2014

Promoting Functions to Type Families in Haskell (extended version)

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

Jan Stolarek

Citation

R. A. Eisenberg, J. Stolarek. Promoting Functions to Type Families in Haskell (extended version). University of Pennsylvania Technical Report MS-CIS-14-09, 2014.

University of Pennsylvania Technical Report MS-CIS-14-09

Promoting Functions to Type Families in Haskell (extended version)[1]

Richard A. Eisenberg
University of Pennsylvania
eir@cis.upenn.edu

Jan Stolarek
Politechnika Łódzka
jan.stolarek@p.lodz.pl

Abstract

Haskell, as implemented in the Glasgow Haskell Compiler (GHC), is enriched with many extensions that support type-level programming, such as promoted datatypes, kind polymorphism, and type families. Yet, the expressiveness of the type-level language remains limited. It is missing many features present at the term level, including **case** expressions, anonymous functions, partially-applied functions, and **let** expressions. In this paper, we present an algorithm – with a proof of correctness – to encode these term-level constructs at the type level. Our approach is automated and capable of promoting a wide array of functions to type families. We also highlight and discuss those term-level features that are not promotable. In so doing, we offer a critique on GHC's existing type system, showing what it is already capable of and where it may want improvement. We believe that delineating the mismatch between GHC's term level and its type level is a key step toward supporting dependently typed programming.

We have implemented our approach as part of the singletons package, available online.

Categories and Subject Descriptors F.3.3 [*Logics And Meanings Of Programs*]: Studies of Program Constructs – Type structure; D.3.1 [*Programming Languages*]: Formal Definitions and Theory – Semantics; D.3.2 [*Programming Languages*]: Language Classifications – Haskell

Keywords Haskell; type-level programming; defunctionalization

1. Introduction

Haskell, especially as implemented in the Glasgow Haskell Compiler (GHC), is endowed with a plethora of facilities for type-level programming. Haskell 98 on its own has type classes (Wadler and Blott 1989), parametric polymorphism, and inferred higher-order kinds. Over the past 15 years or so, more and more features have been added, such as functional dependencies (Jones 2000), first-class polymorphism (Peyton Jones et al. 2007), generalized algebraic datatypes (GADTs) (Cheney and Hinze 2003; Peyton Jones et al. 2006), type families (Chakravarty et al. 2005a,b; Eisenberg et al. 2014), and datatype promotion with kind polymorphism (Yorgey et al. 2012).

Now, we might ask: Are we there yet?

In other words, is type-level programming expressive enough? To begin to answer this question, we must define "enough." In this paper, we choose to interpret "enough" as meaning that type-level programming is at least as expressive as term-level programming. We wish to be able to take any pure term-level program and write an equivalent type-level one.

Our answer to this question: "Almost." As we describe in more detail in Section 4, Haskell's type system as it appears in GHC 7.8 is capable of expressing almost all term-level constructs, including anonymous functions, partially applied functions, **case** and **let** expressions, and even type classes. However, a few key pieces are missing. As described by Yorgey et al. (2012) and expanded on by Weirich et al. (2013), GADTs cannot be promoted. Haskell also lacks higher-order sorts, which would classify the promotion of higher-kinded type variables, including the *m* in *Monad m*. There are other limitations, as well; see Section 5.

Despite these limitations, we have found that a wide array of programs are indeed promotable, using a mechanical translation implemented in Template Haskell (Sheard and Peyton Jones 2002). Our implementation is based on work started by Eisenberg and Weirich (2012) and is part of the singletons package.[2]

Why might we want to promote all these term-level constructs? As Haskell inches ever closer to being dependently typed (Weirich et al. 2013; Gundry 2013; Lindley and McBride 2013), it will become important to identify precisely which term-level constructs are available to be used in dependent contexts – that is, which terms really can be used in types? The present work defines this subset concretely and helps to set the stage for a dependently-typed version of Haskell.

We make the following contributions:

- We describe an enhancement to the singletons library, which promotes term-level definitions to the type level. We focus only on promoting expressions and declarations as defined in chapters 3 and 4 of the Haskell 2010 Language Report (Marlow 2010). Our implementation relies on many extensions of GHC 7.8 but without the need to add new features. (Section 4)
- We delimit exactly what features are *not* promotable under our implementation, and why these would be impossible to promote without further enhancements to Haskell. (Section 5)
- Section 6 describes a formalization of Haskell and presents a proof, given in full in Appendix J, that our promotion algorithm produces well-kinded types. We also show that, if we assume the correctness of our implementation of lambda-lifting, a promoted expression reduces in the same way as the original expression.
- We conclude in Sections 7 and 7.5 with reflections on GHC's current type system and some ideas for the future of Haskell in order to support type-level programming better.

A somewhat unexpected contribution of our work is discovery and posting of nearly 30 GHC bugs. Of these, 15 are related to Template Haskell and 9 to the type checker.

[1] This is a substantial revision to published work (Eisenberg and Stolarek 2014).

[2] `cabal install singletons`. You will need GHC 7.8.2 or higher.

It is our hope that through the use of the singletons library, users will be able to experiment with type-level programming with ease, encouraging the use of a strongly-typed programming style. We, and others, will also gain more experience with code that can operate on both the term and type levels, to better inform the design that may eventually be implemented as part of a dependently-typed version of Haskell.

2. Types and Kinds

Before presenting our main work, it may be helpful to the reader to have a brief review of how promotion currently works in Haskell. This section presents no new results and may be skipped by the expert reader.

2.1 Datakinds

Haskell has long had a notion of *kinds* separate from that of *types*. A term is classified by a type. Thus, *True* has the type *Bool* and $(\lambda x \to length\ x == 0)$ has the type $[a] \to Bool$. A type, in turn, is classified by a kind, where the special kind $\star$ classifies normal types that have values. Thus, *Bool* has kind $\star$, *Maybe* has kind $\star \to \star$, and the *StateT* monad transformer has kind $\star \to (\star \to \star) \to \star \to \star$.

Yorgey et al. (2012) describe how certain Haskell algebraic datatypes can be promoted into new *datakinds*. A simple example is *Bool*. The idea is that a definition

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

introduces a *kind* 'Bool with *types* 'True and 'False.[3] We can now write a datatype like

```
data OperatingSystem (unixLike :: 'Bool) where
  MacOS   :: OperatingSystem 'True
  Linux   :: OperatingSystem 'True
  Windows :: OperatingSystem 'False
```

where we annotate the *OperatingSystem* type with further information that can be used at compile-time.

2.2 Type families

GHC has long supported *open type families* (Chakravarty et al. 2005b), and with the release of version 7.8 comes their closed form (Eisenberg et al. 2014). A type family can be viewed as a function at the type level. As such, type families enable expressive type-level programming. For example, we can easily define an *IsZero* function over type-level natural numbers:

```
data Nat1 = Zero | Succ Nat1
type family IsZero (n :: 'Nat1) :: 'Bool where
  IsZero 'Zero     = 'True
  IsZero ('Succ n) = 'False
```

This new feature of *closed* type families plays a critical role in the present work because they enable *kind inference*. Unlike open type families, closed type families have all of their equations written in one place, and so GHC can use the equations to infer the kinds of the type family arguments and result. Indeed, the *IsZero* example could have been written without the 'Nat_1 and 'Bool kind annotations.

2.3 Kind polymorphism

Yorgey et al. also introduce *kind polymorphism*, which allows for a definition to be abstract in its kinds. For example, we can write a kind-polymorphic *Length* function over type-level lists:

```
type family Length (list :: '[a]) :: 'Nat1 where
  Length '[]        = 'Zero
  Length (x ': xs) = 'Succ (Length xs)
```

In this code, note that *a* is a *kind variable*, as it classifies the type *list*. Thus, *Length* is kind-polymorphic. Kind polymorphism is naturally essential to promoting type-polymorphic functions.

2.4 Type-level literals

Iavor Diatchki has implemented type-level literals into GHC.[4] Two kinds of type-level literals are allowed: natural numbers and strings. The use of a numeric literal in a type will produce a type of kind *Nat* (separate from our Nat_1), and the *GHC.TypeLits* module exports several type families (such as + and ∗) that can manipulate *Nat*s. The use of a string literal at the type level will produce a type of kind *Symbol*. Currently, there are no operations on *Symbol*s other than equality and comparison.

3. Promoting functions

As examples, let's examine a few library functions extracted from the `Data.List` and `Data.Maybe` modules:

```
span :: (a -> Bool) -> [a] -> ([a], [a])
span _ xs@[]  = (xs, xs)
span p xs@(x : xs')
  | p x       = let (ys, zs) = span p xs' in (x : ys, zs)
  | otherwise = ([], xs)

nubBy :: (a -> a -> Bool) -> [a] -> [a]
nubBy eq []       = []
nubBy eq (x : xs) =
  x : nubBy eq (filter (\y -> not (eq x y)) xs)

groupBy :: (a -> a -> Bool) -> [a] -> [[a]]
groupBy _  []       = []
groupBy eq (x : xs) = (x : ys) : groupBy eq zs
  where (ys, zs) = span (eq x) xs

mapMaybe :: (a -> Maybe b) -> [a] -> [b]
mapMaybe _ []       = []
mapMaybe f (x : xs) =
  let rs = mapMaybe f xs in
  case f x of
    Nothing -> rs
    Just r  -> r : rs
```

Now that the programmer has access to datakinds, she might wish to apply the functions above at the type level. These functions are all defined over terms, so she decides to simply rewrite the functions as type families. But she quickly encounters a problem. The functions above use **let** statements, **case** expressions, guards, higher-order functions, lambdas, partial application, **where** clauses, @-patterns and wildcard patterns. None of these features is available at the type level, so translating above definitions to type families is a daunting task.

Nevertheless it is possible to emulate all of these Haskell constructs – and thus implement all of the mentioned functions – at the type level by using only those features described in Section 2. The process of doing this is tedious, so we have extended the singletons library (Eisenberg and Weirich 2012) to do the promotion automatically. Promotion is implemented via Template Haskell and

[3] Diverging somewhat from GHC's parser, we will annotate datakinds with a ' to aid the reader.

[4] `http://www.haskell.org/ghc/docs/7.8.2/html/users_guide/type-level-literals.html`

generates type-level equivalents of definitions supplied by the user. Promotion is performed by invoking the *promote* function:

```
$(promote [d|
  map :: (a → b) → [a] → [b]
  map _ []       = []
  map f (x : xs) = f x : map f xs
 |])
```

A call to *promote* generates a type family implementing the promoted version of *map* as well as some auxiliary definitions required to make it work (details are given in Section 4.3). The functions above are all promotable using *promote*, without any edits.

3.1 A longer example – reordering of type-level lists

Having complex functions easily available at the type level facilitates more programming in types. As a slightly longer example, we consider the following function, *reorderBy*. The *reorderBy* function takes an equivalence predicate and two lists, which we'll call xs_1 and xs_2. The function reorders xs_1 to match the ordering in xs_2, where possible. That is, all elements in xs_1 that are equivalent to elements in xs_2 are brought to the front of the result list, and placed in the same order as those elements in xs_2. Elements in xs_1 not equivalent to anything in xs_2 are left in the same order and moved to the end of the result list. Extra elements in xs_2 are ignored.

Here is an implementation of *reorderBy*:

```
reorderBy :: ∀ a. (a → a → Bool) → [a] → [a] → [a]
reorderBy _  x []     = x
reorderBy eq x (h : t)
  = case extract h x of
      (lst, Nothing) → reorderBy eq lst t
      (lst, Just elt) → elt : (reorderBy eq lst t)
  where
    extract :: a → [a] → ([a], Maybe a)
    extract _ []      = ([], Nothing)
    extract s (h : t)
      | s `eq` h  =   (t, Just s)
      | otherwise = let (resList, resVal) = extract s t
                    in (h : resList, resVal)
```

This function, when promoted, serves a critical role in the *units* library (more fully described by Muranushi and Eisenberg (2014)). That library allows users to type-check their code with respect to units-of-measure, rather like the system developed by Kennedy (1996). A crucial capability of such a library is to type-check the multiplication of two dimensioned quantities. For example, if *v* is a velocity (i.e., a *Length* over a *Time*) and we multiply by *t*, a *Time*, we wish to get a *Length*. Internally, *units* stores the dimensions of a quantity as a type-level list where order is insignificant. When type-checking multiplication, we must combine two such lists, reordering one to match the other in order to avoid duplicating a dimension factor. Reordering is also used to ensure that addition happens between two quantities of the same dimension, once again, neglecting the order of the type-level lists. The type signatures for these operations involve several other concepts related to the *units* library, and a full explanation would take us too far afield.

As demonstrated here, a user can write normal term-level code and have it promoted automatically to the type level. This makes type-level programming much easier because the programmer can write his code using familiar and powerful term-level constructs and our library handles them under the hood. With our library, type-level programming also becomes more reliable: assuming the correctness of our implementation, it is possible to test correctness of term level functions using QuickCheck or HUnit and be confident that the promoted functions generated from tested definitions behave correctly. Testing hand-written type-level code is not as simple.

3.2 Promoted Prelude

Our library provides modules containing promoted functions from the standard Prelude as well as five other modules from the base package: `Data.Bool`, `Data.Either`, `Data.List`, `Data.Maybe` and `Data.Tuple`. These serve both as a convenience for users as well as a test of the robustness of our approach. The five `Data` modules mentioned above export a total of 125 functions. Out of these, we were able to promote 91 simply by wrapping the implementation from the base library in a Template Haskell quote and calling our *promote* function. Out of the 34 unpromotable functions:

- 18 functions are not promotable because they manipulate *Int* or *Integral* type-class values, or because they rely on functions that do so and thus have *Int* in their type signature. However, it is possible to promote all of these functions if they are rewritten to use *Nat*, the kind of type-level numeric literals. For example:

  ```
  $(promoteOnly [d|
    length :: [a] → Nat
    length []       = 0
    length (_ : xs) = 1 + length xs
   |])
  ```

  promotes correctly.
- 6 are not promotable because they use list comprehensions. They become promotable if we rewrite them to explicitly use *map* and *filter* functions.
- 4 functions are not promotable because they operate on strings.
- 5 functions are not promotable because they work with infinite lists and thus generate infinite types, which are not allowed in Haskell.
- 4 functions are not promotable because the promoted function name clashes with existing datatype. See Section 4.1.

Section 5 gives more detail about why the other functions were not promotable. The numbers above don't sum to 34 because some functions fall into several categories. For example, *findIndices* function uses list comprehensions, infinite lists, and integers. Some of the mentioned limitations have workarounds. After applying them we are left with only 7 functions that can't be promoted: 3 that return infinite lists and 4 that work on strings.

4. Promotion algorithm

Up until now, we have seen calls to our *promote* function. This section gives the gory details of how it works, under the hood.

4.1 Naming conventions

Promotion is performed by generating new Haskell definitions from definitions supplied by the user. Thus, we adopt some naming conventions so that programmers can later access the generated type-level definitions. Figure 1 shows typical examples and the full set of special cases. Occasionally, these conventions cause a conflict, such as for the *either* function and the *Either* datatype. In these cases, our version of the Prelude appends an underscore to avoid the conflict. Thus, our promoted *either* function is named *Either_*.

4.2 Preprocessing

The promoted definitions are generated using Template Haskell (Sheard and Peyton Jones 2002). Users quote the code they

Term-level	Promoted	Symbols
map	*Map*	*MapSym0*, *MapSym1*, *MapSym2*
++	:++	:++$, :++$$, :++$$$
Just	'*Just*	*JustSym0*, *JustSym1*
:	':	:$, :$$, :$$$
Special cases:		
[ ]	'[ ]	*NilSym0*
$	$	$$, $$$, $$$$
(,)	'(,)	*Tuple2Sym0*, *Tuple2Sym1*, ...
(#, #)	'(,)	*Tuple2Sym0*, *Tuple2Sym1*, ...
undefined	*Any*	*Any*

Figure 1. Examples demonstrating how names are transformed. See Section 4.3 for more information about symbols.

wish to promote in a declaration quote [*d*| ... |], which converts source Haskell syntax into the Template Haskell abstract syntax tree (AST).

Template Haskell's AST is quite large, as it intends to represent all of the constructs available in Haskell. However, many of these constructs are redundant. For example, Template Haskell maintains the distinction between (*list1* ++ *list2*) and ((++) *list1 list2*), even though these expressions have the same meaning. Thus, to make our task easier we wrote the th-desugar library.[5] This library converts the Template Haskell AST into a smaller core language. For example, **if** expressions are converted to **case** expressions with *True* and *False* branches, and **where** clauses are converted to **let** declarations. This preprocessing step is not mandatory to implement our approach – and in fact initially we did not perform it – but it allows us to focus on promoting a small core set of features instead of dealing with promoting constructs that are just syntactic sugar.

The th-desugar AST is presented in Figure 2 and more fully described in Section 6.1. All Haskell constructs are representable retaining their original meaning in this more condensed AST.

4.3 Functions and partial application at the type level

Functions at the type level and functions at the term level have different syntactic properties in Haskell. At the term level, functions are curried so it is natural to use partially applied functions. By contrast, calls to type-level functions in Haskell must be fully saturated (Chakravarty et al. 2005a), as allowing partially applied type functions wreaks havoc with type inference (see Section 7.1).

So, how to possibly promote a partially applied term-level function? We use the technique of defunctionalization, as first put forward by Reynolds (1972). The fundamental idea of defunctionalization is that functions are represented by opaque *symbols*, which are then applied to their arguments via a special application operator @@. Nested uses of @@ can apply a symbol to multiple arguments. We define @@ to be an ordinary open type family, so that we can add equations for new symbols at any time.

During promotion, we generate symbols for type families and data constructors. The name of a defunctionalization symbol in our implementation is created by appending *Sym0* (for alphanumeric identifiers) or $ (for operators) to the name of the type-level function. Thus, the expression *isJust Nothing* promotes to *IsJustSym0* @@ *NothingSym0* and *map pred* [ ] promotes to *MapSym0* @@ *PredSym0* @@ *NilSym0*. As usual, @@ is left-associative. In these examples, we see that *all* top-level identifiers are promoted to symbols. This is because Template Haskell offers no access to the types of terms, and thus our implementation cannot tell a partially applied function from a fully applied constant. We take the only way out and define, for example, **type** *NothingSym0* = '*Nothing* during promotion. It is then safe and correct to append *every* promoted identifier with *Sym0* or $.

4.3.1 The kind ↠

Because symbols are *not* functions, the kind of a symbol must not be built with →. Instead, we introduce the new kind ↠ (associating to the right, like →) to classify symbols. Thus, the kind of *MapSym0* is (*a* ↠ *b*) ↠ '[*a*] ↠ '[*b*].

Unlike closed promoted datatypes, though, we must be free to create new members of ↠ at any point in the program – it is a fundamentally open kind. Thus, we hook into Haskell's facility to introduce new, opaque, type-level constants through its datatype declaration mechanism. We wish to be able to say

```
data MapSym0 :: (a ↠ b) ↠ '[a] ↠ '[b]
```

using an explicit kind annotation on the datatype declaration. Here, we must be careful, though: all types that contain values must be of kind ⋆ in GHC.[6] Thus, GHC requires that the kind of a datatype end in ... → ⋆, as datatypes are normally meant to hold values. We can now figure out how ↠ must be defined:

```
data TyFun :: ⋆ → ⋆ → ⋆   -- only the promoted form is used
kind a ↠ b = 'TyFun a b → ⋆
```

where the second line uses a hypothetical syntax to introduce a kind synonym. Expanding this definition for ↠, we see that the kind of *MapSym0* indeed ends with ... → ⋆ as required.

In our actual implementation, we have no kind synonyms, and we are left with using the more verbose *TyFun* routinely.

4.3.2 The @@ type family and its instances

The application operator @@ is defined as an open type family; new instances (i.e., equations) of this family can be written at any time. Its definition is quite naturally

```
type family (f :: k1 ↠ k2) @@ (x :: k1) :: k2
```

Rewriting somewhat, the kind of @@ is (*k1* ↠ *k2*) → (*k1* → *k2*) – it converts a symbol into a real function.

To write the instances for our defunctionalized symbols, we must create a new symbol for every level of partial application. For example, *Map* might be totally unapplied, be given one argument, or be given two arguments. Thus, we get three symbols, *MapSym0*, *MapSym1*, and *MapSym2*, with kinds as follows:

```
MapSym0 :: (a ↠ b) ↠ '[a] ↠ '[b]
MapSym1 :: (a ↠ b) → '[a] ↠ '[b]
MapSym2 :: (a ↠ b) → '[a] → '[b]
```

Note how the choice of arrow changes between symbols. *MapSym0* must appear with @@ to use it at all, whereas *MapSym1* takes its first argument without @@. Indeed, the number assigned to a symbol denotes its honest-to-goodness arity as a GHC type.

With these definitions in hand, the instances for @@ are straightforward:

```
type instance MapSym0     @@ f  = MapSym1 f
type instance (MapSym1 f) @@ xs = MapSym2 f xs
type MapSym2 f xs = Map f xs
type family Map (f :: a ↠ b) (xs :: '[a]) :: '[b] where ...
```

The definition for *MapSym2* is not strictly necessary in our scheme; it is included to parallel the non-function case (such as *NothingSym0*, above).

[5] cabal install th-desugar

[6] We ignore here, and throughout, the existence of the kind # that classifies unlifted types.

4.3.3 Kind inference

It is essential that the kinds of the symbols be correct for the promoted code to kind-check. But, given that Template Haskell is not type-aware, how are these kinds determined? At first glance, the problem seems easy: just look at top-level type signatures. After all, it would seem to be a reasonable burden to ask programmers keen on type-level programming to write top-level annotations for their definitions. However, these top-level annotations turn out to be insufficient. As we will shortly see, we use the technique of lambda lifting (Johnsson 1985) to deal with anonymous functions within expressions. Lambda-expressions tend not to have any type annotations, and it would be annoying to users to require annotations here, both on arguments and on the return value. So, we must strive for something better.

To get the kinds right for the symbols, we wish to propagate the kinds up from the type family representing the function at hand. Let's continue to consider the *Map* example. The type family *Map* is given explicit kind annotations (produced from *map*'s top-level type signature), but its kinds could also have been inferred by GHC. Then, the type *MapSym2*, a simple type synonym for *Map*, also gets the correct kinds, via simple kind inference on the definition for *MapSym2*. Thus, we have $MapSym2 :: (a \twoheadrightarrow b) \rightarrow\ '[a] \rightarrow\ '[b]$. To see how *MapSym1* and *MapSym0* get their kinds, let's look at their full definitions:

```
type KindOf (a :: k) = ( 'KProxy :: KProxy k)
    -- defined once for all symbols
data MapSym1 x f where
  MapSym1KindInference
     :: KindOf ((MapSym1 x) @@ arg)
         ~ KindOf (MapSym2 x arg)
     ⇒ MapSym1 x f
data MapSym0 f where
  MapSym0KindInference
     :: KindOf (MapSym0 @@ arg)
         ~ KindOf (MapSym1 arg)
     ⇒ MapSym0 f
```

Much like in the old days before explicit kind annotations, we use dummy data constructors to constrain the kinds of the symbols. The *KindOf* type synonym discards the types, leaving only the kinds. This turns out to be crucial, because the discarded types are ambiguous; without *KindOf*, GHC reports ambiguity errors for these data constructors. In the definition for *MapSym1*, we see that the type variable x is used as an argument to *MapSym2*. This fixes the kind of x to be $(a \twoheadrightarrow b)$. We then see that $KindOf\ ((MapSym1\ x)\ @@\ arg) \sim KindOf\ (MapSym2\ x\ arg)$ So, $(MapSym1\ x)\ @@\ arg$ and $MapSym2\ x\ arg$ must have the same kinds, specifically $'[b]$. Given that @@ has the correct kind, this means that $(MapSym1\ a)$ must have the correct kind (that is, $'[a] \twoheadrightarrow\ '[b]$), and thus that the type variable f has the right kind (that is, $TyFun\ '[a]\ '[b]$, unrolling the definition for $\twoheadrightarrow$). Putting this all together, we see that $MapSym1 :: (a \twoheadrightarrow b) \rightarrow\ '[a] \twoheadrightarrow\ '[b]$, as desired. A similar line of reasoning gives us $MapSym0 :: (a \twoheadrightarrow b) \twoheadrightarrow\ '[a] \twoheadrightarrow\ '[b]$.

4.3.4 η-expansion

There is one corner case we must handle during function promotion. Haskellers often η-reduce their definitions where possible – that is, the type of a function may have more arrows in it than patterns in the function's clauses. A convenient example is *zip*:

```
zip :: [a] → [b] → [(a, b)]
zip = zipWith (,)
```

A naive promotion of *zip* would give us $Zip :: \ '[a] \twoheadrightarrow\ '[b] \twoheadrightarrow\ '[\ '(a, b)]$. This promotion would not correspond to users' intuitions – the kind has the wrong arrows! We would want to be able to say $Zip\ '[Int, Bool]\ '[Char, Double]$ and get $'[\ '(Int, Char), '(Bool, Double)]$. Instead, users would have to use @@ to use *Zip*.

The solution to this is straightforward: η-expansion. When promoting *zip*, we actually promote the following version:

```
zip :: [a] → [b] → [(a, b)]
zip eta₁ eta₂ = zipWith (,) eta₁ eta₂
```

This η-expansion is done only when there is a type signature to signal the need for expansion.

4.4 Datatypes

At the term level, data constructors can be used in any context expecting a function. We want to have the same uniformity at the type level. We rely on GHC's built-in promotion mechanism to promote datatypes, and it does most of the work for us.[7] However, we must generate the defunctionalization symbols manually. For every data constructor, we generate symbols and @@ instances in the same way we generate them for promoted functions. This symbol generation may seem somewhat redundant for promoted data constructors, because they *are* allowed to appear partially applied in programs. Nonetheless, given that $\rightarrow$ and $\twoheadrightarrow$ are distinct kinds, we must defunctionalize the data constructors to achieve uniformity with promoted functions.

By using GHC's mechanism for datatype promotion, we run into one technical snag. During promotion, all arrows $\rightarrow$ become defunctionalized arrows $\twoheadrightarrow$. Since GHC does not apply this transformation during its promotion of datatypes, promoted datatypes that store functions will not work correctly. For example, while promotion of the following *Arr* datatype will succeed, promotion of the *arrApply* function will fail due to a kind mismatch:

```
data Arr a b = Arr (a → b)
arrApply :: Arr a b → a → b
arrApply (Arr f) a = f a
```

We could solve this problem by implementing our own datatype promotion mechanism using Template Haskell. That design would be awkward for the programmer, though, as there would be two promoted versions of each datatype: one generated by GHC and another one by us, with slightly different names.

4.5 case expressions

A **case** expression inspects a scrutinee and selects an appropriate alternative through pattern matching. The only way we can perform pattern matching at the type level is via a type family. Thus, we turn **case** expressions into fresh closed type families. Each alternative in the original **case** becomes a defining equation of the type family. We must, however, remember that case alternatives may use local variables bound outside of the case expression. Since type families are top-level constructs, an equation's RHS can use only bindings introduced by the patterns in its LHS. Therefore, when promoting a **case** expression to a type family, we pass all in-scope bindings as parameters to the type family – much like in lambda lifting. The scrutinee itself is the last parameter.

Here is an example from the `Data.Maybe` module:

```
fromMaybe :: a → Maybe a → a
fromMaybe d x = case x of
  Nothing → d
  Just v  → v
```

[7] We make no attempt to detect datatype definitions that can't be promoted by GHC, for example GADTs. We naturally cannot promote these datatypes either.

This function promotes to the following:[8]

```
type family Case d x scrut where
  Case d x 'Nothing  = d
  Case d x ('Just v) = v
type family FromMaybe (t1 :: a) (t2 :: 'Maybe a) :: a
  where
    FromMaybe d x = Case d x x
```

The **case** expression is promoted to the type family *Case* and its application on the RHS of *FromMaybe*. Local variables *d* and *x*, both in scope at the site of the **case** statement, are passed in, along with the scrutinee, also *x*. In the definition of *Case*, the scrutinee – that is, the third parameter – is matched against, according to the original, unpromoted definition.

It is conceivable to do a dependency check to eliminate the redundant second parameter to *Case*. We have not implemented this as we suspect that benefits of such an optimization would be small, if any.

We also note that, because this type family *Case* is used only once and is fully applied, there is no need to create the defunctionalization symbols for it.

4.6 Lambda expressions

Promoting an anonymous function poses two difficulties. Firstly, lambdas, like all functions, are first-class values that can be passed around and partially applied. Secondly, the body of a lambda can use variables bound in the surrounding scope – the lambda can define a closure. For example, in the *dropWhileEnd* function from the `Data.List` module, *p* is used inside a lambda body but is bound outside of it:

```
dropWhileEnd :: (a → Bool) → [a] → [a]
dropWhileEnd p = foldr (λx xs → if p x && null xs
                                 then []
                                 else x : xs) []
```

Happily, we have already solved both problems, making promotion of lambdas straightforward. A lambda expression promotes to the use of a fresh type family, along with the family's definition. Just like with **case**, all in-scope local variables are turned into explicit parameters. The technique we use here is, of course, lambda lifting (Johnsson 1985).

The major difference between lambdas and **case** expressions is that, for lambdas, we must generate symbols so that the lambda can be partially applied and passed around as a first-class value. The freshness of the type family name prevents a programmer from explicitly calling type families that encode promoted lambdas. The result of promoting *dropWhileEnd* looks like this, omitting the tiresome symbol definitions:

```
type family Case p eta₁ x xs scrut where
  Case p eta₁ x xs 'True  = NilSym0
  Case p eta₁ x xs 'False = (:$) @@ x @@ xs
type family Lambda p eta₁ x xs where
  Lambda p eta₁ x xs = Case p eta₁ x xs
    ((:&&$) @@ (p @@ x) @@ (NullSym0 @@ xs))
type family DropWhileEnd (p :: a ↠ Bool)
                         (eta₁ :: [a]) :: [a] where
  DropWhileEnd p eta₁ =
    (FoldrSym0 @@ (LambdaSym0 @@ p @@ eta₁)
               @@ NilSym0) @@ eta₁
```

[8] Here and elsewhere, we omit various decorations put on generated names to guarantee freshness.

4.7 let statements

A **let** statement introduces a set of (potentially recursive) local bindings. Since there is no local binding construct available at the type level, we must once again lift **let** bindings to the top level. As we have done elsewhere, each **let**-bound name is freshened to guarantee uniqueness. Let-lifting differs in an important respect from **case**- and lambda-lifting: **let**-bound identifiers have an unfolding, unlike **case**- and lambda-bound identifiers. Thus, we do not promote a **let**-bound identifier into a type variable, but instead into a call of the top-level definition generated by the identifier's declaration.

Consider this function:

```
doubleSucc :: Nat₁ → Nat₁
doubleSucc x = let y = Succ x
                   z = Succ y
               in  z
```

In this example, *x* is bound in the scope surrounding the **let**-statement, *y* appears both as a variable binding and on the right-hand side of another binding, namely *z*, while *z* itself appears as a variable binding and inside the body of the **let**. The *y* and *z* bindings will be lifted to become top-level identifiers (type synonyms in this example) that accept *x* as parameter. Since the names of *y* and *z* will be fresh, we must build a substitution from the original **let**-bound identifier to a promoted, freshened identifier applied to all local variables in scope. Thus, the promoted code will look like this:

```
type LetY x = SuccSym0 @@ x
type LetZ x = SuccSym0 @@(LetYSym1 x)
type family DoubleSucc (a :: Nat) :: Nat where
  DoubleSucc x = LetZSym1 x
```

Notice how *x*, which was bound in the scope surrounding the **let**-statement, became an explicit parameter of every lifted **let**-declaration. It is also passed in at every use site of these lifted **let**-bound identifiers.

Recursive **let**-bindings do not pose any particular problem here, as type families may be recursive. A recursive definition that leads to an infinite data structure, however, is problematic – GHC does not permit infinite types. See Section 5 for more discussion.

4.8 Type classes and instances

Type classes enable several different programming capabilities. We review how these manifest at the type level before presenting our promotion strategy.

4.8.1 Ad hoc polymorphism

A Haskell type class enables *ad hoc polymorphism*, where one function can have different implementations at different types. The notion of an explicit type class is made necessary by the lack of a type-case. For example, consider the following bogus definition:

```
sometimesNot :: ∀ a. a → a
sometimesNot x = typecase a of Bool → not x
                               _    → x
```

Here, we check the instantiation for *a* at runtime and make a decision on how to proceed based on the type *a*. This is, of course, not possible in Haskell – it would break both type erasure and parametricity. When a user wants functionality like *sometimesNot*, she uses a type class. The use of this type class then appears in the type of *sometimesNot*:

```
sometimesNot :: SupportsSometimesNot a ⇒ a → a
```

By including the type constraint there, the type advertises that it is not strictly parametric in *a*.

Promoting this concept is made easy by one simple fact: type families are *not* parametric in their kinds! In other words, a type family can pattern-match on the *kinds* of its arguments, not just the types. The following promotion of the original, bogus *sometimesNot* is perfectly valid:

```
type family SometimesNot (x :: a) :: a where
  SometimesNot (x :: Bool) = Not x
  SometimesNot x           = x
```

In this type family, we match on the *kind* of the parameter to choose which equation to use, making this a *kind-indexed* type family. We should note that such action does not cause trouble with type erasure, as both types and kinds are compile-time constructs.

4.8.2 Open functions

A normal Haskell function is *closed*. All of its defining equations must be listed in one place. A type class method, on the other hand, is *open*, allowing its defining equations to be spread across modules. Promoting an open function is thus easy: use an *open* type family.

4.8.3 Method defaulting

Type classes also permit the possibility of method defaults. This is evident in the definition of *Eq*:

```
class Eq a where
  (==), (/=) :: a → a → Bool
  x == y = not (x /= y)
  x /= y = not (x == y)
```

If an instance does not supply a definition for one of these methods, the default is used. Happily, GHC provides a similar capability through associated type families. GHC accepts an associated type family default, much like a method default. The default is used only when an instance does not supply another definition.

4.8.4 Promotion

The first two capabilities above – ad hoc polymorphism and open functions – seem to suggest that we promote a class declaration by rewriting all of its methods as open type families and not to bother promoting the class itself. However, method defaulting, which is much used in practice, tells us that we somehow need to package these type families in a new class definition in order to make the open type families *associated*, and hence defaultable.

To promote a type class, then, we need a *kind class*! Though the syntax is somewhat burdensome, GHC supports kind classes via a poly-kinded type class definition where the type itself is irrelevant. Putting this all together, here is the promotion of *Eq*:[9]

```
data KProxy (a :: ⋆) = KProxy   -- in Data.Proxy
class (kproxy ∼ 'KProxy)
      ⇒ PEq (kproxy :: 'KProxy a) where
  type (x :: a) :== (y :: a) :: Bool
  type (x :: a) :/= (y :: a) :: Bool
  type x :== y = Not (x :/= y)
  type x :/= y = Not (x :== y)
```

We make use here of the type *KProxy*, which when promoted, is a type-level proxy for a kind argument. Its definition restricts its type parameter *a* to be of kind $\star$ so that the type is promotable; GHC does not promote poly-kinded datatypes. However, the type is intended to be used only when promoted.

[9] The definition exactly as stated does not work in GHC 7.8.2, due to a bug in kind-checking associated types. It is reported as GHC bug #9063 and can be worked around via kind annotations on the default definitions.

The class declaration head now takes a type-level proxy for the kind-level argument *a*. In other words, *PEq* is properly a kind class, as desired. (The *kproxy* ∼ '*KProxy* constraint forces the term-level argument *kproxy* to be irrelevant. It is necessary for recursive definitions to type check.)

Instance promotion Given all the work above, promoting instances is quite straightforward: we promote the instance head to use a *KProxy* parameter and promote method bodies just like normal functions. GHC's built-in defaulting mechanism does the defaulting for us.

Constraint promotion How do we promote a constrained function? We simply drop the constraints. Making a type family associated with a class attaches the type family parameters to the class parameters (enabling more kind checking) and allows for defaulting. But, using an associated type family does *not* induce the need for a class constraint. This is crucial, because there is no way of writing a constrained type family instance. Thus, we safely ignore any class constraints during promotion.

If we just drop constraints, couldn't a user call an associated type family at the wrong kind? (For example, consider (:==) at *Bool* → *Bool*.) Yes, this can happen, but nothing bad comes from it – the type family just does not reduce. Types being stuck cause no problems; they are just empty types. This, of course, is quite different from terms being stuck, which generally leads to a crash of some sort.

***Deriving* Eq*,* Ord *and* Bounded** If a datatype derives the *Eq*, *Ord* or *Bounded* classes, we automatically derive the promoted instance. Other derivable classes are currently ignored.

4.9 Other language features

Below we list other language features present in Chapters 3 and 4 of the Haskell 2010 Language Report that were omitted in earlier discussion.

Records: Promotion of records is fully supported. For datatypes declared using record syntax, th-desugar generates appropriate accessor functions. Record update, construction and pattern-matching syntax are desugared into simpler constructs that rely on simple pattern matching, case expressions and datatype construction. There is one restriction on record promotion: a record datatype definition must be promoted in a separate Template Haskell splice from its use sites. This is a limitation in the th-desugar library, which can look up record field names only in a splice that has already been type-checked.

Type signatures on expressions: We promote type-annotated expressions to kind-annotated types.

Errors: The Haskell 2010 Language Report defines `error` and `undefined` functions that cause immediate program termination when evaluated. Both these functions represent $\bot$ and inhabit every type. We don't have type-level expressions that cause type-checking termination when evaluated, but we can have types that belong to any kind. Furthermore, it seems reasonable to equate $\bot$ with a "stuck" type – a type-level expression containing a type family but unable to progress. Thus *error* promotes to the *Error* open type family:

```
type family Error (a :: Symbol) :: k
```

This family has no instances, so it is *always* stuck. Along similar lines, *undefined* promotes to *Any*, a special type in GHC belonging to any kind.

Other syntactic sugar: This catch-all entry includes **if** conditionals, operator sections, and pattern guards. These are eliminated by the th-desugar preprocessing pass, in favour of **case** statements (for conditionals and guards) or lambda-expressions (for sections).

5. Limitations

Earlier work on this subject (Eisenberg and Weirich 2012) listed language features that were either not yet supported by the singletons library or problematic to implement. We can now state that almost all such features are now implemented and fully supported. Exceptions include the following:

Infinite terms: While it is possible to construct infinite terms thanks to laziness, it is not possible to construct infinite types. Therefore, it will not be possible to use any promoted expression that generates such a type. A good example of this is the *iterate* function found in the standard Prelude:

$$\mathit{iterate} :: (a \to a) \to a \to [a]$$
$$\mathit{iterate}\ f\ x = x : \mathit{iterate}\ f\ (f\ x)$$

The promotion itself does not fail, but any attempt to use promoted *Iterate* does. This example also demonstrates another shortcoming of the current implementation. Our algorithm operates in an untyped setting and only reports errors when the algorithm gets stuck. This means we can generate definitions that are unusable. At the moment, the responsibility of identifying such a problem rests on the programmer.

Literals: We rely on GHC's built-in promotion of literals, so our approach is limited by GHC's capabilities. At the moment, promotion of integer literals to type-level *Nat*s is supported, but this approach has drawbacks: negative integer literals do not promote, and the types do not work out – the type *Int* does not promote to the kind *Nat*.

String literals also present a problem, mainly because after GHC promotes them to the type level they are no longer considered lists of characters. This means, for example, that it is impossible to promote code that concatenates two string literals using (++). It seems to us that it is impossible to bridge this gap with the current implementation of type-level strings within GHC.

Datatypes storing functions: We do not support the promotion of datatypes that store functions. See Section 4.4 for details.

do-notation: th-desugar preprocessing desugars **do**-notation along the lines of the desugaring described in the Haskell Report. This creates lambda-expressions composed using monadic bind operators. While lambdas and operators are by themselves promotable, the types of monadic operations pose a problem. They involve a higher-kinded type variable (the *m* in *Monad m*). Haskell's support for kind variables does not have a system of classifying kind variables. That is, there is no such thing as a "higher-sorted" kind variable. If we were to try to promote the type of ($\gg\!=$), we would have to get $ma \to (a \twoheadrightarrow mb) \to mb$. Here, we've removed the need for higher sorts by writing what should be *m a* as the single variable *ma*. But, we have no way of expressing relation between *ma* and *a* in the type signature of a hypothetical ($:\gg\!=$) type family. It is possible to put explicit type annotations on hand-written monadic expressions to guide GHC's kind inference and have them promote correctly. But doing so for desugared **do**-notation would require us to write our own type inference. Thus, **do**-notation is not promotable.

List comprehensions: These are syntactic sugar for monadic notation and thus do not promote for exactly the same reasons as **do**-notation.

Arithmetic sequences: These rely on the *Enum* type class, which is implemented using integers and infinite lists. Integers can be worked around with *Nat*s. Infinite lists however are a more fundamental problem, as we discussed above.

Show and Read type classes: These rely critically on string manipulation, which is not available on type-level *Symbol*s.

Fixity declarations for datatypes: Due to a Template Haskell bug, fixity declarations for capitalized identifiers (including symbols beginning with ":") currently do not work.[10]

6. Formalization and proof

The process we describe in Section 4 is rather involved. In this section, we present a formal grammar for a subset of Haskell and a promotion algorithm over this grammar. We then prove that

- promoting a well-typed, promotable term yields a well-kinded type, and
- assuming lambda-lifting is correct, the semantics of a promoted term lines up with that of the original term.

Both the formal promotion algorithm and the proof are done in two stages. First, we promote (written as a postfix $\Uparrow$) expressions into *extended types*, written $\hat{\tau}$, which contains the grammar of types τ but also includes anonymous functions, and **case** and **let** expressions. We then *reduce* this extended type language into the language of ordinary types through the operation $\lfloor\cdot\rfloor^{\theta}_{\beta}$, discussed more in Section 6.3.

6.1 The formal grammar

The grammar we work with is presented in Figure 2.[11] Much of the first part of this figure – a rendering of the actual AST used in our implementation – is rather standard for Haskell. There are a few points of interest:

Literals: Literals are included as $\langle \mathrm{lit} \rangle$ in the definition of expressions e, as literals form part of the AST used in our implementation. However, as promotion of literals *does* disrupt their typing and semantics, we omit them from the rest of this section.

let declarations: **let**-declarations δ include a bound variable x, an optional signature σ, and a list of function clauses $\overline{\overline{\pi} \mapsto e}$. Note that each clause is a list of patterns $\overline{\pi}$ mapping to a single expression e.

Type family applications: The grammar for types τ includes type family application $F(\overline{\tau})$. This is written with parentheses to emphasize the fact that type families must always appear fully saturated. As implemented in GHC, this is properly part of the syntax, not part of the type system – any use of a bare type family F is malformed.

Kind schemes: Although kind schemes ψ cannot be written in Haskell, a Haskell programmer using kind-polymorphism must consider these, which classify type constructors and promoted data constructors.

Figure 2 includes also the definition for the contexts used in the typing judgements and proofs.

Our notation for lists is optimized for brevity, sometimes at the risk of introducing ambiguity. We frequently simply use an overbar to represent a list. When the length of the list is relevant, we write it as a superscript, thus: $\overline{\tau}^n$. As we never have nested lists, we conflate appending with concatenation: $\overline{\tau}, \tau'$ adds one element to the list $\overline{\tau}$, while $\overline{\tau}, \overline{\tau}'$ concatenates two lists.

[10] See https://ghc.haskell.org/trac/ghc/ticket/9066

[11] Our formalism was developed and typeset using Ott (Sewell et al. 2010)

Metavariables:

Term vars	x, y	Data constructors	K
Type vars	α, β	Type constructors	T
Kind vars	$\mathcal{X}, \mathcal{Y}$	Type families	F

Core th-desugar grammar:

$e ::= x \mid K \mid e_1\ e_2 \mid \lambda x \mapsto e \mid \langle \mathsf{lit} \rangle$	Expressions
$\quad \mid \ \ \textbf{case}\ e_0\ \textbf{of}\ \overline{\pi \mapsto e} \mid \textbf{let}\ \overline{\delta}\ \textbf{in}\ e \mid e :: \tau$	
$\pi ::= x \mid K\ \overline{\pi} \mid \_$	Patterns
$\delta ::= (x :: \sigma)\{\overline{\overline{\pi} \mapsto e}\} \mid x\{\overline{\overline{\pi} \mapsto e}\}$	**let** declarations
$\tau ::= \alpha \mid (\to) \mid \tau_1\ \tau_2 \mid {}'K \mid T \mid \tau :: \kappa \mid F(\overline{\tau})$	Types
$\sigma ::= \forall\, \alpha.\sigma \mid \tau$	Type schemes
$\kappa ::= \mathcal{X} \mid {}'T\ \overline{\kappa} \mid \kappa_1 \to \kappa_2 \mid \star$	Kinds
$\psi ::= \forall\, \mathcal{X}.\psi \mid \kappa$	Kind schemes

Top-level declarations:

$dec ::= \textbf{type}\ F\ \overline{tvb} = \tau$	Declarations
$\quad \mid\ \textbf{type family}\ F\ \overline{tvb}\ \textbf{where}\ \overline{\overline{\tau} \mapsto \tau'}$	
$tvb ::= \alpha \mid \alpha :: \kappa$	Type var. binders

Grammar for extended types:

$\hat{\tau} ::= \alpha \mid (\to) \mid \hat{\tau}_1\ \hat{\tau}_2 \mid {}'K \mid T \mid \hat{\tau} :: \kappa \mid F(\overline{\hat{\tau}})$	Extended types
$\quad \mid \ \ \lambda\alpha \mapsto \hat{\tau} \mid \textbf{case}\ \hat{\tau}_0\ \textbf{of}\ \overline{\tau \mapsto \hat{\tau}'} \mid \textbf{let}\ \overline{\omega}\ \textbf{in}\ \hat{\tau}$	
$\omega ::= (\alpha :: \psi)\{\overline{\overline{\tau} \mapsto \hat{\tau}'}\} \mid \alpha\{\overline{\overline{\tau} \mapsto \hat{\tau}'}\}$	Type-**let** decls.
$\Gamma ::= \varnothing \mid \Gamma, x{:}\tau \mid \Gamma, x{:}\sigma \mid \Gamma, \alpha{:}\kappa \mid \Gamma, \mathcal{X}$	Type contexts
$\hat{\Gamma} ::= \varnothing \mid \hat{\Gamma}, \alpha{:}\kappa \mid \hat{\Gamma}, \alpha{:}\psi \mid \hat{\Gamma}, \mathcal{X}$	Ext. type contexts
$\theta ::= \varnothing \mid \theta, x \mapsto e \mid \theta, \alpha \mapsto \tau$	Substitutions
$\Sigma ::= \overline{\delta}$	Environments

Other notation conventions:

$\mathsf{Sym}_n(K)$ and $\mathsf{Sym}_n(F)$ mean the nth symbol derived from K and F, respectively; these are both type constructors T.
Lambda, Case, and $\mathsf{Let}(\alpha)$ are fresh names for type families F.
$(@@)$ is a type family F; $(\twoheadrightarrow)$ is a type constructor T.
$(\to)$ and $(\twoheadrightarrow)$ associate to the right; $(@@)$ to the left.
$tvs(\Gamma)$ and $kvs(\Gamma)$ extract bound type and kind variables, resp.
$ftv(\overline{\tau})$ and $fkv(\overline{\kappa})$ extract free type and kind variables, resp.

Figure 2. The grammar for the th-desugar subset of Haskell, along with other definitions used in our proof.

Context promotion ($\Gamma\Uparrow := \hat{\Gamma}'$):

$$\begin{aligned}
\varnothing\Uparrow &:= \varnothing \\
(\Gamma, x{:}\tau)\Uparrow &:= \Gamma\Uparrow, x\Uparrow{:}\tau\Uparrow \\
(\Gamma, x{:}\sigma)\Uparrow &:= \Gamma\Uparrow, x\Uparrow{:}\sigma\Uparrow \\
(\Gamma, \alpha{:}\star)\Uparrow &:= \Gamma\Uparrow, \alpha\Uparrow
\end{aligned}$$

Expression promotion ($e\Uparrow := \hat{\tau}$):

$$\begin{aligned}
x\Uparrow &:= x\Uparrow \\
K\Uparrow &:= \mathsf{Sym}_0(K) \\
(e_1\ e_2)\Uparrow &:= e_1\Uparrow\ @@\ e_2\Uparrow \\
(\lambda x \mapsto e)\Uparrow &:= \lambda x\Uparrow \mapsto e\Uparrow \\
(\textbf{case}\ e_0\ \textbf{of}\ \overline{\pi \mapsto e})\Uparrow &:= \ \textbf{case}\ e_0\Uparrow\ \textbf{of}\ \overline{\pi \mapsto e}\Uparrow \\
(\textbf{let}\ \overline{\delta}\ \textbf{in}\ e)\Uparrow &:= \textbf{let}\ \overline{\delta}\Uparrow\ \textbf{in}\ e\Uparrow \\
(e :: \tau)\Uparrow &:= e\Uparrow :: \tau\Uparrow
\end{aligned}$$

Match promotion:

$$(\pi \mapsto e)\Uparrow := \pi\Uparrow \mapsto e\Uparrow$$

Pattern promotion ($\pi\Uparrow := \tau$):

$$\begin{aligned}
x\Uparrow &:= x\Uparrow \\
(K\ \overline{\pi})\Uparrow &:= {}'K\ \overline{\pi}\Uparrow && (K \text{ promotable}) \\
_\Uparrow &:= \alpha && (\alpha \text{ fresh})
\end{aligned}$$

Let declaration promotion ($\delta\Uparrow := \omega$):

$$\begin{aligned}
(x :: \sigma)\{\overline{\overline{\pi} \mapsto e}\}\Uparrow &:= \ (x\Uparrow :: \sigma\Uparrow)\{\overline{\overline{\pi} \mapsto e}\Uparrow\} \\
x\{\overline{\overline{\pi} \mapsto e}\}\Uparrow &:= x\Uparrow\{\overline{\overline{\pi} \mapsto e}\Uparrow\}
\end{aligned}$$

Clause promotion:

$$(\overline{\pi} \mapsto e)\Uparrow := \overline{\pi}\Uparrow \mapsto e\Uparrow$$

Type promotion ($\tau\Uparrow := \kappa$):

$$\begin{aligned}
\alpha\Uparrow &:= \alpha\Uparrow \\
(\tau_1 \to \tau_2)\Uparrow &:= \tau_1\Uparrow \twoheadrightarrow \tau_2\Uparrow \\
(T\ \overline{\tau})\Uparrow &:= {}'T\ \overline{\tau}\Uparrow && (T : \overline{\star} \to \star) \\
(\tau :: \kappa)\Uparrow &:= \tau\Uparrow
\end{aligned}$$

Type scheme promotion ($\sigma\Uparrow := \psi$):

$$\begin{aligned}
(\forall\, \overline{\alpha}.\tau)\Uparrow &:= \forall\, \overline{\alpha}\Uparrow.\tau\Uparrow \\
\tau\Uparrow &:= \tau\Uparrow
\end{aligned}$$

Figure 3. Promotion algorithm. The promotion operator $\Uparrow$ implicitly distributes over lists. The (K promotable) condition refers to whether or not GHC can promote K; see Section 3.3 of Yorgey et al. (2012) for details.

6.2 Promotion algorithm

Figure 3 contains the formal promotion algorithm. This algorithm is appropriately partial. For example, the cases for promoting a type are quite limited; we cannot promote type families or already-promoted data constructors. When no equation in the algorithm statement is applicable for a given τ, then $\tau\Uparrow$ does not exist. If $\tau\Uparrow$ does not exist, then neither does any form containing $\tau\Uparrow$.

Variables are promoted to fresh variables. For example, the variable $x\Uparrow$ is a type variable (like α), but is distinct from other αs. In other aspects, $x\Uparrow$ is an ordinary type variable, making a type scheme like $\forall\, x\Uparrow.\tau$ well-formed.

This algorithm performs defunctionalization. This can be seen in the definitions for $K\Uparrow$, $(e_1\ e_2)\Uparrow$, and $(\tau_1 \to \tau_2)\Uparrow$ – all promoted functions are defunctionalized and must be applied using @@. No expression form promotes to a standard type-level application.

Patterns promote to standard, non-extended types. This fits well with the use of types as patterns when defining type families.

Contexts are promoted to extended contexts. The only difference between an extended context and a regular one is that extended contexts may contain bindings of the form $\alpha{:}\psi$. In Haskell, type variables always have a monomorphic kind; only top-level definitions such as data or type constructors can be kind-polymorphic. Thus, the $\alpha{:}\psi$ form must be excluded from regular contexts. On the other hand, extended types need bindings of this form to support type-level **let** over kind-polymorphic functions.

6.3 Reduction algorithm

After promoting an expression to an extended type, we then *reduce* it back into a regular type. This process entails rewriting the type to fit into the grammar of regular types and emitting top-level **type** and **type family** declarations as appropriate. The algorithm appears in Figure 4. Unlike promotion, reduction is a total operation – it has no possibility of failure.

Reduction on extended types, written $\lfloor \hat{\tau} \rfloor^{\theta}_{\overline{\beta}}$, is parameterized by a list of free type variables $\overline{\beta}$ and a substitution from type variables to types θ. The local variables $\overline{\beta}$ are necessary when working with fresh top-level declarations in order to pass these variables as parameters. The substitution θ maps **let**-bound variables to their lambda-lifted equivalents.

For example, consider *stutter* and its promotion:

```
stutter :: [a] → [a]
stutter (x : xs) =
  let cons ys = x : ys in
```

Reduction of contexts ($\lfloor\hat{\Gamma}\rfloor = \Gamma'$):

$$\begin{aligned} \lfloor\varnothing\rfloor &:= \varnothing \\ \lfloor(\Gamma,\alpha{:}\kappa)\rfloor &:= \lfloor\Gamma\rfloor,\alpha{:}\kappa \\ \lfloor(\Gamma,\alpha{:}\psi)\rfloor &:= \lfloor\Gamma\rfloor \\ \lfloor(\Gamma,\mathcal{X})\rfloor &:= \lfloor\Gamma\rfloor,\mathcal{X} \end{aligned}$$

Reduction of extended types ($\lfloor\hat{\tau}\rfloor^{\theta}_{\overline{\beta}} = \tau'$):

$$\begin{aligned} \lfloor\alpha\rfloor^{\theta}_{\overline{\beta}} &:= \theta(\alpha) \\ \lfloor(\rightarrow)\rfloor^{\theta}_{\overline{\beta}} &:= (\rightarrow) \\ \lfloor\hat{\tau}_1\,\hat{\tau}_2\rfloor^{\theta}_{\overline{\beta}} &:= \lfloor\hat{\tau}_1\rfloor^{\theta}_{\overline{\beta}}\,\lfloor\hat{\tau}_2\rfloor^{\theta}_{\overline{\beta}} \\ \lfloor{}'K\rfloor^{\theta}_{\overline{\beta}} &:= {}'K \\ \lfloor T\rfloor^{\theta}_{\overline{\beta}} &:= T \\ \lfloor\hat{\tau} :: \kappa\rfloor^{\theta}_{\overline{\beta}} &:= \lfloor\hat{\tau}\rfloor^{\theta}_{\overline{\beta}} :: \kappa \\ \lfloor F(\overline{\hat{\tau}})\rfloor^{\theta}_{\overline{\beta}} &:= F(\overline{\lfloor\hat{\tau}\rfloor^{\theta}_{\overline{\beta}}}) \\ \lfloor\lambda\alpha\mapsto\hat{\tau}\rfloor^{\theta}_{\overline{\beta}} &:= \mathsf{Sym}_n(\mathsf{Lambda})\,\overline{\beta}^n \\ &\Rightarrow \mathbf{type}\,\mathsf{Lambda}\,\overline{\beta}\,\alpha = \lfloor\hat{\tau}\rfloor^{\theta}_{\overline{\beta},\alpha} \\ \lfloor\mathbf{case}\,\hat{\tau}_0\,\mathbf{of}\,\overline{\tau\mapsto\hat{\tau}'}\rfloor^{\theta}_{\overline{\beta}} &:= \mathsf{Case}(\overline{\beta},\lfloor\hat{\tau}_0\rfloor^{\theta}_{\overline{\beta}}) \\ &\Rightarrow \mathbf{type\,family}\,\mathsf{Case}\,\overline{\beta}\,\alpha\,\mathbf{where}\,\lceil\overline{\tau\mapsto\hat{\tau}'}\rceil^{\theta}_{\overline{\beta}} \\ &\ \text{where } \alpha \text{ is fresh} \\ \lfloor\mathbf{let}\,\overline{\omega}\,\mathbf{in}\,\hat{\tau}\rfloor^{\theta}_{\overline{\beta}} &:= \lfloor\hat{\tau}\rfloor^{\theta,\theta'}_{\overline{\beta}} \\ &\Rightarrow \forall i, \lceil\omega_i\rceil^{\theta,\theta'}_{\overline{\beta}} \\ &\ \text{where } \theta' = \lfloor\overline{\omega}\rfloor^{\theta}_{\overline{\beta}} \end{aligned}$$

Lifting of type-level **case** match to type family equation:

$$\lceil\tau\mapsto\hat{\tau}'\rceil^{\theta}_{\overline{\beta}} := \mathsf{Case}(\overline{\beta},\tau)\mapsto\lfloor\hat{\tau}'\rfloor^{\theta}_{\overline{\beta},ftv(\tau)}$$

Reduction of type-level **let** decl. to subst. ($\lfloor\omega\rfloor^{\theta}_{\overline{\beta}} = \theta'$):

$$\begin{aligned} \lfloor(\alpha :: \psi)\{\overline{\overline{\tau}\mapsto\hat{\tau}'}\}\rfloor^{\theta}_{\overline{\beta}} &:= \alpha\mapsto\mathsf{Sym}_n(\mathsf{Let}(\alpha))\,\overline{\beta}^n \\ \lfloor\alpha\{\overline{\overline{\tau}\mapsto\hat{\tau}'}\}\rfloor^{\theta}_{\overline{\beta}} &:= \alpha\mapsto\mathsf{Sym}_n(\mathsf{Let}(\alpha))\,\overline{\beta}^n \end{aligned}$$

Lifting of type-level **let** declaration to top-level declaration:

$$\begin{aligned} &\lceil(\alpha :: \forall\overline{\mathcal{X}}.\overline{\kappa}\twoheadrightarrow\kappa_0)\{\overline{\overline{\tau}\mapsto\hat{\tau}'}\}\rceil^{\theta}_{\overline{\beta}} := \\ &\quad \mathbf{type\,family}\,\mathsf{Let}(\alpha)\,\overline{\beta}\,\overline{(\beta'_i :: \kappa_i)}^n\,\mathbf{where}\,{}_{\alpha}\lceil\overline{\overline{\tau}\mapsto\hat{\tau}'}\rceil^{\theta}_{\overline{\beta}} \\ &\quad \text{where } n = \mathsf{length}\,\overline{\tau} \text{ and the } \overline{\beta}' \text{ are fresh} \\ &\lceil\alpha\{\overline{\overline{\tau}\mapsto\hat{\tau}'}\}\rceil^{\theta}_{\overline{\beta}} := \\ &\quad \mathbf{type\,family}\,\mathsf{Let}(\alpha)\,\overline{\beta}\,\overline{\beta}'^n\,\mathbf{where}\,{}_{\alpha}\lceil\overline{\overline{\tau}\mapsto\hat{\tau}'}\rceil^{\theta}_{\overline{\beta}} \\ &\quad \text{where } n = \mathsf{length}\,\overline{\tau} \text{ and the } \overline{\beta}' \text{ are fresh} \end{aligned}$$

Lifting of type-level clauses to type family equations:

$${}_{\alpha}\lceil\overline{\tau}\mapsto\hat{\tau}'\rceil^{\theta}_{\overline{\beta}} := \mathsf{Let}(\alpha)(\overline{\beta},\overline{\tau})\mapsto\lfloor\hat{\tau}'\rfloor^{\theta}_{\overline{\beta},ftv(\overline{\tau})}$$

Figure 4. Reduction algorithm from extended types to regular types. Both operations (reduction and lifting) distribute over lists.

```
    cons (cons xs)
  type family Stutter (xs :: '[a]) :: '[a] where
    Stutter (x ': xs) = LetConsSym2 x xs @@
                          (LetConsSym2 x xs @@ xs)
  type family LetCons x xs ys where
    LetCons x xs ys = (:$) @@ x @@ ys
```

When reducing the body of the **let** (*cons* (*cons xs*)), the type variables $\overline{\beta}$ are *x* and *xs*. This is how these variables are passed into *LetConsSym2*. The substitution θ is $cons\Uparrow \mapsto$ *LetConsSym2 x xs*. Thus, when the reduction algorithm sees $cons\Uparrow$, it knows what to replace it with.

$$\begin{aligned} (\tau_1\rightarrow\tau_2)[\overline{\kappa}/\overline{\alpha}] &:= \tau_1[\overline{\kappa}/\overline{\alpha}]\rightarrow\tau_2[\overline{\kappa}/\overline{\alpha}] \\ (T\,\overline{\tau})[\overline{\kappa}/\overline{\alpha}] &:= {}'T\,\overline{\tau[\overline{\kappa}/\overline{\alpha}]} \\ \alpha_i[\overline{\kappa}/\overline{\alpha}] &:= \kappa_i \end{aligned}$$

Figure 5. Promotion of datatypes in GHC: $\tau[\overline{\kappa}/\overline{\alpha}]$ (adapted from Figure 9 of Yorgey et al. (2012))

We can consider the top-level to be one big **let** expression. Accordingly, there is always a substitution θ during reduction; outside of any local **let**, it is essentially the "make uppercase identifier" function. These substitutions are built by reducing the list of type-level **let**-declarations, as shown in Figure 4.

The figure also contains definitions of the *lifting* operations $\lceil\cdot\rceil^{\theta}_{\overline{\beta}}$, which are used in producing top-level declarations to implement the extra features present in extended types. Other than the clutter of dealing with lots of syntax, there is nothing terribly unexpected here.

6.4 Type-correctness

We define typing judgements for expressions, $\Gamma \vdash e : \tau$, and types, $\Gamma \vdash \tau : \kappa$, based on how users expect Haskell to work. We are unaware of a simple formulation of surface Haskell's type system and so have written this ourselves. There is other work in this area (Faxén 2002; Jones 1999), but the nature of the existing formulations makes them hard to adapt for our purposes. Note that the typing judgements presented here are different than that in work on GHC's core language FC (for example, Sulzmann et al. (2007)), because we are working solely in surface Haskell. The typing rules appear in Appendix A.

Promotion We prove type safety by proving the safety of promotion $\Uparrow$, with respect to typing judgements for extended types $\hat{\Gamma} \vdash_{\mathsf{ext}} \hat{\tau} : \kappa$, in Appendix B. These rules combine the normal typing rules with new rules for the extra type-level forms that closely resemble their term-level equivalents.

We first prove that defunctionalization symbols work as desired:

Lemma (Data constructor symbols)**.** *If $K : \sigma$ and if $\sigma\Uparrow$ exists, then $\mathsf{Sym}_0(K) : \sigma\Uparrow$.*

The proof of this lemma depends on the relationship between our promotion algorithm and GHC's internal promotion algorithm. GHC's version, in Figure 5, is written as a substitution of kinds in for the type variables in a type, as every free type variable must become a kind during GHC's promotion.

We use this fact to prove the following:

Lemma (Promotion to extended types is well-typed)**.** *Let $\Gamma\Uparrow$, $e\Uparrow$, and $\tau\Uparrow$ exist. If $\Gamma \vdash e : \tau$, then $\Gamma\Uparrow \vdash_{\mathsf{ext}} e\Uparrow : \tau\Uparrow$.*

Reduction Having shown that promotion to extended types works, we must now prove that reduction also is well typed. However, reduction depends more critically on the contexts where it is performed. Thus, we introduce the idea of top-level contexts, which simplifies the statements of the lemmas:

Definition (Top-level contexts and substitutions)**.** *Let $\overline{\delta}$ be a set of declarations such that $\varnothing \vdash \overline{\delta} \rightsquigarrow \Gamma_0$ and $\theta_0 = \lfloor\overline{\delta}\Uparrow\rfloor^{\varnothing}_{\varnothing}$. Then, Γ_0 is a top-level context, and θ_0 is the associated top-level substitution.*

This definition uses the judgement $\varnothing \vdash \overline{\delta} \rightsquigarrow \Gamma_0$, which says that the declarations $\overline{\delta}$ are well-typed in an empty context and induce a typing context Γ_0 when the declarations are in scope. The intent is that $\overline{\delta}$ are top-level declarations. The θ_0 mentioned works out in practice to be the "make uppercase identifier" function described above.

Lemma (Type reduction preserves kinds)**.** *Let* Γ_0 *be a top-level context and* θ_0 *its associated substitution. If* $\Gamma_0\Uparrow \vdash_{ext} \hat{\tau} : \kappa$*, then* $\lfloor \Gamma_0\Uparrow \rfloor \vdash \lfloor \hat{\tau} \rfloor^{\theta_0}_{\varnothing} : \kappa$ *and the emitted type declarations are valid.*

Full type-correctness Putting these together yields the following:

Theorem (Promotion is well-typed)**.** *Let* Γ_0 *and* θ_0 *be a top-level context and its associated substitution. If* $\Gamma_0 \vdash e : \tau$*, where* $e\Uparrow$ *and* $\tau\Uparrow$ *exist, then* $\varnothing \vdash \lfloor e\Uparrow \rfloor^{\theta_0}_{\varnothing} : \tau\Uparrow$.

6.5 Semantics

We have shown that promoting a well-typed expression yields a well-kinded type. We must also show that this well-kinded type behaves the same as the original expression. To do so, we define a small-step operational semantics both for expressions and for types. We are unfamiliar with previous work on developing an operational semantics for Haskell. The expression semantics relation, $\Sigma; e \longrightarrow \Sigma'; e'$, is based on an understanding of how Haskell expressions reduce.[12] The step relation tracks an environment Σ, which is just a set of **let**-bound variables for use in lookup. The type-level semantics, $\tau \longrightarrow \tau'$, is a congruence over type family reduction, as type family reduction is the only way that a type "steps."

Conjecture (Promotion preserves semantics for closed terms)**.** *Let* Γ_0 *be a top-level context and* θ_0 *its associated substitution, where* $\Sigma_0 = \overline{\delta}_0$ *are the top-level declarations. If* $\Gamma_0 \vdash e : \tau$, $\Sigma_0; e \longrightarrow^* \Sigma'; e'$*, both* $e\Uparrow$ *and* $\tau\Uparrow$ *exist, and* e' *consists only of data constructors and applications, then* $e'\Uparrow$ *exists and* $\lfloor e\Uparrow \rfloor^{\theta_0}_{\varnothing} \longrightarrow^* \lfloor e'\Uparrow \rfloor^{\varnothing}_{\varnothing}$.

The intuition behind the above conjecture is that an expression well-typed in a top-level context that eventually reduces to an observable value (that is, applied data constructors) promotes to a type that reduces to the promoted form of the value.

Alas, we are unable to prove this conjecture in full because of reduction's dependence on lambda lifting. Proving lambda lifting correct is a large enterprise of itself, and is beyond the scope of this paper. We refer the reader to the work of Fischbach and Hannan (2003), which states a lambda lifting algorithm and proves it correct, at length.

Instead of proving the conjecture above, we settle for proving that an extension of the type-level semantics, $\overline{\omega}; \hat{\tau} \longrightarrow_{ext} \overline{\omega}'; \hat{\tau}'$, supporting extended types, agrees with our term-level semantics:

Theorem (Promotion to extended types preserves semantics)**.** *If* $\Sigma; e \longrightarrow \Sigma'; e'$ *and if* $e\Uparrow$ *exists, then* $\Sigma\Uparrow; e\Uparrow \longrightarrow_{ext} \Sigma'\Uparrow; e'\Uparrow$.

Note that Σ is just a collection of **let**-declarations δ, and can be promoted by the relevant algorithm in Figure 3.

7. Discussion

7.1 Type inference

In Section 4.3, we claim that an unsaturated type family interferes with type inference. The problem stems from the fact that GHC assumes both injectivity and generativity of type application. By injectivity, we mean that if GHC can derive $(a\ b) \sim (a\ c)$, then it can conclude $b \sim c$. Generativity means that if GHC can derive $(a\ b) \sim (c\ d)$, then it can conclude $a \sim c$. In other words, a generative type application creates something new, unequal to anything created with other types.

Type family application is neither injective nor generative. Thus, GHC must ensure that an unapplied type family can never be abstracted over – that is, no type variable can ever be instantiated to a partially-applied type family. If we did perform such an instantiation, GHC's injectivity and generativity assumptions would be invalid, and type inference may arrive at a wrong conclusion.

In this paper, we show a way essentially to manipulate partially-applied type functions. How does this fit with the story above? Critically, the application of a type function in this paper is done explicitly, with the `@@` operator. Thus, a programmer can use unsaturated type functions by explicitly choosing what assumptions hold at each type application. When we say $a\ b$ (normal type application), that application is injective and generative, as usual. If, however, we say a `@@` b, then the application is not necessarily either injective or generative.

This dichotomy works well with GHC's treatment of type family arguments. Recall that `@@` is implemented as an ordinary open type family. Thus, GHC will not break it apart or use the injectivity and generativity assumptions on applications built with `@@`. Happily, this is exactly the behaviour that we want.

The fact that we introduce a new arrow $\twoheadrightarrow$ fits nicely with this, as well. The regular arrow $\rightarrow$, when classifying types, indicates an injective, generative function. Our new arrow $\twoheadrightarrow$ denotes a function without these assumptions. When $\rightarrow$ is used to classify terms, we make no assumptions about the functions involved. It is thus natural to promote the type $\rightarrow$ to the kind $\twoheadrightarrow$, *not* to the kind $\rightarrow$.

7.2 Eliminating symbols

We can go further and argue that GHC's current choice to use juxtaposition for type family application is a design error. The identical appearance of normal application and type family application hides the fact that these are treated differently by GHC. For example, consider these type signatures:

$ex_1 :: Maybe\ a \rightarrow Bool$
$ex_2 :: Foogle\ a \rightarrow Bool$

We know that ex_1's type is unambiguous – that is, we can infer the type a if we know $Maybe\ a$. But, what about ex_2? To know whether the type is ambiguous, we must know how $Foogle$ is defined. Is it a type family, or a type constructor? The answer to that question directly informs ex_2's level of ambiguity. A library author might want to change the nature of $Foogle$ from a type constructor to a type family; now, that change impacts users.

On the other hand, if *all* type families had to be applied explicitly in user code, the difference would be manifest:

$ex_2 :: Foogle$ `@@` $a \rightarrow Bool$

Now, programmers can easily see that ex_2's type is ambiguous and ponder how to fix it.

In the bold new world where type family application is explicit, the appearance of a type family in a program would mean essentially what we mean by a 0-symbol. We can also imagine that GHC could allow `@@` to be used with proper type constructors, as $\rightarrow$ could be considered a sub-type of $\twoheadrightarrow$.

7.3 Semantic differences between terms and types

Terms are evaluated on a by-need basis. How does this translate to types? Type evaluation is non-deterministic and operates differently than term-level evaluation. Indeed, type-level "evaluation" is implemented within GHC by constraint solving: GHC translates a type such as $Vec\ a\ (Pred\ n)$ to $(Pred\ n \sim m) \Rightarrow Vec\ a\ m$ for a fresh m. See Vytiniotis et al. (2011) for details.

Despite this significant difference, we have yet to see any problems play out in our work (neglecting the impossibility of infinite types). It is possible to define type families with non-linear equations (i.e., left-hand sides with a repeated variable) and to define type families over the kind $\star$. Both of these have semantics different than anything seen at the term level. For example, note the somewhat unintuitive rules for simplifying closed type families de-

[12] No attempt is made at modeling Haskell's call-by-need semantics; we settle for call-by-name.

scribed by Eisenberg et al. (2014). However, it seems that by restricting the form of type families to look like promoted term-level functions, we sidestep these problems nicely.

7.4 Features beyond Haskell 2010

We have restricted the scope of our work to include only features mentioned in Chapters 3 and 4 of the Haskell 2010 Report. However, we ourselves enjoy using the many features that GHC supports which fall outside this subset. Many of these features are not possible to promote. Without first-class kind polymorphism (such as higher-rank kinds), we cannot promote higher-rank types. Without kind-level equality, we cannot promote equality constraints, GADTs, or type families; see Weirich et al. (2013) for some theoretical work toward lifting this restriction. Overlapping and incoherent class instances would lead to overlapping open type family equations; these are surely not promotable. Intriguingly, GHC *does* allow functional dependencies among kind variables, so these promote without a problem. We leave it open to future study to determine which other extensions of GHC are promotable.

7.5 Future work

The most tempting direction of future work is to implement a promotion algorithm in GHC directly. With support for partial application in types along the lines of what we propose in Section 7.2, this could be done with much less clutter than we see in this paper. A non-trivial problem in this work is that of namespaces: how can we remain backward compatible while allowing some terms to be used in types? Dealing with naming issues was a recurrent and annoying problem in our work. An important advantage of direct implementation within GHC is that the algorithm would work in a fully typed setting. Instead of generating unusable definitions – as demonstrated in Section 5 – the algorithm could detect errors and report them to the programmer. It would also be possible to correctly promote functions stored inside datatypes.

We would also want a more complete treatment of promoted literals within GHC. The current mismatch between term-level integers and type-level *Nat*s is inconvenient and can prevent promotion of term-level functions to the type level. Similarly, the kind *Symbol* and the type *String* behave too differently to make promotion of *String* functions possible.

With these improvements in place, we would be even closer to enabling dependently typed programming in Haskell, along the lines of the work by Gundry (2013). That work takes care in identifying a subset of Haskell that can be shared between the term level and type level. This subset notably leaves out anonymous and partially-applied functions. The work done here shows that these forms, too, can be included in types and will enable an even more expressive dependently typed Haskell.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. 1116620.

References

M. M. T. Chakravarty, G. Keller, and S. Peyton Jones. Associated type synonyms. In *ACM SIGPLAN International Conference on Functional Programming*, 2005a.

M. M. T. Chakravarty, G. Keller, S. Peyton Jones, and S. Marlow. Associated types with class. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2005b.

J. Cheney and R. Hinze. First-class phantom types. Technical report, Cornell University, 2003.

R. A. Eisenberg and J. Stolarek. Promoting functions to type families in Haskell. In *ACM SIGPLAN Haskell Symposium*, 2014.

R. A. Eisenberg and S. Weirich. Dependently typed programming with singletons. In *ACM SIGPLAN Haskell Symposium*, 2012.

R. A. Eisenberg, D. Vytiniotis, S. Peyton Jones, and S. Weirich. Closed type families with overlapping equations. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2014.

K.-F. Faxén. A static semantics for Haskell. *Journal of Functional Programming*, 12(4-5), July 2002.

A. Fischbach and J. Hannan. Specification and correctness of lambda lifting. *Journal of Functional Programming*, 13(3), May 2003.

A. Gundry. *Type Inference, Haskell and Dependent Types*. PhD thesis, University of Strathclyde, 2013.

T. Johnsson. Lambda lifting: Transforming programs to recursive equations. In *Conference on Functional Programming Languages and Computer Architecture*, 1985.

M. P. Jones. Typing Haskell in Haskell. In *Haskell Workshop*, 1999.

M. P. Jones. Type classes with functional dependencies. In *European Symposium on Programming*, 2000.

A. Kennedy. *Programming Languages and Dimensions*. PhD thesis, University of Cambridge, 1996.

S. Lindley and C. McBride. Hasochism: the pleasure and pain of dependently typed Haskell programming. In *ACM SIGPLAN Haskell Symposium*, 2013.

S. Marlow. Haskell 2010 Language Report. Technical report, 2010.

T. Muranushi and R. A. Eisenberg. Experience report: Type-checking polymorphic units for astrophysics research in Haskell. In *ACM SIGPLAN Haskell Symposium*, 2014.

S. Peyton Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple unification-based type inference for GADTs. In *ACM SIGPLAN International Conference on Functional Programming*, 2006.

S. Peyton Jones, D. Vytiniotis, S. Weirich, and M. Shields. Practical type inference for arbitrary-rank types. *Journal of Functional Programming*, 17(1), Jan. 2007.

J. Reynolds. Definitional interpreters for higher-order programming languages. In *ACM Annual Conference*, 1972.

P. Sewell, F. Zappa Nardelli, S. Owens, G. Peskine, T. Ridge, S. Sarkar, and R. Strniša. Ott: Effective tool support for the working semanticist. *Journal of Functional Programming*, 20(1), Jan. 2010.

T. Sheard and S. Peyton Jones. Template metaprogramming for Haskell. *ACM SIGPLAN Notices*, 37(12), Dec. 2002.

M. Sulzmann, M. M. T. Chakravarty, S. Peyton Jones, and K. Donnelly. System F with type equality coercions. In *ACM SIGPLAN Workshop on Types in Languages Design and Implementation*, 2007.

D. Vytiniotis, S. Peyton Jones, T. Schrijvers, and M. Sulzmann. OutsideIn(X) modular type inference with local assumptions. *Journal of Functional Programming*, 21(4-5), Sept. 2011.

P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1989.

S. Weirich, J. Hsu, and R. A. Eisenberg. System FC with explicit kind equality. In *ACM SIGPLAN International Conference on Functional Programming*, 2013.

B. Yorgey, S. Weirich, J. Cretin, S. Peyton Jones, D. Vytiniotis, and J. P. Magalhães. Giving Haskell a promotion. In *ACM SIGPLAN Workshop on Types in Language Design and Implementation*, 2012.

A. Haskell formalization

A.1 Remaining element of the grammar

$\Phi ::= \forall \overline{\mathcal{X}}.\{\overline{\kappa}\}\kappa_0$ Type family kinds

A.2 Haskell types, kinds, and contexts

$\boxed{\Gamma \vdash \tau : \kappa}$ Type kinding

$$\frac{\alpha{:}\kappa \in \Gamma \qquad \vdash \Gamma}{\Gamma \vdash \alpha : \kappa}\ \textsc{Ty\_Var}$$

$$\frac{\vdash \Gamma}{\Gamma \vdash (\to) : \star \to \star \to \star}\ \textsc{Ty\_Arrow}$$

$$\frac{\Gamma \vdash \tau_1 : \kappa_1 \to \kappa_2 \quad \Gamma \vdash \tau_2 : \kappa_1 \qquad \mathit{fkv}(\kappa_1) \subseteq \mathit{fkv}(\kappa_2)}{\Gamma \vdash \tau_1\, \tau_2 : \kappa_2}\ \textsc{Ty\_App}$$

$$\frac{K : \forall \overline{\alpha}.\tau \quad \Gamma \vdash_{\mathsf{sch}} \forall \overline{\mathcal{X}}.(\tau[\overline{\mathcal{X}}/\overline{\alpha}])\ \mathbf{kind\ scheme} \quad \overline{\Gamma \vdash \kappa_i\ \mathbf{kind}}}{\Gamma \vdash {}'K : \tau[\overline{\kappa}/\overline{\alpha}]}\ \textsc{Ty\_DataCon}$$

$$\frac{T : \forall \overline{\mathcal{X}}.\kappa_0 \qquad \vdash \Gamma \quad \overline{\Gamma \vdash \kappa_i\ \mathbf{kind}}}{\Gamma \vdash T : \kappa_0[\overline{\kappa}/\overline{\mathcal{X}}]}\ \textsc{Ty\_TyCon}$$

$$\frac{\Gamma \vdash \tau : \kappa}{\Gamma \vdash \tau :: \kappa : \kappa}\ \textsc{Ty\_Annot}$$

$$\frac{F : \forall \overline{\mathcal{X}}.\{\overline{\kappa}\}\kappa_0 \qquad \vdash \Gamma \quad \overline{\Gamma \vdash \kappa'_j\ \mathbf{kind}} \quad \overline{\Gamma \vdash \tau_i : \kappa_i[\overline{\kappa'}/\overline{\mathcal{X}}]}}{\Gamma \vdash F(\overline{\tau}) : \kappa_0[\overline{\kappa'}/\overline{\mathcal{X}}]}\ \textsc{Ty\_Fam}$$

$\boxed{\Gamma \vdash_{\mathsf{sch}} \sigma : \kappa}$ Type scheme kinding

$$\frac{\Gamma, \alpha{:}\star \vdash_{\mathsf{sch}} \sigma : \star}{\Gamma \vdash_{\mathsf{sch}} \forall \alpha.\sigma : \star}\ \textsc{TySch\_Poly}$$

$$\frac{\Gamma \vdash \tau : \kappa}{\Gamma \vdash_{\mathsf{sch}} \tau : \kappa}\ \textsc{TySch\_Mono}$$

$\boxed{\Gamma \vdash \kappa\ \mathbf{kind}}$ Kind validity

$$\frac{\mathcal{X} \in \Gamma \qquad \vdash \Gamma}{\Gamma \vdash \mathcal{X}\ \mathbf{kind}}\ \textsc{Ki\_Var}$$

$$\frac{T : \overline{\star} \to \star \qquad \vdash \Gamma \quad \overline{\Gamma \vdash \kappa_i\ \mathbf{kind}}}{\Gamma \vdash {}'T\ \overline{\kappa}\ \mathbf{kind}}\ \textsc{Ki\_TyCon}$$

$$\frac{\Gamma \vdash \kappa_1\ \mathbf{kind} \qquad \Gamma \vdash \kappa_2\ \mathbf{kind}}{\Gamma \vdash \kappa_1 \to \kappa_2\ \mathbf{kind}}\ \textsc{Ki\_Arrow}$$

$$\frac{\vdash \Gamma}{\Gamma \vdash \star\ \mathbf{kind}}\ \textsc{Ki\_Star}$$

$\boxed{\Gamma \vdash_{\mathsf{sch}} \psi\ \mathbf{kind\ scheme}}$ Kind scheme validity

$$\frac{\Gamma, \mathcal{X} \vdash_{\mathsf{sch}} \psi\ \mathbf{kind\ scheme}}{\Gamma \vdash_{\mathsf{sch}} \forall \mathcal{X}.\psi\ \mathbf{kind\ scheme}}\ \textsc{KiSch\_Poly}$$

$$\frac{\Gamma \vdash \kappa\ \mathbf{kind}}{\Gamma \vdash_{\mathsf{sch}} \kappa\ \mathbf{kind\ scheme}}\ \textsc{KiSch\_Mono}$$

$\boxed{\vdash \Gamma}$ Context validity

$$\frac{}{\vdash \varnothing}\ \textsc{Ctx\_Nil}$$

$$\frac{\Gamma \vdash_{\mathsf{sch}} \sigma : \star \qquad x \mathbin{\#} \Gamma}{\vdash \Gamma, x{:}\sigma}\ \textsc{Ctx\_Term}$$

$$\frac{\Gamma \vdash \kappa\ \mathbf{kind} \qquad \alpha \mathbin{\#} \Gamma}{\vdash \Gamma, \alpha{:}\kappa}\ \textsc{Ctx\_Type}$$

$$\frac{\vdash \Gamma \qquad \mathcal{X} \mathbin{\#} \Gamma}{\vdash \Gamma, \mathcal{X}}\ \textsc{Ctx\_Kind}$$

Note the restriction on kind variables in Ty_App. This is perhaps unusual, but harmless. It is necessary in order to say that all kind variables used in a type appear in the type's kind – there are no "hidden" kind variables. This property is, in turn, needed for technical reasons in Lemma 35. The restriction actually matches up with GHC's implementation. When a kind variable would become "hidden" like this, GHC substitutes in *AnyK*, a dummy kind, for the variable.

A.3 Haskell expressions

$\boxed{\Gamma \vdash e : \tau}$ Term typing

$$\frac{x{:}\forall \overline{\alpha}.\tau_0 \in \Gamma \qquad \vdash \Gamma \quad \overline{\Gamma \vdash \tau_i : \star}}{\Gamma \vdash x : \tau_0[\overline{\tau}/\overline{\alpha}]}\ \textsc{Tm\_Var}$$

$$\frac{K : \forall \overline{\alpha}.\tau_0 \qquad \vdash \Gamma \quad \overline{\Gamma \vdash \tau_i : \star}}{\Gamma \vdash K : \tau_0[\overline{\tau}/\overline{\alpha}]}\ \textsc{Tm\_DataCon}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\, e_2 : \tau_2}\ \textsc{Tm\_App}$$

$$\frac{\Gamma, x{:}\tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x \mapsto e : \tau_1 \to \tau_2}\ \textsc{Tm\_Abs}$$

$$\frac{\Gamma \vdash e_0 : \tau_0 \quad \text{for each } \pi_i \mapsto e_i \in \overline{\pi \mapsto e} : \ \vdash \pi_i : \tau_0 \rightsquigarrow \Gamma'_i \quad \Gamma, \Gamma'_i \vdash e_i : \tau}{\Gamma \vdash \mathbf{case}\ e_0\ \mathbf{of}\ \overline{\pi \mapsto e} : \tau}\ \textsc{Tm\_Case}$$

$$\frac{\Gamma \vdash \overline{\delta} \rightsquigarrow \Gamma' \quad \Gamma, \Gamma' \vdash e : \tau}{\Gamma \vdash \mathbf{let}\ \overline{\delta}\ \mathbf{in}\ e : \tau}\ \textsc{Tm\_Let}$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash e :: \tau : \tau}\ \textsc{Tm\_Annot}$$

$\boxed{\vdash \pi : \tau \rightsquigarrow \Gamma}$ Pattern typing

$$\frac{}{\vdash x : \tau \rightsquigarrow x{:}\tau}\ \textsc{Pat\_Var}$$

$$\frac{K : \forall \overline{\alpha}.(\overline{\tau} \to T\ \overline{\alpha}) \quad \overline{\vdash \pi_i : \tau_i[\overline{\tau'}/\overline{\alpha}] \rightsquigarrow \Gamma_i}}{\vdash K\ \overline{\pi} : T\ \overline{\tau'} \rightsquigarrow \overline{\Gamma}}\ \textsc{Pat\_DataCon}$$

$$\frac{}{\vdash \_ : \tau \rightsquigarrow \varnothing}\ \textsc{Pat\_Wild}$$

$\boxed{\Gamma \vdash \overline{\delta} \rightsquigarrow \Gamma'}$ Let declaration group context formation

$$\frac{\vdash \Gamma, \overline{\Gamma'} \quad \text{for each } \delta_i \in \overline{\delta} : \ \Gamma, \overline{\Gamma'} \vdash \delta_i \rightsquigarrow \Gamma'_i}{\Gamma \vdash \overline{\delta} \rightsquigarrow \overline{\Gamma'}}\ \textsc{LetS\_Dec}$$

$\boxed{\Gamma \vdash \delta \rightsquigarrow \Gamma'}$ Let declaration validity

$$\frac{\text{for each } \overline{\pi}_i \mapsto e_i \in \overline{\overline{\pi} \mapsto e} : \ \Gamma \vdash \overline{\pi}_i \mapsto e_i : \sigma}{\Gamma \vdash (x :: \sigma)\{\overline{\overline{\pi} \mapsto e}\} \rightsquigarrow x{:}\sigma}\ \textsc{Let\_Annot}$$

$$\frac{\text{for each } \overline{\pi}_i \mapsto e_i \in \overline{\overline{\pi} \mapsto e} : \ \Gamma \vdash \overline{\pi}_i \mapsto e_i : \sigma}{\Gamma \vdash x\{\overline{\overline{\pi} \mapsto e}\} \rightsquigarrow x{:}\sigma}\ \textsc{Let\_NoAnnot}$$

$\boxed{\Gamma \vdash \mathit{clause} : \sigma}$ Function cluase typing

$$\frac{\begin{array}{c}\overline{\vdash \pi_i : \tau_i \leadsto \Gamma'_i} \\ \overline{\Gamma \vdash \tau_i : \star} \\ \Gamma, \overline{\alpha{:}\star}, \overline{\Gamma'} \vdash e : \tau_0\end{array}}{\Gamma \vdash \overline{\pi} \mapsto e : \forall \overline{\alpha}. \overline{\tau} \to \tau_0} \quad \textsc{Cl\_Clause}$$

A.4 Top-level declaration validity

$\boxed{\vdash \tau : \kappa \leadsto \Gamma}$ Type pattern context formation

$$\frac{}{\vdash \alpha : \kappa \leadsto \alpha{:}\kappa} \quad \textsc{TyPat\_Var}$$

$$\frac{}{\vdash (\to) : \star \to \star \to \star \leadsto \varnothing} \quad \textsc{TyPat\_Arrow}$$

$$\frac{\begin{array}{c}\vdash \tau_1 : \kappa_1 \to \kappa_2 \leadsto \Gamma_1 \\ \vdash \tau_2 : \kappa_1 \leadsto \Gamma_2\end{array}}{\vdash \tau_1\,\tau_2 : \kappa_2 \leadsto \Gamma_1 \cup \Gamma_2} \quad \textsc{TyPat\_App}$$

$$\frac{K : \forall \overline{\alpha}.\tau}{\vdash {}'K : \tau[\overline{\kappa}/\overline{\alpha}] \leadsto \varnothing} \quad \textsc{TyPat\_DataCon}$$

$$\frac{T : \forall \overline{\mathcal{X}}.\kappa_0}{\vdash T : \kappa_0[\overline{\kappa}/\overline{\mathcal{X}}] \leadsto \varnothing} \quad \textsc{TyPat\_TyCon}$$

$$\frac{\begin{array}{c}\vdash \tau : \kappa \leadsto \Gamma_1 \\ \vdash \kappa \leadsto \Gamma_2\end{array}}{\vdash \tau :: \kappa : \kappa \leadsto \Gamma_1, \Gamma_2} \quad \textsc{TyPat\_Annot}$$

$\boxed{\vdash \kappa \leadsto \Gamma}$ Kind pattern context formation

$$\frac{}{\vdash \mathcal{X} \leadsto \mathcal{X}} \quad \textsc{KiPat\_Var}$$

$$\frac{\begin{array}{c}\text{for each } \kappa_i \in \overline{\kappa} : \\ \vdash \kappa_i \leadsto \Gamma_i\end{array}}{\vdash {}'T\,\overline{\kappa} \leadsto \bigcup \Gamma_i} \quad \textsc{KiPat\_TyCon}$$

$$\frac{\begin{array}{c}\vdash \kappa_1 \leadsto \Gamma_1 \\ \vdash \kappa_2 \leadsto \Gamma_2\end{array}}{\vdash \kappa_1 \to \kappa_2 \leadsto \Gamma_1 \cup \Gamma_2} \quad \textsc{KiPat\_Arrow}$$

$$\frac{}{\vdash \star \leadsto \varnothing} \quad \textsc{KiPat\_Star}$$

$\boxed{\vdash \mathit{dec} \leadsto F : \Phi}$ Top-level declaration validity

$$\frac{\begin{array}{c}\vdash \overline{\mathit{tvb}} \leadsto \overline{\kappa}; \Gamma \\ \overline{\mathcal{X}} = \mathit{fkv}(\overline{\kappa}, \kappa_0) \\ \overline{\mathcal{X}}, \Gamma \vdash \tau : \kappa_0\end{array}}{\vdash \mathbf{type}\ F\ \overline{\mathit{tvb}} = \tau \leadsto F : \forall \overline{\mathcal{X}}.\{\overline{\kappa}\}\kappa_0} \quad \textsc{Dec\_TySyn}$$

$$\frac{\begin{array}{c}\vdash \overline{\mathit{tvb}} \leadsto \overline{\kappa}; \Gamma \\ \overline{\mathcal{X}} = \mathit{fkv}(\overline{\kappa}, \kappa_0) \\ \text{for each } \tau_i \mapsto \tau'_i \in \overline{\overline{\tau} \mapsto \tau'} : \\ F \ni \tau_i \mapsto \tau'_i : \forall \overline{\mathcal{X}}.\{\overline{\kappa}\}\kappa_0\end{array}}{\vdash \mathbf{type\ family}\ F\ \overline{\mathit{tvb}}\ \mathbf{where}\ \overline{\overline{\tau} \mapsto \tau'} \leadsto F : \forall \overline{\mathcal{X}}.\{\overline{\kappa}\}\kappa_0} \quad \textsc{Dec\_TyFam}$$

$\boxed{\vdash \overline{\mathit{tvb}} \leadsto \overline{\kappa}; \Gamma}$ Type variable binder context formation

$$\frac{}{\vdash \varnothing \leadsto \varnothing; \varnothing} \quad \textsc{Tvbs\_Nil}$$

$$\frac{\vdash \overline{\mathit{tvb}} \leadsto \overline{\kappa}; \Gamma}{\vdash \overline{\mathit{tvb}}\ \alpha \leadsto \overline{\kappa}, \kappa_0; \Gamma, \alpha{:}\kappa_0} \quad \textsc{Tvbs\_Plain}$$

$$\frac{\vdash \overline{\mathit{tvb}} \leadsto \overline{\kappa}; \Gamma}{\vdash \overline{\mathit{tvb}}\ \alpha :: \kappa_0 \leadsto \overline{\kappa}, \kappa_0; \Gamma, \alpha{:}\kappa} \quad \textsc{Tvbs\_Kinded}$$

$\boxed{F \ni \mathit{equn} : \Phi}$ Type family equation kinding

$$\frac{\begin{array}{c}\overline{\Gamma \vdash \tau_i : \kappa_i} \\ \Gamma \vdash \tau' : \kappa_0\end{array}}{F \ni F(\overline{\tau}) \mapsto \tau' : \forall\, \mathit{fkv}(\overline{\kappa}, \kappa_0).\{\overline{\kappa}\}\kappa_0} \quad \textsc{TF\_Eqn}$$

B. Extended types formalization

$\boxed{\hat{\Gamma} \vdash_{\mathsf{ext}} \hat{\tau} : \kappa}$ Extended type kinding

$$\frac{\begin{array}{c}\alpha{:}\forall \overline{\mathcal{X}}.\kappa_0 \in \hat{\Gamma} \qquad \vdash_{\mathsf{ext}} \hat{\Gamma} \\ \overline{\hat{\Gamma} \vdash_{\mathsf{ext}} \kappa_i\ \mathbf{kind}}\end{array}}{\hat{\Gamma} \vdash_{\mathsf{ext}} \alpha : \kappa_0[\overline{\kappa}/\overline{\mathcal{X}}]} \quad \textsc{TyX\_Var}$$

$$\frac{\vdash_{\mathsf{ext}} \hat{\Gamma}}{\hat{\Gamma} \vdash_{\mathsf{ext}} (\to) : \star \to \star \to \star} \quad \textsc{TyX\_Arrow}$$

$$\frac{\begin{array}{c}\hat{\Gamma} \vdash_{\mathsf{ext}} \hat{\tau}_1 : \kappa_1 \to \kappa_2 \\ \hat{\Gamma} \vdash_{\mathsf{ext}} \hat{\tau}_2 : \kappa_1 \qquad \mathit{fkv}(\kappa_1) \subseteq \mathit{fkv}(\kappa_2)\end{array}}{\hat{\Gamma} \vdash_{\mathsf{ext}} \hat{\tau}_1\,\hat{\tau}_2 : \kappa_2} \quad \textsc{TyX\_App}$$

$$\frac{\begin{array}{l}K : \forall \overline{\alpha}.\tau \\ \hat{\Gamma} \vdash_{\mathsf{ext\ sch}} \forall \overline{\mathcal{X}}.(\tau[\overline{\mathcal{X}}/\overline{\alpha}])\ \mathbf{kind\ scheme} \\ \overline{\hat{\Gamma} \vdash_{\mathsf{ext}} \kappa_i\ \mathbf{kind}}\end{array}}{\hat{\Gamma} \vdash_{\mathsf{ext}} {}'K : \tau[\overline{\kappa}/\overline{\alpha}]} \quad \textsc{TyX\_DataCon}$$

$$\frac{\begin{array}{c}T : \forall \overline{\mathcal{X}}.\kappa_0 \qquad \vdash_{\mathsf{ext}} \hat{\Gamma} \\ \overline{\hat{\Gamma} \vdash_{\mathsf{ext}} \kappa_i\ \mathbf{kind}}\end{array}}{\hat{\Gamma} \vdash_{\mathsf{ext}} T : \kappa_0[\overline{\kappa}/\overline{\mathcal{X}}]} \quad \textsc{TyX\_TyCon}$$

$$\frac{\hat{\Gamma} \vdash_{\mathsf{ext}} \hat{\tau} : \kappa}{\hat{\Gamma} \vdash_{\mathsf{ext}} \hat{\tau} :: \kappa : \kappa} \quad \textsc{TyX\_Annot}$$

$$\frac{\begin{array}{c}F : \forall \overline{\mathcal{X}}.\{\overline{\kappa}\}\kappa_0 \qquad \vdash_{\mathsf{ext}} \hat{\Gamma} \\ \overline{\hat{\Gamma} \vdash_{\mathsf{ext}} \kappa'_j\ \mathbf{kind}} \\ \overline{\hat{\Gamma} \vdash_{\mathsf{ext}} \hat{\tau}_i : \kappa_i[\overline{\kappa}'/\overline{\mathcal{X}}]}\end{array}}{\hat{\Gamma} \vdash_{\mathsf{ext}} F(\overline{\hat{\tau}}) : \kappa_0[\overline{\kappa}'/\overline{\mathcal{X}}]} \quad \textsc{TyX\_Fam}$$

$$\frac{\hat{\Gamma}, \alpha{:}\kappa_1 \vdash_{\mathsf{ext}} \hat{\tau} : \kappa_2}{\hat{\Gamma} \vdash_{\mathsf{ext}} \lambda\alpha \mapsto \hat{\tau} : \kappa_1 \twoheadrightarrow \kappa_2} \quad \textsc{TyX\_Abs}$$

$$\frac{\begin{array}{l}\hat{\Gamma} \vdash_{\mathsf{ext}} \hat{\tau}_0 : \kappa_0 \\ \text{for each } \tau_i \mapsto \hat{\tau}'_i \in \overline{\tau \mapsto \hat{\tau}'} : \\ \quad \vdash \tau_i : \kappa_0 \leadsto \Gamma'_i \\ \quad \hat{\Gamma}^{\mathsf{ki}}, \Gamma'_i \vdash_{\mathsf{ext}} \tau_i : \kappa_0 \\ \quad \hat{\Gamma}, \Gamma'_i \vdash_{\mathsf{ext}} \hat{\tau}'_i : \kappa\end{array}}{\hat{\Gamma} \vdash_{\mathsf{ext}} \mathbf{case}\ \hat{\tau}_0\ \mathbf{of}\ \overline{\tau \mapsto \hat{\tau}'} : \kappa} \quad \textsc{TyX\_Case}$$

$$\frac{\begin{array}{c}\hat{\Gamma} \vdash_{\mathsf{ext}} \overline{\omega} \leadsto \hat{\Gamma}' \\ \hat{\Gamma}, \hat{\Gamma}' \vdash_{\mathsf{ext}} \hat{\tau} : \kappa\end{array}}{\hat{\Gamma} \vdash_{\mathsf{ext}} \mathbf{let}\ \overline{\omega}\ \mathbf{in}\ \hat{\tau} : \kappa} \quad \textsc{TyX\_Let}$$

$\boxed{\hat{\Gamma} \vdash_{\mathsf{ext}} \overline{\omega} \leadsto \hat{\Gamma}'}$ Extended type-let declaration context formation

$$\frac{\begin{array}{l}\vdash_{\mathsf{ext}} \hat{\Gamma}, \overline{\hat{\Gamma}'} \\ \text{for each } \omega_i \in \overline{\omega} : \\ \quad \hat{\Gamma}, \overline{\hat{\Gamma}'} \vdash_{\mathsf{ext}} \omega_i \leadsto \hat{\Gamma}'_i\end{array}}{\hat{\Gamma} \vdash_{\mathsf{ext}} \overline{\omega} \leadsto \overline{\hat{\Gamma}'}} \quad \textsc{TyLets\_Dec}$$

$\boxed{\hat{\Gamma} \vdash_{\mathsf{ext}} \omega \leadsto \hat{\Gamma}'}$ Extended type-let declaration validity

$$\frac{\begin{array}{c}\text{for each } \overline{\tau}_i \mapsto \hat{\tau}'_i \in \overline{\overline{\tau} \mapsto \hat{\tau}'} : \\ \hat{\Gamma} \vdash_{\mathsf{ext}} \overline{\tau}_i \mapsto \hat{\tau}'_i : \psi\end{array}}{\hat{\Gamma} \vdash_{\mathsf{ext}} (\alpha :: \psi)\{\overline{\overline{\tau} \mapsto \hat{\tau}'}\} \rightsquigarrow \alpha{:}\psi} \quad \textsc{TyLet\_Annot}$$

$$\frac{\begin{array}{c}\text{for each } \overline{\tau}_i \mapsto \hat{\tau}'_i \in \overline{\overline{\tau} \mapsto \hat{\tau}'} : \\ \hat{\Gamma} \vdash_{\mathsf{ext}} \overline{\tau}_i \mapsto \hat{\tau}'_i : \psi\end{array}}{\hat{\Gamma} \vdash_{\mathsf{ext}} \alpha\{\overline{\overline{\tau} \mapsto \hat{\tau}'}\} \rightsquigarrow \alpha{:}\psi} \quad \textsc{TyLet\_NoAnnot}$$

$\boxed{\hat{\Gamma} \vdash_{\mathsf{ext}} tclause : \psi}$ Extended type function clause kinding

$$\frac{\begin{array}{c}\overline{\vdash \tau_i : \kappa_i \rightsquigarrow \Gamma'_i} \\ \overline{\hat{\Gamma}^{\mathsf{ki}}, \Gamma'_i \vdash_{\mathsf{ext}} \tau_i : \kappa_i} \\ \hat{\Gamma}, \overline{\mathcal{X}}, \overline{\Gamma'} \vdash_{\mathsf{ext}} \hat{\tau}' : \kappa_0\end{array}}{\hat{\Gamma} \vdash_{\mathsf{ext}} \overline{\tau} \mapsto \hat{\tau}' : \forall \overline{\mathcal{X}}.\overline{\kappa} \twoheadrightarrow \kappa_0} \quad \textsc{TyCl\_Clause}$$

$\boxed{\hat{\Gamma} \vdash_{\mathsf{ext}} \kappa\ \mathbf{kind}}$ Extended context kind validity

$$\frac{\mathcal{X} \in \hat{\Gamma} \qquad \vdash_{\mathsf{ext}} \hat{\Gamma}}{\hat{\Gamma} \vdash_{\mathsf{ext}} \mathcal{X}\ \mathbf{kind}} \quad \textsc{KiX\_Var}$$

$$\frac{\begin{array}{c}T : \overline{\star} \to \star \\ \overline{\hat{\Gamma} \vdash_{\mathsf{ext}} \kappa_i\ \mathbf{kind}}\end{array}}{\hat{\Gamma} \vdash_{\mathsf{ext}} {}'T\ \overline{\kappa}\ \mathbf{kind}} \quad \textsc{KiX\_TyCon}$$

$$\frac{\hat{\Gamma} \vdash_{\mathsf{ext}} \kappa_1\ \mathbf{kind} \qquad \hat{\Gamma} \vdash_{\mathsf{ext}} \kappa_2\ \mathbf{kind}}{\hat{\Gamma} \vdash_{\mathsf{ext}} \kappa_1 \to \kappa_2\ \mathbf{kind}} \quad \textsc{KiX\_Arrow}$$

$$\frac{\vdash_{\mathsf{ext}} \hat{\Gamma}}{\hat{\Gamma} \vdash_{\mathsf{ext}} \star\ \mathbf{kind}} \quad \textsc{KiX\_Star}$$

$\boxed{\hat{\Gamma} \vdash_{\mathsf{ext\ sch}} \psi\ \mathbf{kind\ scheme}}$ Extended context kind scheme validity

$$\frac{\hat{\Gamma}, \mathcal{X} \vdash_{\mathsf{ext\ sch}} \psi\ \mathbf{kind\ scheme}}{\hat{\Gamma} \vdash_{\mathsf{ext\ sch}} \forall \mathcal{X}.\psi\ \mathbf{kind\ scheme}} \quad \textsc{KiSchX\_Poly}$$

$$\frac{\hat{\Gamma} \vdash_{\mathsf{ext}} \kappa\ \mathbf{kind}}{\hat{\Gamma} \vdash_{\mathsf{ext\ sch}} \kappa\ \mathbf{kind\ scheme}} \quad \textsc{KiSchX\_Mono}$$

$\boxed{\vdash_{\mathsf{ext}} \hat{\Gamma}}$ Extended context validity

$$\frac{}{\vdash_{\mathsf{ext}} \varnothing} \quad \textsc{CtxX\_Nil}$$

$$\frac{\hat{\Gamma} \vdash_{\mathsf{ext\ sch}} \psi\ \mathbf{kind\ scheme} \qquad \alpha \# \hat{\Gamma}}{\vdash_{\mathsf{ext}} \hat{\Gamma}, \alpha{:}\psi} \quad \textsc{CtxX\_Type}$$

$$\frac{\vdash_{\mathsf{ext}} \hat{\Gamma} \qquad \mathcal{X} \# \hat{\Gamma}}{\vdash_{\mathsf{ext}} \hat{\Gamma}, \mathcal{X}} \quad \textsc{CtxX\_Kind}$$

$\boxed{\hat{\Gamma} \vdash_{\mathsf{ext}} \theta\ \mathbf{ok}}$ Substitution validity

$$\frac{}{\hat{\Gamma} \vdash_{\mathsf{ext}} \varnothing\ \mathbf{ok}} \quad \textsc{SubstX\_Nil}$$

$$\frac{\begin{array}{c}\hat{\Gamma} \vdash_{\mathsf{ext}} \theta\ \mathbf{ok} \\ \hat{\Gamma} \vdash_{\mathsf{ext}} \alpha : \kappa \qquad \lfloor \hat{\Gamma} \rfloor \vdash \theta(\tau) : \kappa\end{array}}{\hat{\Gamma} \vdash_{\mathsf{ext}} \theta, \alpha \mapsto \tau\ \mathbf{ok}} \quad \textsc{SubstX\_Var}$$

C. Global context consistency

We assume throughout the proof a global context (containing data constructors, type constructors, and type families) with the following properties:

1. If $K : \sigma$, then:
   (a) $\varnothing \vdash_{\mathsf{sch}} \sigma : \star$
   (b) $\sigma = \forall \overline{\alpha}.\overline{\tau} \to T\ \overline{\alpha}$ for some $\overline{\tau}$ and T.
   (c) For each $\tau_i \in \overline{\tau}$ (as defined immediately above), τ_i contains no arrows and no kind annotations.

   Note that the restriction on arrows is significant; this is where the problems discussed in Section 4.4 come into play. The restriction on kind annotations appears for technical reasons and does not restrict the language meaningfully, as annotations are redundant on data constructor types.
2. If $T : \forall \overline{\mathcal{X}}.\kappa$, then:
   (a) $\kappa = \overline{\kappa}' \to \star$ for some $\overline{\kappa}'$.
   (b) $\overline{\mathcal{X}} = \mathit{fkv}(\kappa)$
3. If $F : \forall \overline{\mathcal{X}}.\{\overline{\kappa}\}\kappa_0$, then:
   (a) $\overline{\mathcal{X}} = \mathit{fkv}(\overline{\kappa}, \kappa_0)$
   (b) $\overline{\overline{\mathcal{X}} \vdash \kappa_i\ \mathbf{kind}}$
   (c) $\overline{\mathcal{X}} \vdash \kappa_0\ \mathbf{kind}$
4. $(\texttt{@@}) : \forall \mathcal{X}, \mathcal{Y}.\{\mathcal{X} \twoheadrightarrow \mathcal{Y}, \mathcal{X}\}\mathcal{Y}$
5. $(\twoheadrightarrow) : \star \to \star \to \star$

D. Structural lemmas

Below, the metavariable $\mathcal{J}$ stands for any judgement defined.

Lemma 1 (Weakening)**.**

1. *If $\Gamma \vdash \mathcal{J}$, then $\Gamma, \Gamma' \vdash \mathcal{J}$.*
2. *If $\hat{\Gamma} \vdash_{\mathsf{ext}} \mathcal{J}$, then $\hat{\Gamma}, \hat{\Gamma}' \vdash_{\mathsf{ext}} \mathcal{J}$.*

Proof. By straightforward induction, noting that we can always rename variables in patterns in order to be fresh. ☐

Lemma 2 (Permutation)**.**

1. *If $\Gamma \vdash \mathcal{J}$ and Γ' is a permutation of Γ such that $\vdash \Gamma'$, then $\Gamma' \vdash \mathcal{J}$.*
2. *If $\hat{\Gamma} \vdash_{\mathsf{ext}} \mathcal{J}$ and $\hat{\Gamma}'$ is a permutation of $\hat{\Gamma}$ such that $\vdash_{\mathsf{ext}} \hat{\Gamma}'$, then $\hat{\Gamma}' \vdash_{\mathsf{ext}} \mathcal{J}$.*

Proof. By straightforward induction. ☐

Lemma 3 (Strengthening in type judgements)**.**

1. *If $\Gamma, \Gamma' \vdash \mathcal{J}$ where $\mathcal{J}$ does not mention terms, and if Γ' contains only term bindings, then $\Gamma \vdash \mathcal{J}$.*
2. *If $\hat{\Gamma}, \hat{\Gamma}' \vdash_{\mathsf{ext}} \mathcal{J}$ where $\mathcal{J}$ does not mention terms, and if $\hat{\Gamma}'$ contains only term bindings, then $\hat{\Gamma} \vdash_{\mathsf{ext}} \mathcal{J}$.*

Proof. By straightforward induction, appealing to Lemma 2 as necessary. ☐

Lemma 4 (Context strengthening)**.** *If $\vdash \Gamma, \Gamma'$, then $\vdash \Gamma$.*

Proof. By induction on the length of Γ'. ☐

Lemma 5 (Strengthening – kind variables)**.** *Assume $\overline{\mathcal{X}} \cap \mathit{fkv}(\kappa) = \varnothing$ and that $\overline{\mathcal{X}} \cap \mathit{fkv}(\psi) = \varnothing$.*

1. *If $\Gamma, \overline{\mathcal{X}} \vdash \tau : \kappa$, then $\Gamma \vdash \tau : \kappa$.*
2. *If $\Gamma, \overline{\mathcal{X}} \vdash \kappa\ \mathbf{kind}$, then $\Gamma \vdash \kappa\ \mathbf{kind}$.*
3. *If $\Gamma, \overline{\mathcal{X}} \vdash_{\mathsf{sch}} \psi\ \mathbf{kind\ scheme}$, then $\Gamma \vdash_{\mathsf{sch}} \psi\ \mathbf{kind\ scheme}$.*

Proof. By straightforward mutual induction. In the Ty_App case, it is critical to have the condition $fkv(\kappa_1) \subseteq fkv(\kappa_2)$ in order to be able to apply the induction hypothesis. Also, note the fact that all type variables quantified over a data constructor's type appear in the type itself, and all kind variables quantified over a type constructor's kind appear in the kind itself. We use Lemma 4 to show that shrinking the context does not cause problems. □

Lemma 6 (Application contexts). *If $\alpha{:}\star \in \Gamma$ and $\Gamma \vdash \tau : \kappa$, then α does not appear in an application context in τ.*

Proof. By straightforward induction on $\Gamma \vdash \tau : \kappa$, noting that kinds for type variables are clearly deterministic in the application case. □

Lemma 7 (Application). *If $\tau = \tau_1\,\tau_2$, then there exist τ_0 and $\overline{\tau}'$ such that $\tau = \tau_0\,\overline{\tau}'$ and τ_0 is not an application.*

Proof. Straightforward induction on the structure of τ_1. □

Lemma 8 (Application typing). *If $\Gamma \vdash \tau_1 : \kappa_1 \to \kappa_2$ and $\Gamma \vdash \tau_2 : \kappa_1$, then there exist τ_0 (not an application), $\overline{\tau}'$ (not empty), and $\overline{\kappa}'$ such that $\tau_1\,\tau_2 = \tau_0\,\overline{\tau}'$, $\Gamma \vdash \tau_0 : \overline{\kappa}' \to \kappa_2$ and $\overline{\Gamma \vdash \tau'_i : \kappa'_i}$.*

Proof. Straightforward induction on $\Gamma \vdash \tau_1 : \kappa_1 \to \kappa_2$. □

Lemma 9 (Type substitution in types). *Suppose $\Gamma \vdash \tau' : \kappa'$.*

1. *If $\vdash \Gamma, \alpha{:}\kappa', \Gamma'$, then $\vdash \Gamma, \Gamma'[\tau'/\alpha]$.*
2. *If $\Gamma, \alpha{:}\kappa', \Gamma' \vdash \kappa\ \mathbf{kind}$, then $\Gamma, \Gamma'[\tau'/\alpha] \vdash \kappa\ \mathbf{kind}$.*
3. *If $\Gamma, \alpha{:}\kappa', \Gamma' \vdash_{sch} \psi\ \mathbf{kind\ scheme}$, then $\Gamma, \Gamma'[\tau'/\alpha] \vdash_{sch} \psi\ \mathbf{kind\ scheme}$.*
4. *If $\Gamma, \alpha{:}\kappa', \Gamma' \vdash_{sch} \sigma : \kappa$, then $\Gamma, \Gamma'[\tau'/\alpha] \vdash_{sch} \sigma[\tau'/\alpha] : \kappa$.*
5. *If $\Gamma, \alpha{:}\kappa', \Gamma' \vdash \tau : \kappa$ and $\Gamma \vdash \tau' : \kappa'$, then $\Gamma, \Gamma'[\tau'/\alpha] \vdash \tau[\tau'/\alpha] : \kappa$.*

Proof. Straightforward mutual induction. □

Lemma 10 (Substitution validity respects concatenation). *If $\hat{\Gamma} \vdash_{ext} \theta_1\ \mathbf{ok}$ and $\hat{\Gamma} \vdash_{ext} \theta_2\ \mathbf{ok}$, then $\hat{\Gamma} \vdash_{ext} \theta_1, \theta_2\ \mathbf{ok}$.*

Proof. Straightforward induction on the length of θ_2. □

E. Regularity

Lemma 11 (Pattern contexts). *If $\vdash \pi : \tau \rightsquigarrow \Gamma$, then Γ contains only term bindings.*

Proof. By straightforward induction. □

Lemma 12 (Let contexts). *If $\Gamma \vdash \overline{\delta} \rightsquigarrow \Gamma'$, then Γ' contains only term bindings.*

Proof. Straightforward case analysis on $\Gamma \vdash \delta \rightsquigarrow \Gamma'$. □

Lemma 13 (Kind patterns contexts). *If $\vdash \kappa \rightsquigarrow \Gamma$, then $kvs(\Gamma) = fkv(\kappa)$.*

Proof. By straightforward induction. □

Lemma 14 (Type patterns contexts). *If $\vdash \tau : \kappa \rightsquigarrow \Gamma$, then $tvs(\Gamma) = ftv(\tau)$.*

Proof. By straightforward induction, appealing to Lemma 13 in the TyPat_Annot case. □

Lemma 15 (Regularity of term typing). *If $\Gamma \vdash e : \tau$, then $\Gamma \vdash \tau : \star$.*

Proof. Straightforward induction, appealing to Lemmas 1, 3, 11, and 12. □

Lemma 16 (Regularity of term typing (contexts)). *If $\Gamma \vdash e : \tau$, then $\vdash \Gamma$.*

Proof. Straightforward induction, appealing to Lemma 3 in the Tm_Abs case. □

Lemma 17 (Regularity of kind validity (contexts)). *If $\Gamma \vdash \kappa\ \mathbf{kind}$, then $\vdash \Gamma$.*

Proof. By straightforward induction. □

Lemma 18 (Regularity of kind scheme validity (contexts)). *If $\Gamma \vdash_{sch} \psi\ \mathbf{kind\ scheme}$, then $\vdash \Gamma$.*

Proof. By straightforward induction, appealing to Lemma 17. □

Lemma 19 (Regularity of type kinding (contexts)). *If $\Gamma \vdash \tau : \kappa$, then $\vdash \Gamma$.*

Proof. By straightforward induction, appealing to Lemma 18. □

F. Promotion structural lemmas

Lemma 20 (Promotion commutes with substitution (terms)). *For all e, e', and x, $(e[e'/x])\Uparrow = (e\Uparrow)[e'\Uparrow/x\Uparrow]$.*

Proof. Straightforward induction on the structure of e. □

Lemma 21 (Promotion commutes with substitutions (terms)). *For all e, $\overline{e}'$, and $\overline{x}$, $(e[\overline{e}'/\overline{x}])\Uparrow = (e\Uparrow)[\overline{e}'\Uparrow/\overline{x}\Uparrow]$.*

Proof. Straightforward induction on the length of $\overline{e}'$, appealing to Lemma 20 □

Lemma 22 (Promotion commutes with substitution (types)). *For all τ, τ', and α, such that α never appears in an application position in τ, $(\tau[\tau'/\alpha])\Uparrow = (\tau\Uparrow)[\tau'\Uparrow/\alpha\Uparrow]$.*

Proof. By induction on the structure of τ.

$\tau = \alpha$: Then $\tau[\tau'/\alpha] = \tau'$ and $(\tau\Uparrow)[\tau'\Uparrow/\alpha\Uparrow] = \tau'\Uparrow$, and we are done.

$\tau = (\to)$: Then $\tau[\tau'/\alpha] = (\to)$, and $(\to)\Uparrow$ is not defined. On the right-hand side $(\to)\Uparrow$ is still not defined, so we are done.

$\tau = \tau_1\,\tau_2$: By Lemma 7 we can rewrite $\tau = \tau_0\,\overline{\tau}$, where τ_0 is not an application. Then, proceed by case analysis on τ_0:

- **$\tau_0 = \alpha$:** Impossible, because α is in an application position in τ.
- **$\tau_0 = (\to)$:** Here, we have two cases:
  - **$\overline{\tau} = \tau_3, \tau_4$:** Thus, $\tau = \tau_3 \to \tau_4$ and $(\tau[\tau'/\alpha])\Uparrow = (\tau_3[\tau'/\alpha])\Uparrow \twoheadrightarrow (\tau_4[\tau'/\alpha])\Uparrow$. By the induction hypothesis, we rewrite this as $(\tau_3\Uparrow)[\tau'\Uparrow/\alpha\Uparrow] \twoheadrightarrow (\tau_4\Uparrow)[\tau'\Uparrow/\alpha\Uparrow]$, which is clearly $(\tau\Uparrow)[\tau'\Uparrow/\alpha\Uparrow]$ as desired.
  - **Otherwise:** We get $\tau[\tau'/\alpha] = ((\to)\,\overline{\tau})[\tau'/\alpha] = (\to)\,\overline{\tau}[\tau'/\alpha]$. This last type does not promote. Similarly, $\tau\Uparrow$ itself does not promote on the right hand side, so we are done.
- **$\tau_0 = \tau_3\,\tau_4$:** Impossible, because we assumed that τ_0 is not an application.
- **$\tau_0 = T$:** We have $\tau = T\,\overline{\tau}$ and thus

$$(\tau[\tau'/\alpha])\Uparrow = {}'T\ (\overline{\tau}[\tau'/\alpha])\Uparrow$$

  Both substitution and promotion distribute over lists. We can then use the induction hypothesis and reverse distribution to get $'T\ (\overline{\tau}\Uparrow)[\tau'\Uparrow/\alpha\Uparrow]$ as desired.

Other cases: Neither the left-hand nor right-hand sides are promotable, so we are done.

$\tau = {}'K$**:** Neither side is promotable.

$\tau = T$**:** Trivial.

$\tau = \tau_0 :: \kappa$**:** Here $\tau[\tau'/\alpha] = \tau_0[\tau'/\alpha] :: \kappa$. Then, $(\tau_0[\tau'/\alpha] :: \kappa)\Uparrow = (\tau_0[\tau'/\alpha])\Uparrow$. We apply the induction hypothesis, and we are done.

$\tau = F(\overline{\tau})$**:** Neither side is promotable.

□

Lemma 23 (Promotion commutes with substitutions (types))**.** *For all τ, $\overline{\tau}'$, and $\overline{\alpha}$, such that no $\alpha \in \overline{\alpha}$ ever appears in an application position in τ, $(\tau[\overline{\tau}'/\overline{\alpha}])\Uparrow = (\tau\Uparrow)[\overline{\tau}'\Uparrow/\overline{\alpha}\Uparrow]$.*

Proof. Straightforward induction on the length of the list $\overline{\alpha}$, using Lemma 22. □

G. Symbols

Lemma 24 (Data constructor symbols)**.** *If $K : \sigma$ and if $\sigma\Uparrow$ exists, then $\mathsf{Sym}_0(K) : \sigma\Uparrow$.*

Proof. Expanding definitions, we assume $K : \forall\,\overline{\alpha}.\tau$ and must show $\mathsf{Sym}_0(K) : \forall\,\overline{\alpha}\Uparrow.\tau\Uparrow$. Rewrite τ as $\overline{\tau} \to \tau_0$, where τ_0 is not an arrow type. Then, by the definition of the symbol generation algorithm (described in 4.3 but not formalized), there exist symbols $\mathsf{Sym}_0(K) \ldots \mathsf{Sym}_n(K)$, where $n = \mathsf{length}\,\overline{\tau} + 1$. By construction, $\mathsf{Sym}_n(K) := {}'K$ and $\mathsf{Sym}_n(K) : \forall\,\overline{\mathcal{X}}.(\tau[\overline{\mathcal{X}}/\overline{\alpha}])$, where $\forall\,\overline{\mathcal{X}}.(\tau[\overline{\mathcal{X}}/\overline{\alpha}])$ is the kind scheme derived using GHC's built-in promotion of σ. (See TY_DATACON.) We are free to rename bound variables, so we choose $\overline{\mathcal{X}} = \overline{\alpha}\Uparrow$ and say that $\mathsf{Sym}_n(K) : \forall\,\overline{\alpha}\Uparrow.(\tau[\overline{\alpha}\Uparrow/\overline{\alpha}])$. Let $\kappa = \tau[\overline{\alpha}\Uparrow/\overline{\alpha}]$ and $\overline{\kappa} \to \kappa_0 = \kappa$. We now wish to show that, for all $m \leqslant n$, $\mathsf{Sym}_{n-m}(K) : \forall\,\overline{\alpha}\Uparrow.\overline{\kappa'}^{n-m} \to \overline{\kappa''}^{m} \twoheadrightarrow \kappa_0$, where $\overline{\kappa'}^{n-m}, \overline{\kappa''}^{m} = \overline{\kappa}$. We proceed by induction on m.

Base case: Trivial.

Inductive case: We assume our goal for $m-1$, and we must prove it for m. By construction,

$$(\mathsf{Sym}_{n-m}(K)\,\overline{\beta}^{n-m})\;@@\;\tau' := \mathsf{Sym}_{n-m+1}(K)\,\overline{\beta}^{n-m}\,\tau'$$

for any well-kinded $\overline{\beta}$ and τ'. Our induction hypothesis says that $\mathsf{Sym}_{n-m+1}(K) : \forall\,\overline{\alpha}\Uparrow.\overline{\kappa'}^{n-m+1} \to \overline{\kappa''}^{m-1} \twoheadrightarrow \kappa_0$. Let $\overline{\kappa_1}^{n-m}, \kappa_2 = \overline{\kappa'}^{n-m+1}$, noting that $n - m \geqslant 0$ from $m \leqslant n$. Then, $\varnothing \vdash \mathsf{Sym}_{n-m+1}(K)\,\overline{\beta}^{n-m}\,\tau' : (\overline{\kappa''}^{m-1} \twoheadrightarrow \kappa_0)[\overline{\kappa}_3/\overline{\alpha}\Uparrow]$, where $\overline{\varnothing \vdash \beta_i : \kappa_{1\,i}[\overline{\kappa}_3/\overline{\alpha}\Uparrow]}$ and $\varnothing \vdash \tau' : \kappa_2[\overline{\kappa}_3/\overline{\alpha}\Uparrow]$, for any well-sorted $\overline{\kappa}_3$. Thus, we can say $\varnothing \vdash (\mathsf{Sym}_{n-m}(K)\,\overline{\beta}^{n-m})\;@@\;\tau' : (\overline{\kappa''}^{m-1} \twoheadrightarrow \kappa_0)[\overline{\kappa}_3/\overline{\alpha}\Uparrow]$. From the type of $(@@)$, we must conclude that $\varnothing \vdash \mathsf{Sym}_{n-m}(K)\,\overline{\beta}^{n-m} : (\kappa_2 \twoheadrightarrow \overline{\kappa''}^{m-1} \twoheadrightarrow \kappa_0)[\overline{\kappa}_3/\overline{\alpha}\Uparrow]$. Furthermore, straightforward induction on the length of $\overline{\beta}^{n-m}$ shows us that $\varnothing \vdash \mathsf{Sym}_{n-m}(K) : (\overline{\kappa_1}^{n-m} \to \kappa_2 \twoheadrightarrow \overline{\kappa''}^{m-1} \twoheadrightarrow \kappa_0)[\overline{\kappa}_3/\overline{\alpha}\Uparrow]$. Generalizing over kinds (as is done in datatype definitions such as that for $\mathsf{Sym}_{n-m}(K)$) gives us $\mathsf{Sym}_{n-m}(K) : \forall\,\overline{\alpha}\Uparrow.\overline{\kappa_1}^{n-m} \to \kappa_2 \twoheadrightarrow \overline{\kappa''}^{m-1} \twoheadrightarrow \kappa_0$ as desired.

From the above, we can conclude that $\mathsf{Sym}_0(K) : \forall\,\overline{\alpha}\Uparrow.\overline{\kappa} \twoheadrightarrow \kappa_0$. All that remains is to show that $(\overline{\tau} \to \tau_0)\Uparrow = \overline{\kappa} \twoheadrightarrow \kappa_0$, where we defined $\overline{\kappa} = \overline{\tau}[\overline{\alpha}\Uparrow/\overline{\alpha}]$ and $\kappa_0 = \tau_0[\overline{\alpha}\Uparrow/\overline{\alpha}]$. Assuming that $(\overline{\tau} \to \tau_0)\Uparrow$ exists at all, this comes straight from the definitions of $\Uparrow$ and GHC's internal promotion algorithm, with an induction on the length of $\overline{\tau}$ and then on the structure of each individual τ. We are done. □

Lemma 25 (Type family symbols)**.** *If $F : \forall\,\overline{\mathcal{X}}.\{\overline{\kappa}^n\}\kappa_0$, then, $\forall m$ with $1 \leqslant m \leqslant n$, $\mathsf{Sym}_{n-m}(F) : \forall\,\overline{\mathcal{X}}.\overline{\kappa'}^{n-m} \to \overline{\kappa''}^{m} \twoheadrightarrow \kappa_0$, where $\overline{\kappa}', \overline{\kappa}'' = \overline{\kappa}$.*

Proof. Just as in the induction in the proof above. The difference here is that F is a type function, which must appear fully applied, not a data constructor with arrows in its type. But, other than syntactic differences in the formalism, this not a significant difference here. □

H. Promoting to extended types

Lemma 26 (Promoting types yields kinds)**.** *For all Γ, $(\Gamma^{ty})\Uparrow = \Gamma\Uparrow^{ki}$.*

Proof. Straightforward induction on the length of Γ. □

Lemma 27 (Promoting types is well-kinded)**.** *For all Γ, τ, and σ such that $\Gamma\Uparrow$, $\tau\Uparrow$, and $\sigma\Uparrow$ exist:*

1. *If $\Gamma \vdash \tau : \star$, then $\Gamma\Uparrow \vdash_{\mathsf{ext}} \tau\Uparrow\ \mathbf{kind}$.*
2. *If $\Gamma \vdash_{\mathsf{sch}} \sigma : \star$, then $\Gamma\Uparrow \vdash_{\mathsf{ext\ sch}} \sigma\Uparrow\ \mathbf{kind\ scheme}$.*
3. *If $\vdash \Gamma$, then $\vdash_{\mathsf{ext}} \Gamma\Uparrow$.*

Proof. By mutual induction.

Case TY_VAR: By KIX_VAR.

Case TY_ARROW: Not possible.

Case TY_APP: We use Lemma 8 to get $\Gamma \vdash \tau_0 : \overline{\kappa}' \to \star$ and $\overline{\Gamma \vdash \tau_i' : \kappa_i'}$. We must show $\Gamma\Uparrow \vdash_{\mathsf{ext}} (\tau_0\,\overline{\tau}')\Uparrow\ \mathbf{kind}$. We know τ_0 is not an application and that $\overline{\tau}'$ must not be an empty list. We proceed by case analysis on τ_0:

Case $\tau_0 = (\to)$**:** From the fact that $\tau\Uparrow$ exists, it must be the case that $\overline{\tau}' = \tau_1, \tau_2$. We see that $\Gamma \vdash (\to) : \star \to \star \to \star$, and thus that $\Gamma \vdash \tau_1 : \star$ and $\Gamma \vdash \tau_2 : \star$. We use the induction hypothesis to get $\Gamma\Uparrow \vdash_{\mathsf{ext}} \tau_1\Uparrow\ \mathbf{kind}$ and $\Gamma\Uparrow \vdash_{\mathsf{ext}} \tau_2\Uparrow\ \mathbf{kind}$. By the definition of $\Uparrow$, we see that $(\tau_1 \to \tau_2)\Uparrow = \tau_1\Uparrow \twoheadrightarrow \tau_2\Uparrow$. We also note that $(\twoheadrightarrow) : \star \to \star \to \star$. We apply KIX_TYCON and we are done.

Case $\tau_0 = T$**:** By the side condition allowing $(T\,\overline{\tau}')\Uparrow$ to exist, we know that $\overline{\kappa' = \star}$. Thus, the induction hypothesis gives us $\overline{\Gamma\Uparrow \vdash_{\mathsf{ext}} \tau_i'\Uparrow\ \mathbf{kind}}$. We apply KIX_TYCON and we are done.

Other cases: Not possible, as $(\tau_0\,\overline{\tau}')\Uparrow$ would not exist.

Case TY_DATACON: Not possible.

Case TY_TYCON: Here, $\tau = T$. It must be that $T : \star$. We are done by KIX_TYCON.

Case TY_ANNOT: By induction and the definition of $\Uparrow$.

Case TY_FAM: Not possible.

Case TYSCH_POLY: By induction hypothesis and KISCHX_POLY.

Case TYSCH_MONO: By induction hypothesis and KISCHX_MONO.

Case CTX_NIL: By CTXX_NIL.

Case CTX_TERM: By induction hypothesis and CTXX_TYPE.

Case CTX_TYPE: We appeal to Lemma 17 to get $\vdash \Gamma$, and we then use the induction hypothesis and CTXX_KIND.

Case CTX_KIND: Not possible.

□

Lemma 28 (Arrow-free promotion)**.** *If τ contains no arrows and no kind annotations and $\tau\Uparrow$ exists, then $(\tau\Uparrow)[\overline{\kappa}/\overline{\alpha}\Uparrow] = \tau[\overline{\kappa}/\overline{\alpha}]$.*

Proof. By induction on the structure of τ, appealing to Lemma 7 in the application case. We need the arrow restriction because GHC's promotion and $\Uparrow$ treat arrows differently; we need the kind annotation restriction because GHC's promotion algorithm does not support kind annotations. □

Lemma 29 (Pattern promotion)**.** *If* $\vdash \pi : \tau \rightsquigarrow \Gamma$*,* $\Gamma_0 \vdash \tau : \star$*, and* $\tau\Uparrow$ *exists, then* $\vdash \pi\Uparrow : \tau\Uparrow \rightsquigarrow \Gamma'$ *and* $\Gamma_0\Uparrow, \Gamma' \vdash_{\mathsf{ext}} \pi\Uparrow : \tau\Uparrow$*, where* $\Gamma\Uparrow \subseteq \Gamma'$ *(and thus that* $\Gamma\Uparrow$ *has no bindings of the form* $\alpha{:}\psi$*, which are disallowed in the grammar for* Γ'*).*

Proof. By induction.

Case PAT_VAR: By TYPAT_VAR and TYX_VAR.

Case PAT_DATACON:

$$\frac{K : \forall\,\overline{\alpha}.(\overline{\tau} \to T\,\overline{\alpha}) \qquad \overline{\vdash \pi_i : \tau_i[\overline{\tau}'/\overline{\alpha}] \rightsquigarrow \Gamma_i}}{\vdash K\,\overline{\pi} : T\,\overline{\tau}' \rightsquigarrow \overline{\Gamma}}\ \textsc{Pat\_DataCon}$$

We know $K : \forall\,\overline{\alpha}.(\overline{\tau} \to T\,\overline{\alpha})$ and that, for all i, $\vdash \pi_i : \tau_i[\overline{\tau}'/\overline{\alpha}] \rightsquigarrow \Gamma_i$. We further know that all the bindings in the Γ'_i are distinct – this is implicit in the well-definedness of $\overline{\Gamma}$. (If there could be repeats, we would take an explicit union.) We must show $\vdash\ {}'K\,\overline{\pi}\Uparrow\ :\ {}'T\ \overline{\tau}'\Uparrow\ \rightsquigarrow\ \overline{\Gamma'}$ and $\Gamma_0\Uparrow, \overline{\Gamma'} \vdash_{\mathsf{ext}} {}'K\,\overline{\pi}\Uparrow : {}'T\ \overline{\tau}'\Uparrow$. We wish to perform induction from the end of the list $\overline{\pi}$. Let n be the length of this list. For all $m \leqslant n$, we must show $\vdash {}'K\,\overline{\pi}'^{m}\Uparrow : (\overline{\tau}^{n-m}[\overline{\tau}'/\overline{\alpha}])\Uparrow \to {}'T\ \overline{\tau}'\Uparrow \rightsquigarrow \Gamma'_m$ and $\Gamma_0\Uparrow, \Gamma'_m \vdash_{\mathsf{ext}} {}'K\,\overline{\pi}'^{m}\Uparrow : (\overline{\tau}^{n-m}[\overline{\tau}'/\overline{\alpha}])\Uparrow \to {}'T\ \overline{\tau}'\Uparrow$. We first simplify (using Lemmas 6 and 23) to $\vdash {}'K\,\overline{\pi}'^{m}\Uparrow : (\overline{\tau}^{n-m}\Uparrow)[\overline{\tau}'\Uparrow/\overline{\alpha}\Uparrow] \to {}'T\ \overline{\tau}'\Uparrow \rightsquigarrow \Gamma'_m$ and $\Gamma_0\Uparrow, \Gamma'_m \vdash_{\mathsf{ext}} {}'K\,\overline{\pi}'^{m}\Uparrow : (\overline{\tau}^{n-m}\Uparrow)[\overline{\tau}'\Uparrow/\overline{\alpha}\Uparrow] \to {}'T\ \overline{\tau}'\Uparrow$. Proceed by induction on m.

Base case: We must show $\vdash {}'K : (\overline{\tau}\Uparrow)[\overline{\tau}'\Uparrow/\overline{\alpha}\Uparrow] \to {}'T\ \overline{\tau}'\Uparrow \rightsquigarrow \varnothing$ and $\Gamma_0\Uparrow \vdash_{\mathsf{ext}} {}'K : (\overline{\tau}\Uparrow)[\overline{\tau}'\Uparrow/\overline{\alpha}\Uparrow] \to {}'T\ \overline{\tau}'\Uparrow$. For the first we wish to use TYPAT_DATACON. To do so, we must show that $(\overline{\tau}\Uparrow)[\overline{\tau}'\Uparrow/\overline{\alpha}\Uparrow] \to {}'T\ \overline{\tau}'\Uparrow = (\overline{\tau} \to T\,\overline{\alpha})[\overline{\tau}'\Uparrow/\overline{\alpha}]$ (choosing $\overline{\kappa} = \overline{\tau}\Uparrow$ in the conclusion of TYPAT_DATACON). By the definition of GHC's promotion algorithm, we can reduce the right-hand side of the desired equality to $\overline{\tau}[\overline{\tau}'\Uparrow/\overline{\alpha}] \to {}'T\ \overline{\tau}'\Uparrow$. So, now we just must show $(\overline{\tau}\Uparrow)[\overline{\tau}'\Uparrow/\overline{\alpha}\Uparrow] = \overline{\tau}[\overline{\tau}'\Uparrow/\overline{\alpha}]$. This fact comes directly from Lemma 28, and we are done.

For the second, we wish to use TYX_DATACON. We use Lemma 28 to reduce our goal to $\Gamma_0\Uparrow \vdash_{\mathsf{ext}} {}'K : \overline{\tau}[\overline{\tau}'\Uparrow/\overline{\alpha}] \to {}'T\ \overline{\tau}'\Uparrow$. We can reduce further to $\Gamma_0\Uparrow \vdash_{\mathsf{ext}} {}'K : (\overline{\tau} \to T\,\overline{\alpha})[\overline{\tau}'\Uparrow/\overline{\alpha}]$. The condition on the promotion algorithm that K is promotable means (by definition of promotable) that $\varnothing \vdash_{\mathsf{sch}} \forall\,\overline{\mathcal{X}}.(\tau[\overline{\mathcal{X}}/\overline{\alpha}])$ $\mathbf{kind\ scheme}$ and thus $\Gamma_0\Uparrow \vdash_{\mathsf{ext\ sch}} \forall\,\overline{\mathcal{X}}.(\tau[\overline{\mathcal{X}}/\overline{\alpha}])$ $\mathbf{kind\ scheme}$ as desired. It remains only to show $\Gamma_0\Uparrow \vdash_{\mathsf{ext}} \tau'_i\Uparrow$ $\mathbf{kind}$ for any i. This comes from inversion on $\Gamma_0 \vdash T\,\overline{\tau}' : \star$ (noting that we must have $T : \overline{\star} \to \star$ for K to be promotable, as found by inversion on the $\mathbf{kind\ scheme}$ judgement), followed by Lemma 27.

Inductive case: We can assume that

$$\begin{aligned}\vdash {}'K\,\overline{\pi}'^{m-1}\Uparrow &: \tau_m\Uparrow[\overline{\tau}'\Uparrow/\overline{\alpha}\Uparrow] \\ &\to (\overline{\tau}^{n-m}\Uparrow)[\overline{\tau}'\Uparrow/\overline{\alpha}\Uparrow] \to {}'T\ \overline{\tau}'\Uparrow \rightsquigarrow \Gamma'_{m-1}\end{aligned}$$

and

$$\begin{aligned}\Gamma_0\Uparrow, \Gamma'_{m-1} \vdash_{\mathsf{ext}} {}'K\,\overline{\pi}'^{m-1}\Uparrow &: \tau_m\Uparrow[\overline{\tau}'\Uparrow/\overline{\alpha}\Uparrow] \\ &\to (\overline{\tau}^{n-m}\Uparrow)[\overline{\tau}'\Uparrow/\overline{\alpha}\Uparrow] \to {}'T\ \overline{\tau}'\Uparrow\end{aligned}$$

We must show that

$$\vdash {}'K\,\overline{\pi}'^{m-1}\Uparrow\pi_m\Uparrow : (\overline{\tau}^{n-m}\Uparrow)[\overline{\tau}'\Uparrow/\overline{\alpha}\Uparrow] \to {}'T\ \overline{\tau}'\Uparrow \rightsquigarrow \Gamma'_m$$

and $\Gamma_0\Uparrow, \Gamma'_m \vdash_{\mathsf{ext}} {}'K\,\overline{\pi}'^{m-1}\Uparrow\pi_m\Uparrow : (\overline{\tau}^{n-m}\Uparrow)[\overline{\tau}'\Uparrow/\overline{\alpha}\Uparrow] \to {}'T\ \overline{\tau}'\Uparrow$. For the first, we wish to use TYPAT_APP. We must show that $\vdash \pi_m\Uparrow : \tau_m\Uparrow[\overline{\tau}'\Uparrow/\overline{\alpha}\Uparrow] \rightsquigarrow \Gamma$ and that $\Gamma \cup \Gamma'_{m-1}$ exists. The first of these comes from the outer inductive hypothesis. The second comes from the fact that the term-level patterns mention each variable α_i only once. Thus, we are done.

For the second, we wish to use TYX_APP. The kinding judgements come from the inner and outer inductive hypotheses, respectively. For the restriction on kind variables, we note that our κ_1, namely $\tau_m\Uparrow[\overline{\tau}'\Uparrow/\overline{\alpha}\Uparrow]$, can have free variables only from the $\overline{\tau}'\Uparrow$ as the only free variables in $\tau_m\Uparrow$ must be the $\overline{\alpha}\Uparrow$. All the $\overline{\tau}'\Uparrow$ are mentioned in the result of the data constructor (${}'T\ \overline{\tau}'\Uparrow$), and so there can be no free variables in κ_1 not in κ_2. (In more practical terms, this is saying that there are no existential type variables in our formalization.) We are done.

Thus, we can conclude $\vdash\ {}'K\,\overline{\pi}\Uparrow\ :\ {}'T\ \overline{\tau}'\Uparrow\ \rightsquigarrow\ \overline{\Gamma'}$ and $\Gamma_0\Uparrow, \overline{\Gamma'} \vdash_{\mathsf{ext}} {}'K\,\overline{\pi}\Uparrow : {}'T\ \overline{\tau}'\Uparrow$. We still must show that $\Gamma\Uparrow \subseteq \Gamma'$. This is a direct consequence of the fact, gotten from the induction hypothesis, that each Γ'_i is such that $\Gamma_i\Uparrow \subseteq \Gamma'_i$. We are done.

Case PAT_WILD: By TYPAT_VAR.

□

Lemma 30 (Promotion to extended types is well-typed)**.** *For all* Γ*,* e*,* τ*, and* $\overline{\delta}$*, such that* $\Gamma\Uparrow$*,* $e\Uparrow$*,* $\tau\Uparrow$*, and* $\overline{\delta}\Uparrow$*, exist:*

1. *If* $\Gamma \vdash e : \tau$*, then* $\Gamma\Uparrow \vdash_{\mathsf{ext}} e\Uparrow : \tau\Uparrow$*.*
2. *If* $\Gamma \vdash \overline{\delta} \rightsquigarrow \Gamma'$*, then* $\Gamma\Uparrow \vdash_{\mathsf{ext}} \overline{\delta}\Uparrow \rightsquigarrow \Gamma'\Uparrow$*.*

Proof. By mutual induction. In each case, we adopt the variable names from the rule statement.

Case TM_VAR:

$$\frac{x{:}\forall\,\overline{\alpha}.\tau_0 \in \Gamma \qquad \vdash \Gamma \qquad \overline{\Gamma \vdash \tau_i : \star}}{\Gamma \vdash x : \tau_0[\overline{\tau}/\overline{\alpha}]}\ \textsc{Tm\_Var}$$

We must show $\Gamma\Uparrow \vdash_{\mathsf{ext}} x\Uparrow : (\tau_0[\overline{\tau}/\overline{\alpha}])\Uparrow$. We wish to use Lemma 23, but we must establish that no $\alpha \in \overline{\alpha}$ appears in an application position in τ_0. From $\vdash \Gamma$, we know that $\Gamma \vdash_{\mathsf{sch}} \forall\,\overline{\alpha}.\tau_0 : \star$ and therefore that $\Gamma, \overline{\alpha{:}\star} \vdash \tau_0 : \star$. Thus, by Lemma 6, we see that no α_i can appear in an application context in τ_0. Thus, we use Lemma 23 to see that we must show $\Gamma\Uparrow \vdash_{\mathsf{ext}} x\Uparrow : (\tau_0\Uparrow)[\overline{\tau}\Uparrow/\overline{\alpha}\Uparrow]$. We will use TYX_VAR. We must show $x\Uparrow{:}\forall\,\overline{\alpha}\Uparrow.\tau_0\Uparrow \in \Gamma\Uparrow$, $\vdash_{\mathsf{ext}} \Gamma\Uparrow$, and $\overline{\Gamma\Uparrow \vdash_{\mathsf{ext}} \tau_i\Uparrow\ \mathbf{kind}}$. The first is direct from the definition of $\Uparrow$, and the rest are by Lemma 27.

Case TM_DATACON:

$$\frac{K : \forall\,\overline{\alpha}.\tau_0 \qquad \vdash \Gamma \qquad \overline{\Gamma \vdash \tau_i : \star}}{\Gamma \vdash K : \tau_0[\overline{\tau}/\overline{\alpha}]}\ \textsc{Tm\_DataCon}$$

We must show $\Gamma\Uparrow \vdash_{\mathsf{ext}} K\Uparrow : (\tau_0[\overline{\tau}/\overline{\alpha}])\Uparrow$, or $\Gamma\Uparrow \vdash_{\mathsf{ext}} \mathsf{Sym}_0(K) : (\tau_0\Uparrow)[\overline{\tau}\Uparrow/\overline{\alpha}\Uparrow]$, appealing to the rules about data constructor types in the global contexts, and then Lemmas 6 and 23. We wish to use TYX_TYCON. We know $\vdash_{\mathsf{ext}} \Gamma\Uparrow$ by the induction hypothesis and $\mathsf{Sym}_0(K) : \forall\,\overline{\alpha}\Uparrow.\tau_0\Uparrow$ by Lemma 24. We conclude $\overline{\Gamma\Uparrow \vdash_{\mathsf{ext}} \tau_i\Uparrow\ \mathbf{kind}}$ by Lemma 27, and we are done.

Case TM_APP:

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}\ \textsc{Tm\_App}$$

We must show $\Gamma\Uparrow \vdash_{\mathsf{ext}} e_1\Uparrow @@ e_2\Uparrow : \tau_2\Uparrow$. The induction hypothesis gives us $\Gamma\Uparrow \vdash_{\mathsf{ext}} e_1\Uparrow : \tau_1\Uparrow \twoheadrightarrow \tau_2\Uparrow$ and $\Gamma\Uparrow \vdash_{\mathsf{ext}} e_2\Uparrow : \tau_1\Uparrow$. We wish to use TYX_FAM. By definition, $(@@) : \forall \mathcal{X}, \mathcal{Y}.\{\mathcal{X} \twoheadrightarrow \mathcal{Y}, \mathcal{X}\}\mathcal{Y}$. Choose $\tau_1\Uparrow, \tau_2\Uparrow = \overline{\kappa}'$. We must yet show $\vdash_{\mathsf{ext}} \Gamma\Uparrow$, $\Gamma\Uparrow \vdash_{\mathsf{ext}} \tau_1\Uparrow\ \mathbf{kind}$, and $\Gamma\Uparrow \vdash_{\mathsf{ext}} \tau_2\Uparrow\ \mathbf{kind}$. We apply Lemmas 15, 16, and 27, and we are done.

Case TM_ABS: Straightforward use of induction hypothesis and definition of $\Uparrow$.

Case TM_CASE:

$$\frac{\begin{array}{l}\Gamma \vdash e_0 : \tau_0 \\ \text{for each } \pi_i \mapsto e_i \in \overline{\pi \mapsto e} : \\ \quad \vdash \pi_i : \tau_0 \rightsquigarrow \Gamma'_i \\ \quad \Gamma, \Gamma'_i \vdash e_i : \tau\end{array}}{\Gamma \vdash \mathbf{case}\ e_0\ \mathbf{of}\ \overline{\pi \mapsto e} : \tau}\ \textsc{Tm\_Case}$$

We must show $\Gamma\Uparrow \vdash_{\mathsf{ext}} \mathbf{case}\ e_0\Uparrow\ \mathbf{of}\ \overline{\pi \mapsto e}\Uparrow : \tau\Uparrow$. We will use TYX_CASE. The induction hypothesis gives us $\Gamma\Uparrow \vdash_{\mathsf{ext}} e_0\Uparrow : \tau_0\Uparrow$. Now, fix i. We must show $\vdash \pi_i\Uparrow : \tau_0\Uparrow \rightsquigarrow \Gamma''_i$, $\Gamma\Uparrow^{\mathsf{ki}}, \Gamma''_i \vdash \pi_i\Uparrow : \tau_0\Uparrow$, and $\Gamma\Uparrow, \Gamma''_i \vdash_{\mathsf{ext}} e_i\Uparrow : \tau\Uparrow$. The first two of these comes from Lemma 29 (using Lemma 15 to establish that $\Gamma \vdash \tau_0 : \star$ and therefore that $\Gamma^{\mathsf{ty}} \vdash \tau_0 : \star$ by Lemma 3, using Lemma 26 to simplify), along with the fact that $\Gamma'_i\Uparrow \subseteq \Gamma''_i$. Thus, we know that $(\Gamma, \Gamma'_i)\Uparrow \subseteq \Gamma\Uparrow, \Gamma''_i$. The induction hypothesis gives us $(\Gamma, \Gamma'_i)\Uparrow \vdash_{\mathsf{ext}} e_i\Uparrow : \tau\Uparrow$, and we are done by weakening (Lemma 1).

Case TM_LET: Straightforward use of the induction hypothesis and definition of $\Uparrow$.

Case TM_ANNOT: By induction and TYX_ANNOT.

Case LETS_DEC:

$$\frac{\begin{array}{l}\vdash \Gamma, \overline{\Gamma'} \\ \text{for each } \delta_i \in \overline{\delta} : \\ \quad \Gamma, \overline{\Gamma'} \vdash \delta_i \rightsquigarrow \Gamma'_i\end{array}}{\Gamma \vdash \overline{\delta} \rightsquigarrow \overline{\Gamma'}}\ \textsc{Lets\_Dec}$$

We wish to use TYLETS_DEC. We must show $\vdash_{\mathsf{ext}} (\Gamma, \overline{\Gamma'})\Uparrow$; this comes from the induction hypothesis. Now, fix i. We must show $(\Gamma, \overline{\Gamma'})\Uparrow \vdash_{\mathsf{ext}} \delta_i\Uparrow \rightsquigarrow \Gamma'_i\Uparrow$. Proceed by inversion on $\Gamma, \overline{\Gamma'} \vdash \delta_i \rightsquigarrow \Gamma'_i$:

Case LET_ANNOT:

$$\frac{\begin{array}{l}\text{for each } \overline{\pi}_i \mapsto e_i \in \overline{\overline{\pi} \mapsto e} : \\ \quad \Gamma \vdash \overline{\pi}_i \mapsto e_i : \sigma\end{array}}{\Gamma \vdash (x :: \sigma)\{\overline{\overline{\pi} \mapsto e}\} \rightsquigarrow x{:}\sigma}\ \textsc{Let\_Annot}$$

We must show $(\Gamma, \overline{\Gamma'})\Uparrow \vdash_{\mathsf{ext}} (x\Uparrow :: \sigma\Uparrow)\{\overline{\overline{\pi} \mapsto e}\Uparrow\} \rightsquigarrow \Gamma'_i$ We wish to use TYLET_ANNOT. Fix j. We must show $(\Gamma, \overline{\Gamma'})\Uparrow \vdash_{\mathsf{ext}} \overline{\pi}_j\Uparrow \mapsto e_j\Uparrow : \sigma\Uparrow$. We wish to use TYCL_CLAUSE. Rewrite σ to be $\forall \overline{\alpha}.\overline{\tau} \to \tau_0$. Thus, $\sigma\Uparrow = \forall \overline{\alpha}\Uparrow.\overline{\tau}\Uparrow \twoheadrightarrow \tau_0\Uparrow$. Then, we must show the following:

$\overline{\vdash \pi_{j\,k}\Uparrow : \tau_k\Uparrow \rightsquigarrow \Gamma_{1\,k}}$**:** Fix k. Inversion on $\Gamma, \overline{\Gamma'} \vdash \overline{\pi}_j \mapsto e_j : \forall \overline{\alpha}.\overline{\tau} \to \tau_0$ gives us $\vdash \pi_{j\,k} : \tau_k \rightsquigarrow \Gamma_{2\,k}$. We use Lemma 29 to get $\vdash \pi_{j\,k}\Uparrow : \tau_k\Uparrow \rightsquigarrow \Gamma_{1\,k}$ where $\Gamma_{2\,k}\Uparrow \subseteq \Gamma_{1\,k}$.

$\overline{(\Gamma, \overline{\Gamma'})\Uparrow^{\mathsf{ki}}, \Gamma_{1\,k} \vdash_{\mathsf{ext}} \pi_{j\,k}\Uparrow : \tau_k\Uparrow}$**:** From Lemma 29, noting that $\Gamma, \overline{\Gamma'} \vdash \tau_k : \star$ comes from inversion on CL_CLAUSE, and then applying Lemmas 3 and 26.

$(\Gamma, \overline{\Gamma'})\Uparrow, \overline{\alpha}\Uparrow, \overline{\Gamma_1} \vdash_{\mathsf{ext}} e_j\Uparrow : \tau_0\Uparrow$**:** Inversion gives us

$$\Gamma, \overline{\Gamma'}, \overline{\alpha{:}\star}, \overline{\Gamma_2} \vdash e_j : \tau_0$$

. The induction hypothesis then gives us (simplifying in the context) $(\Gamma, \overline{\Gamma'})\Uparrow, \overline{\alpha}\Uparrow, \overline{\Gamma_2}\Uparrow \vdash_{\mathsf{ext}} e_j\Uparrow : \tau_0\Uparrow$. By weakening $\overline{\Gamma_2}\Uparrow$ to $\overline{\Gamma_1}$, we get the desired judgement.

Case LET_NOANNOT: Similar to case above.

□

I. Reduction of extended types

Lemma 31 (Context reduction)**.** *If $\overline{\beta_i{:}\kappa_i}$ are the type variable bindings in Γ and $\vdash \Gamma$, then $fkv(\overline{\kappa}) \subset \Gamma$.*

Proof. By straightforward induction. □

Lemma 32 (Reduction substitution)**.** *If $\hat{\Gamma} \vdash_{\mathsf{ext}} \theta\ \mathbf{ok}$, then, for every α such that $\alpha \mapsto \tau \in \theta$, if $\hat{\Gamma} \vdash_{\mathsf{ext}} \alpha : \kappa$, then $\lfloor\hat{\Gamma}\rfloor \vdash \theta(\alpha) : \kappa$.*

Proof. Straightforward induction on $\hat{\Gamma} \vdash_{\mathsf{ext}} \theta\ \mathbf{ok}$. □

Lemma 33 (Type reduction preserves kind validity)**.**

1. *If $\hat{\Gamma} \vdash_{\mathsf{ext}} \kappa\ \mathbf{kind}$, then $\lfloor\hat{\Gamma}\rfloor \vdash \kappa\ \mathbf{kind}$.*
2. *If $\hat{\Gamma} \vdash_{\mathsf{ext\,sch}} \psi\ \mathbf{kind\ scheme}$, then $\lfloor\hat{\Gamma}\rfloor \vdash_{\mathsf{sch}} \psi\ \mathbf{kind\ scheme}$.*
3. *If $\vdash_{\mathsf{ext}} \hat{\Gamma}$, then $\vdash \lfloor\hat{\Gamma}\rfloor$.*

Proof. Straightforward induction, noting that $\lfloor\hat{\Gamma}\rfloor \subseteq \hat{\Gamma}$. □

Lemma 34 (Easy type reduction preserves kinds)**.** *If $\Gamma \vdash_{\mathsf{ext}} \tau : \kappa$, then $\Gamma \vdash \tau : \kappa$.*

Proof. Straightforward induction, appealing to Lemma 33 and noting that $\lfloor\Gamma\rfloor = \Gamma$. Note that this lemma covers proper types τ, not extended types $\hat{\tau}$, so the hard cases cannot happen. □

Lemma 35 (Type reduction preserves kinds)**.** *Suppose $\hat{\Gamma} \vdash_{\mathsf{ext}} \theta\ \mathbf{ok}$, and that θ maps every variable α such that $\alpha{:}\forall \mathcal{X}.\psi \in \hat{\Gamma}$.*

1. *If $\hat{\Gamma} \vdash_{\mathsf{ext}} \hat{\tau} : \kappa$, then $\lfloor\hat{\Gamma}\rfloor \vdash \lfloor\hat{\tau}\rfloor^{\theta}_{tvs(\lfloor\hat{\Gamma}\rfloor)} : \kappa$ and the emitted top-level declarations are well-kinded; that is, $\overline{\vdash dec \rightsquigarrow F : \Phi}$ for every dec (and some F and Φ) emitted during reduction.*
2. *If $\hat{\Gamma} \vdash_{\mathsf{ext}} \overline{\omega} \rightsquigarrow \hat{\Gamma}'$, then $\hat{\Gamma}, \hat{\Gamma}' \vdash_{\mathsf{ext}} \lfloor\overline{\omega}\rfloor^{\theta}_{tvs(\lfloor\hat{\Gamma}\rfloor)}\ \mathbf{ok}$.*
3. *If $\hat{\Gamma} \vdash_{\mathsf{ext}} (\alpha :: \psi)\{\overline{\overline{\tau} \mapsto \hat{\tau}'}\} \rightsquigarrow \alpha{:}\psi$, where $\psi = \forall \overline{\mathcal{X}}.\overline{\kappa} \twoheadrightarrow \kappa_0$, then $\vdash \lceil(\alpha :: \psi)\{\overline{\overline{\tau} \mapsto \hat{\tau}'}\}\rceil^{\theta}_{\overline{\beta}} \rightsquigarrow \mathit{Let}(\alpha) : \forall \overline{\mathcal{Y}}, \overline{\mathcal{X}}.\{\overline{\kappa}', \overline{\kappa}\}\kappa_0$, where $\overline{\beta_i{:}\kappa'_i}$ are the type variables in $\lfloor\hat{\Gamma}\rfloor$, $\overline{\mathcal{Y}} = fkv(\overline{\kappa}')$, and the length of $\overline{\kappa}$ is the same as the length of $\overline{\tau}$.*
4. *If $\hat{\Gamma} \vdash_{\mathsf{ext}} \alpha\{\overline{\overline{\tau} \mapsto \hat{\tau}'}\} \rightsquigarrow \alpha{:}\psi$, where $\psi = \forall \overline{\mathcal{X}}.\overline{\kappa} \twoheadrightarrow \kappa_0$, then $\vdash \lceil\alpha\{\overline{\overline{\tau} \mapsto \hat{\tau}'}\}\rceil^{\theta}_{\overline{\beta}} \rightsquigarrow \mathit{Let}(\alpha) : \forall \overline{\mathcal{Y}}, \overline{\mathcal{X}}.\{\overline{\kappa}', \overline{\kappa}\}\kappa_0$, where $\overline{\beta_i{:}\kappa'_i}$ are the type variables in $\lfloor\hat{\Gamma}\rfloor$, $\overline{\mathcal{Y}} = fkv(\overline{\kappa}')$, and the length of $\overline{\kappa}$ is the same as the length of $\overline{\tau}$.*

Proof. By induction.

Case TYX_VAR:

$$\frac{\begin{array}{l}\alpha{:}\forall \overline{\mathcal{X}}.\kappa_0 \in \hat{\Gamma} \qquad \vdash_{\mathsf{ext}} \hat{\Gamma} \\ \overline{\hat{\Gamma} \vdash_{\mathsf{ext}} \kappa_i\ \mathbf{kind}}\end{array}}{\hat{\Gamma} \vdash_{\mathsf{ext}} \alpha : \kappa_0[\overline{\kappa}/\overline{\mathcal{X}}]}\ \textsc{TyX\_Var}$$

We must show $\lfloor\hat{\Gamma}\rfloor \vdash \theta(\alpha) : \kappa_0[\overline{\kappa}/\overline{\mathcal{X}}]$. If θ maps α, then we are done by Lemma 32. Otherwise, let's consider the case when $\overline{\mathcal{X}}$ is non-empty. Here, we have a contradiction, because we assume that every such variable α is mapped in θ. So, $\overline{\mathcal{X}}$ must be empty. We use Lemma 33 to get $\vdash \lfloor\hat{\Gamma}\rfloor$ and we are done by TY_VAR.

Case TYX_ARROW: By induction and TY_ARROW.

Case TyX_App: By induction and Ty_App.
Case TyX_DataCon: By induction and Ty_DataCon.
Case TyX_TyCon: By induction and Ty_TyCon.
Case TyX_Annot: By induction and Ty_Annot.
Case TyX_Fam: By induction and Ty_Fam.
Case TyX_Abs:

$$\frac{\hat{\Gamma}, \alpha{:}\kappa_1 \vdash_{\mathsf{ext}} \hat{\tau} : \kappa_2}{\hat{\Gamma} \vdash_{\mathsf{ext}} \lambda\alpha \mapsto \hat{\tau} : \kappa_1 \twoheadrightarrow \kappa_2} \quad \text{TyX\_Abs}$$

We first show that $\vdash \mathbf{type}\, \mathsf{Lambda}\, \overline{\beta}\, \alpha = \lfloor \hat{\tau} \rfloor^{\theta}_{\overline{\beta},\alpha} \rightsquigarrow \mathsf{Lambda} : \forall \overline{\mathcal{X}}.\{\overline{\kappa}, \kappa_1\}\kappa_2$, where $\overline{\beta_i{:}\kappa_i}$ are the type variable bindings in $\lfloor \hat{\Gamma} \rfloor$ and $\overline{\mathcal{X}} = \mathit{fkv}(\overline{\kappa}, \kappa_1, \kappa_2)$.
It is straightforward to see that $\vdash \overline{\beta}\, \alpha \rightsquigarrow \overline{\kappa}, \kappa_1; \overline{\beta_i{:}\kappa_i}, \alpha{:}\kappa_1$; this comes directly from the (rather permissive) rules for checking type variable binders. We must now show $\overline{\mathcal{X}}, \overline{\beta_i{:}\kappa_i}, \alpha{:}\kappa_1 \vdash \lfloor \hat{\tau} \rfloor^{\theta}_{\overline{\beta},\alpha} : \kappa_2$. The induction hypothesis gives us $\lfloor \hat{\Gamma} \rfloor, \alpha{:}\kappa_1 \vdash \lfloor \hat{\tau} \rfloor^{\theta}_{\overline{\beta},\alpha} : \kappa_2$. We apply Lemma 5 to get the desired context above (noting that $\lfloor \hat{\Gamma} \rfloor$ may contain extra kind variables that do not appear elsewhere in the context), and we can conclude $\mathsf{Lambda} : \forall \overline{\mathcal{X}}.\{\overline{\kappa}, \kappa_1\}\kappa_2$.
We must show $\lfloor \hat{\Gamma} \rfloor \vdash \mathsf{Sym}_n(\mathsf{Lambda})\, \overline{\beta}^n : \kappa_1 \twoheadrightarrow \kappa_2$. Thus, by Lemma 25, $\mathsf{Sym}_n(\mathsf{Lambda}) : \overline{\kappa} \to \kappa_1 \twoheadrightarrow \kappa_2$ and thus $\lfloor \hat{\Gamma} \rfloor \vdash \mathsf{Sym}_n(\mathsf{Lambda})\, \overline{\beta}^n : \kappa_1 \twoheadrightarrow \kappa_2$ as desired.
Case TyX_Case:

$$\frac{\begin{array}{l} \hat{\Gamma} \vdash_{\mathsf{ext}} \hat{\tau}_0 : \kappa_0 \\ \text{for each } \tau_i \mapsto \hat{\tau}'_i \in \overline{\tau \mapsto \hat{\tau}'} : \\ \quad \vdash \tau_i : \kappa_0 \rightsquigarrow \Gamma'_i \\ \quad \hat{\Gamma}^{\mathsf{ki}}, \Gamma'_i \vdash_{\mathsf{ext}} \tau_i : \kappa_0 \\ \quad \hat{\Gamma}, \Gamma'_i \vdash_{\mathsf{ext}} \hat{\tau}'_i : \kappa \end{array}}{\hat{\Gamma} \vdash_{\mathsf{ext}} \mathbf{case}\, \hat{\tau}_0\, \mathbf{of}\, \overline{\tau \mapsto \hat{\tau}'} : \kappa} \quad \text{TyX\_Case}$$

We first show that $\vdash \mathbf{type\, family}\, \mathsf{Case}\, \overline{\beta}\, \alpha\, \mathbf{where}\, \lceil \overline{\tau \mapsto \hat{\tau}'} \rceil^{\theta}_{\overline{\beta}} \rightsquigarrow \mathsf{Case} : \forall \overline{\mathcal{X}}.\{\overline{\kappa}', \kappa_0\}\kappa$, where $\overline{\beta_j{:}\kappa'_j}$ are the type variable bindings in $\lfloor \hat{\Gamma} \rfloor$ and $\overline{\mathcal{X}} = \mathit{fkv}(\overline{\kappa}', \kappa_0, \kappa)$. To use Dec_TyFam, we just show $\mathsf{Case} \ni \lceil \tau_i \mapsto \hat{\tau}'_i \rceil^{\theta}_{\overline{\beta}} : \forall \overline{\mathcal{X}}.\{\overline{\kappa}', \kappa_0\}\kappa$ for all i. Fix i. We must show $\mathsf{Case} \ni \mathsf{Case}(\overline{\beta}, \tau_i) \mapsto \lfloor \hat{\tau}'_i \rfloor^{\theta}_{\overline{\beta}, \mathit{ftv}(\tau_i)} : \forall \overline{\mathcal{X}}.\{\overline{\kappa}', \kappa_0\}\kappa$. In turn, this requires $\overline{\Gamma_1 \vdash \beta_j : \kappa'_j}$, $\Gamma_1 \vdash \tau_i : \kappa_0$, and $\Gamma_1 \vdash \lfloor \hat{\tau}'_i \rfloor^{\theta}_{\overline{\beta}, \mathit{ftv}(\tau)} : \kappa$ for some Γ_1.
We know by inversion on the original typing judgement that $\hat{\Gamma}^{\mathsf{ki}}, \Gamma'_i \vdash_{\mathsf{ext}} \tau_i : \kappa_0$. We note that $\hat{\Gamma}^{\mathsf{ki}}$ is a proper context, not an extended one. So, we use Lemma 34 to get $\hat{\Gamma}^{\mathsf{ki}}, \Gamma'_i \vdash \tau_i : \kappa_0$. Inversion also gives us $\hat{\Gamma}, \Gamma'_i \vdash_{\mathsf{ext}} \hat{\tau}'_i : \kappa$. Thus, the induction hypothesis yields $\lfloor \hat{\Gamma} \rfloor, \Gamma'_i \vdash \lfloor \hat{\tau}'_i \rfloor^{\theta}_{\overline{\beta}, \mathit{ftv}(\tau_i)} : \kappa_0$, noting that $\mathit{tvs}(\Gamma'_i) = \mathit{ftv}(\tau_i)$ by Lemma 14.
Putting all this together, we can choose $\Gamma_1 = \lfloor \hat{\Gamma} \rfloor, \Gamma'_i$, and we have established $\mathsf{Case} : \forall \overline{\mathcal{X}}.\{\overline{\kappa}', \kappa_0\}\kappa$.
We must show $\lfloor \hat{\Gamma} \rfloor \vdash \mathsf{Case}(\overline{\beta}, \lfloor \hat{\tau}_0 \rfloor^{\theta}_{\overline{\beta}}) : \kappa$. Inversion on the original typing judgement gives us $\hat{\Gamma} \vdash_{\mathsf{ext}} \hat{\tau}_0 : \kappa_0$. Then, the induction hypothesis yields $\lfloor \hat{\Gamma} \rfloor \vdash \lfloor \hat{\tau}_0 \rfloor^{\theta}_{\overline{\beta}} : \kappa_0$. We use Lemma 19 to get $\vdash \lfloor \hat{\Gamma} \rfloor$, and then we are done by Ty_Fam. (The substitution in that rule is chosen to be the identity operation. That is $\overline{\kappa}' = \overline{\mathcal{X}}$.)
Case TyX_Let:

$$\frac{\begin{array}{c} \hat{\Gamma} \vdash_{\mathsf{ext}} \overline{\omega} \rightsquigarrow \hat{\Gamma}' \\ \hat{\Gamma}, \hat{\Gamma}' \vdash_{\mathsf{ext}} \hat{\tau} : \kappa \end{array}}{\hat{\Gamma} \vdash_{\mathsf{ext}} \mathbf{let}\, \overline{\omega}\, \mathbf{in}\, \hat{\tau} : \kappa} \quad \text{TyX\_Let}$$

We use the induction hypotheses (3) and (4) to show that the produced declarations are well-typed. To do so, we must first show that $\hat{\Gamma} \vdash_{\mathsf{ext}} \theta, \theta'\ \mathbf{ok}$, which comes from induction hypothesis (2) and Lemma 10.
Now, we must show $\lfloor \hat{\Gamma} \rfloor \vdash \lfloor \hat{\tau} \rfloor^{\theta,\theta'}_{\overline{\beta}} : \kappa$, where we know $\hat{\Gamma}, \overline{\hat{\Gamma}'} \vdash_{\mathsf{ext}} \hat{\tau} : \kappa$. We note that $\lfloor \overline{\hat{\Gamma}'} \rfloor = \varnothing$ and that $\hat{\Gamma}, \overline{\hat{\Gamma}'} \vdash_{\mathsf{ext}} \theta, \theta'\ \mathbf{ok}$ as shown above. We apply the induction hypothesis and we are done.
Case TyLets_Dec:

$$\frac{\begin{array}{l} \vdash_{\mathsf{ext}} \hat{\Gamma}, \overline{\hat{\Gamma}'} \\ \text{for each } \omega_i \in \overline{\omega} : \\ \quad \hat{\Gamma}, \overline{\hat{\Gamma}'} \vdash_{\mathsf{ext}} \omega_i \rightsquigarrow \hat{\Gamma}'_i \end{array}}{\hat{\Gamma} \vdash_{\mathsf{ext}} \overline{\omega} \rightsquigarrow \overline{\hat{\Gamma}'}} \quad \text{TyLets\_Dec}$$

We must show that $\hat{\Gamma}, \hat{\Gamma}' \vdash_{\mathsf{ext}} \lfloor \overline{\omega} \rfloor^{\theta}_{\overline{\beta}}\ \mathbf{ok}$, where $\overline{\beta_i{:}\kappa'_i}$ are the type variable bindings in $\lfloor \hat{\Gamma} \rfloor$. We appeal to Lemma 10 to discover that we must show, for any i, that $\hat{\Gamma}, \hat{\Gamma}' \vdash_{\mathsf{ext}} \lfloor \omega_i \rfloor^{\theta}_{\overline{\beta}}\ \mathbf{ok}$. We have two cases:
Case TyLet_Annot:

$$\frac{\begin{array}{l} \text{for each } \overline{\tau}_i \mapsto \hat{\tau}'_i \in \overline{\overline{\tau} \mapsto \hat{\tau}'} : \\ \quad \hat{\Gamma} \vdash_{\mathsf{ext}} \overline{\tau}_i \mapsto \hat{\tau}'_i : \psi \end{array}}{\hat{\Gamma} \vdash_{\mathsf{ext}} (\alpha :: \psi)\{\overline{\overline{\tau} \mapsto \hat{\tau}'}\} \rightsquigarrow \alpha{:}\psi} \quad \text{TyLet\_Annot}$$

We must show $\hat{\Gamma}, \hat{\Gamma}' \vdash_{\mathsf{ext}} \alpha \mapsto \mathsf{Sym}_n(\mathsf{Let}(\alpha))\, \overline{\beta}^n\ \mathbf{ok}$. Rewrite ψ as $\forall \overline{\mathcal{X}}^m.\overline{\kappa} \twoheadrightarrow \kappa_0$ and choose arbitrary well-sorted $\overline{\kappa}''^m$. We see that $\hat{\Gamma}, \hat{\Gamma}' \vdash_{\mathsf{ext}} \alpha : \kappa[\overline{\kappa}''/\overline{\mathcal{X}}]$. We must show that $\hat{\Gamma}, \hat{\Gamma}' \vdash_{\mathsf{ext}} \mathsf{Sym}_n(\mathsf{Let}(\alpha))\, \overline{\beta} : (\overline{\kappa} \twoheadrightarrow \kappa_0)[\overline{\kappa}''/\overline{\mathcal{X}}]$. The induction hypothesis (3) tells us $\mathsf{Let}(\alpha) : \forall \overline{\mathcal{Y}}, \overline{\mathcal{X}}.\{\overline{\kappa}', \overline{\kappa}\}\kappa_0$, where $\overline{\mathcal{Y}} = \mathit{fkv}(\overline{\kappa}')$. Lemma 25 then tells us that $\mathsf{Sym}_n(\mathsf{Let}(\alpha)) : \forall \overline{\mathcal{Y}}, \overline{\mathcal{X}}.\overline{\kappa}' \to \overline{\kappa} \twoheadrightarrow \kappa_0$. Thus, we can conclude that $\hat{\Gamma}, \hat{\Gamma}' \vdash_{\mathsf{ext}} \mathsf{Sym}_n(\mathsf{Let}(\alpha))\, \overline{\beta} : (\overline{\kappa} \twoheadrightarrow \kappa_0)[\overline{\kappa}''/\overline{\mathcal{X}}]$ as desired, noting that the variables $\overline{\mathcal{Y}}$ do not appear in $(\overline{\kappa} \twoheadrightarrow \kappa_0)$.
Case TyLet_NoAnnot: Similar to above case.
Case TyLet_Annot:

$$\frac{\begin{array}{l} \text{for each } \overline{\tau}_i \mapsto \hat{\tau}'_i \in \overline{\overline{\tau} \mapsto \hat{\tau}'} : \\ \quad \hat{\Gamma} \vdash_{\mathsf{ext}} \overline{\tau}_i \mapsto \hat{\tau}'_i : \psi \end{array}}{\hat{\Gamma} \vdash_{\mathsf{ext}} (\alpha :: \psi)\{\overline{\overline{\tau} \mapsto \hat{\tau}'}\} \rightsquigarrow \alpha{:}\psi} \quad \text{TyLet\_Annot}$$

We must show that $\vdash \lceil (\alpha :: \psi)\{\overline{\overline{\tau} \mapsto \hat{\tau}'}\} \rceil^{\theta}_{\overline{\beta}} \rightsquigarrow \mathsf{Let}(\alpha) : \forall \overline{\mathcal{Y}}, \overline{\mathcal{X}}.\{\overline{\kappa}', \overline{\kappa}\}\kappa_0$, where $\psi = \forall \overline{\mathcal{X}}.\overline{\kappa} \twoheadrightarrow \kappa_0$, $\overline{\beta_i{:}\kappa'_i}$ are the type variables in $\lfloor \hat{\Gamma} \rfloor$, $\overline{\mathcal{Y}} = \mathit{fkv}(\overline{\kappa}')$, and the length of $\overline{\kappa}$ is the same as the length of $\overline{\tau}$.
Simplifying the declaration gives us

$$\mathbf{type\, family}\, \mathsf{Let}(\alpha)\, \overline{\beta}\, \overline{(\beta'_j :: \kappa_j)}\, \mathbf{where}\, {}_{\alpha}\lceil \overline{\overline{\tau} \mapsto \hat{\tau}'} \rceil^{\theta,\theta'}_{\overline{\beta}}$$

with fresh $\overline{\beta}'$. We can see that $\vdash \overline{\beta}\, \overline{(\beta'_j :: \kappa_j)} \rightsquigarrow \overline{\kappa}', \overline{\kappa}; \Gamma_0$. To use Dec_TyFam, we must only show that, for all k, $\mathsf{Let}(\alpha) \ni {}_{\alpha}\lceil \overline{\tau}_k \mapsto \hat{\tau}'_k \rceil^{\theta}_{\overline{\beta}} : \forall \overline{\mathcal{X}}.\{\overline{\kappa}', \overline{\kappa}\}\kappa_0$.
Fix k. We reduce the equation to be $\mathsf{Let}(\alpha)(\overline{\beta}, \overline{\tau}_k) \mapsto \lfloor \hat{\tau}'_k \rfloor^{\theta}_{\overline{\beta}, \mathit{ftv}(\overline{\tau}_k)}$.
Thus, we have to show, for all m, $\overline{\Gamma_1 \vdash \beta_m : \kappa'_m}$ for all j, $\Gamma_1 \vdash \tau_{k\,j} : \kappa_j$, and $\Gamma_1 \vdash \lfloor \hat{\tau}'_k \rfloor^{\theta}_{\overline{\beta}, \mathit{ftv}(\overline{\tau}_k)} : \kappa_{i\,0}$ for some Γ_1. The first two sets of these judgements are derived much like in the **case** case, noting that we get $(\hat{\Gamma}, \overline{\hat{\Gamma}'})^{\mathsf{ki}}, \Gamma''_{k\,j} \vdash \tau_{k\,j} : \kappa_j$ from inversion on TyCl_Clause.
For the third judgement, we know $\hat{\Gamma}, \overline{\hat{\Gamma}'}, \overline{\mathcal{X}}', \overline{\Gamma''_k} \vdash_{\mathsf{ext}} \hat{\tau}'_k : \kappa_0$. We wish to use the induction hypothesis. Our goal mentions $\lfloor \hat{\tau}'_k \rfloor^{\theta}_{\overline{\beta}, \mathit{ftv}(\overline{\tau}_k)}$. Is list of variables in the reduction correct? Yes.

We note that $\lfloor \overline{\hat{\Gamma}'} \rfloor$ is empty, because $\overline{\hat{\Gamma}'}$ contains only bindings of the form $\alpha{:}\psi$, as it is formed from the definitions in the **let**. (Here, we assume that we can tell the difference between $\forall\,\varnothing.\kappa$ and κ.) We get no type variables from $\overline{\mathcal{X}}'$. And, from Lemma 14, we know that $tvs(\overline{\Gamma''_k}) = ftv(\overline{\tau}_k)$ as desired. Thus, we use the induction hypothesis to get $\lfloor\hat{\Gamma}\rfloor, \overline{\mathcal{X}}', \overline{\Gamma''_k} \vdash \lfloor \hat{\tau}'_k \rfloor^{\theta}_{\overline{\beta}, ftv(\overline{\tau}_k)} : \kappa_0$. We choose $\Gamma_1 = \lfloor\hat{\Gamma}\rfloor, \overline{\mathcal{X}}', \overline{\Gamma''_k}$ and we can apply TF_EQN and thus DEC_TYFAM. We can conclude that the declaration is well-typed, as desired.

Case TYLET_NOANNOT: Similar to case above.

□

J. Promotion is well-typed

Lemma 36 (Top-level substitutions are valid)**.** *If θ_0 is a top-level substitution associated with Γ_0, then $\Gamma_0\Uparrow \vdash_{\mathsf{ext}} \theta_0$ $\mathbf{ok}$.*

Proof. Direct from Lemma 35. □

Theorem 37 (Promotion is well-typed)**.** *Let Γ_0 and θ_0 be a top-level context and its associated substitution. If $\Gamma_0 \vdash e : \tau$, where $e\Uparrow$ and $\tau\Uparrow$ exist, then $\varnothing \vdash \lfloor e\Uparrow \rfloor^{\theta_0}_{\varnothing} : \tau\Uparrow$.*

Proof. Direct from Lemmas 30 and 35, using Lemma 36. □

The θ_0 in the above proof takes every function to its promoted equivalent. It is roughly the "make uppercase identifier" function.

K. Promotion preserves semantics

Theorem 38 (Promotion preserves semantics)**.** *If $\Sigma; e \longrightarrow \Sigma'; e'$ and if $e\Uparrow$ exists, then $\Sigma\Uparrow; e\Uparrow \longrightarrow_{\mathsf{ext}} \Sigma'\Uparrow; e'\Uparrow$.*

Proof. By straightforward induction, appealing to Lemma 21. □