vdash^{*}_{\mathsf{ty}} \mathrm{t} : \kappa \rightsquigarrow \tau \dashv \Omega$

   Then $\Sigma; \Psi, \Omega \vDash_{\mathsf{ty}} \tau : \kappa$*.*
2. *If* $\Sigma \vDash_{\mathsf{ctx}} \Psi\ \mathsf{ok}$ *and* $\Sigma; \Psi \vdash_{\mathsf{pt}} \mathrm{s} \rightsquigarrow \sigma \dashv \Omega$*, then* $\Sigma; \mathsf{Rel}(\Psi, \Omega) \vDash_{\mathsf{ty}} \sigma : \mathbf{Type}$*.*
3. *If* $\Sigma; \Psi \vDash_{\mathsf{ty}} \tau_1 : \Pi_\nu a{:}_\rho \kappa_1.\, \kappa_2$ *and* $\Sigma; \Psi; \rho \vdash^{*}_{\mathsf{arg}} \mathrm{t}_2 : \kappa_1 \rightsquigarrow \psi_2; \tau_2 \dashv \Omega$*, then* $\Sigma; \Psi, \Omega \vDash_{\mathsf{ty}} \tau_1\, \psi_2 : \kappa_2[\tau_2/a]$*.*
4. *If* $\Sigma; \mathsf{Rel}(\Psi) \vDash_{\mathsf{ty}} \kappa : \mathbf{Type}$*,* $\Sigma; \Psi \vDash_{\mathsf{ty}} \tau_0 : \Pi\Delta.\, H\,\overline{\tau}$*,* $\Sigma; \mathsf{Rel}(\Psi) \vDash_{\mathsf{ty}} H\,\overline{\tau} : \mathbf{Type}$*, and* $\Sigma; \Psi;\ \Pi\Delta.\, H\,\overline{\tau}; \tau_0 \vdash_{\mathsf{alt}} \mathrm{alt} : \kappa \rightsquigarrow \mathit{alt} \dashv \Omega$*, then* $\Sigma; \Psi, \Omega;\ \Pi\Delta.\, H\,\overline{\tau} \vdash^{\tau_0}_{\mathsf{alt}} \mathit{alt} : \kappa$*.*
5. *If* $\Sigma; \mathsf{Rel}(\Psi) \vDash_{\mathsf{ty}} \kappa : \mathbf{Type}$*,* $\Sigma; \Psi \vDash_{\mathsf{ty}} \tau_0 : \Pi\Delta.\, H\,\overline{\tau}$*,* $\Sigma; \mathsf{Rel}(\Psi) \vDash_{\mathsf{ty}} H\,\overline{\tau} : \mathbf{Type}$*, and* $\Sigma; \Psi; \kappa_0; \tau_0 \vdash_{\mathsf{altc}} \mathrm{alt} : \kappa \rightsquigarrow \mathit{alt} \dashv \Omega$*, then* $\Sigma; \Psi, \Omega; \kappa_0 \vdash^{\tau_0}_{\mathsf{alt}} \mathit{alt} : \kappa$*.*
6. *If* $\Sigma \vDash_{\mathsf{ctx}} \Psi\ \mathsf{ok}$ *and* $\Sigma; \Psi \vdash_{\mathsf{q}} \mathrm{qvar} \rightsquigarrow a : \kappa; \nu \dashv \Omega$*, then* $\Sigma; \mathsf{Rel}(\Psi, \Omega) \vDash_{\mathsf{ty}} \kappa : \mathbf{Type}$*.*
7. *If* $\Sigma \vDash_{\mathsf{ctx}} \Psi\ \mathsf{ok}$ *and* $\Sigma; \Psi \vdash_{\mathsf{aq}} \mathrm{aqvar} \rightsquigarrow a : \kappa \dashv \Omega$*, then* $\Sigma; \mathsf{Rel}(\Psi, \Omega) \vDash_{\mathsf{ty}} \kappa : \mathbf{Type}$*.*
8. *If* $\Sigma; \Psi \vDash_{\mathsf{ty}} \tau_0 : \kappa$ *and* $\Sigma; \Psi \vdash_{\mathsf{aq}} \mathrm{aqvar} : \kappa \rightsquigarrow a : \kappa'; x.\tau \dashv \Omega$*, then* $\Sigma; \Psi, \Omega \vDash_{\mathsf{ty}} \tau[\tau_0/x] : \kappa'$*.*

Proof. Proceed by induction on the structure of the type inference derivation.

Case ITy_Inst:

$$\frac{\begin{array}{l}\Sigma; \Psi \vdash^{*}_{\mathsf{ty}} \mathrm{t} \rightsquigarrow \tau : \kappa \dashv \Omega_1 \\ \vdash^{\mathsf{Spec}}_{\mathsf{inst}} \kappa \rightsquigarrow \overline{\psi}; \kappa' \dashv \Omega_2\end{array}}{\Sigma; \Psi \vdash_{\mathsf{ty}} \mathrm{t} \rightsquigarrow \tau\,\overline{\psi} : \kappa' \dashv \Omega_1, \Omega_2}\ \textsc{ITy\_Inst}$$

The induction hypothesis gives us $\Sigma; \Psi, \Omega_1 \vDash_{\mathsf{ty}} \tau : \kappa$. Lemma E.36 then gives us $\Sigma; \Psi, \Omega_1, \Omega_2 \vDash_{\mathsf{ty}} \tau\,\overline{\psi} : \kappa'$ as desired.

Case ITy_Var: By Ty_Var and Lemma E.36.

Case ITy_App:

$$\frac{\begin{array}{l}\Sigma; \Psi \uparrow^{\mathsf{ty}} \mathrm{t}_1 \rightsquigarrow \tau_1 : \kappa_0 \dashv \Omega_1 \\ \uparrow^{\mathsf{fun}} \kappa_0; \mathsf{Rel} \rightsquigarrow \gamma; \Pi; a; \rho; \kappa_1; \kappa_2 \dashv \Omega_2 \\ \Sigma; \Psi, \Omega_1, \Omega_2; \rho \uparrow^{*}_{\mathsf{arg}} \mathrm{t}_2 : \kappa_1 \rightsquigarrow \psi_2; \tau_2 \dashv \Omega_3\end{array}}{\Sigma; \Psi \uparrow^{*}_{\mathsf{ty}} \mathrm{t}_1\, \mathrm{t}_2 \rightsquigarrow (\tau_1 \rhd \gamma)\, \psi_2 : \kappa_2[\tau_2/a] \dashv \Omega_1, \Omega_2, \Omega_3} \quad \textsc{ITy\_App}$$

The induction hypothesis tells us that $\Sigma; \Psi, \Omega_1 \Downarrow_{\mathsf{ty}} \tau_1 : \kappa_0$. Thus $\Sigma; \mathsf{Rel}(\Psi, \Omega_1) \Downarrow_{\mathsf{ty}} \kappa_0 : \mathbf{Type}$ by Lemma E.17. Lemma E.37 tells us that $\Sigma; \mathsf{Rel}(\Psi, \Omega_1, \Omega_2) \Downarrow_{\mathsf{co}} \gamma : \kappa_0 \sim \Pi_{\mathsf{Req}} a{:}_{\rho}\kappa_1.\, \kappa_2$. Rule Ty_Cast gives us $\Sigma; \Psi, \Omega_1, \Omega_2 \Downarrow_{\mathsf{ty}} \tau_1 \rhd \gamma : \Pi_{\mathsf{Req}} a{:}_{\rho}\kappa_1.\, \kappa_2$. Another use of the induction hypothesis (for the $\uparrow^{*}_{\mathsf{arg}}$ premise) gives us our desired outcome.

Case ITy_AppSpec: By induction.

Case ITy_Annot: By induction.

Case ITy_Case:

$$\frac{\begin{array}{l}\Sigma; \Psi \uparrow^{\mathsf{ty}} \mathrm{t}_0 \rightsquigarrow \tau_0 : \kappa_0 \dashv \Omega_0 \\ \Sigma; \Psi, \Omega_0 \uparrow^{\mathsf{scrut}} \overline{\mathrm{alt}}; \kappa_0 \rightsquigarrow \gamma; \Delta; H'; \overline{\tau} \dashv \Omega_0' \\ \mathsf{fresh}\, \alpha \qquad \Omega' = \Omega_0, \Omega_0', \alpha{:}_{\mathsf{Irrel}}\mathbf{Type} \\ \forall i,\ \Sigma; \Psi, \Omega'; \text{'}\Pi\Delta.\, H'\, \overline{\tau}; \tau_0 \rhd \gamma \uparrow^{\mathsf{alt}} \mathrm{alt}_i : \alpha \rightsquigarrow alt_i \dashv \Omega_i \\ \overline{alt}' = \mathsf{make\_exhaustive}(\overline{alt}; \kappa)\end{array}}{\Sigma; \Psi \uparrow^{*}_{\mathsf{ty}} \mathbf{case}\, \mathrm{t}_0\, \mathbf{of}\, \overline{\mathrm{alt}} \rightsquigarrow \mathbf{case}_{\alpha}\, (\tau_0 \rhd \gamma)\, \mathbf{of}\, \overline{alt}' : \alpha \dashv \Omega', \overline{\Omega}} \quad \textsc{ITy\_Case}$$

The induction hypothesis tells us that $\Sigma; \Psi, \Omega_0 \Downarrow_{\mathsf{ty}} \tau_0 : \kappa_0$. Lemma E.38 tells us that $\Sigma; \Psi, \Omega_0, \Omega_0' \Downarrow_{\mathsf{ty}} \tau_0 \rhd \gamma : \text{'}\Pi\Delta.\, H'\, \overline{\tau}$ and $\Sigma; \mathsf{Rel}(\Psi, \Omega_0, \Omega_0') \Downarrow_{\mathsf{ty}} H'\, \overline{\tau} : \mathbf{Type}$. Rule Ctx_UTyVar gives us $\Sigma \Downarrow_{\mathsf{ctx}} \Omega'\ \mathsf{ok}$. The induction hypothesis (for $\uparrow^{\mathsf{alt}}$) tells us that, $\forall i,\ \Sigma; \Psi, \Omega', \Omega_i; \text{'}\Pi\Delta.\, H'\, \overline{\tau} \Downarrow^{\tau_0 \rhd \gamma}_{\mathsf{alt}} alt_i : \alpha$. Lemma E.39 then tells us that $\overline{alt}'$ are well-formed and exhaustive. Lemma E.9 (and Lemma E.8 on the $\Downarrow_{\mathsf{alt}}$ judgments) allows us to combine all the Ω_i into $\overline{\Omega}$. We are done by Ty_Case.

Case ITy_Lam:

$$\frac{\begin{array}{l}\Sigma; \Psi \uparrow^{\mathsf{q}} \mathrm{qvar} \rightsquigarrow a : \kappa_1; \nu \dashv \Omega_1 \\ \Sigma; \Psi, \Omega_1, a{:}_{\mathsf{Rel}}\kappa_1 \uparrow^{*}_{\mathsf{ty}} \mathrm{t} \rightsquigarrow \tau : \kappa_2 \dashv \Omega_2 \\ \Omega_2 \hookrightarrow a{:}_{\mathsf{Rel}}\kappa_1 \rightsquigarrow \Omega_2'; \xi\end{array}}{\Sigma; \Psi \uparrow^{*}_{\mathsf{ty}} \lambda \mathrm{qvar}.\, \mathrm{t} \rightsquigarrow \lambda a{:}_{\mathsf{Rel}}\kappa_1.\, (\tau[\xi]) : \Pi_{\nu} a{:}_{\mathsf{Rel}}\kappa_1.\, (\kappa_2[\xi]) \dashv \Omega_1, \Omega_2'} \quad \textsc{ITy\_Lam}$$

The induction hypothesis (on $\uparrow^{\mathsf{q}}$) tells us that $\Sigma; \mathsf{Rel}(\Psi, \Omega_1) \Downarrow_{\mathsf{ty}} \kappa_1 : \mathbf{Type}$. Thus $\Sigma \Downarrow_{\mathsf{ctx}} \Psi, \Omega_1, a{:}_{\mathsf{Rel}}\kappa_1\ \mathsf{ok}$ and we can use the induction hypothesis to get $\Sigma; \Psi, \Omega_1, a{:}_{\mathsf{Rel}}\kappa_1, \Omega_2 \Downarrow_{\mathsf{ty}} \tau : \kappa_2$. By Lemma E.35, we get $\Sigma; \Psi, \Omega_1, \Omega_2', a{:}_{\mathsf{Rel}}\kappa_1 \Downarrow_{\mathsf{ty}} \tau[\xi] : \kappa_2[\xi]$ and thus $\Sigma; \Psi, \Omega_1, \Omega_2' \Downarrow_{\mathsf{ty}} \lambda a{:}_{\mathsf{Rel}}\kappa_1.\, (\tau[\xi]) : \Pi_{\nu} a{:}_{\mathsf{Rel}}\kappa_1.\, (\kappa_2[\xi])$ as desired.

Case ITy_LamIrrel: Like previous case.

Case ITy_Arrow: By induction and Lemma E.9

Case ITy_MArrow: By induction and Lemma E.9

Case ITy_Fix:

$$\frac{\begin{array}{l}\Sigma;\Psi \overset{}{\underset{\mathsf{ty}}{\Rightarrow}} \mathrm{t} \rightsquigarrow \tau : \kappa \dashv \Omega_1 \\ \overset{}{\underset{\mathsf{fun}}{\Rightarrow}}\ \kappa; \mathsf{Rel} \rightsquigarrow \gamma; \tilde{\Pi}; a; \mathsf{Rel}; \kappa_1; \kappa_2 \dashv \Omega_2 \\ \Sigma; \mathsf{Rel}(\Psi, \Omega_1, \Omega_2) \models_{\mathsf{ty}} \kappa_2 : \mathbf{Type} \\ \mathsf{fresh}\ \iota \qquad \Omega\ =\ \Omega_1, \Omega_2, \iota{:}\kappa_2 \sim \kappa_1\end{array}}{\Sigma;\Psi \overset{*}{\underset{\mathsf{ty}}{\Rightarrow}} \mathbf{fix}\,\mathrm{t} \rightsquigarrow \mathbf{fix}\,(\tau \rhd (\gamma \fatsemi \tilde{\Pi} a{:}_{\mathsf{Rel}}\langle\kappa_1\rangle.\,\iota)) : \kappa_1 \dashv \Omega}\ \textsc{ITy\_Fix}$$

The induction hypothesis gives us $\Sigma;\Psi,\Omega_1 \models_{\mathsf{ty}} \tau : \kappa$ and thus $\Sigma;\mathsf{Rel}(\Psi,\Omega_1) \models_{\mathsf{ty}} \kappa : \mathbf{Type}$ by Lemma E.17. Lemma E.37 gives us $\Sigma;\mathsf{Rel}(\Psi,\Omega_1,\Omega_2) \models_{\mathsf{co}} \gamma : \kappa \sim \tilde{\Pi}_{\mathsf{Req}} a{:}_{\mathsf{Rel}}\kappa_1.\,\kappa_2$ and then Ty_Cast tells us $\Sigma;\Psi,\Omega_1,\Omega_2 \models_{\mathsf{ty}} \tau \rhd \gamma : \tilde{\Pi}_{\mathsf{Req}} a{:}_{\mathsf{Rel}}\kappa_1.\,\kappa_2$. Thus, Lemma E.8 tells us $\Sigma \models_{\mathsf{ctx}} \Psi,\Omega_1,\Omega_2\ \mathsf{ok}$. In order to prove $\Sigma \models_{\mathsf{ctx}} \Omega\ \mathsf{ok}$, we must show $\Sigma;\mathsf{Rel}(\Psi,\Omega_1,\Omega_2) \models_{\mathsf{ty}} \kappa_2 : \mathbf{Type}$ and $\Sigma;\mathsf{Rel}(\Psi,\Omega_1,\Omega_2) \models_{\mathsf{ty}} \kappa_1 : \mathbf{Type}$. The first of these is a premise to ITy_Fix. To get the second, we use Lemma E.17 to get $\Sigma;\mathsf{Rel}(\Psi,\Omega_1,\Omega_2) \models_{\mathsf{ty}} \tilde{\Pi}_{\mathsf{Req}} a{:}_{\mathsf{Rel}}\kappa_1.\,\kappa_2 : \mathbf{Type}$ and then invert. We can conclude $\Sigma \models_{\mathsf{ctx}} \Omega\ \mathsf{ok}$ by Ctx_UCoVar.

Inversion on $\Sigma;\Psi,\Omega_1,\Omega_2 \models_{\mathsf{ty}} \tau \rhd \gamma : \tilde{\Pi}_{\mathsf{Req}} a{:}_{\mathsf{Rel}}\kappa_1.\,\kappa_2$ tells us that $\Sigma;\mathsf{Rel}(\Psi,\Omega) \models_{\mathsf{co}} \gamma : \kappa \sim \tilde{\Pi}_{\mathsf{Req}} a{:}_{\mathsf{Rel}}\kappa_1.\,\kappa_2$. We can further see (by Co_PiTy) that $\Sigma;\mathsf{Rel}(\Psi,\Omega) \models_{\mathsf{co}} \tilde{\Pi} a{:}_{\mathsf{Rel}}\langle\kappa_1\rangle.\,\iota : (\tilde{\Pi} a{:}_{\mathsf{Rel}}\kappa_1.\,\kappa_2) \sim (\tilde{\Pi} a{:}_{\mathsf{Rel}}\kappa_1.\,(\kappa_1[a \rhd \mathbf{sym}\,\langle\kappa_1\rangle/a]))$ However, because $a \# \kappa_1$ (by Lemma E.11), that last substitution has no effect, and so we conclude $\Sigma;\mathsf{Rel}(\Psi,\Omega) \models_{\mathsf{co}} \tilde{\Pi} a{:}_{\mathsf{Rel}}\langle\kappa_1\rangle.\,\iota : (\tilde{\Pi} a{:}_{\mathsf{Rel}}\kappa_1.\,\kappa_2) \sim (\tilde{\Pi} a{:}_{\mathsf{Rel}}\kappa_1.\,\kappa_1)$ and thus $\Sigma;\Psi,\Omega \models_{\mathsf{ty}} \tau \rhd (\gamma \fatsemi \tilde{\Pi} a{:}_{\mathsf{Rel}}\langle\kappa_1\rangle.\,\iota) : \tilde{\Pi} a{:}_{\mathsf{Rel}}\kappa_1.\,\kappa_1$. Finally, Ty_Fix gives us $\Sigma;\Psi,\Omega \models_{\mathsf{ty}} \mathbf{fix}\,(\tau \rhd (\gamma \fatsemi \tilde{\Pi} a{:}_{\mathsf{Rel}}\langle\kappa_1\rangle.\,\iota)) : \kappa_1$ as desired.

Case ITy_Let:

$$\frac{\begin{array}{l}\Sigma;\Psi \overset{*}{\underset{\mathsf{ty}}{\Rightarrow}} \mathrm{t}_1 \rightsquigarrow \tau_1 : \kappa_1 \dashv \Omega \\ \Sigma;\Psi,\Omega,x{:}_{\mathsf{Rel}}\kappa_1 \overset{*}{\underset{\mathsf{ty}}{\Rightarrow}} \mathrm{t}_2 \rightsquigarrow \tau_2 : \kappa_2 \dashv \Omega_2 \\ \Omega_2 \hookrightarrow x{:}_{\mathsf{Rel}}\kappa_1 \rightsquigarrow \Omega_2'; \xi\end{array}}{\Sigma;\Psi \overset{*}{\underset{\mathsf{ty}}{\Rightarrow}} \mathbf{let}\,x := \mathrm{t}_1\,\mathbf{in}\,\mathrm{t}_2 \rightsquigarrow (\lambda x{:}_{\mathsf{Rel}}\kappa_1.\,(\tau_2[\xi]))\,\tau_1 : \kappa_2[\xi][\tau_1/x] \dashv \Omega, \Omega_2'}\ \textsc{ITy\_Let}$$

The induction hypothesis gives us $\Sigma;\Psi,\Omega \models_{\mathsf{ty}} \tau_1 : \kappa_1$. Lemma E.17 tells us $\Sigma;\mathsf{Rel}(\Psi,\Omega) \models_{\mathsf{ty}} \kappa_1 : \mathbf{Type}$ and thus that $\Sigma \models_{\mathsf{ctx}} \Psi,\Omega,x{:}_{\mathsf{Rel}}\kappa_1\ \mathsf{ok}$. Another use of the induction hypothesis gives us $\Sigma;\Psi,\Omega,x{:}_{\mathsf{Rel}}\kappa_1,\Omega_2 \models_{\mathsf{ty}} \tau_2 : \kappa_2$. Lemma E.35 then gives us $\Sigma;\Psi,\Omega,\Omega_2',x{:}_{\mathsf{Rel}}\kappa_1 \models_{\mathsf{ty}} \tau_2[\xi] : \kappa_2[\xi]$ and thus $\Sigma;\Psi,\Omega,\Omega_2' \models_{\mathsf{ty}} \lambda x{:}_{\mathsf{Rel}}\kappa_1.\,(\tau_2[\xi]) : \tilde{\Pi} x{:}_{\mathsf{Rel}}\kappa_1.\,(\kappa_2[\xi])$ Rule Ty_AppRel gives us $\Sigma;\Psi,\Omega,\Omega_2' \models_{\mathsf{ty}} (\lambda x{:}_{\mathsf{Rel}}\kappa_1.\,(\tau_2[\xi]))\,\tau_1 : \kappa_2[\xi][\tau_1/x]$ as desired.

Case ITyC_Case: Similar to the case for ITy_Case. The only differences are the definition of Ω' (which is simpler in this case) and the use of $\overset{}{\underset{\mathsf{altc}}{\Rightarrow}}$ in place of $\overset{}{\underset{\mathsf{alt}}{\Rightarrow}}$. Both $\overset{}{\underset{\mathsf{alt}}{\Rightarrow}}$ and $\overset{}{\underset{\mathsf{altc}}{\Rightarrow}}$ are proven sound via the induction hypothesis.

Case ITyC_LamDep:

$$\frac{\begin{array}{l}\vdash^{\rightarrow}_{\mathsf{fun}}\ \kappa; \mathsf{Rel} \rightsquigarrow \gamma; \tilde{\Pi}; a; \mathsf{Rel}; \kappa_1; \kappa_2 \dashv \Omega_0 \\ \neg(a \# \kappa_2) \\ \Sigma; \mathsf{Rel}(\Psi) \vdash^{\rightarrow}_{\mathsf{pt}} \mathrm{s} \rightsquigarrow \kappa_1' \dashv \Omega_1 \\ \Omega = \Omega_0, \Omega_1, \iota{:}\kappa_1 \sim \kappa_1' \\ \Sigma; \Psi, \Omega, b{:}_{\mathsf{Rel}}\kappa_1' \vdash^{*}_{\mathsf{ty}} \mathrm{t} : \kappa_2[b \rhd \mathbf{sym}\,\iota/a] \rightsquigarrow \tau \dashv \Omega_2 \\ \Omega_2 \hookrightarrow b{:}_{\mathsf{Rel}}\kappa_1' \rightsquigarrow \Omega_2'; \xi \\ \eta = \kappa_2[(a \rhd \iota) \rhd \mathbf{sym}\,\iota/a] \approx_{\langle \mathbf{Type} \rangle} \kappa_2 \\ \tau_0 = (\lambda a{:}_{\mathsf{Rel}}\kappa_1.\, (\tau[\xi][a \rhd \iota/b] \rhd \eta)) \rhd \mathbf{sym}\,\gamma\end{array}}{\Sigma; \Psi \vdash^{\rightarrow}_{\mathsf{ty}} \lambda(a :: \mathrm{s}).\, \mathrm{t} : \kappa \rightsquigarrow \tau_0 \dashv \Omega, \Omega_2'}\ \textsc{ITyC\_LamDep}$$

We have assumed $\Sigma; \mathsf{Rel}(\Psi) \vDash_{\mathsf{ty}} \kappa : \mathbf{Type}$ and thus can use Lemma E.37 to get $\Sigma; \mathsf{Rel}(\Psi, \Omega_0) \vDash_{\mathsf{co}} \gamma : \kappa \sim \tilde{\Pi}_{\mathsf{Req}} a{:}_{\mathsf{Rel}}\kappa_1.\, \kappa_2$. (The $\neg(a \# \kappa_2)$ premise is not used in this rule; it is used to filter out which cases are handled in the next one.) By Lemma E.8, we have $\Sigma \vDash_{\mathsf{ctx}} \mathsf{Rel}(\Psi)\ \mathsf{ok}$ and thus can use the induction hypothesis to get $\Sigma; \mathsf{Rel}(\Psi, \Omega_1) \vDash_{\mathsf{ty}} \kappa_1' : \mathbf{Type}$. We must now prove that $\Sigma; \mathsf{Rel}(\Psi, \Omega), b{:}_{\mathsf{Rel}}\kappa_1' \vDash_{\mathsf{ty}} \kappa_2[b \rhd \mathbf{sym}\,\iota/a] : \mathbf{Type}$. First, we prove that $\Sigma \vDash_{\mathsf{ctx}} \mathsf{Rel}(\Psi, \Omega), b{:}_{\mathsf{Rel}}\kappa_1'\ \mathsf{ok}$. For this, it is left to prove only that $\Sigma; \mathsf{Rel}(\Psi, \Omega_0, \Omega_1) \vDash_{\mathsf{ty}} \kappa_1 : \mathbf{Type}$. This we can get from Lemma E.18, inversion of Ty_Pi, and Lemma E.4. The inversion of Ty_Pi also tells us that $\Sigma; \mathsf{Rel}(\Psi, \Omega_0), a{:}_{\mathsf{Rel}}\kappa_1 \vDash_{\mathsf{ty}} \kappa_2 : \mathbf{Type}$. Lemma E.9 allows us to weaken this to $\Sigma; \mathsf{Rel}(\Psi, \Omega), b{:}_{\mathsf{Rel}}\kappa_1', a{:}_{\mathsf{Rel}}\kappa_1 \vDash_{\mathsf{ty}} \kappa_2 : \mathbf{Type}$. We can see that $\Sigma; \mathsf{Rel}(\Psi, \Omega), b{:}_{\mathsf{Rel}}\kappa_1' \vDash_{\mathsf{ty}} b \rhd \mathbf{sym}\,\iota : \kappa_1$. We thus use Lemma E.13 to get $\Sigma; \mathsf{Rel}(\Psi, \Omega), b{:}_{\mathsf{Rel}}\kappa_1' \vDash_{\mathsf{ty}} \kappa_2[b \rhd \mathbf{sym}\,\iota/a] : \mathbf{Type}$ as desired. We then use the induction hypothesis to get $\Sigma; \Psi, \Omega, b{:}_{\mathsf{Rel}}\kappa_1', \Omega_2 \vDash_{\mathsf{ty}} \tau : \kappa_2[b \rhd \mathbf{sym}\,\iota/a]$. Lemma E.35 allows us to rewrite this to $\Sigma; \Psi, \Omega, \Omega_2', b{:}_{\mathsf{Rel}}\kappa_1' \vDash_{\mathsf{ty}} \tau[\xi] : \kappa_2[b \rhd \mathbf{sym}\,\iota/a][\xi]$, but Lemma E.34 tells us the $[\xi]$ in the kind has no effect. Lemma E.9 allows us to weaken this to $\Sigma; \Psi, \Omega, \Omega_2', a{:}_{\mathsf{Rel}}\kappa_1, b{:}_{\mathsf{Rel}}\kappa_1' \vDash_{\mathsf{ty}} \tau[\xi] : \kappa_2[b \rhd \mathbf{sym}\,\iota/a]$. We can see that $\Sigma; \Psi, \Omega, \Omega_2', a{:}_{\mathsf{Rel}}\kappa_1 \vDash_{\mathsf{ty}} a \rhd \iota : \kappa_1'$ and thus we can use Lemma E.13 to get $\Sigma; \Psi, \Omega, \Omega_2', a{:}_{\mathsf{Rel}}\kappa_1 \vDash_{\mathsf{ty}} \tau[\xi][a \rhd \iota/b] : \kappa_2[b \rhd \mathbf{sym}\,\iota/a][a \rhd \iota/b]$. Inlining substitutions, we can rewrite the kind to $\kappa_2[(a \rhd \iota) \rhd \mathbf{sym}\,\iota/a]$. We can then see that $\Sigma; \Psi, \Omega, \Omega_2', a{:}_{\mathsf{Rel}}\kappa_1 \vDash_{\mathsf{ty}} \tau[\xi][a \rhd \iota/b] \rhd \eta : \kappa_2$ and by Ty_Lam that $\Sigma; \Psi, \Omega, \Omega_2' \vDash_{\mathsf{ty}} \lambda a{:}_{\mathsf{Rel}}\kappa_1.\, (\tau[\xi][a \rhd \iota/b] \rhd \eta) : \tilde{\Pi} a{:}_{\mathsf{Rel}}\kappa_1.\, \kappa_2$. A use of Ty_Cast gives us $\Sigma; \Psi, \Omega, \Omega_2' \vDash_{\mathsf{ty}} (\lambda a{:}_{\mathsf{Rel}}\kappa_1.\, (\tau[\xi][a \rhd \iota/b] \rhd \eta)) \rhd \mathbf{sym}\,\gamma : \kappa$ as desired.

Case ITyC_Lam:

$$\frac{\begin{array}{l}\vdash^{\rightarrow}_{\mathsf{fun}}\ \kappa; \mathsf{Rel} \rightsquigarrow \gamma; \tilde{\Pi}; a; \mathsf{Rel}; \kappa_1; \kappa_2 \dashv \Omega_0 \\ \Sigma; \Psi \vdash^{\rightarrow}_{\mathsf{aq}} \mathrm{aqvar} : \kappa_1 \rightsquigarrow b : \kappa_1'; x.\tau_1 \dashv \Omega_1 \\ \Sigma; \Psi, \Omega_0, \Omega_1, b{:}_{\mathsf{Rel}}\kappa_1' \vdash^{*}_{\mathsf{ty}} \mathrm{t} : \kappa_2 \rightsquigarrow \tau \dashv \Omega_2 \\ \Omega_2 \hookrightarrow b{:}_{\mathsf{Rel}}\kappa_1' \rightsquigarrow \Omega_2'; \xi \\ \Omega' = \Omega_0, \Omega_1, \Omega_2'\end{array}}{\Sigma; \Psi \vdash^{\rightarrow}_{\mathsf{ty}} \lambda \mathrm{aqvar}.\, \mathrm{t} : \kappa \rightsquigarrow (\lambda a{:}_{\mathsf{Rel}}\kappa_1.\, \tau[\xi][\tau_1[a/x]/b]) \rhd \mathbf{sym}\,\gamma \dashv \Omega'}\ \textsc{ITyC\_Lam}$$

Lemma E.37 tells us $\Sigma; \mathsf{Rel}(\Psi, \Omega_0) \vDash_{\mathsf{co}} \gamma : \kappa \sim \tilde{\Pi}_{\mathsf{Req}}\, a{:}_{\mathsf{Rel}}\kappa_1.\, \kappa_2$. Lemma E.18 and inversions tell us $\Sigma; \mathsf{Rel}(\Psi, \Omega_0) \vDash_{\mathsf{ty}} \kappa_1 : \mathbf{Type}$ and $\Sigma; \mathsf{Rel}(\Psi, \Omega_0), a{:}_{\mathsf{Rel}}\kappa_1 \vDash_{\mathsf{ty}} \kappa_2 : \mathbf{Type}$. We can conclude $\Sigma \vDash_{\mathsf{ctx}} \mathsf{Rel}(\Psi, \Omega_0), a{:}_{\mathsf{Rel}}\kappa_1\ \mathsf{ok}$ and thus (using Lemma E.28) $\Sigma; \Psi, \Omega_0, a{:}_{\mathsf{Rel}}\kappa_1 \vDash_{\mathsf{ty}} a : \kappa_1$. The induction hypothesis on $\vDash_{\mathsf{aq}}$ then tells us $\Sigma; \Psi, \Omega_0, a{:}_{\mathsf{Rel}}\kappa_1, \Omega_1 \vDash_{\mathsf{ty}} \tau_1[a/x] : \kappa_1'$. We can see by the construction of Ω_1 and κ_1' that $a \# \Omega_1$ and $a \# \kappa_1'$. Because we are in rule ITyC_Lam, it means that ITyC_LamDep does not apply. This can be for one of two reasons, and thus we now have two cases:

Case aqvar $= a$ **(unannotated binder):** In this case, we see (by IAQ-VarC_Var) that $\kappa_1' = \kappa_1$. We can choose $a = b$ by the α-renaming. Thus, $\Sigma; \mathsf{Rel}(\Psi, \Omega_0), b{:}_{\mathsf{Rel}}\kappa_1' \vDash_{\mathsf{ty}} \kappa_2 : \mathbf{Type}$.

Case $a \# \kappa_2$**:** We now use Lemma E.10 to get $\Sigma; \mathsf{Rel}(\Psi, \Omega_0) \vDash_{\mathsf{ty}} \kappa_2 : \mathbf{Type}$.

Regardless of which case above we are in, we now must prove $\Sigma \vDash_{\mathsf{ctx}} \Psi, \Omega_0, \Omega_1, b{:}_{\mathsf{Rel}}\kappa_1'\ \mathsf{ok}$. To do this, we must show only that $\Sigma; \mathsf{Rel}(\Psi, \Omega_0, \Omega_1) \vDash_{\mathsf{ty}} \kappa_1' : \mathbf{Type}$, which comes from Lemma E.17 and Lemma E.10. We can then use Lemma E.9 to get $\Sigma; \mathsf{Rel}(\Psi, \Omega_0, \Omega_1), b{:}_{\mathsf{Rel}}\kappa_1' \vDash_{\mathsf{ty}} \kappa_2 : \mathbf{Type}$. The induction hypothesis now applies to get $\Sigma; \Psi, \Omega_0, \Omega_1, b{:}_{\mathsf{Rel}}\kappa_1', \Omega_2 \vDash_{\mathsf{ty}} \tau : \kappa_2$. Lemma E.35 tells us $\Sigma; \Psi, \Omega_0, \Omega_1, \Omega_2', b{:}_{\mathsf{Rel}}\kappa_1' \vDash_{\mathsf{ty}} \tau[\xi] : \kappa_2[\xi]$, but Lemma E.34 tells us the $[\xi]$ in the kind has no effect. Lemma E.9 gives us $\Sigma; \Psi, \Omega_0, a{:}_{\mathsf{Rel}}\kappa_1, \Omega_1, \Omega_2', b{:}_{\mathsf{Rel}}\kappa_1' \vDash_{\mathsf{ty}} \tau[\xi] : \kappa_2$. We can thus use Lemma E.13 to get $\Sigma; \Psi, \Omega_0, a{:}_{\mathsf{Rel}}\kappa_1, \Omega_1, \Omega_2' \vDash_{\mathsf{ty}} \tau[\xi][\tau_1[a/x]/b] : \kappa_2[\tau_1[a/x]/b]$, but Lemma E.11 tells us the substitution in the kind has no effect. Noting that, by analysis stemming from our two cases previously, $a \# \Omega_2'$, we can reshuffle the context to be $\Psi, \Omega_0, \Omega_1, \Omega_2', a{:}_{\mathsf{Rel}}\kappa_1$ and thus conclude $\Sigma; \Psi, \Omega_0, \Omega_1, \Omega_2' \vDash_{\mathsf{ty}} \lambda a{:}_{\mathsf{Rel}}\kappa_1.\, \tau[\xi][\tau_1[a/x]/b] : \tilde{\Pi} a{:}_{\mathsf{Rel}}\kappa_1.\, \kappa_2$. Thus Ty_Cast gives us $\Sigma; \Psi, \Omega_0, \Omega_1, \Omega_2' \vDash_{\mathsf{ty}} (\lambda a{:}_{\mathsf{Rel}}\kappa_1.\, \tau[\xi][\tau_1[a/x]/b]) \rhd \mathbf{sym}\, \gamma : \kappa$ as desired.

Case ITyC_LamIrrelDep: Like case for ITyC_LamDep.

Case ITyC_LamIrrel: Like case for ITyC_Lam.

Case ITyC_Fix:

$$\frac{\Sigma; \Psi \vDash_{\mathsf{ty}} \mathrm{t} : \tilde{\Pi}_{\mathsf{Req}}\, a{:}_{\mathsf{Rel}}\kappa.\, \kappa \rightsquigarrow \tau \dashv \Omega}{\Sigma; \Psi \vDash_{\mathsf{ty}} \mathbf{fix}\, \mathrm{t} : \kappa \rightsquigarrow \mathbf{fix}\, \tau \dashv \Omega}\ \text{ITyC\_Fix}$$

We know $\Sigma; \mathsf{Rel}(\Psi) \vDash_{\mathsf{ty}} \kappa : \mathbf{Type}$. We can thus conclude by Ty_Pi that $\Sigma; \mathsf{Rel}(\Psi) \vDash_{\mathsf{ty}} \tilde{\Pi} a{:}_{\mathsf{Rel}}\kappa.\, \kappa : \mathbf{Type}$. We thus use the induction hypothesis to get $\Sigma; \Psi, \Omega \vDash_{\mathsf{ty}} \tau : \tilde{\Pi} a{:}_{\mathsf{Rel}}\kappa.\, \kappa$. Thus we are done by Ty_Fix.

Case ITyC_Infer:

$$\frac{\begin{array}{l}\Sigma; \Psi \stackrel{*}{\Rightarrow}_{\mathsf{ty}} \mathrm{t} \rightsquigarrow \tau : \kappa_1 \dashv \Omega \\ \stackrel{\Rightarrow}{\mathsf{pre}} \kappa_2 \rightsquigarrow \Delta; \kappa_2'; \tau_2 \\ \Omega \hookrightarrow \Delta \rightsquigarrow \Omega'; \xi_1 \\ \kappa_1[\xi_1] \leq^* \kappa_2' \rightsquigarrow \tau_2' \dashv \Omega_2 \\ \Omega_2 \hookrightarrow \Delta \rightsquigarrow \Omega_2'; \xi_2\end{array}}{\Sigma; \Psi \Rightarrow_{\mathsf{ty}} \mathrm{t} : \kappa_2 \rightsquigarrow \tau_2\,(\lambda\Delta.\, \tau_2'[\xi_2]\,\tau[\xi_1]) \dashv \Omega', \Omega_2'} \quad \text{ITyC\_Infer}$$

The induction hypothesis tells us that $\Sigma; \Psi, \Omega \vDash_{\mathsf{ty}} \tau : \kappa_1$. We have assumed $\Sigma; \mathsf{Rel}(\Psi) \vDash_{\mathsf{ty}} \kappa_2 : \mathbf{Type}$. We can thus use Lemma E.40 to get $\Sigma; \Psi \vDash_{\mathsf{ty}} \tau_2 : \tilde{\Pi} x{:}_{\mathsf{Rel}}(\tilde{\Pi}\Delta.\, \kappa_2').\, \kappa_2$. Lemma E.17 and inversion gives us $\Sigma; \mathsf{Rel}(\Psi, \Delta) \vDash_{\mathsf{ty}} \kappa_2' : \mathbf{Type}$ and thus Lemma E.8 and Lemma E.28 give us $\Sigma \vDash_{\mathsf{ctx}} \Psi, \Delta\ \mathsf{ok}$. We can thus use Lemma E.9 to get $\Sigma; \Psi, \Delta, \Omega \vDash_{\mathsf{ty}} \tau : \kappa_1$ and then Lemma E.35 to get $\Sigma; \Psi, \Omega', \Delta \vDash_{\mathsf{ty}} \tau[\xi_1] : \kappa_1[\xi_1]$. Lemma E.17 tells us $\Sigma; \mathsf{Rel}(\Psi, \Omega', \Delta) \vDash_{\mathsf{ty}} \kappa_1[\xi_1] : \mathbf{Type}$ and Lemma E.9 tells us $\Sigma; \mathsf{Rel}(\Psi, \Omega', \Delta) \vDash_{\mathsf{ty}} \kappa_2' : \mathbf{Type}$. We can thus use Lemma E.41 to get $\Sigma; \Psi, \Omega', \Delta, \Omega_2 \vDash_{\mathsf{ty}} \tau_2' : \tilde{\Pi} x{:}_{\mathsf{Rel}}(\kappa_1[\xi_1]).\, \kappa_2'$. Lemma E.35 then tells us $\Sigma; \Psi, \Omega', \Omega_2', \Delta \vDash_{\mathsf{ty}} \tau_2'[\xi_2] : (\tilde{\Pi} x{:}_{\mathsf{Rel}}(\kappa_1[\xi_1]).\, \kappa_2')[\xi_2]$, but Lemma E.34 tells us that the $[\xi_2]$ in the kind has no effect (because neither κ_2' nor κ_1 nor ξ_1 can mention anything bound in Ω_2). We thus have $\Sigma; \Psi, \Omega', \Omega_2', \Delta \vDash_{\mathsf{ty}} \tau_2'[\xi_2] : \tilde{\Pi} x{:}_{\mathsf{Rel}}(\kappa_1[\xi_1]).\, \kappa_2'$. Ty_AppRel (with Lemma E.9) tells us $\Sigma; \Psi, \Omega', \Omega_2', \Delta \vDash_{\mathsf{ty}} \tau_2'[\xi_2]\,\tau[\xi_1] : \kappa_2'[\tau[\xi_1]/x]$ but Lemma E.11 tells us that the substitution in the kind has no effect. Multiple uses of Ty_Lam gives us $\Sigma; \Psi, \Omega', \Omega_2' \vDash_{\mathsf{ty}} \lambda\Delta.\, \tau_2'[\xi_2]\,\tau[\xi_1] : \tilde{\Pi}\Delta.\, \kappa_2'$. Yet another use of Lemma E.9 and Ty_AppRel gives us $\Sigma; \Psi, \Omega', \Omega_2' \vDash_{\mathsf{ty}} \tau_2\,(\lambda\Delta.\, \tau_2'[\xi_2]\,\tau[\xi_1]) : \kappa_2[\lambda\Delta.\, \tau_2'[\xi_2]\,\tau[\xi_1]/x]$, where Lemma E.11 tells us that the substitution in the kind has no effect. We are thus done.

Invisible λ/Λ cases: Like corresponding visible λ/Λ cases. Note that the difference between the $\Rightarrow_{\mathsf{ty}}$ and $\stackrel{*}{\Rightarrow}_{\mathsf{ty}}$ checking judgments is relevant for user-facing issues of type inference (e.g., principal types), not the soundness we are proving here.

Case ITyC_Let: Similar to case for ITy_Let. The only difference is that the expected type is propagated down.

Case ITyC_Skol:

$$\frac{\begin{array}{l}\nu \leq \mathsf{Spec} \\ \Sigma; \Psi, \$a{:}_\rho \kappa_1 \stackrel{*}{\Rightarrow}_{\mathsf{ty}} \mathrm{t} : \kappa_2 \rightsquigarrow \tau \dashv \Omega \\ \Omega \hookrightarrow \$a{:}_\rho \kappa_1 \rightsquigarrow \Omega'; \xi\end{array}}{\Sigma; \Psi \stackrel{*}{\Rightarrow}_{\mathsf{ty}} \mathrm{t} : \tilde{\Pi}_\nu \$a{:}_\rho \kappa_1.\, \kappa_2 \rightsquigarrow \lambda\$a{:}_\rho \kappa_1.\, \tau[\xi] \dashv \Omega'} \quad \text{ITyC\_Skol}$$

We have assumed $\Sigma; \mathsf{Rel}(\Psi) \vDash_{\mathsf{ty}} \tilde{\Pi}_\nu \$a{:}_\rho \kappa_1.\, \kappa_2 : \mathbf{Type}$. Inversion gives us $\Sigma; \mathsf{Rel}(\Psi), \$a{:}_{\mathsf{Rel}} \kappa_1 \vDash_{\mathsf{ty}} \kappa_2 : \mathbf{Type}$, and we can thus use the induction hypothesis to get $\Sigma; \Psi, \$a{:}_\rho \kappa_1, \Omega \vDash_{\mathsf{ty}} \tau : \kappa_2$. Lemma E.35 tells us $\Sigma; \Psi, \Omega', \$a{:}_\rho \kappa_1 \vDash_{\mathsf{ty}} \tau[\xi] : \kappa_2[\xi]$, but Lemma E.34 tells us that the $[\xi]$ in the kind has no effect. We can thus

conclude $\Sigma; \Psi, \Omega' \vDash_{\mathsf{ty}} \lambda\$a{:}_\rho\kappa_1.\,(\tau[\xi]) : \underset{\sim}{\Pi}_\nu \$a{:}_\rho\kappa_1.\,\kappa_2$ as desired.

Case ITyC_Otherwise: By induction.

Case IPtC_Pi:

$$\frac{\begin{array}{l}\vdash_{\mathsf{pi}} \mathrm{quant} \rightsquigarrow \Pi; \rho \\ \Sigma; \Psi \vdash_{\mathsf{q}} \mathrm{qvar} \rightsquigarrow a : \kappa; \nu \dashv \Omega \\ \Sigma; \Psi, \Omega, a{:}_\rho\kappa \vdash_{\mathsf{pt}} \mathrm{s} \rightsquigarrow \sigma \dashv \Omega_2 \\ \Omega_2 \hookrightarrow a{:}_\rho\kappa \rightsquigarrow \Omega_2'; \xi\end{array}}{\Sigma; \Psi \vdash_{\mathsf{pt}} \forall\, \mathrm{qvar}.\, \mathrm{s} \rightsquigarrow \Pi_\nu a{:}_\rho\kappa.\,(\sigma[\xi]) \dashv \Omega, \Omega_2'} \quad \text{IPtC\_Pi}$$

The induction hypothesis (on $\vdash_{\mathsf{q}}$) tells us $\Sigma; \mathsf{Rel}(\Psi, \Omega) \vDash_{\mathsf{ty}} \kappa : \mathbf{Type}$. Thus $\Sigma \vDash_{\mathsf{ctx}} \mathsf{Rel}(\Psi, \Omega, a{:}_\rho\kappa)\ \mathsf{ok}$ and we can use the induction hypothesis (on $\vdash_{\mathsf{pt}}$) to get $\Sigma; \mathsf{Rel}(\Psi, \Omega, a{:}_\rho\kappa, \Omega_2) \vDash_{\mathsf{ty}} \sigma : \mathbf{Type}$. Lemma E.35 gives us $\Sigma; \mathsf{Rel}(\Psi, \Omega, \Omega_2', a{:}_\rho\kappa) \vDash_{\mathsf{ty}} \sigma[\xi] : \mathbf{Type}$ and thus $\Sigma; \mathsf{Rel}(\Psi, \Omega, \Omega_2') \vDash_{\mathsf{ty}} \underset{\sim}{\Pi} a{:}_\rho\kappa.\,(\sigma[\xi]) : \mathbf{Type}$ as desired.

Case IPtC_Constrained:

$$\frac{\begin{array}{l}\Sigma; \Psi \vdash_{\mathsf{ty}} \mathrm{t} : \mathbf{Type} \rightsquigarrow \tau \dashv \Omega_1 \\ \Sigma; \Psi, \Omega_1, \$a{:}_{\mathsf{Rel}}\tau \vdash_{\mathsf{pt}} \mathrm{s} \rightsquigarrow \sigma \dashv \Omega_2 \\ \Omega_2 \hookrightarrow \$a{:}_{\mathsf{Rel}}\tau \rightsquigarrow \Omega_2'; \xi\end{array}}{\Sigma; \Psi \vdash_{\mathsf{pt}} \mathrm{t} \Rightarrow \mathrm{s} \rightsquigarrow \underset{\sim}{\Pi}_{\mathsf{Inf}} \$a{:}_{\mathsf{Rel}}\tau.\,(\sigma[\xi]) \dashv \Omega_1, \Omega_2'} \quad \text{IPtC\_Constrained}$$

Lemma C.38, Lemma E.9 and Lemma E.3 tell us $\Sigma; \mathsf{Rel}(\Psi) \vDash_{\mathsf{ty}} \mathbf{Type} : \mathbf{Type}$ and thus we can use the induction hypothesis on $\vdash_{\mathsf{ty}}$ to get $\Sigma; \Psi, \Omega_1 \vDash_{\mathsf{ty}} \tau : \mathbf{Type}$. We thus have $\Sigma \vDash_{\mathsf{ctx}} \Psi, \Omega_1, \$a{:}_{\mathsf{Rel}}\tau\ \mathsf{ok}$ and can use the induction hypothesis on $\vdash_{\mathsf{pt}}$ to get $\Sigma; \mathsf{Rel}(\Psi, \Omega_1, \$a{:}_{\mathsf{Rel}}\tau, \Omega_2) \vDash_{\mathsf{ty}} \sigma : \mathbf{Type}$. Lemma E.35 gives us $\Sigma; \mathsf{Rel}(\Psi, \Omega_1, \Omega_2', \$a{:}_{\mathsf{Rel}}\tau) \vDash_{\mathsf{ty}} \sigma[\xi] : \mathbf{Type}$ and thus $\Sigma; \mathsf{Rel}(\Psi, \Omega_1, \Omega_2') \vDash_{\mathsf{ty}} \underset{\sim}{\Pi}_{\mathsf{Inf}} \$a{:}_{\mathsf{Rel}}\tau.\,(\sigma[\xi]) : \mathbf{Type}$ as desired.

Case IPtC_Mono: By induction.

Case IArg_Rel: By induction and straightforward use of typing rules.

Case IArg_Irrel: By induction and straightforward use of typing rules.

Case IAlt_Con:

$$\frac{\begin{array}{l}\Sigma \vDash_{\mathsf{tc}} H : \Delta_1; \Delta_2; H' \qquad \Delta_3, \Delta_4 = \Delta_2[\overline{\tau}/\mathsf{dom}(\Delta_1)] \\ \mathsf{dom}(\Delta_3) = \overline{x} \qquad \mathsf{dom}(\Delta_4) = \mathsf{dom}(\Delta') \\ \mathsf{match}_{\{\mathsf{dom}(\Delta_3)\}}(\mathsf{types}(\Delta_4); \mathsf{types}(\Delta')) = \mathsf{Just}\,\theta \\ \Sigma; \Psi, \Delta_3 \vdash_{\mathsf{ty}} \mathrm{t} : \kappa \rightsquigarrow \tau \dashv \Omega \\ \Omega \hookrightarrow \Delta_3 \rightsquigarrow \Omega'; \xi \\ \Delta_3' = \Delta_3, c{:}\tau_0 \sim H_{\{\overline{\tau}\}}\,\overline{x}\end{array}}{\Sigma; \Psi; \text{'}\Pi\Delta'.\, H'\,\overline{\tau}; \tau_0 \vdash_{\mathsf{alt}} H\,\overline{x} \to \mathrm{t} : \kappa \rightsquigarrow H \to \lambda\Delta_3'.\,(\tau[\xi]) \dashv \Omega'} \quad \text{IAlt\_Con}$$

We wish to prove $\Sigma; \Psi, \Omega'; \text{'}\Pi\Delta'.\, H'\,\overline{\tau} \vDash^{\tau_0}_{\mathsf{alt}} H \to \lambda\Delta_3, (c{:}\tau_0 \sim H_{\{\overline{\tau}\}}\,\overline{x}).\,(\tau[\xi]) : \kappa$,

given the premises above along with

- $\Sigma; \mathsf{Rel}(\Psi) \vDash_{\mathsf{ty}} \kappa : \mathbf{Type}$
- $\Sigma; \Psi \vDash_{\mathsf{ty}} \tau_0 : \Pi\Delta'.\, H'\,\overline{\tau}$
- $\Sigma; \mathsf{Rel}(\Psi) \vDash_{\mathsf{ty}} H'\,\overline{\tau} : \mathbf{Type}$

We will use Alt_Match. This requires the following:

$\Sigma \vdash_{\mathsf{tc}} H : \Delta_1; \Delta_2; H'$**:** This is a premise above.

$\Delta_3, \Delta_4 = \Delta_2[\overline{\tau}/\mathsf{dom}(\Delta_1)]$**:** This is a premise above.

$\mathsf{dom}(\Delta_4) = \mathsf{dom}(\Delta')$**:** This is a premise above.

$\mathsf{match}_{\{\mathsf{dom}(\Delta_3)\}}(\mathsf{types}(\Delta_4); \mathsf{types}(\Delta')) = \mathsf{Just}\,\theta$**:** This is a premise above.

$\Sigma; \Psi, \Omega' \vDash_{\mathsf{ty}} \lambda\Delta_3, (c{:}\tau_0 \sim H_{\{\overline{\tau}\}}\,\overline{x}).\,(\tau[\xi]) : \Pi\Delta_3, c{:}\tau_0 \sim H_{\{\overline{\tau}\}}\,\mathsf{dom}(\Delta_3).\,\kappa$**:** Let $\Psi' = \Psi, \Omega', \Delta_3, c{:}\tau_0 \sim H_{\{\overline{\tau}\}}\,\overline{x}$. We must show only that $\Sigma; \Psi' \vDash_{\mathsf{ty}} \tau[\xi] : \kappa$. (Note that $\mathsf{dom}(\Delta_3) = \overline{x}$, which is the one discrepancy between the quantified contexts above.) To use the induction hypothesis on $\vDash_{\mathsf{ty}}$, we must show $\Sigma; \mathsf{Rel}(\Psi, \Delta_3) \vDash_{\mathsf{ty}} \kappa : \mathbf{Type}$, which means we must show only that $\Sigma \vDash_{\mathsf{ctx}} \Psi, \Delta_3\ \mathsf{ok}$ and then use Lemma E.9. Lemma C.40 gives us $\Sigma \vdash_{\mathsf{ctx}} \Delta_1, \Delta_2\ \mathsf{ok}$ and by Lemma E.3, $\Sigma \vDash_{\mathsf{ctx}} \Delta_1, \Delta_2\ \mathsf{ok}$. Lemma C.77 gives us $\Sigma \vdash_{\mathsf{tc}} H' : \varnothing; \mathsf{Rel}(\Delta_1); \mathbf{Type}$. We know $\Sigma; \mathsf{Rel}(\Psi) \vDash_{\mathsf{ty}} H'\,\overline{\tau} : \mathbf{Type}$. By Lemma C.42 (easily updated to use $\vDash$ judgments), we can see that $\Sigma; \mathsf{Rel}(\Psi) \vDash_{\mathsf{vec}} \overline{\tau} : \mathsf{Rel}(\Delta_1)$ and thus Lemma E.15 tells us $\Sigma \vDash_{\mathsf{ctx}} \Delta_2[\overline{\tau}/\mathsf{dom}(\Delta_1)]\ \mathsf{ok}$ and by Lemma E.8 and Lemma E.9, $\Sigma \vDash_{\mathsf{ctx}} \Psi, \Delta_3\ \mathsf{ok}$ as desired. We have concluded that $\Sigma; \mathsf{Rel}(\Psi, \Delta_3) \vDash_{\mathsf{ty}} \kappa : \mathbf{Type}$ and so can use the induction hypothesis to get $\Sigma; \Psi, \Delta_3, \Omega \vDash_{\mathsf{ty}} \tau : \kappa$. Lemma E.35 tells us $\Sigma; \Psi, \Omega', \Delta_3 \vDash_{\mathsf{ty}} \tau[\xi] : \kappa[\xi]$, but Lemma E.34 tells us that the $[\xi]$ in the conclusion has no effect. The last step here is to use weakening to add the binding for c to the context. This requires proving only that $\Sigma; \mathsf{Rel}(\Psi, \Omega', \Delta_3) \vDash_{\mathsf{ty}} H_{\{\overline{\tau}\}}\,\overline{x} : \Pi\Delta_4.\, H'\,\overline{\tau}$. We can see that $\Sigma; \mathsf{Rel}(\Psi, \Omega', \Delta_3) \vDash_{\mathsf{ty}} H_{\{\overline{\tau}\}} : \Pi\Delta_3, \Delta_4.\, H'\,\overline{\tau}$ by Ty_Con. We are thus done by Lemma C.31 (easily updated for $\vDash$ judgments).

Case IAlt_Default: By induction and Alt_Default.

Case IAltC_Con: This case is identical to that for IAlt_Con. The difference is the assumptions that can be made when solving for the unification variables in Ω, which does not affect the course of this proof.

Case IAltC_Default: Similar to the case for IAlt_Default.

Case IQVar_Req: By induction.

Case IQVar_Spec: By induction.

Case IAQVar_Var:

$$\frac{\mathsf{fresh}\,\beta}{\Sigma; \Psi \vDash_{\mathsf{aq}} a \rightsquigarrow a : \beta \dashv \beta{:}_{\mathsf{Irrel}}\mathbf{Type}}\ \text{IAQVar\_Var}$$

We must show only that $\Sigma; \mathsf{Rel}(\Psi), \beta{:}_{\mathsf{Rel}}\mathbf{Type} \vDash_{\mathsf{ty}} \beta : \mathbf{Type}$. This is true by Ty_UVar.

Case IAQVar_Annot: By induction.

Case IAQVarC_Var:

$$\frac{}{\Sigma; \Psi \vdash_{\mathsf{aq}} a : \kappa \rightsquigarrow a : \kappa; x.x \dashv \varnothing} \quad \text{IAQVarC\_Var}$$

Given $\Sigma; \Psi \vDash_{\mathsf{ty}} \tau_0 : \kappa$, we must show $\Sigma; \Psi \vDash_{\mathsf{ty}} x[\tau_0/x] : \kappa$. By assumption.

Case IAQVarC_Annot:

$$\frac{\begin{array}{l}\Sigma; \mathsf{Rel}(\Psi) \vdash_{\mathsf{pt}} \mathrm{s} \rightsquigarrow \sigma \dashv \Omega_1 \\ \kappa \leq \sigma \rightsquigarrow \tau \dashv \Omega_2\end{array}}{\Sigma; \Psi \vdash_{\mathsf{aq}} (a :: \mathrm{s}) : \kappa \rightsquigarrow a : \sigma; x.\tau\, x \dashv \Omega_1, \Omega_2} \quad \text{IAQVarC\_Annot}$$

Given $\Sigma; \Psi \vDash_{\mathsf{ty}} \tau_0 : \kappa$, we must show $\Sigma; \Psi, \Omega_1, \Omega_2 \vDash_{\mathsf{ty}} (\tau\, x)[\tau_0/x] : \sigma$, which can be rewritten to $\Sigma; \Psi, \Omega_1, \Omega_2 \vDash_{\mathsf{ty}} \tau\, \tau_0 : \sigma$. We know $\Sigma \vDash_{\mathsf{ctx}} \Psi\ \mathsf{ok}$ by Lemma E.8. The induction hypothesis then tells us $\Sigma; \mathsf{Rel}(\Psi, \Omega_1) \vDash_{\mathsf{ty}} \sigma : \mathbf{Type}$. Lemma E.17 tells us $\Sigma; \mathsf{Rel}(\Psi) \vDash_{\mathsf{ty}} \kappa : \mathbf{Type}$. We can thus use Lemma E.41 to get $\Sigma; \Psi, \Omega_1, \Omega_2 \vDash_{\mathsf{ty}} \tau : \tilde{\Pi} x{:}_{\mathsf{Rel}}\kappa.\, \sigma$. Rule Ty_AppRel gives us $\Sigma; \Psi, \Omega_1, \Omega_2 \vDash_{\mathsf{ty}} \tau\, \tau_0 : \sigma[\tau_0/x]$, but Lemma E.11 tells us that the substitution in the kind has no effect. We are done.

□

Lemma E.43 (Declarations)**.** *If* $\Sigma \vdash_{\mathsf{ctx}} \Gamma\ \mathsf{ok}$ *and* $\Sigma; \Gamma \vdash_{\mathsf{decl}} \mathrm{decl} \rightsquigarrow x : \kappa := \tau$*, then* $\mathsf{dom}(\mathrm{decl}) = x$ *and* $\Sigma; \Gamma \vDash_{\mathsf{ty}} \tau : \kappa$*.*

Proof. By case analysis on the type inference judgment.

Case IDecl_Synthesize:

$$\frac{\begin{array}{l}\Sigma; \Gamma \vdash_{\mathsf{ty}} \mathrm{t} \rightsquigarrow \tau : \kappa \dashv \Omega \\ \Sigma; \Gamma \vdash_{\mathsf{solv}} \Omega \rightsquigarrow \Delta; \Theta \\ \tau' = \lambda\Delta.\, (\tau[\Theta]) \qquad \kappa' = \tilde{\Pi}_{\mathsf{Inf}}\Delta.\, (\kappa[\Theta])\end{array}}{\Sigma; \Gamma \vdash_{\mathsf{decl}} x := \mathrm{t} \rightsquigarrow x : \kappa' := \tau'} \quad \text{IDecl\_Synthesize}$$

By Lemma E.3, we have $\Sigma \vDash_{\mathsf{ctx}} \Gamma\ \mathsf{ok}$. We then use Lemma E.42 to get $\Sigma; \Gamma, \Omega \vDash_{\mathsf{ty}} \tau : \kappa$. Lemma E.8 gives us $\Sigma \vDash_{\mathsf{ctx}} \Gamma, \Omega\ \mathsf{ok}$. Property E.24 tells us that Θ is idempotent, that $\Sigma \vDash_{\mathsf{ctx}} \Gamma, \Delta\ \mathsf{ok}$, and that $\Sigma; \Gamma, \Delta \vDash_{\mathsf{z}} \Theta : \Omega$. Lemma E.9 gives us $\Sigma; \Gamma, \Delta, \Omega \vDash_{\mathsf{ty}} \tau : \kappa$. We can then use Lemma E.23 to get $\Sigma; \Gamma, \Delta \vDash_{\mathsf{ty}} \tau[\Theta] : \kappa[\Theta]$. Rule Ty_Lam (used repeatedly) gives us $\Sigma; \Gamma \vDash_{\mathsf{ty}} \tau' : \kappa'$. Lemma E.3 gives $\Sigma; \Gamma \vdash_{\mathsf{ty}} \tau' : \kappa'$ as desired.

Case IDecl_Check:

$$\frac{\begin{array}{l}\Sigma;\Gamma \vdash^{\Rightarrow}_{\mathsf{pt}} \mathrm{s} \rightsquigarrow \sigma \dashv \Omega_1 \\ \Sigma;\mathsf{Rel}(\Gamma) \vdash^{\Rightarrow}_{\mathsf{solv}} \mathsf{Rel}(\Omega_1) \rightsquigarrow \Delta_1;\Theta_1 \\ \sigma' = \Pi_{\mathsf{Inf}}\Delta_1.\,(\sigma[\Theta_1]) \\ \Sigma;\Gamma \vdash^{\Rightarrow *}_{\mathsf{ty}} \mathrm{t} : \sigma' \rightsquigarrow \tau \dashv \Omega_2 \\ \Sigma;\Gamma \vdash^{\Rightarrow}_{\mathsf{solv}} \Omega_2 \rightsquigarrow \varnothing;\Theta_2 \\ \tau' = \tau[\Theta_2]\end{array}}{\Sigma;\Gamma \vdash^{\Rightarrow}_{\mathsf{decl}} x :: \mathrm{s} := \mathrm{t} \rightsquigarrow x : \sigma' := \tau'} \quad \textsc{IDecl\_Check}$$

Lemma E.3 provides $\Sigma \vDash_{\mathsf{ctx}} \Gamma\ \mathsf{ok}$. We then use Lemma E.42 to get $\Sigma;\mathsf{Rel}(\Gamma,\Omega_1) \vDash_{\mathsf{ty}} \sigma : \mathbf{Type}$. Lemma E.8 gives us $\Sigma \vDash_{\mathsf{ctx}} \mathsf{Rel}(\Gamma,\Omega_1)\ \mathsf{ok}$. Property E.24 tells us Θ_1 is idempotent, $\Sigma \vDash_{\mathsf{ctx}} \mathsf{Rel}(\Gamma),\Delta_1\ \mathsf{ok}$, and $\Sigma;\mathsf{Rel}(\Gamma),\Delta_1 \vDash_{\mathsf{z}} \Theta_1 : \mathsf{Rel}(\Omega_1)$. Lemma E.9 gives us $\Sigma;\mathsf{Rel}(\Gamma),\Delta_1,\mathsf{Rel}(\Omega_1) \vDash_{\mathsf{ty}} \sigma : \mathbf{Type}$. Lemma E.23 then says that $\Sigma;\mathsf{Rel}(\Gamma),\Delta_1 \vDash_{\mathsf{ty}} \sigma[\Theta_1] : \mathbf{Type}$. Lemma C.6 gives us $\Sigma;\mathsf{Rel}(\Gamma),\mathsf{Rel}(\Delta_1) \vDash_{\mathsf{ty}} \sigma[\Theta_1] : \mathbf{Type}$ and thus we can use Ty_Pi repeatedly to get $\Sigma;\mathsf{Rel}(\Gamma) \vDash_{\mathsf{ty}} \sigma' : \mathbf{Type}$. A second use of Lemma E.42 gives us $\Sigma;\Gamma,\Omega_2 \vDash_{\mathsf{ty}} \tau : \sigma'$. A second use of Property E.24 tells us Θ_2 is idempotent and $\Sigma;\Gamma \vDash_{\mathsf{z}} \Theta_2 : \Omega_2$. We can thus use Lemma E.23 once again to tell us $\Sigma;\Gamma \vDash_{\mathsf{ty}} \tau[\Theta_2] : \sigma'[\Theta_2]$, except that Lemma E.11 tells us that zonking the kind has no effect. Thus $\Sigma;\Gamma \vDash_{\mathsf{ty}} \tau' : \sigma'$, and Lemma E.3 gives us $\Sigma;\Gamma \vdash_{\mathsf{ty}} \tau' : \sigma'$ as desired.

□

Theorem E.44 (Full program elaboration is sound)**.** *If* $\Sigma \vdash_{\mathsf{ctx}} \Gamma\ \mathsf{ok}$ *and* $\Sigma;\Gamma \vdash^{\Rightarrow}_{\mathsf{prog}} \mathrm{prog} \rightsquigarrow \Gamma';\theta$*, then:*

1. $\Sigma \vdash_{\mathsf{ctx}} \Gamma,\Gamma'\ \mathsf{ok}$
2. $\Sigma;\Gamma \vdash_{\mathsf{subst}} \theta : \Gamma'$
3. $\mathsf{dom}(\mathrm{prog}) \subseteq \mathsf{dom}(\Gamma')$

Proof. By induction on the type inference judgment.

Case IProg_Nil: Trivial.

Case IProg_Decl:

$$\frac{\begin{array}{c}\Sigma;\Gamma \vdash^{\Rightarrow}_{\mathsf{decl}} \mathrm{decl} \rightsquigarrow x : \kappa := \tau \\ \Sigma;\Gamma, x{:}_{\mathsf{Rel}}\kappa,\, c{:}x \sim \tau \vdash^{\Rightarrow}_{\mathsf{prog}} \mathrm{prog} \rightsquigarrow \Gamma';\theta\end{array}}{\Sigma;\Gamma \vdash^{\Rightarrow}_{\mathsf{prog}} \mathrm{decl};\mathrm{prog} \rightsquigarrow x{:}_{\mathsf{Rel}}\kappa,\, c{:}x \sim \tau,\Gamma';(\tau/x,\langle\tau\rangle/c)\circ\theta} \quad \textsc{IProg\_Decl}$$

Lemma E.43 tells us that $x = \mathsf{dom}(\mathrm{decl})$ and $\Sigma;\Gamma \vdash_{\mathsf{ty}} \tau : \kappa$. We must show $\Sigma \vdash_{\mathsf{ctx}} \Gamma, x{:}_{\mathsf{Rel}}\kappa, c{:}x \sim \tau\ \mathsf{ok}$. To do this, we need only $\Sigma;\mathsf{Rel}(\Gamma) \vdash_{\mathsf{ty}} \kappa : \mathbf{Type}$, which we get from Lemma C.43. We can then use the induction hypothesis to get $\Sigma \vdash_{\mathsf{ctx}} \Gamma, x{:}_{\mathsf{Rel}}\kappa, c{:}x \sim \tau,\Gamma'\ \mathsf{ok}$, $\Sigma;\Gamma, x{:}_{\mathsf{Rel}}\kappa, c{:}x \sim \tau \vdash_{\mathsf{subst}} \theta : \Gamma'$, and

$\mathsf{dom}(\text{prog}) \subseteq \mathsf{dom}(\Gamma')$. We already have that the outgoing context is well-formed and the domain condition. We need only show that $\Sigma; \Gamma \vdash_{\mathsf{subst}} (\tau/x, \langle\tau\rangle/c) \circ \theta : x{:}_{\mathsf{Rel}}\kappa, c{:}x \sim \tau, \Gamma'$. Let $\theta' = (\tau/x, \langle\tau\rangle/c) \circ \theta$. We will work backwards. The last step will be SUBST_TYREL. We must show $\Sigma; \Gamma \vdash_{\mathsf{ty}} x[\theta'] : \kappa$ and $\Sigma; \Gamma \vdash_{\mathsf{subst}} \theta' : (c{:}x \sim \tau, \Gamma')[\tau/x]$. We have already established the former (noting that $x[\theta'] = \tau$). We rewrite the latter as $\Sigma; \Gamma \vdash_{\mathsf{subst}} \theta' : c{:}\tau \sim \tau, \Gamma'[\tau/x]$. This will be shown by SUBST_CO. We must then show $\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{co}} c[\theta'] : \tau \sim \tau$ and $\Sigma; \Gamma \vdash_{\mathsf{subst}} \theta' : \Gamma'[\tau/x][\langle\tau\rangle/c]$. The former is straightforwardly by CO_REFL and Lemma C.6. For the latter, recall that we know $\Sigma; \Gamma, x{:}_{\mathsf{Rel}}\kappa, c{:}x \sim \tau \vdash_{\mathsf{subst}} \theta : \Gamma'$. Thus, two uses of Lemma E.29 gives us $\Sigma; \Gamma \vdash_{\mathsf{subst}} \theta' : \Gamma'[\tau/x][\langle\tau\rangle/c]$ as desired. We are done.

□

E.10 Conservativity with respect to OUTSIDEIN

.

This section assumes the introduction to Section 6.8.2, where much of the conservativity argument is given.

Claim E.45 (Expressions)**.** *If* $\Gamma \overset{\text{OI}}{\Rightarrow} \mathrm{t} : \kappa \rightsquigarrow Q_{\mathrm{w}}$ *under axiom set* $\mathcal{Q}$ *and signature* Σ*, then* $\Sigma; \Gamma, \mathsf{encode}(\mathcal{Q}) \Rightarrow_{\mathsf{ty}} \mathrm{t} \rightsquigarrow \cdot : \kappa \dashv \overline{\alpha}{:}_{\mathsf{Irrel}}\mathbf{Type}, \mathsf{encode}(Q_{\mathrm{w}})$ *where* $\overline{\alpha} = \mathsf{fuv}(\kappa) \cup \mathsf{fuv}(Q_{\mathrm{w}})$.

Proof sketch. **Case VARCON:** OUTSIDEIN fully instantiates all variables, producing unification variables for any quantified type variables and emitting wanted constraints for any constraint in the variable's type. BAKE does the same, via its $\Rightarrow_{\mathsf{inst}}$ judgment.

Case APP: In this case, $\mathrm{t} = \mathrm{t}_1\,\mathrm{t}_2$. OUTSIDEIN's is a fairly typicaly application form rule, but using constraints to assert that the type of t_1 is indeed a function. BAKE works similarly, using its $\Rightarrow_{\mathsf{fun}}$ judgment to assert that a type is a Π-type. Of course, OUTSIDEIN's treatment of t_2 uses synthesis while BAKE's uses its checking judgment.

Case ABS: This rule coresponds quite closely to BAKE's ITY_LAM rule. Note that OUTSIDEIN does not permit annotations on λ-bound variables, simplifying the treatment of abstractions. Furthermore, BAKE must use its generalization judgment (written with $\hookrightarrow$) to handle its unification variables, while OUTSIDEIN does not need to have this complication.

Case CASE: Contrast OUTSIDEIN's rule with BAKE's:

$$\frac{\begin{array}{l}\Gamma \vdash e : \tau \rightsquigarrow C \quad \beta, \overline{\gamma} \text{ fresh} \\ K_i{:}\forall \overline{a}\overline{b}_i.Q_i \Rightarrow \overline{v}_i \to \mathtt{T}\,\overline{a} \quad \overline{b}_i \text{ fresh} \\ \Gamma, (\overline{x_i{:}[\overline{a \mapsto \gamma}]v_i}) \vdash e_i : \tau_i \rightsquigarrow C_i \quad \overline{\delta}_i = fuv(\tau_i, C_i) - fuv(\Gamma, \overline{\gamma}) \\ C_i' = \begin{cases} C_i \wedge \tau_i \sim \beta & \text{if } \overline{b}_i = \epsilon \text{ and } Q_i = \epsilon \\ \exists \overline{\delta}_i.([\overline{a \mapsto \gamma}]Q_i \supset C_i \wedge \tau_i \sim \beta) & \text{otherwise} \end{cases}\end{array}}{\Gamma \vdash \mathtt{case}\, e\, \mathtt{of}\, \{\overline{K_i\, \overline{x}_i \to e_i}\} : \beta \rightsquigarrow C \wedge (\mathtt{T}\,\overline{\gamma} \sim \tau) \wedge (\bigwedge C_i')} \text{ CASE}$$

$$\frac{\begin{array}{l}\Sigma; \Psi \overset{}{\underset{\mathsf{ty}}{\Rightarrow}} \mathrm{t}_0 \rightsquigarrow \tau_0 : \kappa_0 \dashv \Omega_0 \\ \Sigma; \Psi, \Omega_0 \overset{}{\underset{\mathsf{scrut}}{\Rightarrow}} \overline{\mathrm{alt}}; \kappa_0 \rightsquigarrow \gamma; \Delta; H'; \overline{\tau} \dashv \Omega_0' \\ \mathsf{fresh}\,\alpha \qquad \Omega' = \Omega_0, \Omega_0', \alpha{:}_{\mathsf{Irrel}}\mathbf{Type} \\ \forall i,\ \Sigma; \Psi, \Omega'; \text{'}\Pi\Delta.\, H'\,\overline{\tau}; \tau_0 \rhd \gamma \overset{}{\underset{\mathsf{alt}}{\Rightarrow}} \mathrm{alt}_i : \alpha \rightsquigarrow alt_i \dashv \Omega_i \\ \overline{alt}' = \mathsf{make\_exhaustive}(\overline{alt}; \kappa)\end{array}}{\Sigma; \Psi \overset{*}{\underset{\mathsf{ty}}{\Rightarrow}} \mathbf{case}\, \mathrm{t}_0\, \mathbf{of}\, \overline{\mathrm{alt}} \rightsquigarrow \mathbf{case}_\alpha\, (\tau_0 \rhd \gamma)\, \mathbf{of}\, \overline{alt}' : \alpha \dashv \Omega', \overline{\Omega}} \text{ ITY\_CASE}$$

$$\frac{\begin{array}{l}\Sigma \overset{}{\underset{\mathsf{tc}}{\vdash}} H : \Delta_1; \Delta_2; H' \qquad \Delta_3, \Delta_4 = \Delta_2[\overline{\tau}/\mathsf{dom}(\Delta_1)] \\ \mathsf{dom}(\Delta_3) = \overline{x} \qquad \mathsf{dom}(\Delta_4) = \mathsf{dom}(\Delta') \\ \mathsf{match}_{\{\mathsf{dom}(\Delta_3)\}}(\mathsf{types}(\Delta_4); \mathsf{types}(\Delta')) = \mathsf{Just}\,\theta \\ \Sigma; \Psi, \Delta_3 \overset{}{\underset{\mathsf{ty}}{\Rightarrow}} \mathrm{t} : \kappa \rightsquigarrow \tau \dashv \Omega \\ \Omega \hookrightarrow \Delta_3 \rightsquigarrow \Omega'; \xi \\ \Delta_3' = \Delta_3, c{:}\tau_0 \sim H_{\{\overline{\tau}\}}\,\overline{x}\end{array}}{\Sigma; \Psi; \text{'}\Pi\Delta'.\, H'\,\overline{\tau}; \tau_0 \overset{}{\underset{\mathsf{alt}}{\Rightarrow}} H\,\overline{x} \to \mathrm{t} : \kappa \rightsquigarrow H \to \lambda\Delta_3'.\, (\tau[\xi]) \dashv \Omega'} \text{ IALT\_CON}$$

The first premises line up well, with both checking the scrutinee. Both rules then must ensure that the scrutinee's type is headed by a type constant ($\mathtt{T}$ in OUTSIDEIN, H in BAKE). This is done via the emission of a constraint in OUTSIDEIN (the $\mathtt{T}\,\overline{\gamma} \sim \tau$ constraint in the conclusion) and the use of $\overset{}{\underset{\mathsf{scrut}}{\Rightarrow}}$ in BAKE. One reason for a difference in treatment here is that BAKE wishes to use any information available because of the existence of its checking judgment, whereas OUTSIDEIN is free to invent new, uninformative unification variables ($\overline{\gamma}$). This difference makes BAKE produce the unification variables only when the scrutinee's type (κ_0) is not already manifestly the right shape.

Both rules then check the individual alternatives, which have to look up the constructor (K_i and H, respectively) in the environment. PICO gathers the three sorts of existentials together in Δ_2; OUTSIDEIN expands this out to the $\overline{b}_i$, Q_i, and $\overline{v}_i$. OUTSIDEIN does not permit unsaturated matching, so we can treat Δ_4 as empty. OUTSIDEIN does not consider scoped type variables; it thus only brings in $\overline{x}_i$ of types $\overline{v}_i$ while checking each alternative; BAKE brings all of Δ_3 into scope. OUTSIDEIN checks the synthesized type of each alternative, τ_i against the overall result type β; BAKE ensures the types of the alternatives line up by using a checking judgment against the result kind κ. (Note that, like OUTSIDEIN, this result kind is a unification variable. We can see that the κ in

IAlt_Con is the α from ITy_Case.)

The constraint C_i' emitted by OutsideIn is delicately constructed. If there are no existential type variables and no local constraints, OutsideIn emits a simple constraint. Otherwise, it has to emit an implication constraint. OutsideIn's implication constraints allow unification variables to be local to a certain constraint; the treatment of implication constraints is somewhat different than that of simple constraints. In particular, when solving an implication constraint, any unification variables that arose outside of that implication are considered untouchable—they cannot then be unified. (See Section 6.3.3.) In order to avoid imposing the untouchability restriction, OutsideIn makes a simple constraint when possible. Due to Bake's more uniform treatment of implications, this distinction is not necessary; untouchability is informed by the order of unification variables in the context passed to the solver.

Bake makes its version of implication constraints via the generalization judgment (written with $\hookrightarrow$). All of the constraints generated while checking the alternative are quantified over variables in Δ_3; this precisely corresponds to the apapearance of the Q_i to the left of the $\supset$ in OutsideIn's constraint C_i'. (Recall that Q_i is a component of Bake's Δ_3.) Bake also adds another equality assumption into its Δ_3'; this equality has to do with dependent pattern matching (Section 4.3.3) and has no place in the non-dependent language of OutsideIn.

Case Let: Other than Bake's generalization step, these rules line up perfectly.

Cases LetA and GLetA: These cases cover annotated **let**s, where the bound variable is also given a type. I do not consider this form separately, instead preferring to use the checking judgments to handle this case.

□

E.11 Conservativity with respect to System SB

This section compares Bake with System SB, as presented by Eisenberg et al. [33, Figure 8]. Omitted elaborations are denoted with $\cdot$. Please refer to the introduction to Section 6.8.3 for changes needed to both System SB and Bake in order to prove the following claims.

Claim E.46 (System SB Subsumption)**.** *If* $\kappa_1 \leq_{\mathsf{dsk}} \kappa_2$*, then* $\kappa_1 \leq \kappa_2 \rightsquigarrow \cdot \dashv \Omega$*.*

Proof sketch. Note that this refers to the judgment in Eisenberg et al. [33, bottom of Figure 9]. If we assume all relevances are Rel, the rules of the Bake subsumption judgments are identical to those of the SB judgments. The one exception is the comparison between DSK_Refl and ISub_Unify, where Bake emits an equality constraint instead of simply asserting that the two types are equal. This variance is addressed by the tweaks above, and thus we are done. □

Claim E.47 (Conservativity with respect to System SB). *Assume* $\Psi \approx \Gamma$.

1. *If* $\Gamma \vdash_{\mathsf{sb}} \mathrm{t} \Rightarrow \kappa$, *then* $\Sigma; \Psi \Vdash_{\mathsf{ty}} \mathrm{t} \rightsquigarrow \cdot : \kappa \dashv \Omega$.
2. *If* $\Gamma \vdash^*_{\mathsf{sb}} \mathrm{t} \Rightarrow \kappa$, *then* $\Sigma; \Psi \Vdash^*_{\mathsf{ty}} \mathrm{t} \rightsquigarrow \cdot : \kappa \dashv \Omega$.
3. *If* $\Gamma \vdash_{\mathsf{sb}} \mathrm{t} \Leftarrow \kappa$, *then* $\Sigma; \Psi \Vdash_{\mathsf{ty}} \mathrm{t} : \kappa \rightsquigarrow \cdot \dashv \Omega$.
4. *If* $\Gamma \vdash^*_{\mathsf{sb}} \mathrm{t} \Leftarrow \kappa$, *then* $\Sigma; \Psi \Vdash^*_{\mathsf{ty}} \mathrm{t} : \kappa \rightsquigarrow \cdot \dashv \Omega$.

Proof sketch. By induction on the input derivation. This remains only a proof sketch because I have not concretely defined the tweaks above, nor have I given a full accounting of how types written in SB source are checked to become Pico types.

Case SB_Abs: We know here that $\mathrm{t} = \lambda x.\, \mathrm{t}_0$, $\kappa = \Pi_{\mathsf{Req}} x{:}_{\mathsf{Rel}} \alpha.\, \kappa_2$, and that $\Gamma, x{:}_{\mathsf{Rel}} \alpha \vdash^*_{\mathsf{sb}} \mathrm{t}_0 \Rightarrow \kappa_2$. We will use ITy_Lam. By IQVar_Req and IAQVar_Var, we can see that $\Sigma; \Psi \Vdash_{\mathsf{q}} x \rightsquigarrow x : \alpha; \mathsf{Req} \dashv \alpha{:}_{\mathsf{Irrel}} \mathbf{Type}$. The induction hypothesis tells us that $\Sigma; \Psi, \alpha{:}_{\mathsf{Irrel}} \mathbf{Type}, x{:}_{\mathsf{Rel}} \alpha \Vdash^*_{\mathsf{ty}} \mathrm{t}_0 \rightsquigarrow \cdot : \kappa_2 \dashv \Omega$. We can thus use ITy_Lam to get $\Sigma; \Psi \Vdash^*_{\mathsf{ty}} \lambda x.\, \mathrm{t}_0 \rightsquigarrow \cdot : \Pi_{\mathsf{Req}} x{:}_{\mathsf{Rel}} \alpha.\, \kappa_2 \dashv \alpha{:}_{\mathsf{Irrel}} \mathbf{Type}, \Omega$. We then must use ITy_Inst to remove the star from the judgment; however, given that the kind starts with a visible binder, instantiation has no effect, and we are thus done.

Case SB_InstS: By ITy_Inst, noting that Bake's instantiation operation $\Vdash_{\mathsf{inst}}$ behaves exactly like the manual instantation in SB_InstS's conclusion.

Case SB_Var: By ITy_Var.

Case SB_App: By ITy_App, noting that all visible quantification in System SB is relevant, and thus the $\Vdash^*_{\mathsf{arg}}$ judgment reduces to $\Vdash^*_{\mathsf{ty}}$.

Case SB_TApp: By ITy_AppSpec.

Case SB_Annot: Here, we know $\mathrm{t} = (\Lambda @ \overline{a}.\, \mathrm{t}_0) :: \mathrm{s}_0$ where $\Sigma; \Psi \Vdash_{\mathsf{pt}} \mathrm{s}_0 \rightsquigarrow \kappa \dashv \Omega_1$ and $\kappa = \Pi_{\mathsf{Spec}} \overline{a}{:}_{\mathsf{Irrel}} \mathbf{Type}, \overline{b}{:}_{\mathsf{Irrel}} \mathbf{Type}.\, \kappa_0$. Furthermore, we know $\Gamma, \overline{a}{:}_{\mathsf{Irrel}} \mathbf{Type} \vdash^*_{\mathsf{sb}} \mathrm{t}_0 \Leftarrow \kappa_0$. The induction hypothesis tells us $\Sigma; \Psi, \overline{a}{:}_{\mathsf{Irrel}} \mathbf{Type}, \$\overline{b}{:}_{\mathsf{Irrel}} \mathbf{Type} \Vdash^*_{\mathsf{ty}} \mathrm{t}_0 : \kappa_0 \rightsquigarrow \cdot \dashv \Omega_2$. We wish to use ITyC_Skol (repeatedly) to get from that last judgment to $\Sigma; \Psi, \overline{a}{:}_{\mathsf{Irrel}} \mathbf{Type} \Vdash^*_{\mathsf{ty}} \mathrm{t}_0 : \Pi_{\mathsf{Spec}} \$\overline{b}{:}_{\mathsf{Irrel}} \mathbf{Type}.\, \kappa_0 \rightsquigarrow \cdot \dashv \Omega_3$. Rename $\$\overline{b}$ to $\overline{b}$. We now wish to use ITyC_LamInvisIrrel (repeatedly). Consider one use. We can see that $\Sigma; \Psi \Vdash_{\mathsf{aq}} a : \mathbf{Type} \rightsquigarrow a : \mathbf{Type}; x.x \dashv \varnothing$. We know $\Sigma; \Psi, \overline{a}{:}_{\mathsf{Irrel}} \mathbf{Type} \Vdash^*_{\mathsf{ty}} \mathrm{t}_0 : \Pi_{\mathsf{Spec}} \overline{b}{:}_{\mathsf{Irrel}} \mathbf{Type}.\, \kappa_0 \rightsquigarrow \cdot \dashv \Omega_3$ and can thus conclude $\Sigma; \Psi \Vdash^*_{\mathsf{ty}} \Lambda @ a.\, \mathrm{t}_0 : \Pi_{\mathsf{Spec}} a{:}_{\mathsf{Irrel}} \mathbf{Type}, \overline{b}{:}_{\mathsf{Irrel}} \mathbf{Type}.\, \kappa_0 \rightsquigarrow \cdot \dashv \Omega_4$. Repeat this process to get $\Sigma; \Psi \Vdash^*_{\mathsf{ty}} \Lambda @ \overline{a}.\, \mathrm{t}_0 : \Pi_{\mathsf{Spec}} \overline{a}{:}_{\mathsf{Irrel}} \mathbf{Type}, \overline{b}{:}_{\mathsf{Irrel}} \mathbf{Type}.\, \kappa_0 \rightsquigarrow \cdot \dashv \Omega_5$. This can be rewritten as $\Sigma; \Psi \Vdash^*_{\mathsf{ty}} \Lambda @ \overline{a}.\, \mathrm{t}_0 : \kappa \rightsquigarrow \cdot \dashv \Omega_5$. Thus ITy_Annot gives us $\Sigma; \Psi \Vdash^*_{\mathsf{ty}} (\Lambda @ \overline{a}.\, \mathrm{t}_0) :: \mathrm{s}_0 \rightsquigarrow \cdot : \kappa \dashv \Omega_6$ as desired.

Case SB_Let: By ITy_Let.

Case SB_DLet: By ITyC_Let. Note that System SB's decision to put SB_DLet in the unstarred judgment is immaterial, as pointed out by Eisenberg et al. [33, footnote 13].

Case SB_DAbs: By ITyC_Lam.

Case SB_Infer: By ITyC_Infer and Claim E.46.

Case SB_DeepSkol: Recall that according to the tweaks I have made to this algorithm, this rule now does shallow skolemization. We are done by ITyC_Skol.

□

Appendix F

Proofs about PICO$^{\equiv}$

F.1 The PICO$^{\equiv}$ type system

The PICO$^{\equiv}$ system is introduced in Section 7.2; its rules, in full, are as follows:

$\boxed{\phi \equiv \phi'}$

$$\frac{\tau_1 \equiv \tau_1' \qquad \kappa_1 \equiv \kappa_1' \qquad \kappa_2 \equiv \kappa_2' \qquad \tau_2 \equiv \tau_2'}{\tau_1 \ {}^{\kappa_1}\!\sim^{\kappa_2} \tau_2 \equiv \tau_1' \ {}^{\kappa_1'}\!\sim^{\kappa_2'} \tau_2} \quad \text{DE\_PROP}$$

$\boxed{alt \equiv alt'}$

$$\frac{\tau \equiv \tau'}{\pi \to \tau \equiv \pi \to \tau'} \quad \text{DE\_ALT}$$

$\boxed{\overline{\psi} \equiv \overline{\psi}'}$

$$\frac{}{\varnothing \equiv \varnothing} \quad \text{DE\_VECNIL}$$

$$\frac{\tau \equiv \tau' \qquad \overline{\psi} \equiv \overline{\psi}'}{\tau, \overline{\psi} \equiv \tau', \overline{\psi}'} \quad \text{DE\_VECTYREL}$$

$$\frac{\tau \equiv \tau' \qquad \overline{\psi} \equiv \overline{\psi}'}{\{\tau\}, \overline{\psi} \equiv \{\tau'\}, \overline{\psi}'} \quad \text{DE\_VECTYIRREL}$$

$$\frac{\overline{\psi} \equiv \overline{\psi}'}{\gamma, \overline{\psi} \equiv \gamma', \overline{\psi}'} \quad \text{DE\_VECCO}$$

$\boxed{\Gamma \equiv \Gamma'}$

$$\frac{}{\varnothing \equiv \varnothing} \quad \text{DE\_CtxNil}$$

$$\frac{\kappa \equiv \kappa' \qquad \Gamma \equiv \Gamma'}{a{:}_\rho \kappa, \Gamma \equiv a{:}_\rho \kappa', \Gamma'} \quad \text{DE\_CtxTy}$$

$$\frac{\phi \equiv \phi' \qquad \Gamma \equiv \Gamma'}{c{:}\phi, \Gamma \equiv c{:}\phi', \Gamma'} \quad \text{DE\_CtxCo}$$

$\boxed{\Sigma; \Gamma \Vdash_{\mathsf{ty}} \tau : \kappa}$ Type formation

$$\frac{\Sigma \Vdash_{\mathsf{ctx}} \Gamma\ \mathsf{ok} \qquad a{:}_{\mathsf{Rel}} \kappa \in \Gamma}{\Sigma; \Gamma \Vdash_{\mathsf{ty}} a : \kappa} \quad \text{DTy\_Var}$$

$$\frac{\begin{array}{l}\Sigma \vdash_{\mathsf{tc}} H : \Delta_1; \Delta_2; H' \qquad \Sigma \Vdash_{\mathsf{ctx}} \Gamma\ \mathsf{ok} \\ \Sigma; \mathsf{Rel}(\Gamma) \Vdash_{\mathsf{vec}} \overline{\tau} : \mathsf{Rel}(\Delta_1)\end{array}}{\Sigma; \Gamma \Vdash_{\mathsf{ty}} H_{\{\overline{\tau}\}} : \Pi(\Delta_2[\overline{\tau}/\mathsf{dom}(\Delta_1)]).\, H'\, \overline{\tau}} \quad \text{DTy\_Con}$$

$$\frac{\begin{array}{ll}\Sigma; \Gamma \Vdash_{\mathsf{ty}} \tau_1 : \kappa_0 & \kappa_0 \overset{\rightarrow}{\equiv} \Pi a{:}_{\mathsf{Rel}} \kappa_1.\, \kappa_2 \\ \Sigma; \Gamma \Vdash_{\mathsf{ty}} \tau_2 : \kappa_1' & \kappa_1 \equiv \kappa_1'\end{array}}{\Sigma; \Gamma \Vdash_{\mathsf{ty}} \tau_1\, \tau_2 : \kappa_2[\tau_2/a]} \quad \text{DTy\_AppRel}$$

$$\frac{\begin{array}{ll}\Sigma; \Gamma \Vdash_{\mathsf{ty}} \tau_1 : \kappa_0 & \kappa_0 \overset{\rightarrow}{\equiv} \Pi a{:}_{\mathsf{Irrel}} \kappa_1.\, \kappa_2 \\ \Sigma; \mathsf{Rel}(\Gamma) \Vdash_{\mathsf{ty}} \tau_2 : \kappa_1' & \kappa_1 \equiv \kappa_1'\end{array}}{\Sigma; \Gamma \Vdash_{\mathsf{ty}} \tau_1\, \{\tau_2\} : \kappa_2[\tau_2/a]} \quad \text{DTy\_AppIrrel}$$

$$\frac{\begin{array}{ll}\Sigma; \Gamma \Vdash_{\mathsf{ty}} \tau : \kappa_0 & \kappa_0 \overset{\rightarrow}{\equiv} \Pi c{:}\phi.\, \kappa \\ \Sigma; \mathsf{Rel}(\Gamma) \Vdash_{\mathsf{co}} \gamma : \phi' & \phi \equiv \phi'\end{array}}{\Sigma; \Gamma \Vdash_{\mathsf{ty}} \tau\, \gamma : \kappa[\gamma/c]} \quad \text{DTy\_CApp}$$

$$\frac{\Sigma; \Gamma, \mathsf{Rel}(\delta) \Vdash_{\mathsf{ty}} \kappa : \tau \qquad \tau \equiv \mathbf{Type}}{\Sigma; \Gamma \Vdash_{\mathsf{ty}} \Pi\delta.\, \kappa : \mathbf{Type}} \quad \text{DTy\_Pi}$$

$$\frac{\begin{array}{ll}\Sigma; \mathsf{Rel}(\Gamma) \Vdash_{\mathsf{co}} \gamma : \kappa_1 \sim \kappa_2 & \\ \Sigma; \Gamma \Vdash_{\mathsf{ty}} \tau : \kappa_1' & \Sigma; \mathsf{Rel}(\Gamma) \Vdash_{\mathsf{ty}} \kappa_2 : \sigma \\ \kappa_1 \equiv \kappa_1' & \sigma \equiv \mathbf{Type}\end{array}}{\Sigma; \Gamma \Vdash_{\mathsf{ty}} \tau \rhd \gamma : \kappa_2} \quad \text{DTy\_Cast}$$

$$\frac{\begin{array}{l}\Sigma; \mathsf{Rel}(\Gamma) \Vdash_{\mathsf{ty}} \kappa : \tau_0 \qquad \tau_0 \equiv \mathbf{Type} \qquad \Sigma; \Gamma \Vdash_{\mathsf{ty}} \tau : \sigma \\ \sigma \overset{\rightarrow}{\equiv} \tilde{\Pi}\Delta.\, H\,\overline{\sigma} \\ \Sigma; \mathsf{Rel}(\Gamma) \Vdash_{\mathsf{ty}} H\,\overline{\sigma} : \tau_1 \\ \tau_1 \equiv \mathbf{Type} \\ \forall i,\ \Sigma; \Gamma; \tilde{\Pi}\Delta.\, H\,\overline{\sigma} \Vdash^{\tau}_{\mathsf{alt}} alt_i : \kappa \\ \overline{alt} \text{ are exhaustive and distinct for } H, \text{ (w.r.t. } \Sigma)\end{array}}{\Sigma; \Gamma \Vdash_{\mathsf{ty}} \mathbf{case}_{\kappa}\,\tau\,\mathbf{of}\,\overline{alt} : \kappa}\ \textsc{DTy\_Case}$$

$$\frac{\Sigma; \Gamma, \delta \Vdash_{\mathsf{ty}} \tau : \kappa}{\Sigma; \Gamma \Vdash_{\mathsf{ty}} \lambda\delta.\,\tau : \tilde{\Pi}\delta.\,\kappa}\ \textsc{DTy\_Lam}$$

$$\frac{\Sigma; \Gamma \Vdash_{\mathsf{ty}} \tau : \tilde{\Pi} a{:}_{\mathsf{Rel}}\kappa_1.\,\kappa_2 \qquad \kappa_1 \equiv \kappa_2}{\Sigma; \Gamma \Vdash_{\mathsf{ty}} \mathbf{fix}\,\tau : \kappa}\ \textsc{DTy\_Fix}$$

$$\frac{\begin{array}{l}\Sigma; \mathsf{Rel}(\Gamma) \Vdash_{\mathsf{co}} \gamma : H_{1\{\overline{\tau}_1\}}\,\overline{\psi}_1 \sim H_{2\{\overline{\tau}_2\}}\,\overline{\psi}_2 \qquad H_1 \neq H_2 \\ \Sigma; \mathsf{Rel}(\Gamma) \Vdash_{\mathsf{ty}} \tau : \kappa \qquad \kappa \equiv \mathbf{Type}\end{array}}{\Sigma; \Gamma \Vdash_{\mathsf{ty}} \mathbf{absurd}\,\gamma\,\tau : \tau}\ \textsc{DTy\_Absurd}$$

$\boxed{\Sigma; \Gamma; \sigma \Vdash^{\tau}_{\mathsf{alt}} alt : \kappa}$ Case alternatives

$$\frac{\begin{array}{l}\Sigma \vdash_{\mathsf{tc}} H : \Delta_1; \Delta_2; H' \qquad \Delta_3, \Delta_4 = \Delta_2[\overline{\sigma}/\mathsf{dom}(\Delta_1)] \\ \mathsf{dom}(\Delta_4) = \mathsf{dom}(\Delta') \\ \mathsf{match}_{\{\mathsf{dom}(\Delta_3)\}}(\mathsf{types}(\Delta_4); \mathsf{types}(\Delta')) = \mathsf{Just}\,\theta \\ \Sigma; \Gamma \Vdash_{\mathsf{ty}} \tau : \kappa_0 \qquad \kappa_0 \equiv \tilde{\Pi}\Delta_3, c{:}\tau_0 \sim H_{\{\overline{\sigma}\}}\,\mathsf{dom}(\Delta_3).\,\kappa\end{array}}{\Sigma; \Gamma; \tilde{\Pi}\Delta'.\, H'\,\overline{\sigma} \Vdash^{\tau_0}_{\mathsf{alt}} H \to \tau : \kappa}\ \textsc{DAlt\_Match}$$

$$\frac{\Sigma; \Gamma \Vdash_{\mathsf{ty}} \tau : \kappa' \qquad \kappa' \equiv \kappa}{\Sigma; \Gamma; \sigma \Vdash^{\tau_0}_{\mathsf{alt}} \_ \to \tau : \kappa}\ \textsc{DAlt\_Default}$$

$\boxed{\Sigma; \Gamma \Vdash_{\mathsf{co}} \gamma : \phi}$ Coercion formation

$$\frac{\Sigma \Vdash_{\mathsf{ctx}} \Gamma\ \mathsf{ok} \qquad c{:}\phi \in \Gamma}{\Sigma; \Gamma \Vdash_{\mathsf{co}} c : \phi}\ \textsc{DCo\_Var}$$

$$\frac{\Sigma; \Gamma \Vdash_{\mathsf{ty}} \tau : \kappa}{\Sigma; \Gamma \Vdash_{\mathsf{co}} \langle\tau\rangle : \tau \sim \tau}\ \textsc{DCo\_Refl}$$

$$\frac{\Sigma; \Gamma \Vdash_{\mathsf{co}} \gamma : \tau_1 \sim \tau_2}{\Sigma; \Gamma \Vdash_{\mathsf{co}} \mathbf{sym}\,\gamma : \tau_2 \sim \tau_1}\ \textsc{DCo\_Sym}$$

$$\frac{\Sigma;\Gamma \Vdash_{\mathsf{co}} \gamma_1 : \tau_1 \ {}^{\kappa_1}\!\sim^{\kappa_2} \tau_2 \qquad \Sigma;\Gamma \Vdash_{\mathsf{co}} \gamma_2 : \tau_2' \ {}^{\kappa_2'}\!\sim^{\kappa_3} \tau_3 \qquad \tau_2 \equiv \tau_2' \qquad \kappa_2 \equiv \kappa_2'}{\Sigma;\Gamma \Vdash_{\mathsf{co}} \gamma_1 \fatsemi \gamma_2 : \tau_1 \sim \tau_3} \ \text{DCo\_Trans}$$

$$\frac{\Sigma;\Gamma \Vdash_{\mathsf{co}} \eta : \kappa_1 \sim \kappa_2 \qquad \lfloor \tau_1 \rfloor = \lfloor \tau_2 \rfloor \qquad \Sigma;\Gamma \Vdash_{\mathsf{ty}} \tau_1 : \kappa_1' \qquad \Sigma;\Gamma \Vdash_{\mathsf{ty}} \tau_2 : \kappa_2' \qquad \kappa_1 \equiv \kappa_1' \qquad \kappa_2 \equiv \kappa_2'}{\Sigma;\Gamma \Vdash_{\mathsf{co}} \tau_1 \approx_\eta \tau_2 : \tau_1 \sim \tau_2} \ \text{DCo\_Coherence}$$

$$\frac{\forall i,\ \Sigma;\Gamma \Vdash_{\mathsf{co}} \gamma_i : \sigma_i \sim \sigma_i' \qquad \Sigma;\Gamma \Vdash_{\mathsf{ty}} H_{\{\overline{\sigma}\}} : \kappa_1 \qquad \Sigma;\Gamma \Vdash_{\mathsf{ty}} H_{\{\overline{\sigma}'\}} : \kappa_2}{\Sigma;\Gamma \Vdash_{\mathsf{co}} H_{\{\overline{\gamma}\}} : H_{\{\overline{\sigma}\}} \sim H_{\{\overline{\sigma}'\}}} \ \text{DCo\_Con}$$

$$\frac{\Sigma;\Gamma \Vdash_{\mathsf{co}} \gamma_1 : \tau_1 \sim \tau_2 \qquad \Sigma;\Gamma \Vdash_{\mathsf{co}} \gamma_2 : \sigma_1 \sim \sigma_2 \qquad \Sigma;\Gamma \Vdash_{\mathsf{ty}} \tau_1\,\sigma_1 : \kappa_1 \qquad \Sigma;\Gamma \Vdash_{\mathsf{ty}} \tau_2\,\sigma_2 : \kappa_2}{\Sigma;\Gamma \Vdash_{\mathsf{co}} \gamma_1\,\gamma_2 : \tau_1\,\sigma_1 \sim \tau_2\,\sigma_2} \ \text{DCo\_AppRel}$$

$$\frac{\Sigma;\Gamma \Vdash_{\mathsf{co}} \gamma_1 : \tau_1 \sim \tau_2 \qquad \Sigma;\Gamma \Vdash_{\mathsf{co}} \gamma_2 : \sigma_1 \sim \sigma_2 \qquad \Sigma;\Gamma \Vdash_{\mathsf{ty}} \tau_1\,\{\sigma_1\} : \kappa_1 \qquad \Sigma;\Gamma \Vdash_{\mathsf{ty}} \tau_2\,\{\sigma_2\} : \kappa_2}{\Sigma;\Gamma \Vdash_{\mathsf{co}} \gamma_1\,\{\gamma_2\} : \tau_1\,\{\sigma_1\} \sim \tau_2\,\{\sigma_2\}} \ \text{DCo\_AppIrrel}$$

$$\frac{\Sigma;\Gamma \Vdash_{\mathsf{co}} \gamma_0 : \tau_1 \sim \tau_2 \qquad \Sigma;\Gamma \Vdash_{\mathsf{ty}} \tau_1\,\gamma_1 : \kappa_1 \qquad \Sigma;\Gamma \Vdash_{\mathsf{ty}} \tau_2\,\gamma_2 : \kappa_2}{\Sigma;\Gamma \Vdash_{\mathsf{co}} \gamma_0\,(\gamma_1,\gamma_2) : \tau_1\,\gamma_1 \sim \tau_2\,\gamma_2} \ \text{DCo\_CApp}$$

$$\frac{\Sigma;\Gamma \Vdash_{\mathsf{co}} \eta : \kappa_1 \ {}^{\mathbf{Type}}\!\sim^{\mathbf{Type}} \kappa_2 \qquad \Sigma;\Gamma, a{:}_{\mathsf{Rel}}\kappa_1 \Vdash_{\mathsf{co}} \gamma : \sigma_1 \ {}^{\mathbf{Type}}\!\sim^{\mathbf{Type}} \sigma_2}{\Sigma;\Gamma \Vdash_{\mathsf{co}} \Pi a{:}_\rho \eta.\,\gamma : (\Pi a{:}_\rho \kappa_1.\,\sigma_1) \sim (\Pi a{:}_\rho \kappa_2.\,(\sigma_2[a \rhd \mathbf{sym}\,\eta/a]))} \ \text{DCo\_PiTy}$$

$$\frac{\Sigma;\Gamma \Vdash_{\mathsf{co}} \eta_1 : \tau_1 \sim \tau_2 \qquad \Sigma;\Gamma \Vdash_{\mathsf{co}} \eta_2 : \sigma_1 \sim \sigma_2 \qquad \Sigma;\Gamma, c{:}\tau_1 \sim \sigma_1 \Vdash_{\mathsf{co}} \gamma : \kappa_1 \ {}^{\mathbf{Type}}\!\sim^{\mathbf{Type}} \kappa_2 \qquad c \ \tilde{\#}\ \gamma \qquad \eta_3 = \eta_1 \fatsemi c \fatsemi \mathbf{sym}\,\eta_2}{\Sigma;\Gamma \Vdash_{\mathsf{co}} \Pi c{:}(\eta_1,\eta_2).\,\gamma : (\Pi c{:}\tau_1 \sim \sigma_1.\,\kappa_1) \sim (\Pi c{:}\tau_2 \sim \sigma_2.\,(\kappa_2[\eta_3/c]))} \ \text{DCo\_PiCo}$$

$$\frac{\begin{array}{l}\Sigma;\Gamma \Vdash_{\mathsf{co}} \eta : \kappa_1 \sim \kappa_2 \qquad \Sigma;\Gamma \Vdash_{\mathsf{co}} \gamma_0 : \tau_1 \sim \tau_2 \\ \forall i,\ \Sigma;\Gamma \Vdash_{\mathsf{co}} \gamma_i : \sigma_i \sim \sigma_i' \\ \overline{alt}_1 \;=\; \overline{\pi_i \to \sigma_i} \qquad \overline{alt}_2 \;=\; \overline{\pi_i \to \sigma_i'} \\ \Sigma;\Gamma \Vdash_{\mathsf{ty}} \mathbf{case}_{\kappa_1}\, \tau_1\, \mathbf{of}\, \overline{alt}_1 : \kappa_1 \qquad \Sigma;\Gamma \Vdash_{\mathsf{ty}} \mathbf{case}_{\kappa_2}\, \tau_2\, \mathbf{of}\, \overline{alt}_2 : \kappa_2\end{array}}{\Sigma;\Gamma \Vdash_{\mathsf{co}} \mathbf{case}_{\eta}\, \gamma_0\, \mathbf{of}\, \overline{\pi_i \to \gamma_i} : \mathbf{case}_{\kappa_1}\, \tau_1\, \mathbf{of}\, \overline{alt}_1 \sim \mathbf{case}_{\kappa_2}\, \tau_2\, \mathbf{of}\, \overline{alt}_2}\ \textsc{DCo\_Case}$$

$$\frac{\begin{array}{l}\Sigma;\Gamma \Vdash_{\mathsf{co}} \eta : \kappa_1 \sim \kappa_2 \\ \Sigma;\Gamma, a{:}_{\rho}\kappa_1 \Vdash_{\mathsf{co}} \gamma : \tau_1 \sim \tau_2 \\ \Sigma;\Gamma, a{:}_{\rho}\kappa_1 \Vdash_{\mathsf{ty}} \tau_1 : \sigma_1 \qquad \Sigma;\Gamma, a{:}_{\rho}\kappa_1 \Vdash_{\mathsf{ty}} \tau_2 : \sigma_2\end{array}}{\Sigma;\Gamma \Vdash_{\mathsf{co}} \lambda a{:}_{\rho}\eta.\, \gamma : \lambda a{:}_{\rho}\kappa_1.\, \tau_1 \sim \lambda a{:}_{\rho}\kappa_2.\, (\tau_2[a \rhd \mathbf{sym}\, \eta/a])}\ \textsc{DCo\_Lam}$$

$$\frac{\begin{array}{l}\Sigma;\Gamma \Vdash_{\mathsf{co}} \eta_1 : \tau_1 \sim \tau_2 \qquad \Sigma;\Gamma \Vdash_{\mathsf{co}} \eta_2 : \sigma_1 \sim \sigma_2 \\ \Sigma;\Gamma, c{:}\tau_1 \sim \sigma_1 \Vdash_{\mathsf{co}} \gamma : \kappa_1 \sim \kappa_2 \qquad c \mathrel{\tilde{\#}} \gamma \\ \eta_3 \;=\; \eta_1 \fatsemi c \fatsemi \mathbf{sym}\, \eta_2\end{array}}{\Sigma;\Gamma \Vdash_{\mathsf{co}} \lambda c{:}(\eta_1, \eta_2).\, \gamma : (\lambda c{:}\tau_1 \sim \sigma_1.\, \kappa_1) \sim (\lambda c{:}\tau_2 \sim \sigma_2.\, (\kappa_2[\eta_3/c]))}\ \textsc{DCo\_CLam}$$

$$\frac{\begin{array}{l}\Sigma;\Gamma \Vdash_{\mathsf{co}} \gamma : \tau_1 \sim \tau_2 \\ \Sigma;\Gamma \Vdash_{\mathsf{ty}} \mathbf{fix}\, \tau_1 : \kappa_1 \qquad \Sigma;\Gamma \Vdash_{\mathsf{ty}} \mathbf{fix}\, \tau_2 : \kappa_2\end{array}}{\Sigma;\Gamma \Vdash_{\mathsf{co}} \mathbf{fix}\, \gamma : \mathbf{fix}\, \tau_1 \sim \mathbf{fix}\, \tau_2}\ \textsc{DCo\_Fix}$$

$$\frac{\begin{array}{l}\Sigma;\Gamma \Vdash_{\mathsf{co}} \gamma_1 : H_{1\{\overline{\tau}_1\}}\, \overline{\psi}_1 \sim H'_{1\{\overline{\tau}'_1\}}\, \overline{\psi}'_1 \qquad H_1 \neq H'_1 \\ \Sigma;\Gamma \Vdash_{\mathsf{co}} \gamma_2 : H_{2\{\overline{\tau}_2\}}\, \overline{\psi}_2 \sim H'_{2\{\overline{\tau}'_2\}}\, \overline{\psi}'_2 \qquad H_2 \neq H'_2 \\ \Sigma;\Gamma \Vdash_{\mathsf{co}} \eta : \kappa_1 \mathrel{^{\tau_1}\!\sim^{\tau_2}} \kappa_2 \qquad \tau_1 \equiv \mathbf{Type} \qquad \tau_2 \equiv \mathbf{Type}\end{array}}{\Sigma;\Gamma \Vdash_{\mathsf{co}} \mathbf{absurd}\, (\gamma_1, \gamma_2)\, \eta : \mathbf{absurd}\, \gamma_1\, \kappa_1 \sim \mathbf{absurd}\, \gamma_2\, \kappa_2}\ \textsc{DCo\_Absurd}$$

$$\frac{\begin{array}{l}\Sigma;\Gamma \Vdash_{\mathsf{co}} \gamma : \tau_1 \sim \tau_2 \\ \tau_1 \mathrel{\vec{\equiv}} \Pi a{:}_{\rho}\kappa_1.\, \sigma_1 \qquad \tau_2 \mathrel{\vec{\equiv}} \Pi a{:}_{\rho}\kappa_2.\, \sigma_2\end{array}}{\Sigma;\Gamma \Vdash_{\mathsf{co}} \mathbf{argk}\, \gamma : \kappa_1 \sim \kappa_2}\ \textsc{DCo\_ArgK}$$

$$\frac{\begin{array}{l}\Sigma;\Gamma \Vdash_{\mathsf{co}} \gamma : \kappa_1 \sim \kappa_2 \\ \kappa_1 \mathrel{\vec{\equiv}} \Pi c{:}(\tau_1 \sim \tau'_1).\, \sigma_1 \qquad \kappa_2 \mathrel{\vec{\equiv}} \Pi c{:}(\tau_2 \sim \tau'_2).\, \sigma_2\end{array}}{\Sigma;\Gamma \Vdash_{\mathsf{co}} \mathbf{argk}_1\, \gamma : \tau_1 \sim \tau_2}\ \textsc{DCo\_CArgK1}$$

$$\frac{\begin{array}{l}\Sigma;\Gamma \Vdash_{\mathsf{co}} \gamma : \kappa_1 \sim \kappa_2 \\ \kappa_1 \mathrel{\vec{\equiv}} \Pi c{:}(\tau_1 \sim \tau'_1).\, \sigma_1 \qquad \kappa_2 \mathrel{\vec{\equiv}} \Pi c{:}(\tau_2 \sim \tau'_2).\, \sigma_2\end{array}}{\Sigma;\Gamma \Vdash_{\mathsf{co}} \mathbf{argk}_2\, \gamma : \tau'_1 \sim \tau'_2}\ \textsc{DCo\_CArgK2}$$

$$\frac{\begin{array}{l}\Sigma;\Gamma \Vdash_{\mathsf{co}} \gamma : \tau_1 \sim \tau_2 \\ \tau_1 \overset{\rightarrow}{\equiv} \lambda a{:}_\rho \kappa_1.\, \sigma_1 \qquad \tau_2 \overset{\rightarrow}{\equiv} \lambda a{:}_\rho \kappa_2.\, \sigma_2\end{array}}{\Sigma;\Gamma \Vdash_{\mathsf{co}} \mathbf{argk}\, \gamma : \kappa_1 \sim \kappa_2}\ \textsc{DCo\_ArgKLam}$$

$$\frac{\begin{array}{l}\Sigma;\Gamma \Vdash_{\mathsf{co}} \gamma : \kappa_1 \sim \kappa_2 \\ \kappa_1 \overset{\rightarrow}{\equiv} \lambda c{:}(\tau_1 \sim \tau_1').\, \sigma_1 \qquad \kappa_2 \overset{\rightarrow}{\equiv} \lambda c{:}(\tau_2 \sim \tau_2').\, \sigma_2\end{array}}{\Sigma;\Gamma \Vdash_{\mathsf{co}} \mathbf{argk}_1\, \gamma : \tau_1 \sim \tau_2}\ \textsc{DCo\_CArgKLam1}$$

$$\frac{\begin{array}{l}\Sigma;\Gamma \Vdash_{\mathsf{co}} \gamma : \kappa_1 \sim \kappa_2 \\ \kappa_1 \overset{\rightarrow}{\equiv} \lambda c{:}(\tau_1 \sim \tau_1').\, \sigma_1 \qquad \kappa_2 \overset{\rightarrow}{\equiv} \lambda c{:}(\tau_2 \sim \tau_2').\, \sigma_2\end{array}}{\Sigma;\Gamma \Vdash_{\mathsf{co}} \mathbf{argk}_2\, \gamma : \tau_1' \sim \tau_2'}\ \textsc{DCo\_CArgKLam2}$$

$$\frac{\begin{array}{l}\Sigma;\Gamma \Vdash_{\mathsf{co}} \gamma : \phi \qquad |\Delta_1| = |\Delta_2| = n \\ \phi \overset{\rightarrow}{\equiv} \Pi\Delta_1.\, \tau_1 \sim \Pi\Delta_2.\, \tau_2 \\ \Sigma;\Gamma \Vdash_{\mathsf{ty}} \tau_1 : \sigma_1 \qquad \sigma_1 \equiv \mathbf{Type} \\ \Sigma;\Gamma \Vdash_{\mathsf{ty}} \tau_2 : \sigma_2 \qquad \sigma_2 \equiv \mathbf{Type}\end{array}}{\Sigma;\Gamma \Vdash_{\mathsf{co}} \mathbf{res}^n\, \gamma : \tau_1 \sim \tau_2}\ \textsc{DCo\_Res}$$

$$\frac{\begin{array}{l}\Sigma;\Gamma \Vdash_{\mathsf{co}} \gamma : \phi \qquad |\Delta_1| = |\Delta_2| = n \\ \phi \overset{\rightarrow}{\equiv} \lambda\Delta_1.\, \tau_1 \sim \lambda\Delta_2.\, \tau_2 \\ \Sigma;\Gamma \Vdash_{\mathsf{ty}} \tau_1 : \kappa_1 \qquad \Sigma;\Gamma \Vdash_{\mathsf{ty}} \tau_2 : \kappa_2\end{array}}{\Sigma;\Gamma \Vdash_{\mathsf{co}} \mathbf{res}^n\, \gamma : \tau_1 \sim \tau_2}\ \textsc{DCo\_ResLam}$$

$$\frac{\begin{array}{l}\Sigma;\Gamma \Vdash_{\mathsf{co}} \gamma : \phi_1 \\ \phi_1 \overset{\rightarrow}{\equiv} \Pi a{:}_{\mathsf{Rel}} \kappa_1.\, \sigma_1 \sim \Pi a{:}_{\mathsf{Rel}} \kappa_2.\, \sigma_2 \\ \Sigma;\Gamma \Vdash_{\mathsf{co}} \eta : \tau_1 \sim^{\kappa_1'}_{\kappa_2'} \tau_2 \\ \kappa_1 \equiv \kappa_1' \qquad \kappa_2 \equiv \kappa_2'\end{array}}{\Sigma;\Gamma \Vdash_{\mathsf{co}} \gamma @ \eta : \sigma_1[\tau_1/a] \sim \sigma_2[\tau_2/a]}\ \textsc{DCo\_InstRel}$$

$$\frac{\begin{array}{l}\Sigma;\Gamma \Vdash_{\mathsf{co}} \gamma : \phi_1 \\ \phi_1 \overset{\rightarrow}{\equiv} \Pi a{:}_{\mathsf{Irrel}} \kappa_1.\, \sigma_1 \sim \Pi a{:}_{\mathsf{Irrel}} \kappa_2.\, \sigma_2 \\ \Sigma;\Gamma \Vdash_{\mathsf{co}} \eta : \tau_1 \sim^{\kappa_1'}_{\kappa_2'} \tau_2 \\ \kappa_1 \equiv \kappa_1' \qquad \kappa_2 \equiv \kappa_2'\end{array}}{\Sigma;\Gamma \Vdash_{\mathsf{co}} \gamma @ \{\eta\} : \sigma_1[\tau_1/a] \sim \sigma_2[\tau_2/a]}\ \textsc{DCo\_InstIrrel}$$

$$\frac{\begin{array}{l}\Sigma;\Gamma \Vdash_{\mathsf{co}} \eta_1 : \phi_3 \\ \phi_3 \overset{\rightarrow}{\equiv} \Pi c{:}\phi_1.\,\sigma_1 \sim \Pi c{:}\phi_2.\,\sigma_2 \\ \Sigma;\Gamma \Vdash_{\mathsf{co}} \gamma_1 : \phi_1' \qquad \phi_1 \equiv \phi_1' \qquad \Sigma;\Gamma \Vdash_{\mathsf{co}} \gamma_2 : \phi_2' \qquad \phi_2 \equiv \phi_2'\end{array}}{\Sigma;\Gamma \Vdash_{\mathsf{co}} \eta_1@(\gamma_1,\gamma_2) : \sigma_1[\gamma_1/c] \sim \sigma_2[\gamma_2/c]}\ \textsc{DCo\_CInst}$$

$$\frac{\begin{array}{l}\Sigma;\Gamma \Vdash_{\mathsf{co}} \gamma : \phi_1 \\ \phi_1 \overset{\rightarrow}{\equiv} \lambda a{:}_{\mathsf{Rel}}\kappa_1.\,\tau_1 \sim \lambda a{:}_{\mathsf{Rel}}\kappa_2.\,\tau_2 \\ \Sigma;\Gamma \Vdash_{\mathsf{co}} \eta : \sigma_1 \ {}^{\kappa_1'}\!\sim^{\kappa_2'} \sigma_2 \qquad \kappa_1 \equiv \kappa_1' \qquad \kappa_2 = \kappa_2'\end{array}}{\Sigma;\Gamma \Vdash_{\mathsf{co}} \gamma@\eta : \tau_1[\sigma_1/a] \sim \tau_2[\sigma_2/a]}\ \textsc{DCo\_InstLamRel}$$

$$\frac{\begin{array}{l}\Sigma;\Gamma \Vdash_{\mathsf{co}} \gamma : \phi_1 \\ \phi_1 \overset{\rightarrow}{\equiv} \lambda a{:}_{\mathsf{Irrel}}\kappa_1.\,\tau_1 \sim \lambda a{:}_{\mathsf{Irrel}}\kappa_2.\,\tau_2 \\ \Sigma;\Gamma \Vdash_{\mathsf{co}} \eta : \sigma_1 \ {}^{\kappa_1'}\!\sim^{\kappa_2'} \sigma_2 \qquad \kappa_1 \equiv \kappa_1' \qquad \kappa_2 \equiv \kappa_2'\end{array}}{\Sigma;\Gamma \Vdash_{\mathsf{co}} \gamma@\{\eta\} : \tau_1[\sigma_1/a] \sim \tau_2[\sigma_2/a]}\ \textsc{DCo\_InstLamIrrel}$$

$$\frac{\begin{array}{l}\Sigma;\Gamma \Vdash_{\mathsf{co}} \gamma : \phi_3 \\ \phi_3 \overset{\rightarrow}{\equiv} \lambda c{:}\phi_1.\,\sigma_1 \sim \lambda c{:}\phi_2.\,\sigma_2 \\ \Sigma;\Gamma \Vdash_{\mathsf{co}} \eta_1 : \phi_1' \qquad \phi_1 \equiv \phi_1' \\ \Sigma;\Gamma \Vdash_{\mathsf{co}} \eta_2 : \phi_2' \qquad \phi_2 \equiv \phi_2'\end{array}}{\Sigma;\Gamma \Vdash_{\mathsf{co}} \gamma@(\eta_1,\eta_2) : \sigma_1[\eta_1/c] \sim \sigma_2[\eta_2/c]}\ \textsc{DCo\_CInstLam}$$

$$\frac{\begin{array}{l}\Sigma;\Gamma \Vdash_{\mathsf{co}} \gamma : \phi \qquad \phi \overset{\rightarrow}{\equiv} H_{\{\overline{\kappa}\}}\,\overline{\psi} \sim H_{\{\overline{\kappa}'\}}\,\overline{\psi}' \\ \psi_i = \tau \qquad \psi_i' = \sigma \\ \Sigma;\Gamma \Vdash_{\mathsf{ty}} \tau : \kappa_1 \qquad \Sigma;\Gamma \Vdash_{\mathsf{ty}} \sigma : \kappa_2\end{array}}{\Sigma;\Gamma \Vdash_{\mathsf{co}} \mathbf{nth}_i\,\gamma : \tau \sim \sigma}\ \textsc{DCo\_NthRel}$$

$$\frac{\begin{array}{l}\Sigma;\Gamma \Vdash_{\mathsf{co}} \gamma : \phi \qquad \phi \overset{\rightarrow}{\equiv} H_{\{\overline{\kappa}\}}\,\overline{\psi} \sim H_{\{\overline{\kappa}'\}}\,\overline{\psi}' \\ \psi_i = \{\tau\} \qquad \psi_i' = \{\sigma\} \\ \Sigma;\mathsf{Rel}(\Gamma) \Vdash_{\mathsf{ty}} \tau : \kappa_1 \qquad \Sigma;\mathsf{Rel}(\Gamma) \Vdash_{\mathsf{ty}} \sigma : \kappa_2\end{array}}{\Sigma;\Gamma \Vdash_{\mathsf{co}} \mathbf{nth}_i\,\gamma : \tau \sim \sigma}\ \textsc{DCo\_NthIrrel}$$

$$\frac{\begin{array}{l}\Sigma;\Gamma \Vdash_{\mathsf{co}} \gamma : \phi \\ \phi \stackrel{\rightarrow}{\equiv} \tau_1\_\psi_1 \sim \tau_2\_\psi_2 \\ \Sigma;\Gamma \Vdash_{\mathsf{ty}} \tau_1 : \kappa_0 \qquad \kappa_0 \stackrel{\rightarrow}{\equiv} \Pi\delta_1.\,\kappa_1 \\ \Sigma;\Gamma \Vdash_{\mathsf{ty}} \tau_2 : \kappa_0' \qquad \kappa_0' \stackrel{\rightarrow}{\equiv} \Pi\delta_2.\,\kappa_2 \\ \Sigma;\Gamma \Vdash_{\mathsf{co}} \eta : \phi' \\ \phi' \equiv \Pi\delta_1.\,\kappa_1 \sim \Pi\delta_2.\,\kappa_2\end{array}}{\Sigma;\Gamma \Vdash_{\mathsf{co}} \mathbf{left}_\eta\,\gamma : \tau_1 \sim \tau_2}\ \textsc{DCo\_Left}$$

$$\frac{\begin{array}{l}\Sigma;\Gamma \Vdash_{\mathsf{co}} \gamma : \phi \\ \phi \stackrel{\rightarrow}{\equiv} \tau_1\_\sigma_1 \sim \tau_2\_\sigma_2 \\ \Sigma;\Gamma \Vdash_{\mathsf{ty}} \sigma_1 : \kappa_1 \qquad \Sigma;\Gamma \Vdash_{\mathsf{ty}} \sigma_2 : \kappa_2 \\ \Sigma;\Gamma \Vdash_{\mathsf{co}} \eta : \phi' \qquad \phi' \equiv \kappa_1 \sim \kappa_2\end{array}}{\Sigma;\Gamma \Vdash_{\mathsf{co}} \mathbf{right}_\eta\,\gamma : \sigma_1 \sim \sigma_2}\ \textsc{DCo\_RightRel}$$

$$\frac{\begin{array}{l}\Sigma;\Gamma \Vdash_{\mathsf{co}} \gamma : \phi \\ \phi \stackrel{\rightarrow}{\equiv} \tau_1\_\{\sigma_1\} \sim \tau_2\_\{\sigma_2\} \\ \Sigma;\Gamma \Vdash_{\mathsf{ty}} \sigma_1 : \kappa_1 \qquad \Sigma;\Gamma \Vdash_{\mathsf{ty}} \sigma_2 : \kappa_2 \\ \Sigma;\Gamma \Vdash_{\mathsf{co}} \eta : \phi' \qquad \phi' \equiv \kappa_1 \sim \kappa_2\end{array}}{\Sigma;\Gamma \Vdash_{\mathsf{co}} \mathbf{right}_\eta\,\gamma : \sigma_1 \sim \sigma_2}\ \textsc{DCo\_RightIrrel}$$

$$\frac{\Sigma;\Gamma \Vdash_{\mathsf{co}} \gamma : \tau_1 \ {}^{\kappa_1}\!\sim^{\kappa_2} \tau_2}{\Sigma;\Gamma \Vdash_{\mathsf{co}} \mathbf{kind}\,\gamma : \kappa_1 \sim \kappa_2}\ \textsc{DCo\_Kind}$$

$$\frac{\Sigma;\Gamma \Vdash_{\mathsf{ty}} \tau : \kappa \qquad \Sigma;\Gamma \Vdash_{\mathsf{ty}} \tau' : \kappa' \qquad \kappa \equiv \kappa' \qquad \Sigma;\Gamma \Vdash_{\mathsf{s}} \tau \longrightarrow \tau'}{\Sigma;\Gamma \Vdash_{\mathsf{co}} \mathbf{step}\,\tau : \tau \sim \tau'}\ \textsc{DCo\_Step}$$

$\boxed{\Sigma;\Gamma \Vdash_{\mathsf{prop}} \phi\ \mathsf{ok}}$ Proposition formation

$$\frac{\begin{array}{l}\Sigma;\Gamma \Vdash_{\mathsf{ty}} \tau_1 : \kappa_1 \\ \Sigma;\Gamma \Vdash_{\mathsf{ty}} \tau_2 : \kappa_2\end{array}}{\Sigma;\Gamma \Vdash_{\mathsf{prop}} \tau_1 \ {}^{\kappa_1}\!\sim^{\kappa_2} \tau_2\ \mathsf{ok}}\ \textsc{DProp\_Equality}$$

$\boxed{\Sigma;\Gamma \Vdash_{\mathsf{vec}} \overline{\psi} : \Delta}$ Type vector formation

$$\frac{\Sigma \Vdash_{\mathsf{ctx}} \Gamma\ \mathsf{ok}}{\Sigma;\Gamma \Vdash_{\mathsf{vec}} \varnothing : \varnothing}\ \textsc{DVec\_Nil}$$

$$\frac{\Sigma;\Gamma \Vdash_{\mathsf{ty}} \tau : \kappa' \qquad \kappa \equiv \kappa' \qquad \Sigma;\Gamma \Vdash_{\mathsf{vec}} \overline{\psi} : \Delta[\tau/a]}{\Sigma;\Gamma \Vdash_{\mathsf{vec}} \tau, \overline{\psi} : a{:}_{\mathsf{Rel}}\kappa, \Delta}\ \textsc{DVec\_TyRel}$$

$$\frac{\Sigma;\mathsf{Rel}(\Gamma) \Vdash_{\mathsf{ty}} \tau : \kappa' \qquad \kappa \equiv \kappa' \qquad \Sigma;\Gamma \Vdash_{\mathsf{vec}} \overline{\psi} : \Delta[\tau/a]}{\Sigma;\Gamma \Vdash_{\mathsf{vec}} \{\tau\}, \overline{\psi} : a{:}_{\mathsf{Irrel}}\kappa, \Delta}\ \textsc{DVec\_TyIrrel}$$

$$\frac{\Sigma;\mathsf{Rel}(\Gamma) \Vdash_{\mathsf{co}} \gamma : \phi' \qquad \phi \equiv \phi' \qquad \Sigma;\Gamma \Vdash_{\mathsf{vec}} \overline{\psi} : \Delta[\gamma/c]}{\Sigma;\Gamma \Vdash_{\mathsf{vec}} \gamma, \overline{\psi} : c{:}\phi, \Delta}\ \textsc{DVec\_Co}$$

$\boxed{\Sigma \Vdash_{\mathsf{ctx}} \Gamma\ \mathsf{ok}}$ Context formation

$$\frac{\vdash_{\mathsf{sig}} \Sigma\ \mathsf{ok}}{\Sigma \Vdash_{\mathsf{ctx}} \varnothing\ \mathsf{ok}}\ \textsc{DCtx\_Nil}$$

$$\frac{\Sigma;\mathsf{Rel}(\Gamma) \Vdash_{\mathsf{ty}} \kappa : \tau \qquad \tau \equiv \mathbf{Type} \qquad a \# \Gamma \qquad \Sigma \Vdash_{\mathsf{ctx}} \Gamma\ \mathsf{ok}}{\Sigma \Vdash_{\mathsf{ctx}} \Gamma, a{:}_{\rho}\kappa\ \mathsf{ok}}\ \textsc{DCtx\_TyVar}$$

$$\frac{\Sigma;\mathsf{Rel}(\Gamma) \Vdash_{\mathsf{prop}} \phi\ \mathsf{ok} \qquad c \# \Gamma \qquad \Sigma \Vdash_{\mathsf{ctx}} \Gamma\ \mathsf{ok}}{\Sigma \Vdash_{\mathsf{ctx}} \Gamma, c{:}\phi\ \mathsf{ok}}\ \textsc{DCtx\_CoVar}$$

$\boxed{\Sigma;\Gamma \Vdash_{\mathsf{s}} \sigma \longrightarrow \sigma'}$ Small-step operational semantics

$$\frac{}{\Sigma;\Gamma \Vdash_{\mathsf{s}} (\lambda a{:}_{\mathsf{Rel}}\kappa.\, \sigma_1)\, \sigma_2 \longrightarrow \sigma_1[\sigma_2/a]}\ \textsc{DS\_BetaRel}$$

$$\frac{}{\Sigma;\Gamma \Vdash_{\mathsf{s}} (\lambda a{:}_{\mathsf{Irrel}}\kappa.\, v_1)\, \{\sigma_2\} \longrightarrow v_1[\sigma_2/a]}\ \textsc{DS\_BetaIrrel}$$

$$\frac{}{\Sigma;\Gamma \Vdash_{\mathsf{s}} (\lambda c{:}\phi.\, \sigma)\, \gamma \longrightarrow \sigma[\gamma/c]}\ \textsc{DS\_CBeta}$$

$$\frac{alt_i \;=\; H \to \tau_0}{\Sigma;\Gamma \Vdash_{\mathsf{s}} \mathbf{case}_{\kappa}\, H_{\{\overline{\tau}\}}\, \overline{\psi}\, \mathbf{of}\, \overline{alt} \longrightarrow \tau_0\, \overline{\psi}\, \langle H_{\{\overline{\tau}\}}\, \overline{\psi}\rangle}\ \textsc{DS\_Match}$$

$$\frac{alt_i \;=\; \_ \to \sigma \qquad \text{no alternative in } \overline{alt} \text{ matches } H}{\Sigma;\Gamma \Vdash_{\mathsf{s}} \mathbf{case}_{\kappa}\, H_{\{\overline{\tau}\}}\, \overline{\psi}\, \mathbf{of}\, \overline{alt} \longrightarrow \sigma}\ \textsc{DS\_Default}$$

$$\frac{alt_i \;=\; \_ \to \sigma \qquad \text{no alternative in } \overline{alt} \text{ matches } H}{\Sigma; \Gamma \Vdash_{\mathsf{s}} \mathbf{case}_{\kappa}\, H_{\{\overline{\tau}\}}\, \overline{\psi} \rhd \gamma\, \mathbf{of}\, \overline{alt} \longrightarrow \sigma} \quad \text{DS\_DefaultCo}$$

$$\frac{\tau \;=\; \lambda a{:}_{\mathsf{Rel}}\kappa.\, \sigma}{\Sigma; \Gamma \Vdash_{\mathsf{s}} \mathbf{fix}\, \tau \longrightarrow \sigma[\mathbf{fix}\, \tau / a]} \quad \text{DS\_Unroll}$$

$$\frac{}{\Sigma; \Gamma \Vdash_{\mathsf{s}} (v \rhd \gamma_1) \rhd \gamma_2 \longrightarrow v \rhd (\gamma_1 \fatsemi \gamma_2)} \quad \text{DS\_Trans}$$

$$\frac{\Sigma; \Gamma, a{:}_{\mathsf{Irrel}}\kappa \Vdash_{\mathsf{s}} \sigma \longrightarrow \sigma'}{\Sigma; \Gamma \Vdash_{\mathsf{s}} \lambda a{:}_{\mathsf{Irrel}}\kappa.\, \sigma \longrightarrow \lambda a{:}_{\mathsf{Irrel}}\kappa.\, \sigma'} \quad \text{DS\_IrrelAbs\_Cong}$$

$$\frac{\Sigma; \Gamma \Vdash_{\mathsf{s}} \sigma \longrightarrow \sigma'}{\Sigma; \Gamma \Vdash_{\mathsf{s}} \sigma\, \psi \longrightarrow \sigma'\, \psi} \quad \text{DS\_App\_Cong}$$

$$\frac{\Sigma; \Gamma \Vdash_{\mathsf{s}} \sigma \longrightarrow \sigma'}{\Sigma; \Gamma \Vdash_{\mathsf{s}} \sigma \rhd \gamma \longrightarrow \sigma' \rhd \gamma} \quad \text{DS\_Cast\_Cong}$$

$$\frac{\Sigma; \Gamma \Vdash_{\mathsf{s}} \sigma \longrightarrow \sigma'}{\Sigma; \Gamma \Vdash_{\mathsf{s}} \mathbf{case}_{\tau}\, \sigma\, \mathbf{of}\, \overline{alt} \longrightarrow \mathbf{case}_{\tau}\, \sigma'\, \mathbf{of}\, \overline{alt}} \quad \text{DS\_Case\_Cong}$$

$$\frac{\Sigma; \Gamma \Vdash_{\mathsf{s}} \tau \longrightarrow \tau'}{\Sigma; \Gamma \Vdash_{\mathsf{s}} \mathbf{fix}\, \tau \longrightarrow \mathbf{fix}\, \tau'} \quad \text{DS\_Fix\_Cong}$$

$$\frac{\begin{array}{l}\Sigma; \mathsf{Rel}(\Gamma) \Vdash_{\mathsf{co}} \gamma_0 : \phi \\ \phi \overset{\rightarrow}{\equiv} \Pi a{:}_{\mathsf{Rel}}\kappa.\, \sigma \sim \Pi a{:}_{\mathsf{Rel}}\kappa'.\, \sigma' \\ \gamma_1 \;=\; \mathbf{sym}\,(\mathbf{argk}\, \gamma_0) \qquad \gamma_2 \;=\; \gamma_0 @ (\tau \rhd \gamma_1 \approx_{\mathbf{sym}\, \gamma_1} \tau)\end{array}}{\Sigma; \Gamma \Vdash_{\mathsf{s}} (v \rhd \gamma_0)\, \tau \longrightarrow v\, (\tau \rhd \gamma_1) \rhd \gamma_2} \quad \text{DS\_PushRel}$$

$$\frac{\begin{array}{l}\Sigma; \mathsf{Rel}(\Gamma) \Vdash_{\mathsf{co}} \gamma_0 : \phi \\ \phi \overset{\rightarrow}{\equiv} \Pi a{:}_{\mathsf{Irrel}}\kappa.\, \sigma \sim \Pi a{:}_{\mathsf{Irrel}}\kappa'.\, \sigma' \\ \gamma_1 \;=\; \mathbf{sym}\,(\mathbf{argk}\, \gamma_0) \qquad \gamma_2 \;=\; \gamma_0 @ (\tau \rhd \gamma_1 \approx_{\mathbf{sym}\, \gamma_1} \tau)\end{array}}{\Sigma; \Gamma \Vdash_{\mathsf{s}} (v \rhd \gamma_0)\, \{\tau\} \longrightarrow v\, \{\tau \rhd \gamma_1\} \rhd \gamma_2} \quad \text{DS\_PushIrrel}$$

$$\frac{\begin{array}{l}\Sigma; \mathsf{Rel}(\Gamma) \Vdash_{\mathsf{co}} \gamma_0 : \phi_0 \\ \phi_0 \overset{\rightarrow}{\equiv} \Pi c{:}\phi.\, \sigma \sim \Pi c{:}\phi'.\, \sigma' \\ \gamma_1 \;=\; \mathbf{argk}_1\, \gamma_0 \qquad \gamma_2 \;=\; \mathbf{argk}_2\, \gamma_0 \\ \eta' \;=\; \gamma_1 \fatsemi \eta \fatsemi \mathbf{sym}\, \gamma_2 \qquad \gamma_3 \;=\; \gamma_0 @ (\eta', \eta)\end{array}}{\Sigma; \Gamma \Vdash_{\mathsf{s}} (v \rhd \gamma_0)\, \eta \longrightarrow v\, \eta' \rhd \gamma_3} \quad \text{DS\_CPush}$$

$$\frac{\begin{array}{l}\gamma_1 \;=\; \tilde{\Pi} a{:}_{\mathsf{Irrel}} \langle \kappa \rangle.\, \gamma \qquad \gamma_2 \;=\; \tau_1 \approx_{\langle \mathbf{Type} \rangle} \tau_2 \\ \tau_1 \;=\; \tilde{\Pi} a{:}_{\mathsf{Irrel}} \kappa.\, (\kappa_1 [a \rhd \mathbf{sym}\, \langle \kappa \rangle / a]) \qquad \tau_2 \;=\; \tilde{\Pi} a{:}_{\mathsf{Irrel}} \kappa.\, \kappa_1\end{array}}{\Sigma; \Gamma \Vdash_{\mathsf{s}} \lambda a{:}_{\mathsf{Irrel}} \kappa.\, (v \rhd \gamma) \longrightarrow (\lambda a{:}_{\mathsf{Irrel}} \kappa.\, v) \rhd (\gamma_1 \fatsemi \gamma_2)} \quad \text{DS\_APush}$$

$$\frac{\begin{array}{l}\gamma_1 \;=\; \gamma_0 @ (a \approx_{\gamma_2} a \rhd \gamma_2) \fatsemi \mathbf{sym}\, \gamma_2 \\ \gamma_2 \;=\; \mathbf{argk}\, \gamma_0\end{array}}{\Sigma; \Gamma \Vdash_{\mathsf{s}} \mathbf{fix}\, ((\lambda a{:}_{\mathsf{Rel}} \kappa.\, \sigma) \rhd \gamma_0) \longrightarrow (\mathbf{fix}\, (\lambda a{:}_{\mathsf{Rel}} \kappa.\, (\sigma \rhd \gamma_1))) \rhd \gamma_2} \quad \text{DS\_FPush}$$

$$\frac{\begin{array}{l}\Sigma \vdash_{\mathsf{tc}} H : \overline{a}{:}_{\mathsf{Irrel}} \overline{\kappa}; \Delta; H' \qquad \Delta \;=\; \Delta_1, \Delta_2 \qquad n \;=\; |\Delta_2| \\ \kappa \;=\; \text{'}\Pi \overline{a}{:}_{\mathsf{Irrel}} \overline{\kappa}, \Delta.\, H'\, \overline{a} \\ \sigma \;=\; \text{'}\Pi (\Delta_2 [\overline{\tau}/\overline{a}][\overline{\psi}/\mathsf{dom}(\Delta_1)]).\, H'\, \overline{\tau} \\ \sigma' \;=\; \text{'}\Pi (\Delta_2 [\overline{\tau}'/\overline{a}][\overline{\psi}'/\mathsf{dom}(\Delta_1)]).\, H'\, \overline{\tau}' \\ \Sigma; \mathsf{Rel}(\Gamma) \Vdash_{\mathsf{co}} \eta : \phi \qquad \phi \equiv \sigma \sim \sigma' \\ \Sigma; \mathsf{Rel}(\Gamma) \Vdash_{\mathsf{vec}} \overline{\tau}' : \overline{a}{:}_{\mathsf{Rel}} \overline{\kappa} \\ \forall i,\; \gamma_i \;=\; \mathsf{build\_kpush\_co}(\langle \kappa \rangle @ (\mathbf{nths}\, (\mathbf{res}^n\, \eta)); \overline{\psi}_{1 \ldots i-1}) \\ \forall i,\; \psi_i' \;=\; \mathsf{cast\_kpush\_arg}(\psi_i; \gamma_i) \\ H \to \kappa' \in \overline{alt}\end{array}}{\Sigma; \Gamma \Vdash_{\mathsf{s}} \mathbf{case}_{\kappa_0}\, (H_{\{\overline{\tau}\}}\, \overline{\psi}) \rhd \eta\, \mathbf{of}\, \overline{alt} \longrightarrow \mathbf{case}_{\kappa_0}\, H_{\{\overline{\tau}'\}}\, \overline{\psi}'\, \mathbf{of}\, \overline{alt}} \quad \text{DS\_KPush}$$

F.2 Properties of ≡

Section 7.2 stated some properties of ≡ somewhat informally. Here are the more formal descriptions:

Property F.1 (Formal statement of Property 7.3)**.** *If* $\Sigma; \Gamma \vdash_{\mathsf{ty}} \tau : \kappa$, $\Sigma; \Gamma \vdash_{\mathsf{ty}} \tau' : \kappa$, *and* $\tau \equiv \tau'$, *then there exists* γ *such that* $\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{co}} \gamma : \tau \sim \tau'$.

Property F.2 (Formal statement of Property 7.4)**.** *If* $\overline{\psi} \equiv \overline{\psi}'$, *then* $\tau[\overline{\psi}/\overline{z}] \equiv \tau[\overline{\psi}'/\overline{z}]$.

Lemma F.3 (Transporting coercions)**.** *If* $\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{co}} \gamma : \phi$, $\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{prop}} \phi'\ \mathsf{ok}$, *and* $\phi \equiv \phi'$, *then there exists* γ' *such that* $\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{co}} \gamma' : \phi'$.

Proof. Lemma C.44 tells us that $\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{prop}} \phi\ \mathsf{ok}$. Let $\phi = \tau_1\ {}^{\kappa_1}\!\sim^{\kappa_2}\ \tau_2$ and $\phi' = \tau_1'\ {}^{\kappa_1'}\!\sim^{\kappa_2'}\ \tau_2'$. We can conclude all of the following by inversion:

- $\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{ty}} \tau_1 : \kappa_1$
- $\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{ty}} \tau_2 : \kappa_2$
- $\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{ty}} \tau_1' : \kappa_1'$
- $\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{ty}} \tau_2' : \kappa_2'$
- $\tau_1 \equiv \tau_1'$
- $\tau_2 \equiv \tau_2'$

By Property F.1, we can get γ_1 and γ_2 such that $\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{co}} \gamma_1 : \tau_1 \sim \tau_1'$ and $\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{co}} \gamma_2 : \tau_2 \sim \tau_2'$. Thus, $\Sigma; \mathsf{Rel}(\Gamma) \vdash_{\mathsf{co}} \mathbf{sym}\, \gamma_1 \fatsemi \gamma \fatsemi \gamma_2 : \phi'$ as desired. □

We also regularly need to extract certain bits of a type or proposition, via an extraction operator $\overset{\rightarrow}{\equiv}$. Extraction has these properties:

Property F.4 (Extraction respects $\equiv$)**.**

1. *If* $\tau \overset{\rightarrow}{\equiv} \tau'$ *then* $\tau \equiv \tau'$.
2. *If* $\phi \overset{\rightarrow}{\equiv} \phi'$ *then* $\phi \equiv \phi'$.

Property F.5 (Extraction can be chained with $\equiv$)**.**

1. *If* $\tau \equiv \tau'$ *and* $\tau' \overset{\rightarrow}{\equiv} \tau''$, *then* $\tau \overset{\rightarrow}{\equiv} \tau''$.
2. *If* $\phi \equiv \phi'$ *and* $\phi' \overset{\rightarrow}{\equiv} \phi''$, *then* $\phi \overset{\rightarrow}{\equiv} \phi''$.

Property F.6 (Extraction is deterministic)**.**

1. *If* $\tau \overset{\rightarrow}{\equiv} \tau_1$ *and* $\tau \overset{\rightarrow}{\equiv} \tau_2$, *then* $\tau_1 = \tau_2$.
2. *If* $\phi \overset{\rightarrow}{\equiv} \phi_1$ *and* $\phi \overset{\rightarrow}{\equiv} \phi_2$, *then* $\phi_1 = \phi_2$.

Property F.7 (Extraction is well-typed)**.**

1. *If* $\Sigma; \Gamma \vdash_{\mathsf{ty}} \tau : \kappa$ *and* $\tau \overset{\rightarrow}{\equiv} \tau'$, *then* $\Sigma; \Gamma \vdash_{\mathsf{ty}} \tau' : \kappa'$
2. *If* $\Sigma; \Gamma \vdash_{\mathsf{prop}} \phi\ \mathsf{ok}$ *and* $\phi \overset{\rightarrow}{\equiv} \phi'$, *then* $\Sigma; \Gamma \vdash_{\mathsf{prop}} \phi'\ \mathsf{ok}$.

F.3 Lemmas adapted from Appendix C

Lemma F.8 (Scoping)**.** *(as Lemma C.12, but with reference to $\Vdash$ judgments)*

Proof. Similar to the proof for Lemma C.12. □

Lemma F.9 (Context regularity)**.** *If:*

1. $\Sigma; \Gamma \Vdash^{}_{\mathsf{ty}} \tau : \kappa$*, OR*
2. $\Sigma; \Gamma \Vdash^{}_{\mathsf{co}} \gamma : \phi$*, OR*
3. $\Sigma; \Gamma \Vdash^{}_{\mathsf{prop}} \phi\ \mathsf{ok}$*, OR*
4. $\Sigma; \Gamma; \sigma_0 \Vdash^{\tau_0}_{\mathsf{alt}} alt : \kappa$*, OR*
5. $\Sigma; \Gamma \Vdash^{}_{\mathsf{vec}} \overline{\psi} : \Delta$*, OR*
6. $\Sigma \Vdash^{}_{\mathsf{ctx}} \Gamma\ \mathsf{ok}$

Then $\Sigma \Vdash^{}_{\mathsf{ctx}} \mathsf{prefix}(\Gamma)\ \mathsf{ok}$ and $\vdash^{}_{\mathsf{sig}} \Sigma\ \mathsf{ok}$, where $\mathsf{prefix}(\Gamma)$ is an arbitrary prefix of Γ. Furthermore, both resulting derivations are no larger than the input derivations.

Proof. Straightforward mutual induction. □

F.4 Soundness of Pico$^{\equiv}$

The following lemma also defines the $\lceil \cdot \rceil$ operation. This deterministic operation is defined to be the existentially-quantified output of the clauses of the following lemma, as labeled. Note that making sense of $\lceil \cdot \rceil$ requires that the argument be well-formed (that is, the premises of the clause defining the operation must be satisfied). For example, $\lceil \tau \rceil$ is not just an operation over a type τ, but it also requires $\Sigma; \Gamma \Vdash^{}_{\mathsf{ty}} \tau : \kappa$ as an input.

Lemma F.10 (Pico$^{\equiv}$ is sound)**.** *The uses of $\lceil \Gamma \rceil$ below depend on Lemma F.9 above.*

1. *If $\Sigma \Vdash^{}_{\mathsf{ctx}} \Gamma\ \mathsf{ok}$, then $\Sigma \vdash^{}_{\mathsf{ctx}} \Gamma'\ \mathsf{ok}$ where $\Gamma' \equiv \Gamma$. Let $\lceil \Gamma \rceil \triangleq \Gamma'$. Furthermore, $\mathsf{Rel}(\lceil \Gamma \rceil) = \lceil \mathsf{Rel}(\Gamma) \rceil$ and if $\Gamma = \Gamma_0, \delta$, then $\lceil \Gamma \rceil = \lceil \Gamma_0 \rceil, \delta'$.*
2. *If $\Sigma; \Gamma \Vdash^{}_{\mathsf{ty}} \tau : \kappa$, then $\Sigma; \lceil \Gamma \rceil \vdash^{}_{\mathsf{ty}} \tau' : \kappa'$ where $\tau' \equiv \tau$ and $\kappa' \equiv \kappa$. Let $\lceil \tau \rceil \triangleq \tau'$.*
3. *If $\Sigma; \Gamma \Vdash^{}_{\mathsf{co}} \gamma : \phi$, then $\Sigma; \lceil \Gamma \rceil \vdash^{}_{\mathsf{co}} \gamma' : \phi'$ where $\phi \equiv \phi'$.*
4. *If $\Sigma; \Gamma; \sigma \Vdash^{\tau}_{\mathsf{alt}} alt : \kappa$ and we have τ' and κ' such that $\tau' \equiv \tau$ and $\kappa' \equiv \kappa$, then $\Sigma; \lceil \Gamma \rceil; \sigma \vdash^{\tau'}_{\mathsf{alt}} alt' : \kappa'$ where $alt' \equiv alt$. Let $\lceil alt \rceil \triangleq alt'$.*
5. *If $\Sigma; \Gamma \Vdash^{}_{\mathsf{prop}} \phi\ \mathsf{ok}$, then $\Sigma; \lceil \Gamma \rceil \vdash^{}_{\mathsf{prop}} \phi'\ \mathsf{ok}$ where $\phi' \equiv \phi$. Let $\lceil \phi \rceil \triangleq \phi'$.*
6. *If $\Sigma; \Gamma \Vdash^{}_{\mathsf{vec}} \overline{\psi} : \Delta$, then $\Sigma; \lceil \Gamma \rceil \vdash^{}_{\mathsf{vec}} \overline{\psi}' : \Delta$ where $\overline{\psi}' \equiv \overline{\psi}$. Let $\lceil \overline{\psi} \rceil \triangleq \overline{\psi}'$.*
7. *If $\Sigma; \Gamma \Vdash^{}_{\mathsf{s}} \sigma \longrightarrow \tau$, $\Sigma; \Gamma \Vdash^{}_{\mathsf{ty}} \sigma : \kappa$, and $\Sigma; \Gamma \Vdash^{}_{\mathsf{ty}} \tau : \kappa$, then $\Sigma; \lceil \Gamma \rceil \vdash^{}_{\mathsf{s}} \lceil \sigma \rceil \longrightarrow \lceil \tau \rceil$.*

Proof. By induction on the typing derivations.

Case DCtx_Nil: Immediate.

Case DCtx_TyVar:

$$\frac{\begin{array}{ll}\Sigma; \mathsf{Rel}(\Gamma) \Vdash_{\mathsf{ty}} \kappa : \tau & \tau \equiv \mathbf{Type} \\ a \# \Gamma & \Sigma \Vdash_{\mathsf{ctx}} \Gamma\ \mathsf{ok}\end{array}}{\Sigma \Vdash_{\mathsf{ctx}} \Gamma, a{:}_\rho \kappa\ \mathsf{ok}}\ \textsc{DCtx\_TyVar}$$

We must show $\Sigma \vdash_{\mathsf{ctx}} \lceil\Gamma\rceil, a{:}_\rho \kappa'\ \mathsf{ok}$ for some $\kappa' \equiv \kappa$. Note that $\lceil\Gamma\rceil$ is well-formed by the induction hypothesis. The induction hypothesis gives us $\lceil\kappa\rceil$ such that $\Sigma; \mathsf{Rel}(\lceil\Gamma\rceil) \vdash_{\mathsf{ty}} \lceil\kappa\rceil : \tau'$ such that $\tau' \equiv \tau$. By transitivity of $\equiv$ (Property 7.1), $\tau' \equiv \mathbf{Type}$. We have $\Sigma; \mathsf{Rel}(\lceil\Gamma\rceil) \vdash_{\mathsf{ty}} \tau' : \mathbf{Type}$ (by Lemma C.43) and $\Sigma; \mathsf{Rel}(\lceil\Gamma\rceil) \vdash_{\mathsf{ty}} \mathbf{Type} : \mathbf{Type}$ (by Lemma C.38 and Lemma C.10). We then use Property F.1 to get γ such that $\Sigma; \mathsf{Rel}(\lceil\Gamma\rceil) \vdash_{\mathsf{co}} \gamma : \tau' \sim \mathbf{Type}$. Choose $\kappa' = \lceil\kappa\rceil \rhd \gamma$. We see that $\Sigma; \mathsf{Rel}(\lceil\Gamma\rceil) \vdash_{\mathsf{ty}} \lceil\kappa\rceil \rhd \gamma : \mathbf{Type}$ as desired. Property 7.5 tells us that $\kappa \equiv \lceil\kappa\rceil \rhd \gamma$, and so we are done.

Case DCtx_CoVar: By induction.

Case DTy_Var: By induction.

Case DTy_Con: By induction. Note that relating the result type (well-typed in Pico) to the input type (well-typed in Pico$^{\equiv}$) by $\equiv$ requires congruence, Property F.2. Congruence is similarly used in many other cases.

Case DTy_AppRel:

$$\frac{\begin{array}{ll}\Sigma; \Gamma \Vdash_{\mathsf{ty}} \tau_1 : \kappa_0 & \kappa_0 \overset{\rightarrow}{\equiv} \Pi a{:}_{\mathsf{Rel}} \kappa_1.\, \kappa_2 \\ \Sigma; \Gamma \Vdash_{\mathsf{ty}} \tau_2 : \kappa_1' & \kappa_1 \equiv \kappa_1'\end{array}}{\Sigma; \Gamma \Vdash_{\mathsf{ty}} \tau_1\, \tau_2 : \kappa_2[\tau_2/a]}\ \textsc{DTy\_AppRel}$$

The induction hypothesis tells us $\Sigma; \lceil\Gamma\rceil \vdash_{\mathsf{ty}} \lceil\tau_1\rceil : \kappa_0'$ where $\kappa_0' \equiv \kappa_0$, and $\Sigma; \lceil\Gamma\rceil \vdash_{\mathsf{ty}} \lceil\tau_2\rceil : \kappa_1''$ where $\kappa_1'' \equiv \kappa_1'$. By Property F.5, we get $\kappa_0' \overset{\rightarrow}{\equiv} \Pi a{:}_{\mathsf{Rel}} \kappa_1.\, \kappa_2$. By Lemma C.43, we have $\Sigma; \mathsf{Rel}(\lceil\Gamma\rceil) \vdash_{\mathsf{ty}} \kappa_0' : \mathbf{Type}$. Thus by Property F.7, we get $\Sigma; \mathsf{Rel}(\lceil\Gamma\rceil) \vdash_{\mathsf{ty}} \Pi a{:}_{\mathsf{Rel}} \kappa_1.\, \kappa_2 : \sigma$. Inversion tells us that $\sigma = \mathbf{Type}$. We can thus use Property 7.1 and Property F.1 to get γ_1 such that $\Sigma; \mathsf{Rel}(\lceil\Gamma\rceil) \vdash_{\mathsf{co}} \gamma_1 : \kappa_0' \sim \Pi a{:}_{\mathsf{Rel}} \kappa_1.\, \kappa_2$.

Now, inversion and Lemma C.7 tells us $\Sigma; \mathsf{Rel}(\lceil\Gamma\rceil) \vdash_{\mathsf{ty}} \kappa_1 : \mathbf{Type}$ and Lemma C.43 tells us $\Sigma; \mathsf{Rel}(\lceil\Gamma\rceil) \vdash_{\mathsf{ty}} \kappa_1'' : \mathbf{Type}$. Thus Property 7.1 and Property F.1 give us γ_2 such that $\Sigma; \mathsf{Rel}(\lceil\Gamma\rceil) \vdash_{\mathsf{co}} \gamma_2 : \kappa_1'' \sim \kappa_1$. We now choose $\lceil\tau_1\, \tau_2\rceil \triangleq (\lceil\tau_1\rceil \rhd \gamma_1)\, (\lceil\tau_2\rceil \rhd \gamma_2)$ and we can see that $\Sigma; \lceil\Gamma\rceil \vdash_{\mathsf{ty}} \lceil\tau_1\, \tau_2\rceil : \kappa_2[\lceil\tau_2\rceil \rhd \gamma_2/a]$ as desired. Relating this output kind to the input kind is achieved by Property F.2 and Property 7.5.

Case DTy_AppIrrel: Similar to previous case.

Case DTy_CApp: Similar to previous cases, but appealing to Lemma F.3.

Case DTy_Pi:

$$\frac{\Sigma; \Gamma, \mathsf{Rel}(\delta) \Vdash_{\mathsf{ty}} \kappa : \tau \qquad \tau \equiv \mathbf{Type}}{\Sigma; \Gamma \Vdash_{\mathsf{ty}} \Pi\delta.\, \kappa : \mathbf{Type}} \quad \textsc{DTy\_Pi}$$

The induction hypothesis gives us $\Sigma; \lceil \Gamma, \mathsf{Rel}(\delta) \rceil \vdash_{\mathsf{ty}} \lceil \kappa \rceil : \tau'$ where $\tau' \equiv \tau$. Lemma C.43 tells us $\Sigma; \lceil \mathsf{Rel}(\Gamma, \delta) \rceil \vdash_{\mathsf{ty}} \tau' : \mathbf{Type}$. We know $\Sigma; \lceil \mathsf{Rel}(\Gamma, \delta) \rceil \vdash_{\mathsf{ty}} \mathbf{Type} : \mathbf{Type}$ by Lemma C.38 and Lemma C.10. We can thus use Property F.1 to get γ such that $\Sigma; \lceil \mathsf{Rel}(\Gamma, \delta) \rceil \vdash_{\mathsf{co}} \gamma : \tau' \sim \mathbf{Type}$. We can conclude $\Sigma; \lceil \Gamma, \mathsf{Rel}(\delta) \rceil \vdash_{\mathsf{ty}} \lceil \kappa \rceil \rhd \gamma : \mathbf{Type}$. By the extra condition in the induction hypothesis for contexts, we can rewrite this to $\Sigma; \lceil \Gamma \rceil, \mathsf{Rel}(\delta') \vdash_{\mathsf{ty}} \lceil \kappa \rceil \rhd \gamma : \mathbf{Type}$, where $\delta' \equiv \delta$. We can now use TY_PI to conclude $\Sigma; \lceil \Gamma \rceil \vdash_{\mathsf{ty}} \Pi\delta'.\, (\lceil \kappa \rceil \rhd \gamma) : \mathbf{Type}$ as desired. Here, $\lceil \Pi\delta.\, \kappa \rceil \triangleq \Pi\delta'.\, (\lceil \kappa \rceil \rhd \gamma)$.

Case DTy_Cast:

$$\frac{\begin{array}{l}\Sigma; \mathsf{Rel}(\Gamma) \Vdash_{\mathsf{co}} \gamma : \kappa_1 \sim \kappa_2 \\ \Sigma; \Gamma \Vdash_{\mathsf{ty}} \tau : \kappa_1' \qquad \Sigma; \mathsf{Rel}(\Gamma) \Vdash_{\mathsf{ty}} \kappa_2 : \sigma \\ \kappa_1 \equiv \kappa_1' \qquad \sigma \equiv \mathbf{Type}\end{array}}{\Sigma; \Gamma \Vdash_{\mathsf{ty}} \tau \rhd \gamma : \kappa_2} \quad \textsc{DTy\_Cast}$$

The induction hypothesis gives us:

- $\Sigma; \mathsf{Rel}(\lceil \Gamma \rceil) \vdash_{\mathsf{co}} \gamma' : \kappa_1'' \sim \kappa_2''$ with $\kappa_1'' \equiv \kappa_1$ and $\kappa_2'' \equiv \kappa_2$
- $\Sigma; \lceil \Gamma \rceil \vdash_{\mathsf{ty}} \lceil \tau \rceil : \kappa_1'''$ where $\kappa_1''' \equiv \kappa_1'$
- $\Sigma; \mathsf{Rel}(\lceil \Gamma \rceil) \vdash_{\mathsf{ty}} \lceil \kappa_2 \rceil : \sigma'$ where $\sigma' \equiv \sigma$

Lemma C.44, Lemma C.43, Property 7.1, and Property F.1 give us γ_0 such that $\Sigma; \mathsf{Rel}(\lceil \Gamma \rceil) \vdash_{\mathsf{co}} \gamma_0 : \kappa_1''' \sim \kappa_1''$. We can thus use TY_CAST to get $\Sigma; \lceil \Gamma \rceil \vdash_{\mathsf{ty}} \lceil \tau \rceil \rhd \gamma_0 : \kappa_2''$ as desired.

Case DTy_Case: Along the lines of similar cases. We need Lemma F.8 to establish that the result type of the scrutinee does not mention the bound telescope Δ.

Other cases: Proceed as above. At this point, we have seen a variety of constructs and how to handle them. The remaining cases are similar, using the properties of $\equiv$ and $\overset{\rightarrow}{\equiv}$ to get from a typing derivation in PICO$^{\equiv}$ to one in PICO.

□

Bibliography

[1] Thorsten Altenkirch, Conor McBride, and James McKinna. Why dependent types matter. 2005. URL `http://www.strictlypositive.org/ydtm.ps.gz`.

[2] Lennart Augustsson. *Compiling pattern matching*, pages 368–381. Springer Berlin Heidelberg, Berlin, Heidelberg, 1985. ISBN 978-3-540-39677-2. doi: 10.1007/3-540-15975-4_48. URL `http://dx.doi.org/10.1007/3-540-15975-4_48`.

[3] Lennart Augustsson. Cayenne—a language with dependent types. In *Proc. ACM SIGPLAN International Conference on Functional Programming*, ICFP '98, pages 239–250. ACM, 1998.

[4] Christiaan P. R. Baaij. *Digital Circuits in CλaSH: Functional Specification and Type-Directed Synthesis*. PhD thesis, University of Twente, 2015.

[5] Patrick Bahr. Composing and decomposing data types: A closed type families implementation of data types à la carte. In *Workshop on Generic Programming*. ACM, 2014.

[6] Henk Barendregt. Introduction to generalized type systems. *J. Funct. Program.*, 1(2):125–154, 1991.

[7] Bruno Barras and Bruno Bernardo. The implicit calculus of constructions as a programming language with dependent types. In Roberto Amadio, editor, *Foundations of Software Science and Computational Structures*, FOSSACS 2008, pages 365–379, Budapest, Hungary, 2008. Springer Berlin Heidelberg.

[8] Edwin Brady. Programming and reasoning with algebraic effects and dependent types. In *International Conference on Functional Programming*. ACM, 2013.

[9] Edwin Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *J. Funct. Prog.*, 23, 2013.

[10] Edwin Brady, Conor McBride, and James McKinna. Inductive families need not store their indices. In Stefano Berardi, Mario Coppo, and Ferruccio Damiani, editors, *Types for Proofs and Programs*, volume 3085 of *Lecture Notes in Computer Science*. 2004.

[11] Joachim Breitner, Richard A. Eisenberg, Simon Peyton Jones, and Stephanie Weirich. Safe zero-cost coercions for Haskell. *J. Funct. Program.*, 26:1–79, 2016.

[12] Luca Cardelli. A polymorphic λ-calculus with Type:Type. Technical report, DEC/Compaq Systems Research Center, 1986. Research report 10.

[13] Chris Casinghino, Vilhelm Sjöberg, and Stephanie Weirich. Step-indexed normalization for a language with general recursion. In Proc. 4th Workshop on *Mathematically Structured Functional Programming,* Tallinn, Estonia, pages 25–39, 2012.

[14] Chris Casinghino, Vilhelm Sjöberg, and Stephanie Weirich. Combining proofs and programs in a dependently typed language. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14. ACM, 2014.

[15] Manuel M. T. Chakravarty, Gabriele Keller, and Simon Peyon Jones. Associated type synonyms. In *International Conference on Functional Programming*, ICFP '05. ACM, 2005.

[16] Manuel M. T. Chakravarty, Gabriele Keller, Simon Peyton Jones, and Simon Marlow. Associated types with class. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2005.

[17] David Raymond Christiansen. A pretty printer that says what it means. Talk, Haskell Implementors Workshop, Vancouver, BC, Canada, 2015. URL `https://www.youtube.com/watch?v=m7BBCcIDXSg`.

[18] Dominique Clément, Thierry Despeyroux, Gilles Kahn, and Joëlle Despeyroux. A simple applicative language: Mini-ML. In *Conference on LISP and Functional Programming*, LFP '86. ACM, 1986.

[19] Thierry Coquand and Gérard Huet. The calculus of constructions. *Information and Computation*, 76(2):95 – 120, 1988. ISSN 0890-5401. doi: http://dx.doi.org/10.1016/0890-5401(88)90005-3. URL `http://www.sciencedirect.com/science/article/pii/0890540188900053`.

[20] Luis Damas. *Type Assignment in Programming Languages*. PhD thesis, University of Edinburgh, 1985.

[21] Nicolaas Govert de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *Indagationes Mathematicae (Proceedings)*, 75(5):381âĂŞ392, 1972. ISSN 1385-7258. doi: 10.1016/1385-7258(72)90034-0. URL `http://dx.doi.org/10.1016/1385-7258(72)90034-0`.

[22] Iavor S. Diatchki. Improving haskell types with SMT. In *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell*, Haskell '15. ACM, 2015.

[23] Larry Diehl and Tim Sheard. Generic lookup and update for infinitary inductive-recursive types. In *Proceedings of the 1st International Workshop on Type-Driven Development*, TyDe 2016. ACM, 2016.

[24] Joshua Dunfield and Neelakantan R. Krishnaswami. Complete and easy bidirectional typechecking for higher-rank polymorphism. In *International Conference on Functional Programming*, ICFP '13. ACM, 2013.

[25] Richard A. Eisenberg. Dependent types in haskell. Presentation to New York City Haskell Users' Group, Oct 2014. URL `https://github.com/goldfirere/nyc-hug-oct2014`.

[26] Richard A. Eisenberg. System FC, as implemented in GHC. Technical Report MS-CIS-15-09, University of Pennsylvania, 2015. URL `https://github.com/ghc/ghc/blob/master/docs/core-spec/core-spec.pdf`.

[27] Richard A. Eisenberg. An overabundance of equality: Implementing kind equalities into haskell. Technical Report MS-CIS-15-10, University of Pennsylvania, 2015. URL `http://www.cis.upenn.edu/~eir/papers/2015/equalities/equalities.pdf`.

[28] Richard A. Eisenberg and Simon Peyton Jones. Levity polymorphism. Draft, 2016. URL `http://cs.brynmawr.edu/~rae/papers/2017/levity/levity.pdf`.

[29] Richard A. Eisenberg and Jan Stolarek. Promoting functions to type families in Haskell. In *ACM SIGPLAN Haskell Symposium*, 2014.

[30] Richard A. Eisenberg and Stephanie Weirich. Dependently typed programming with singletons. In *ACM SIGPLAN Haskell Symposium*, 2012.

[31] Richard A. Eisenberg, Dimitrios Vytiniotis, Simon Peyton Jones, and Stephanie Weirich. Closed type families with overlapping equations (extended version). Technical Report MS-CIS-13-10, University of Pennsylvania, 2013.

[32] Richard A. Eisenberg, Dimitrios Vytiniotis, Simon Peyton Jones, and Stephanie Weirich. Closed type families with overlapping equations. In *Principles of Programming Languages*, POPL '14. ACM, 2014.

[33] Richard A. Eisenberg, Stephanie Weirich, and Hamidhasan Ahmed. Visible type application. In *European Symposium on Programming (ESOP)*, LNCS. Springer-Verlag, 2016.

[34] Richard A. Eisenberg, Stephanie Weirich, and Hamidhasan Ahmed. Visible type application (extended version). 2016. URL `http://www.cis.upenn.edu/~eir/papers/2016/type-app/visible-type-app-extended.pdf`.

[35] Jeff Epstein, Andrew P. Black, and Simon Peyton Jones. Towards haskell in the cloud. In *Haskell Symposium*. ACM, 2011.

[36] Jean-Yves Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris 7, 1972.

[37] Adam Gundry. *Type Inference, Haskell and Dependent Types*. PhD thesis, University of Strathclyde, 2013.

[38] Adam Gundry. A typechecker plugin for units of measure: Domain-specific constraint solving in GHC Haskell. In *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell*, Haskell '15. ACM, 2015.

[39] Cordelia V. Hall, Kevin Hammond, Simon L. Peyton Jones, and Philip L. Wadler. Type classes in haskell. *ACM Trans. Program. Lang. Syst.*, 18(2), March 1996.

[40] Bastiaan Heeren, Daan Leijen, and Arjan van IJzendoorn. Helium, for learning haskell. In *Workshop on Haskell*. ACM, 2003.

[41] Fritz Henglein. Type inference with polymorphic recursion. *ACM Trans. Program. Lang. Syst.*, 15(2):253–289, April 1993. ISSN 0164-0925. doi: 10.1145/169701.169692. URL `http://doi.acm.org/10.1145/169701.169692`.

[42] Jason J. Hickey. Formal objects in type theory using very dependent types. In *Foundations of Object Oriented Languages (FOOL '96)*, 1996. URL `http://www.cis.upenn.edu/~bcpierce/FOOL/FOOL3.html`.

[43] J. Roger Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146, 1969.

[44] Martin Hofmann and Thomas Streicher. The groupoid interpretation of type theory. In Giovanni Sambin and Jan M. Smith, editors, *Twenty-Five Years of Constructive Type Theory*, Oxford Logic Guides, Venice, Italy, 1995. Oxford University Press.

[45] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A history of haskell: Being lazy with class. In *Conference on History of Programming Languages*, 2007.

[46] John Hughes. Generalising monads to arrows. *Sci. Comput. Program.*, 37(1-3), May 2000.

[47] Chung-Kil Hur. Agda with excluded middle is inconsistent. Email to Agda mailing list, January 2010. URL `https://lists.chalmers.se/pipermail/agda/2010/001526.html`.

[48] Antonius J. C. Hurkens. *A simplification of Girard's paradox*, pages 266–278. Springer Berlin Heidelberg, Berlin, Heidelberg, 1995. ISBN 978-3-540-49178-1. doi: 10.1007/BFb0014058. URL `http://dx.doi.org/10.1007/BFb0014058`.

[49] Mark P. Jones. Type classes with functional dependencies. In *European Symposium on Programming*, 2000.

[50] Stefan Kahrs and Connor Smith. Non-Omega-Overlapping TRSs are UN. In Delia Kesner and Brigitte Pientka, editors, *1st International Conference on Formal Structures for Computation and Deduction (FSCD 2016)*, volume 52 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 22:1–22:17, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-95977-010-1. doi: http://dx.doi.org/10.4230/LIPIcs.FSCD.2016.22. URL `http://drops.dagstuhl.de/opus/volltexte/2016/5996`.

[51] Georgios Karachalias, Tom Schrijvers, Dimitrios Vytiniotis, and Simon Peyton Jones. GADTs meet their match. In *International Conference on Functional Programming*, ICFP '15. ACM, 2015.

[52] Oleg Kiselyov, Ralf Lämmel, and Keean Schupke. Strongly typed heterogeneous collections. In *Proc. 2004 ACM SIGPLAN Workshop on Haskell*, Haskell '04, pages 96–107. ACM, 2004.

[53] Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: A practical design pattern for generic programming. In *Workshop on Types in Languages Design and Implementation*. ACM, 2003.

[54] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, LPAR'16, pages 348–370. Springer Berlin Heidelberg, 2010.

[55] Sheng Liang, Paul Hudak, and Mark Jones. Monad transformers and modular interpreters. In *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95. ACM, 1995.

[56] Sam Lindley and Conor McBride. Hasochism: the pleasure and pain of dependently typed Haskell programming. In *ACM SIGPLAN Haskell Symposium*, 2013.

[57] Zhaohui Luo. *An Extended Calculus of Constructions*. PhD thesis, University of Edinburgh, 1990.

[58] Conor McBride. *Dependently Typed Functional Programs and their Proofs*. PhD thesis, University of Edinburgh, 1999. URL `http://strictlypositive.org/thesis.pdf`.

[59] Conor McBride. Faking it: Simulating dependent types in Haskell. *J. Funct. Program.*, 12(5):375–392, July 2002.

[60] Conor McBride. The Strathclyde Haskell Enhancement. `https://personal.cis.strath.ac.uk/conor.mcbride/pub/she/`, 2011.

[61] Conor Thomas McBride. How to keep your neighbours in order. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14. ACM, 2014.

[62] Lambert Meertens. Incremental polymorphic type checking in B. In *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '83. ACM, 1983.

[63] Robin Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17, 1978.

[64] Alexandre Miquel. The implicit calculus of constructions: Extending pure type systems with an intersection type binder and subtyping. In Samson Abramsky, editor, *Typed Lambda Calculi and Applications*, volume 2044 of *Lecture Notes in Computer Science*, pages 344–359. Springer Berlin Heidelberg, 2001.

[65] Nathan Mishra-Linger and Tim Sheard. Erasure and polymorphism in pure type systems. In *Foundations of Software Science and Computational Structures (FoSSaCS)*. Springer, 2008.

[66] J. Garrett Morris and Mark P. Jones. Instance chains: type class programming without overlapping instances. In *ACM SIGPLAN International Conference on Functional Programming*, 2010.

[67] Alan Mycroft. *Polymorphic type schemes and recursive definitions*. Springer, Berlin, Heidelberg, 1984.

[68] Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

[69] Nicolas Oury and Wouter Swierstra. The power of Pi. In *Proc. 13th ACM SIGPLAN international conference on Functional programming*, ICFP '08, pages 39–50. ACM, 2008.

[70] Conrad Parker. Type-level instant insanity. *The Monad.Reader*, (8), 2007.

[71] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.

[72] Simon Peyton Jones and Mark Shields. Lexically-scoped type variables. Draft, 2004. URL `http://research.microsoft.com/en-us/um/people/simonpj/papers/scoped-tyvars/`.

[73] Simon Peyton Jones, Andrew Tolmach, and Tony Hoare. Playing by the rules: rewriting as a practical optimisation technique in GHC. In *Haskell Workshop*, pages 203–233, 2001.

[74] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Mark Shields. Practical type inference for arbitrary-rank types. *Journal of Functional Programming*, 17(1), January 2007.

[75] Simon Peyton Jones, Stephanie Weirich, Richard A. Eisenberg, and Dimitrios Vytiniotis. A reflection on types. In *A list of successes that can change the world*, LNCS. Springer, 2016. A festschrift in honor of Phil Wadler.

[76] Matthew Pickering, Gergő Érdi, Simon Peyton Jones, and Richard A. Eisenberg. Pattern synonyms. In *ACM SIGPLAN Haskell Symposium*. ACM, 2016.

[77] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, MA, 2002.

[78] Benjamin C. Pierce and David N. Turner. Local type inference. *ACM Trans. Program. Lang. Syst.*, 22(1), January 2000.

[79] François Pottier and Didier Rémy. *Advanced Topics in Types and Programming Languages*, chapter The Essence of ML Type Inference, pages 387–489. The MIT Press, 2005.

[80] Bertrand Russell. Mathematical llogic as based on a theory of types. *Amer. J. Math.*, 30:222–262, 1908.

[81] Alejandro Serrano Mena. *Beginning Haskell: A Project-Based Approach*. Apress, 2013.

[82] Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strniša. Ott: Effective tool support for the working semanticist. *Journal of Functional Programming*, 20(1), January 2010.

[83] Tim Sheard and Simon Peyton Jones. Template meta-programming for Haskell. In *Proc. 2002 ACM SIGPLAN workshop on Haskell*, Haskell '02, pages 1–16. ACM, 2002.

[84] Vincent Simonet and François Pottier. A constraint-based approach to guarded algebraic data types. *ACM Trans. Program. Lang. Syst.*, 29(1), January 2007.

[85] Vilhelm Sjöberg and Stephanie Weirich. Programming up to congruence. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15. ACM, 2015.

[86] Jan Stolarek, Simon Peyon Jones, and Richard A. Eisenberg. Injective type families for Haskell. In *Haskell Symposium*, Haskell '15. ACM, 2015.

[87] Martin Sulzmann, Manuel M. T. Chakravarty, Simon Peyton Jones, and Kevin Donnelly. System F with type equality coercions. In *Types in languages design and implementation*, TLDI '07. ACM, 2007.

[88] Martin Sulzmann, Gregory J. Duck, Simon Peyton-Jones, and Peter J. Stuckey. Understanding functional dependencies via constraint handling rules. *J. Funct. Program.*, 17(1), January 2007.

[89] Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoue, and Santiago Zanella-Béguelin. Dependent types and multi-monadic effects in F*. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16. ACM, 2016.

[90] Matúš Tejiščák and Edwin Brady. Practical erasure in dependently typed languages. Draft, 2015. URL `http://eb.host.cs.st-andrews.ac.uk/drafts/dtp-erasure-draft.pdf`.

[91] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. `https://homotopytypetheory.org/book`, Institute for Advanced Study, 2013.

[92] Floris van Doorn, Herman Geuvers, and Freek Wiedijk. Explicit convertibility proofs in pure type systems. In *Proceedings of the Eighth ACM SIGPLAN International Workshop on Logical Frameworks and Meta-languages: Theory and Practice*, LFMTP '13, pages 25–36, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2382-6. doi: 10.1145/2503887.2503890. URL `http://doi.acm.org/10.1145/2503887.2503890`.

[93] Niki Vazou, Eric L. Seidel, and Ranjit Jhala. LiquidHaskell: Experience with refinement types in the real world. In *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell*, Haskell '14. ACM, 2014.

[94] Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton Jones. Refinement types for Haskell. In *International Conference on Functional Programming*, ICFP '14. ACM, 2014.

[95] Niki Vazou, Alexander Bakst, and Ranjit Jhala. Bounded refinement types. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015. ACM, 2015.

[96] Dimitrios Vytiniotis and Simon Peyton Jones. Evidence Normalization in System FC. In Femke van Raamsdonk, editor, *24th International Conference on Rewriting Techniques and Applications (RTA 2013)*, volume 21 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 20–38, Dagstuhl, Germany, 2013. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-939897-53-8. doi: http://dx.doi.org/10.4230/LIPIcs.RTA.2013.20. URL `http://drops.dagstuhl.de/opus/volltexte/2013/4050`.

[97] Dimitrios Vytiniotis, Stephanie Weirich, and Simon Peyton Jones. Boxy types: Inference for higher-rank types and impredicativity. In *International Conference on Functional Programming*, ICFP '06. ACM, 2006.

[98] Dimitrios Vytiniotis, Simon Peyton Jones, and Tom Schrijvers. Let should not be generalized. In *Types in Language Design and Implementation*, TLDI '10. ACM, 2010.

[99] Dimitrios Vytiniotis, Simon Peyton Jones, Tom Schrijvers, and Martin Sulzmann. OutsideIn(X): Modular type inference with local assumptions. *Journal of Functional Programming*, 21(4-5), September 2011.

[100] Dimitrios Vytiniotis, Simon Peyton Jones, and José Pedro Magalhães. Equality proofs and deferred type errors: A compiler pearl. In *International Conference on Functional Programming*, ICFP '12. ACM, 2012.

[101] Philip Wadler. Compiling pattern matching. In Simon Peyton Jones, editor, *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987.

[102] Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad-hoc. In *POPL*, pages 60–76. ACM, 1989.

[103] Geoffrey Washburn and Stephanie Weirich. Boxes Go Bananas: Encoding higher-order abstract syntax with parametric polymorphism. In *International Conference on Functional Programming*. ACM, 2003.

[104] Stephanie Weirich. Paradoxical typecase. Presentation to WG2.8., November 2012. URL `http://www.cis.upenn.edu/~sweirich/talks/wg28-paradoxes.pdf`.

[105] Stephanie Weirich, Justin Hsu, and Richard A. Eisenberg. System FC with explicit kind equality. In *International Conference on Functional Programming*, ICFP '13. ACM, 2013.

[106] Stephanie Weirich, Justin Hsu, and Richard A. Eisenberg. System FC with explicit kind equality (extended version). Technical report, University of Pennsylvania, 2013. URL `http://www.cis.upenn.edu/~sweirich/nokinds-extended.pdf`.

[107] Brent A. Yorgey, Stephanie Weirich, Julien Cretin, Simon Peyton Jones, Dimitrios Vytiniotis, and José Pedro Magalhães. Giving Haskell a promotion. In *Types in Language Design and Implementation*, TLDI '12. ACM, 2012.