System FC, as implemented in GHC

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

Let us know how access to this document benefits you.

Follow this and additional works at: http://repository.brynmawr.edu/compsci_pubs

Part of the Computer Sciences Commons

Custom Citation

This paper is posted at Scholarship, Research, and Creative Work at Bryn Mawr College. http://repository.brynmawr.edu/compsci_pubs/15

For more information, please contact repository@brynmawr.edu.
1 Introduction

This document presents the typing system of System FC, very closely to how it is implemented in GHC. Care is taken to include only those checks that are actually written in the GHC code. It should be maintained along with any changes to this type system.

Who will use this? Any implementer of GHC who wants to understand more about the type system can look here to see the relationships among constructors and the different types used in the implementation of the type system. Note that the type system here is quite different from that of Haskell—these are the details of the internal language, only.

At the end of this document is a hypothetical operational semantics for GHC. It is hypothetical because GHC does not strictly implement a concrete operational semantics anywhere in its code. While all the typing rules can be traced back to lines of real code, the operational semantics do not, in general, have as clear a provenance.

There are a number of details elided from this presentation. The goal of the formalism is to aid in reasoning about type safety, and checks that do not work toward this goal were omitted. For example, various scoping checks (other than basic context inclusion) appear in the GHC code but not here.

2 Grammar

2.1 Metavariables

We will use the following metavariables:

- \( x, c \) Term-level variable names
- \( \alpha, \beta \) Type-level variable names
- \( N \) Type-level constructor names
- \( M \) Axiom rule names
- \( i, j, k, a, b, c \) Indices to be used in lists

2.2 Literals

Literals do not play a major role, so we leave them abstract:

\[
\text{lit} ::= \text{Literals, basicTypes/Literal.lhs:Literal}
\]

We also leave abstract the function \( \text{basicTypes/Literal.lhs:literalType} \) and the judgment \( \text{coreSyn/CoreLint.lhs:lintTyLit} \) (written \( \Gamma \vdash \text{tylit lit} : \kappa \)).

2.3 Variables

GHC uses the same datatype to represent term-level variables and type-level variables:

\[
z ::= \text{Term or type name}
\]

\[
| \alpha \quad \text{Type-level name}
\]

---

\(^1\)This document was originally prepared by Richard Eisenberg (eir@cis.upenn.edu), but it should be maintained by anyone who edits the functions or data structures mentioned in this file. Please feel free to contact Richard for more information.
Variable names, basicTypes/Var.lhs:Var

2.4 Expressions

The datatype that represents expressions:

\[
\begin{align*}
\mathit{e, u} & ::= & \text{Expressions, coreSyn/CoreSyn.lhs:Expr} \\
\mid & \mathit{n} & \text{Var: Variable} \\
\mid & \mathit{\text{lit}} & \text{Lit: Literal} \\
\mid & \mathit{e_1 e_2} & \text{App: Application} \\
\mid & \lambda n . e & \text{Lam: Abstraction} \\
\mid & \text{let binding in } e & \text{Let: Variable binding} \\
\mid & \text{case } e \text{ as } n \text{ return } \tau \text{ of } \mathit{\text{alt}_i^i} & \text{Case: Pattern match} \\
\mid & e \triangleright \gamma & \text{Cast: Cast} \\
\mid & e \{\text{tick}\} & \text{Tick: Internal note} \\
\mid & \tau & \text{Type: Type} \\
\mid & \gamma & \text{Coercion: Coercion}
\end{align*}
\]

There are a few key invariants about expressions:

- The right-hand sides of all top-level and recursive \texttt{let}s must be of lifted type.
- The right-hand side of a non-recursive \texttt{let} and the argument of an application may be of unlifted type, but only if the expression is ok-for-speculation. See \texttt{#let_app_invariant#} in coreSyn/CoreSyn.lhs.
- We allow a non-recursive \texttt{let} for bind a type variable.
- The case for a \texttt{case} must come first.
- The list of case alternatives must be exhaustive.
- Types and coercions can only appear on the right-hand-side of an application.
- The \(\tau\) form of an expression must not then turn out to be a coercion. In other words, the payload inside of a \texttt{Type} constructor must not turn out to be built with \texttt{CoercionTy}.

Bindings for \texttt{let} statements:

\[
\begin{align*}
\mathit{binding} & ::= & \text{Let-bindings, coreSyn/CoreSyn.lhs:Bind} \\
\mid & \mathit{n = e} & \text{NonRec: Non-recursive binding} \\
\mid & \text{rec } \mathit{n_i := e_i}^i & \text{Rec: Recursive binding}
\end{align*}
\]

Case alternatives:

\[
\begin{align*}
\mathit{alt} & ::= & \text{Case alternative, coreSyn/CoreSyn.lhs:Alt} \\
\mid & \mathit{K }\pi^i \rightarrow e & \text{Constructor applied to fresh names}
\end{align*}
\]

Constructors as used in patterns:

\[
\begin{align*}
\mathit{K} & ::= & \text{Constructors used in patterns, coreSyn/CoreSyn.lhs:AltCon} \\
\mid & K & \text{DataAlt: Data constructor} \\
\mid & \mathit{\text{lit}} & \text{LitAlt: Literal (such as an integer or character)}
\end{align*}
\]
Notes that can be inserted into the AST. We leave these abstract:

\[ \text{tick} ::= \text{Internal notes, coreSyn/CoreSyn.lhs:Tickish} \]

A program is just a list of bindings:

\[ \text{program} ::= \text{A System FC program, coreSyn/CoreSyn.lhs:CoreProgram} \]

\[ \text{binding, } ^i \text{ List of bindings} \]

2.5 Types

\[ \tau, \kappa, \sigma, \phi ::= \text{Types/kinds, types/TyCoRep.lhs:Type} \]

\[ n \quad \text{TyVarTy: Variable} \]

\[ \tau_1 \tau_2 \quad \text{AppTy: Application} \]

\[ T \tau_1 \quad \text{TyConApp: Application of type constructor} \]

\[ \tau_1 \rightarrow \tau_2 \quad \text{ForAllTy (Anon ...) ...: Function} \]

\[ \forall n. \tau \quad \text{ForAllTy (Named ...) ...: Type and coercion polymorphism} \]

\[ \text{lit} \quad \text{LitTy: Type-level literal} \]

\[ \tau \triangleright \gamma \quad \text{CastTy: Kind cast} \]

\[ \gamma \quad \text{CoercionTy: Coercion used in type} \]

ForAllTys are represented in two different ways, depending on whether the ForAllTy is anonymous (written \( \tau_1 \rightarrow \tau_2 \)) or named (written \( \forall n. \tau \)).

There are some invariants on types:

- The name used in a type must be a type-level name (TyVar).
- The type \( \tau_1 \) in the form \( \tau_1 \tau_2 \) must not be a type constructor \( T \). It should be another application or a type variable.
- The form \( T \tau_1 \) (TyConApp) does not need to be saturated.
- A saturated application of \( (\rightarrow) \tau_1 \tau_2 \) should be represented as \( \tau_1 \rightarrow \tau_2 \). This is a different point in the grammar, not just pretty-printing. The constructor for a saturated \( (\rightarrow) \) is ForAllTy.
- A type-level literal is represented in GHC with a different datatype than a term-level literal, but we are ignoring this distinction here.
- A coercion used as a type should appear only in the right-hand side of an application.

Note that the use of the \( T \tau_1 \) form and the \( \tau_1 \rightarrow \tau_2 \) form are purely representational. The metatheory would remain the same if these forms were removed in favor of \( \tau_1 \tau_2 \). Nevertheless, we keep all three forms in this documentation to accurately reflect the implementation.

The Named variant of a Binder (the first argument to a ForAllTy) also tracks visibility of arguments. Visibility affects only source Haskell, and is omitted from this presentation.

We use the notation \( \tau_1 \sim_{\#} \tau_2 \) to stand for \( (\sim_{\#}) \kappa_1 \kappa_2 \tau_1 \tau_2 \).

2.6 Coercions

\[ \gamma, \eta ::= \text{Coercions, types/TyCoRep.lhs:Coercion} \]

\[ (\tau)_\rho \quad \text{Refl: Reflexivity} \]
\[ T_\rho \gamma_1 \gamma_2 \]

TyConAppCo: Type constructor application

\[ \forall \eta \gamma \]

AppCo: Application

\[ n \]

ForAllCo: Polymorphism

\[ C \text{ind} \gamma_1 \]

CoVarCo: Variable

\[ \text{prov} \langle \tau_1, \tau_2 \rangle_\eta \]

UnivCo: Universal coercion

\[ \text{sym} \gamma \]

SymCo: Symmetry

\[ \gamma_1 \gamma_2 \]

TransCo: Transitivity

\[ \mu \tau_1 \tau_2 \]

AxiomRuleCo: Axiom-rule application (for type-nats)

\[ \text{nth} \gamma \]

NthCo: Projection (0-indexed)

\[ \gamma \hat{\eta} \]

InstCo: Instantiation

\[ \gamma \bowtie \eta \]

CoherenceCo: Coherence

\[ \text{kind} \gamma \]

KindCo: Kind extraction

\[ \text{sub} \gamma \]

SubCo: Sub-role — convert nominal to representational

Invariants on coercions:

- \( \langle \tau_1 \tau_2 \rangle_\rho \) is used; never \( \langle \tau_1 \rangle_\rho \langle \tau_2 \rangle_\eta \).

- If \( \langle T \rangle_\rho \) is applied to some coercions, at least one of which is not reflexive, use \( T_\rho \gamma_1 \gamma_2 \ldots \).

- The \( T \) in \( T_\rho \gamma_1 \gamma_2 \) is never a type synonym, though it could be a type function.

- Every non-reflexive coercion coerces between two distinct types.

- The name in a coercion must be a term-level name (\text{Id}).

- The contents of \( \langle \tau \rangle_\rho \) must not be a coercion. In other words, the payload in a \text{Refl} must not be built with \text{CoercionTy}.

The \text{UnivCo} constructor takes several arguments: the two types coerced between, a coercion relating these types' kinds, a role for the universal coercion, and a provenance. The provenance states what created the universal coercion:

\[
\text{prov} ::= \text{UnivCo provenance, types/TyCoRep.lhs:UnivCoProvenance}
\]

| \text{unsafe} From \text{unsafeCoerce#} |
| \text{phant} From the need for a phantom coercion |
| \text{irrel} From proof irrelevance |

Roles label what equality relation a coercion is a witness of. Nominal equality means that two types are identical (have the same name); representational equality means that two types have the same representation (introduced by newtypes); and phantom equality includes all types. See [http://ghc.haskell.org/trac/ghc/wiki/Roles](http://ghc.haskell.org/trac/ghc/wiki/Roles) and [http://research.microsoft.com/en-us/um/people/simonpj/papers/ext-f/coercible.pdf](http://research.microsoft.com/en-us/um/people/simonpj/papers/ext-f/coercible.pdf) for more background.

\[
\rho ::= \text{Roles, types/CoAxiom.lhs:Role}
\]

| N Nominal |
| R Representational |
| P Phantom |

Is it a left projection or a right projection?

\[
\text{LorR} ::= \text{left or right deconstructor, types/TyCoRep.lhs:LeftOrRight}
\]

| left CLeft: Left projection |
| right CRight: Right projection |
Axioms:

\[ C ::= \text{Axioms}, \quad \text{types/TyCon.lhs:CoAxiom} \]
\[ T_p \overset{\text{CoAxiom}}{\to} \quad \text{CoAxiom: Axiom} \]

\[ axBranch, \ axBranch \cdot' ::= \text{Axiom branches}, \quad \text{types/TyCon.lhs:CoAxBranch} \]
\[ \forall \rho_i. (\tau_j \to \sigma) \overset{\text{CoAxBranch}}{\to} \quad \text{CoAxBranch: Axiom branch} \]

The left-hand sides \( \tau_j \) of different branches of one axiom must all have the same length.

The definition for \( axBranch \) above does not include the list of incompatible branches (field \text{cab_incomps} of \text{CoAxBranch}), as that would unduly clutter this presentation. Instead, as the list of incompatible branches can be computed at any time, it is checked for in the judgment \text{no_conflict}. See Section 4.16.

Axiom rules, produced by the type-nats solver:

\[ \mu ::= \text{CoAxiomRules}, \quad \text{types/CoAxiom.lhs:CoAxiomRule} \]
\[ M(i, \tau_j, \rho') \overset{\text{CoAxiomRule}}{\to} \quad \text{Named rule, with parameter info} \]

An axiom rule \( \mu = M(i, \tau_j, \rho') \) is an axiom name \( M \), with a type arity \( i \), a list of roles \( \tau_j \) for its coercion parameters, and an output role \( \rho' \). The definition within GHC also includes a field named \text{coaxrProves} which computes the output coercion from a list of types and a list of coercions. This is elided in this presentation, as we simply identify axiom rules by their names \( M \). See also \text{typecheck/TcTypeNats.lhs:mkBinAxiom} and \text{typecheck/TcTypeNats.lhs:mkAxiom1}.

In \text{CO_UNIVCO}, function \text{compatibleUnBoxedTys} stands for following checks:

- both types are unboxed;
- types should have same size;
- both types should be either integral or floating;
- coercion between vector types are not allowed;
- unboxed tuples should have same length and each element should be coercible to appropriate element of the target tuple;

For function implementation see \text{coreSyn/CoreLint.lhs:checkTypes}. For further discussion see \url{https://ghc.haskell.org/trac/ghc/wiki/BadUnsafeCoercions}.

2.7 Type constructors

Type constructors in GHC contain lots of information. We leave most of it out for this formalism:

\[ T ::= \text{Type constructors, types/TyCon.lhs:TyCon} \]
\[ \to \quad \text{FunTyCon: Arrow} \]
\[ N^\kappa \quad \text{AlgTyCon, TupleTyCon, SynTyCon: algebraic, tuples, families, and synonyms} \]
\[ H \quad \text{PrimTyCon: Primitive tycon} \]
\[ 'K \quad \text{PromotedDataCon: Promoted data constructor} \]

We include some representative primitive type constructors. There are many more in \text{prelude/TysPrim.lhs}.

\[ H ::= \text{Primitive type constructors, prelude/TysPrim.lhs} \]
\[ \text{Int}_# \quad \text{Unboxed Int (intPrimTyCon)} \]
\[ (\sim_#) \quad \text{Unboxed equality (eqPrimTyCon)} \]
Note that although GHC contains distinct type constructors \( \mathord{*} \) and \( \textit{Constraint} \), this formalism treats only \( \mathord{*} \). These two type constructors are considered wholly equivalent. In particular the function \( \text{eqType} \) returns \( \text{True} \) when comparing \( \mathord{*} \) and \( \textit{Constraint} \). We need them both because they serve different functions in source Haskell.

**TYPE**  The type system is rooted at the special constant **TYPE** and the (quite normal) datatype \( \textbf{data} \) \textit{Levity} = \textit{Lifted} | \textit{Unlifted} . The type of **TYPE** is \( \textit{Levity} \rightarrow \text{**TYPE**`Lifted} \). The idea is that **TYPE`Lifted** classifies lifted types and **TYPE`Unlifted** classifies unlifted types. Indeed \( \mathord{*} \) is just a plain old type synonym for **TYPE`Lifted**, and \( \# \) is just a plain old type synonym for **TYPE`Unlifted**.

### 3 Contexts

The functions in \texttt{coreSyn/CoreLint.lhs} use the \texttt{LintM} monad. This monad contains a context with a set of bound variables \( \Gamma \). The formalism treats \( \Gamma \) as an ordered list, but GHC uses a set as its representation.

\[
\Gamma ::= \text{List of bindings, } \text{coreSyn/CoreLint.lhs:LintM} \\
| n \quad \text{Single binding} \\
| \Gamma_i \quad \text{Context concatenation}
\]

We assume the Barendregt variable convention that all new variables are fresh in the context. In the implementation, of course, some work is done to guarantee this freshness. In particular, adding a new type variable to the context sometimes requires creating a new, fresh variable name and then applying a substitution. We elide these details in this formalism, but see \texttt{types/Type.lhs:substTyVarBndr} for details.

### 4 Typing judgments

The following functions are used from GHC. Their names are descriptive, and they are not formalized here: \texttt{types/TyCon.lhs:tyConKind}, \texttt{types/TyCon.lhs:tyConArity}, \texttt{basicTypes/DataCon.lhs:dataConTyCon}, \texttt{types/TyCon.lhs:isNewTyCon}, \texttt{basicTypes/DataCon.lhs:dataConRepType}.

#### 4.1 Program consistency

Check the entire bindings list in a context including the whole list. We extract the actual variables (with their types/kinds) from the bindings, check for duplicates, and then check each binding.

\[
\proves{\textbf{prog} \textit{program}} \quad \text{Program typing, coreSyn/CoreLint.lhs:lintCoreBindings}
\]

\[
\Gamma = \frac{\vars{of} \textit{binding}_i}{\text{no_duplicates} \ \textit{binding}_i} \\
\frac{\Gamma \vdash \textit{bind} \ \textit{binding}_i}{\ddouble{\textbf{prog} \textit{binding}_i}} \quad \text{PROG\_COREBINDINGS}
\]
Here is the definition of `vars_of`, taken from `coreSyn/CoreSyn.lhs:bindersOf`:

\[
\begin{align*}
\text{vars_of } n &= e \\
\text{vars_of } \text{rec } n_i &= e_i
\end{align*}
\]

### 4.2 Binding consistency

\[\Gamma \vdash \text{bind} n \leftarrow e\]

<table>
<thead>
<tr>
<th>[\Gamma \vdash \text{bind} n \leftarrow e]</th>
<th>[\Gamma \vdash n = e]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binding_NonRec</td>
<td></td>
</tr>
</tbody>
</table>

\[\Gamma \vdash \text{bind} n_i \leftarrow e_i\]

<table>
<thead>
<tr>
<th>[\Gamma \vdash \text{bind} n_i \leftarrow e_i]</th>
<th>[\Gamma \vdash \text{rec } n_i = e_i]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binding_Rec</td>
<td></td>
</tr>
</tbody>
</table>

### 4.3 Expression typing

\[\Gamma \vdash \text{tm } e : \tau\]

<table>
<thead>
<tr>
<th>[\Gamma \vdash \text{tm } e : \tau]</th>
<th>[\tau = \text{literalType lit}]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tm_VAR</td>
<td>Tm_LIT</td>
</tr>
</tbody>
</table>

\[\Gamma \vdash \text{tm } e : \sigma\]

<table>
<thead>
<tr>
<th>[\Gamma \vdash \text{tm } e : \sigma]</th>
<th>[\Gamma \vdash \text{co } \gamma : \sigma^{\kappa_1 \sim \kappa_2} \tau]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tm_Cast</td>
<td></td>
</tr>
</tbody>
</table>

In the GHC source, this function contains a number of other checks, such as for strictness and exportability. See the source code for further information.
\[ \phi = \sigma_1^{\kappa_1 \sim \#_2} \sigma_2 \]
\[ \Gamma \vdash_\text{Tm} \phi \text{ ok} \]
\[ \Gamma, e^\phi \vdash_\text{Tm} e : \tau \]
\[ \Gamma \vdash_\text{Tm} \lambda e^\phi. e : \forall e^\phi. \tau \]

**TM_LAMCO**

\[ \Gamma \vdash_\text{Tm} e : \sigma \]
\[ \sigma = \ast \lor \sigma = \# \]
\[ \Gamma \vdash_\text{ty} \tau : \text{TYPE} \sigma_2 \]
\[ \Gamma, z^\sigma ; \sigma \vdash_\text{th} \text{alt}_i : \tau_i \]
\[ \Gamma \vdash_\text{Tm} \text{case} e \text{ as } z^\sigma \text{ return } \tau \text{ of } \text{alt}_i \]
\[ \Gamma \vdash_\text{co} \gamma : \tau_1^{\kappa_1 \sim \kappa_2} \tau_2 \]
\[ \Gamma \vdash_\text{Tm} \gamma : \tau_1^{\kappa_1 \sim \kappa_2} \_N \tau_2 \]

**TM_COERCION**

\[ \Gamma \vdash_\text{co} \gamma : \tau_1^{\kappa_1 \sim \kappa_2} \_ \# \tau_2 \]
\[ \Gamma \vdash_\text{Tm} \gamma : (\sim_{\#_2}) \kappa_1 \kappa_2 \tau_1 \tau_2 \]

**TM_COERCIONREP**

- Some explication of TM_LETREC is helpful: The idea behind the second premise \((\Gamma, \Gamma_i^{\prime} \vdash_\text{ty} \sigma_i : \kappa_i^{-1})\) is that we wish to check each substituted type \(\sigma_i^\prime\) in a context containing all the types that come before it in the list of bindings. The \(\Gamma_i^{\prime}\) are contexts containing the names and kinds of all type variables (and term variables, for that matter) up to the \(i\)th binding. This logic is extracted from coreSyn/CoreLint.lhs:lintAndScopeIds.
- The GHC source code checks all arguments in an application expression all at once using coreSyn/CoreSyn.lhs:collectArgs and coreSyn/CoreLint.lhs:lintCoreArgs. The operation has been unfolded for presentation here.
- If a tick contains breakpoints, the GHC source performs additional (scoping) checks.
- The rule for case statements also checks to make sure that the alternatives in the case are well-formed with respect to the invariants listed above. These invariants do not affect the type or evaluation of the expression, so the check is omitted here.
- The GHC source code for TM_VAR contains checks for a dead id and for one-tuples. These checks are omitted here.

### 4.4 Kinding

\[ \Gamma \vdash_\text{ty} \tau : \kappa \]

Kinding, coreSyn/CoreLint.lhs:lintType

\[ z^\kappa \in \Gamma \]
\[ \Gamma \vdash_\text{ty} z^\kappa : \kappa \]

**TY_TYVARTY**
4.5 Kind validity

![Formulas and proofs related to kind validity and type inference.]

Kind validity, coreSyn/CoreLint.lhs:lintKind

Γ ⊩ k κ ok  K_STAR

Γ ⊩ ty τ 1 : κ 1
Γ ⊩ ty τ 2 : κ 2
Γ ⊩ app (τ 2 : κ 2) : κ 1 ⊣→ κ  
Γ ⊩ ty τ 1 τ 2 : κ  Ty_APPTY

Γ ⊩ ty τ 1 : κ 1
Γ ⊩ ty τ 2 : κ 2
Γ ⊩ → κ 1 → κ 2 : κ  
Γ ⊩ ty τ 1 → τ 2 : κ  Ty_FUNTY

(¬ (isUnLiftedTyCon T) ∨ length τ 1 = tyConArity T)
Γ ⊩ ty τ 1 : κ 1
Γ ⊩ tyLitLit : tyConKind T ⊣→ κ  
Γ ⊩ ty τ 1 i : κ
Γ ⊩ app (τ 1 i : κ 1) i : τConKind T ⊣→ κ  Ty_TYCONAPP

Γ ⊩ k κ 1 ok
Γ ⊩ z κ 1 τ : TYPE σ
¬ (z ∈ f v (σ))
Γ ⊩ ∀ z κ 1 τ : TYPE σ
Ty_FORALLTY

Γ ⊩ tyLitLit : κ
Γ ⊩ tyLitLit : κ  Ty_LITTY

Γ ⊩ ty τ 1 : κ 1
Γ ⊩ co γ : κ 1 ⊣→ κ 2
Γ ⊩ ty τ ⊿ γ : κ 2  Ty_CASTTY

Γ ⊩ |z| tyConKind T ⊣→ κ 1 κ 2
Γ ⊩ ty τ : κ 1
Γ ⊩ ty τ : κ 1
Γ ⊩ ty |z| : κ 1 ⊣→ κ 2 κ 2
Γ ⊩ ty |z| : κ 1 κ 2  Ty_COERCIONTY_NOM

Γ ⊩ |z| tyConKind T ⊣→ κ 1 κ 2
Γ ⊩ ty τ : κ 1 κ 2
Γ ⊩ ty τ : κ 1 κ 2
Γ ⊩ ty |z| : (κ 1 κ 2) τ 1 τ 2
Γ ⊩ ty |z| : (κ 1 κ 2) τ 1 τ 2  Ty_COERCIONTY_REPR
4.6 Coercion typing

In the coercion typing judgment, the \( \# \) marks are left off the equality operators to reduce clutter. This is not actually inconsistent, because the GHC function that implements this check, `lintCoercion`, actually returns five separate values (the two kinds, the two types, and the role), not a type with head \( \sim \) or \( \sim_R \).

Note that the difference between these two forms of equality is interpreted in the rules `Co_CoVarCoNom` and `Co_CoVarCoRepr`.

\[
\Gamma \vdash \gamma : \tau_1 \sim_{\rho} \tau_2 \quad \text{Coercion typing, coreSyn/CoreLint.lhs:lintCoercion}
\]

\[
\Gamma \vdash \gamma : \kappa \\
\Gamma \vdash \tau : \kappa \\
\Gamma \vdash \langle \tau \rangle_\rho : \tau \sim_{\rho} \tau \quad \text{CO_REFL}
\]

\[
\begin{align*}
\Gamma & \vdash \gamma_1 : \sigma_1 \sim_{\rho_1} \kappa_1^\prime \\
\Gamma & \vdash \gamma_2 : \sigma_2 \sim_{\rho_2} \kappa_2^\prime \\
\Gamma & \vdash \kappa_1 \rightarrow \kappa_2 : \kappa \\
\Gamma & \vdash \kappa_1^\prime \rightarrow \kappa_2^\prime : \kappa^\prime
\end{align*}
\]

\[
\Gamma \vdash \langle \rightarrow \rangle_\rho \gamma_1 \gamma_2 : (\sigma_1 \rightarrow \sigma_2) \sim_{\rho} (\tau_1 \rightarrow \tau_2) \quad \text{CO_TyConAppCoFunTy}
\]

\[
T \neq (\rightarrow) \\
\bar{\pi}_i^1 = \text{take(length }\bar{\pi}_i^1\text{, tyConRolesX }\rho T) \\
\Gamma \vdash \gamma_i : \kappa_i \sim_{\rho_i} \kappa_i^\prime \\
\Gamma \vdash \sigma_i^1 : \kappa_i \sim_{\rho_i} \kappa_i^\prime \\
\Gamma \vdash \tau_i \sim_{\rho_i} \tau_i^1 \\
\Gamma \vdash \tau_i^1 \sim_{\rho_i} \tau_i \\
\Gamma \vdash \bar{\sigma}_i^1 \sim_{\rho_i} \bar{\tau}_i^1 \quad \text{CO_TyConAppCo}
\]

\[
\begin{align*}
\Gamma & \vdash \gamma_1 : \sigma_1 \sim_{\rho} \kappa_1^2 \sigma_2 \\
\Gamma & \vdash \gamma_2 : \tau_1 \sim_{\rho} \kappa_2^2 \tau_2 \\
\Gamma & \vdash \bar{\tau}_1 \sim_{\rho} \bar{\tau}_2 \\
\Gamma & \vdash \bar{\tau}_1 \sim_{\rho} \bar{\tau}_2
\end{align*}
\]

\[
\Gamma \vdash \gamma_1 \gamma_2 : (\sigma_1 \tau_1) \sim_{\rho} (\sigma_2 \tau_2) \quad \text{CO_APPCoX}
\]

\[
\begin{align*}
\Gamma & \vdash \gamma_1 : \sigma_1 \sim_{\rho} \kappa_1^2 \sigma_2 \\
\Gamma & \vdash \gamma_2 \gamma_2 : \tau_1 \sim_{\rho} \kappa_2^2 \tau_2 \\
\Gamma & \vdash \bar{\tau}_1 \sim_{\rho} \bar{\tau}_2 \\
\Gamma & \vdash \bar{\tau}_1 \sim_{\rho} \bar{\tau}_2
\end{align*}
\]

\[
\Gamma \vdash \gamma_1 \gamma_2 : (\sigma_1 \tau_1) \sim_{\rho} (\sigma_2 \tau_2) \quad \text{CO_APPCoPhantom}
\]
\[
\begin{align*}
\Gamma \vdash_{co} & \gamma : (T \sigma_j^j)^{\kappa_1 \sim^{\rho^j}_N \kappa'_1} (T \tau_j^j) \\
\text{length } \sigma_j^j = \text{length } \tau_j^j \\
i < \text{length } \sigma_j^j \\
\Gamma \vdash_{ty} \sigma_1 : \kappa_2 \\
\Gamma \vdash_{ty} \tau_1 : \kappa'_2 \\
\neg (\exists \gamma \text{ s.t. } \sigma_1 = \gamma) \\
\neg (\exists \gamma \text{ s.t. } \tau_1 = \gamma) \\
\rho' = (\text{tyConRolesX} \rho T)[i] \\
\end{align*}
\]

\[
\Gamma \vdash_{co} \text{nth}^i \gamma : \sigma_i \kappa_2 \sim^{\rho^i}_N \kappa'_2 \tau_i
\]

\[
\begin{align*}
\Gamma \vdash_{co} & \gamma : (\forall \sigma_1. \tau_1)^{\kappa_3 \sim^{\rho}_N \kappa_1} (\forall \sigma_2. \tau_2) \\
\Gamma \vdash_{co} \text{nth}^0 \gamma : \kappa_3 \sim^{\rho}_N \kappa_2 \\
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash_{co} & \gamma_1 : (\sigma_1 \sigma_2)^{\kappa_3 \sim^{\rho}_N (\tau_1 \tau_2)} \\
\Gamma \vdash_{ty} \sigma_1 : \kappa_1 \\
\Gamma \vdash_{ty} \tau_1 : \kappa'_1 \\
\end{align*}
\]

\[
\Gamma \vdash_{co} \text{left} \gamma : \sigma_1 \kappa_1 \sim^{\rho}_N \tau_1
\]

\[
\begin{align*}
\Gamma \vdash_{co} & \gamma_1 : (\sigma_1 \sigma_2)^{\kappa_3 \sim^{\rho}_N (\tau_1 \tau_2)} \\
\Gamma \vdash_{ty} \sigma_2 : \kappa_2 \\
\Gamma \vdash_{ty} \tau_2 : \kappa'_2 \\
\neg (\exists \gamma \text{ s.t. } \sigma_2 = \gamma) \\
\neg (\exists \gamma \text{ s.t. } \tau_2 = \gamma) \\
\end{align*}
\]

\[
\Gamma \vdash_{co} \text{right} \gamma : \sigma_2 \kappa_2 \sim^{\rho}_N \tau_2
\]

\[
\begin{align*}
\Gamma \vdash_{co} & \gamma \&\eta : (\tau_1 \sigma_1 \mapsto \tau_1 \mapsto \sigma_1)^{\kappa_3 \sim^{\rho}_N (\tau_2 \sigma_2 \mapsto \tau_2 \mapsto \sigma_2)} \\
\Gamma \vdash_{co} \eta : \sigma_1 \kappa_1 \sim^{\rho}_N \sigma_2 \\
\end{align*}
\]

\[
\begin{align*}
C = T_{\rho_0} \ axBranch_k^k \\
0 \leq \text{ind } < \text{length } axBranch_k^k \\
\forall \mu, \mu', (\sigma_j^j \mapsto \mu_1) = (axBranch_k^k)[\text{ind}] \\
\Gamma \vdash_{\text{ask}} \mu_1 \mapsto {^{\rho_0}_N \mu_1'} \sim (\text{subst}_1, \text{subst}_2) \\
\sigma_2 \mapsto \text{subst}_1(\sigma_1) \\
\neg \text{conflict}(C, \sigma_2 \sigma_2, \text{ind}, \text{ind} - 1) \\
\tau_2 = \text{subst}_2(\tau_1) \\
\sigma_2 = T \sigma_j^j \\
\Gamma \vdash_{ty} \sigma_2 : \kappa \\
\Gamma \vdash_{ty} \tau_2 : \kappa' \\
\end{align*}
\]

\[
\Gamma \vdash_{co} C \text{ind} \mapsto {^{\rho_0}_N \mu_1'} : \sigma_2 \kappa \sim^{\rho_0}_N \tau_2
\]
\[
\begin{align*}
\Gamma \vdash_{\text{co}} \gamma : \tau_1 \kappa_1 \sim_{\rho} \kappa_2 \\
\Gamma \vdash_{\text{by}} \tau_1 \triangleright \eta : \kappa_1' \quad &\quad \text{Co\_Coherence}\text{Co} \\
\Gamma \vdash_{\text{co}} \gamma \triangleright \eta : \tau_1 \triangleright \eta \kappa_1 \sim_{\rho} \kappa_2 \\
\Gamma \vdash_{\text{co}} \gamma : \tau_1 \kappa_1 \sim_{\rho} \kappa_2 \\
\Gamma \vdash_{\text{co}} \text{kind } \gamma : \kappa_1 \ast \sim_{\eta} \kappa_2 \\
\Gamma \vdash_{\text{co}} \text{sub } \gamma : \sigma \sim_{\eta} \tau \\
\Gamma \vdash_{\text{co}} \mu = \text{M}_{(i, j), \rho'} \\
\Gamma \vdash_{\text{by}} \tau_1 : \kappa_1' \\
\Gamma \vdash_{\text{co}} \gamma_j : \sigma_j \sim_{\rho_j} \kappa'_j \\
\text{Just } (\tau'_1, \tau'_2) = \text{coaxrProves } \mu \tau_1 \mid (\sigma_j, \kappa'_j) \\
\Gamma \vdash_{\text{by}} \tau'_1 : \kappa_0 \\
\Gamma \vdash_{\text{by}} \tau'_2 : \kappa'_0 \\
\Gamma \vdash_{\text{co}} \mu \tau_1 \mid (\tau'_1 \kappa_0 \sim_{\rho} \tau'_2) \\
\end{align*}
\]

See Section 4.15 for more information about \texttt{tyConRolesX}, and see Section 2.6 for more information about \texttt{coaxrProves}.

### 4.7 Name consistency

There are two very similar checks for names, one declared as a local function:

\[
\begin{align*}
\Gamma \vdash_{\text{n}} \text{n ok} & \quad \text{Name consistency check, coreSyn/CoreLint.lhs:lintSingleBinding#lintBinder} \\
\Gamma \vdash_{\text{by}} \tau : \kappa \\
\kappa = \ast \lor \kappa = \# \\
\Gamma \vdash_{\text{n}} x^\tau \text{ ok} & \quad \text{NAME\_ID} \\
\Gamma \vdash_{\text{n}} \alpha^\kappa \text{ ok} & \quad \text{NAME\_TYVAR} \\
\Gamma \vdash_{\text{bnd}} \text{n ok} & \quad \text{Binding consistency, coreSyn/CoreLint.lhs:lintBinder} \\
\Gamma \vdash_{\text{by}} \tau : \kappa \\
\kappa = \ast \lor \kappa = \# \\
\Gamma \vdash_{\text{bnd}} x^\tau \text{ ok} & \quad \text{BINDING\_ID} \\
\end{align*}
\]
\[ \Gamma \vdash \kappa \text{ ok} \quad \text{BINDING\_TYVAR} \]

4.8 Substitution consistency

\[ \Gamma \vdash \text{subst } n \mapsto \tau \text{ ok} \quad \text{Substitution consistency, coreSyn/CoreLint.lhs:lintTyKind} \]

\[ \Gamma \vdash \tau : \kappa \quad \text{SUBST\_TYPE} \]

4.9 Case alternative consistency

\[ \Gamma; \sigma \vdash \text{alt } \text{alt } : \tau \quad \text{Case alternative consistency, coreSyn/CoreLint.lhs:lintCoreAlt} \]

\[ \frac{\Gamma \vdash \text{tm } e : \tau}{\Gamma; \sigma \vdash \text{alt } \omega \rightarrow e : \tau} \quad \text{ALT\_DEFAULT} \]

\[ \sigma = \text{literalType } \text{lit} \]

\[ \frac{\Gamma \vdash \text{tm } e : \tau}{\Gamma; \sigma \vdash \text{alt } \text{lit } \rightarrow e : \tau} \quad \text{ALT\_LITALT} \]

\[ T = \text{dataConTyCon } K \]

\[ \neg (\text{isNewTyCon } T) \]

\[ \tau_1 = \text{dataConRepType } K \]

\[ \tau_2 = \tau_1 \{ \sigma_j^i \} \]

\[ \Gamma \vdash \text{bnd } n_i \text{ ok} \]

\[ \Gamma' = \Gamma; \tau_i \]

\[ \Gamma' \vdash \text{altbnd } n_i \rightarrow \tau_2 \rightarrow T \sigma_j^j \]

\[ \Gamma' \vdash \text{tm } e : \tau \]

\[ \frac{}{\Gamma; T \sigma_j^j \vdash \text{alt } K \tau_i \rightarrow e : \tau} \quad \text{ALT\_DATAALT} \]

4.10 Telescope substitution

\[ \tau' = \tau \{ \sigma_l \} \quad \text{Telescope substitution, types/Type.lhs:applyTys} \]

\[ \frac{}{\tau = \tau \{ \} } \quad \text{APPLYTYS\_EMPTY} \]
\[
\begin{align*}
\tau' &= \tau[\sigma_i]\{i\} \\
\tau'' &= \tau'[n \mapsto \sigma] \\
\tau''' &= (\forall \tau)(\sigma, \sigma_i) \\
\end{align*}
\]

\text{APPLY_TYS_TY}

### 4.11 Case alternative binding consistency

\[\Gamma \vdash \text{altbnd} \ : \ \tau_1 \leadsto \tau_2\]  
Case alternative binding consistency, coreSyn/CoreLint.lhs:lintAltBinders

\[\Gamma \vdash \text{altbnd} \ : \ \tau \leadsto \tau\]  
ALT_BINDERS_EMPTY

\[\begin{align*}
\Gamma \vdash \text{subst} \ \beta^\kappa \mapsto \alpha^\kappa \text{ ok} \\
\Gamma \vdash \text{altbnd} \ \tau_i \ : \ \tau[\beta^\kappa \mapsto \alpha^\kappa] \leadsto \sigma \\
\end{align*}\]

ALT_BINDERS_TYVAR

\[\begin{align*}
\Gamma \vdash \text{altbnd} \ \tau_i \ : \ \tau[z^\phi \mapsto c^\phi] \leadsto \sigma \\
\end{align*}\]

ALT_BINDERS_ID_COERCION

\[\begin{align*}
\Gamma \vdash \text{altbnd} \ \tau_i \ : \ \tau \leadsto \sigma \\
\end{align*}\]

ALT_BINDERS_ID_TERM

### 4.12 Arrow kinding

\[\Gamma \vdash \kappa_1 \rightarrow \kappa_2 \ : \ \kappa\]  
Arrow kinding, coreSyn/CoreLint.lhs:lintArrow

\[\begin{align*}
\kappa_1 &\in \{\ast, \#\} \\
\kappa_2 &= \text{TYPE} \sigma \\
\end{align*}\]

ARROW_KIND

### 4.13 Type application kinding

\[\Gamma \vdash \text{app} \ : \ \tau \leadsto \tau\]  
Type application kinding, coreSyn/CoreLint.lhs:lint_app

\[\Gamma \vdash \text{app} \ : \ \kappa \leadsto \kappa\]  
APP_EMPTY
4.14 Axiom argument kinding

\[ \Gamma \vdash \text{axk} \left[ n_i \rho \mapsto \tau \right] \Rightarrow (\text{subst}_1, \text{subst}_2) \]

Axiom argument kinding, coreSyn/CoreLint.lhs:check_ki

\[ \Gamma \vdash \text{axk} \left[ \cdot \mapsto \cdot \right] \Rightarrow (\cdot, \cdot) \]

AXIOM_KIND_EMPTY

\[ \Gamma \vdash \text{axk} \left[ n_i \rho \mapsto \tau \right] \Rightarrow (\text{subst}_1, \text{subst}_2) \]

\[ n = z^\kappa \]

\[ \Gamma \vdash \text{co} \gamma_0 : \tau_1 \text{subst}_1(\kappa) \Rightarrow \tau_2 \]

\[ \Gamma \vdash \text{axk} \left[ n_i \rho \mapsto \tau, \gamma_0 \right] \Rightarrow (\text{subst}_1 [n \mapsto \tau_1], \text{subst}_2 [n \mapsto \tau_2]) \]

AXIOM_KIND_ARG

4.15 Roles

During type-checking, role inference is carried out, assigning roles to the arguments of every type constructor. The function \( \text{tyConRoles} \) extracts these roles. Also used in other judgments is \( \text{tyConRolesX} \), which is the same as \( \text{tyConRoles} \), but with an arbitrary number of \( \text{N} \) at the end, to account for potential oversaturation.

The checks encoded in the following judgments are run from typecheck/TcTyClsDecls.lhs:checkValidTyCon when \(-\text{dcore-lint}\) is set.

validRoles \( T \) Type constructor role validity, typecheck/TcTyClsDecls.lhs:checkValidRoles

\[ \forall \tau_i \Rightarrow \text{tyConDataCons} T \]
\[ \forall \tau_j \Rightarrow \text{tyConRoles} T \]
\[ \text{validDcRoles} \tau_i \Rightarrow \text{tyConDataCons} T \]
\[ \text{validRoles} T \]

validDcRoles \( \tau_a \Rightarrow \tau \) Data constructor role validity, typecheck/TcTyClsDecls.lhs:check_dc_roles

\[ \forall \tau_a \Rightarrow \forall \tau_b \Rightarrow \tau_c \Rightarrow T \Rightarrow \text{dataConRepType} K \]
\[ \tau_a \Rightarrow \text{dataConRepType} K \]

validDcRoles \( \tau_a \Rightarrow \tau \)

\[ \text{validDcRoles} \tau_a \Rightarrow \tau \]

CDR_ARGS
In the following judgment, the role $\rho$ is an input, not an output. The metavariable $\Omega$ denotes a role context, as shown here:

$$\Omega ::= \text{Mapping from type variables to roles} \mid \text{List of bindings}$$

$$\Omega \triangleright \tau : \rho$$  \hspace{1cm} Type role validity, typecheck/TeTyClsDecls.lhs:check_ty_roles

$$\frac{\Omega(n) = \rho'}{\rho' \leq \rho} \quad \text{CTR_TyVarTy}$$

$$\overline{\tau_i} = \text{tyConRoles } T$$

$$\rho_i \in \{N,R\} \implies \Omega \triangleright \tau_i : \rho_i$$

$$\frac{\Omega \triangleright \tau_i : \rho_i \quad \text{CTR_TyConAppRep}}{\Omega \triangleright \tau : \overline{\tau_i} : R}$$

$$\frac{\Omega \triangleright \tau_i : N_i \quad \text{CTR_TyConAppNom}}{\Omega \triangleright \tau : \overline{\tau_i} : N}$$

$$\frac{\Omega \triangleright \tau_1 : \rho \quad \Omega \triangleright \tau_2 : N}{\Omega \triangleright \tau_1 \tau_2 : \rho} \quad \text{CTR_AppTy}$$

$$\frac{\Omega \triangleright \tau_1 : \rho \quad \Omega \triangleright \tau_2 : \rho}{\Omega \triangleright \tau_1 \rightarrow \tau_2 : \rho} \quad \text{CTR_FunTy}$$

$$\frac{\Omega, n : N \triangleright \tau : \rho}{\Omega \triangleright \forall n. \tau : \rho} \quad \text{CTR_ForAllTy}$$

$$\frac{\Omega \triangleright \text{lit} : \rho}{\Omega \triangleright \tau : \rho} \quad \text{CTR_LitTy}$$

$$\frac{\Omega \triangleright \tau : \rho \quad \Omega \triangleright \tau \triangleright \gamma : \rho}{\Omega \triangleright \gamma : P} \quad \text{CTR_CoercionTy}$$

These judgments depend on a sub-role relation:
\( \rho_1 \leq \rho_2 \) Sub-role relation, types/Coercion.lhs:ltRole

\[ \begin{align*}
N \leq \rho & \quad \text{RLT\_NOMINAL} \\
\rho \leq P & \quad \text{RLT\_PHANTOM} \\
\rho \leq \rho & \quad \text{RLT\_REFL}
\end{align*} \]

4.16 Branched axiom conflict checking

The following judgment is used within Co_AxiomInstCo to make sure that a type family application cannot unify with any previous branch in the axiom. The actual code scans through only those branches that are flagged as incompatible. These branches are stored directly in the \texttt{axBranch}. However, it is cleaner in this presentation to simply check for compatibility here.

\( \text{no\_conflict}(C, \sigma_j^i, \text{ind}_1, \text{ind}_2) \) Branched axiom conflict checking, types/OptCoercion.lhs:checkAxInstCo and types/FamInstEnv.lhs:compatibleBranches

\[ \begin{align*}
\text{no\_conflict}(C, \sigma_i^i, \text{ind}, -1) & \quad \text{NoCONFLICT\_NoBranch} \\
\text{no\_conflict}(C, \sigma_j^j, \text{ind}_1, \text{ind}_2 - 1) & \quad \text{NoCONFLICT\_Incompat} \\
\text{no\_conflict}(C, \sigma_j^j, \text{ind}_1, \text{ind}_2) & \quad \text{NoCONFLICT\_CompatApart} \\
\text{no\_conflict}(C, \sigma_j^j, \text{ind}_1, \text{ind}_2) & \quad \text{NoCONFLICT\_CompatCoincident}
\end{align*} \]
The judgment \texttt{apart} checks to see whether two lists of types are surely apart. \texttt{apart}(\tau_i, \sigma_i), where \tau_i is a list of types and \sigma_i is a list of type patterns (as in a type family equation), first flattens the \tau_i using \texttt{types/FamInstEnv.lhs:flattenTys} and then checks to see if \texttt{types/Unify.lhs:tcUnifyTysFG} returns \texttt{SurelyApart}. Flattening takes all type family applications and replaces them with fresh variables, taking care to map identical type family applications to the same fresh variable.

The algorithm \texttt{unify} is implemented in \texttt{types/Unify.lhs}. It performs a standard unification, returning a substitution upon success.

5 Operational semantics

5.1 Disclaimer

GHC does not implement an operational semantics in any concrete form. Most of the rules below are implied by algorithms in, for example, the simplifier and optimizer. Yet, there is no one place in GHC that states these rules, analogously to \texttt{CoreLint.lhs}. Nevertheless, these rules are included in this document to help the reader understand System FC.

5.2 The context \(\Sigma\)

We use a context \(\Sigma\) to keep track of the values of variables in a (mutually) recursive group. Its definition is as follows:

\[\Sigma ::= \cdot | \Sigma, [n \mapsto e]\]

The presence of the context \(\Sigma\) is solely to deal with recursion. If your use of FC does not require modeling recursion, you will not need to track \(\Sigma\).

5.3 Operational semantics rules

\[
\begin{align*}
\Sigma \vdash_{op} e & \rightarrow e' \\
\Sigma(n) = e & \quad \text{S.VAR} \\
\Sigma \vdash_{op} e_1 & \rightarrow e'_1 \\
\Sigma \vdash_{op} e_1 e_2 & \rightarrow e_1' e_2' & \text{S.APP} \\
\Sigma \vdash_{op} (\lambda n. e_1) e_2 & \rightarrow e_1 [n \mapsto e_2] & \text{S.BETA} \\
\end{align*}
\]

\[\begin{align*}
\gamma_0 &= \text{sym}(\text{nth}^0 \gamma) \\
\gamma_1 &= \text{nth}^1 \gamma \\
\exists \tau \text{ s.t. } e_2 = \tau & \\
\exists \gamma \text{ s.t. } e_2 = \gamma & \\
\Sigma \vdash_{op} ((\lambda n. e_1) \triangleright \gamma) e_2 & \rightarrow (\lambda n. e_1 \triangleright \gamma_1) (e_2 \triangleright \gamma_0) & \text{S.PUSH}
\end{align*}\]
\[ \sum \vdash_{op} ((\lambda n. e) \triangleright \gamma) \rightarrow (\lambda n. (e \triangleright \gamma \ n)) \tau \quad \text{S\_TPush} \]

\[ \gamma_0 = \text{nth}^1 (\text{nth}^0 \gamma) \]
\[ \gamma_1 = \text{sym} (\text{nth}^2 (\text{nth}^0 \gamma)) \]
\[ \gamma_2 = \text{nth}^1 \gamma \]
\[ \sum \vdash_{op} ((\lambda n. e) \triangleright \gamma) \gamma' \rightarrow (\lambda n. e \triangleright \gamma_2) (\gamma_0 \triangleright \gamma' \triangleright \gamma_1) \quad \text{S\_CPush} \]

\[ \sum \vdash_{op} (e \triangleright \gamma_1) \triangleright \gamma_2 \rightarrow e \triangleright (\gamma_1 \triangleright \gamma_2) \quad \text{S\_TRANS} \]

\[ \sum \vdash_{op} e \rightarrow e' \]
\[ \sum \vdash_{op} e \triangleright \gamma \rightarrow e' \triangleright \gamma \]
\[ \sum \vdash_{op} e \rightarrow e' \]
\[ \sum \vdash_{op} e (\{\text{tick}\}) \rightarrow e' (\{\text{tick}\}) \quad \text{S\_Tick} \]

\[ \sum \vdash_{op} \text{case } e \text{ as } n \text{ return } \tau \text{ of } \text{alt}_i \rightarrow \text{case } e' \text{ as } n \text{ return } \tau \text{ of } \text{alt}_i \]

\[ \sum \vdash_{op} \text{case } e \text{ as } n \text{ return } \tau \text{ of } \text{alt}_i \rightarrow \text{case } e' \text{ as } n \text{ return } \tau \text{ of } \text{alt}_i \]

\[ \sum \vdash_{op} \text{case } e \rightarrow e' \quad \text{S\_CASE} \]

\[ \sum \vdash_{op} \text{case } \text{alt}_j = K \alpha_b \rightarrow \gamma \rightarrow u \quad \text{S\_MatchData} \]

\[ \sum \vdash_{op} \text{case } \text{alt}_j = \text{lit} \rightarrow u \quad \text{S\_MatchLit} \]

\[ \sum \vdash_{op} \text{case } e \text{ as } n \text{ return } \tau \text{ of } \text{alt}_i \rightarrow u [n \mapsto \text{lit}] \]

\[ \sum \vdash_{op} \text{case } e \text{ as } n \text{ return } \tau \text{ of } \text{alt}_i \rightarrow u [n \mapsto \text{alt}_i] \]

\[ \sum \vdash_{op} \text{case } e \text{ as } n \text{ return } \tau \text{ of } \text{alt}_i \rightarrow \text{case } e' \text{ as } n \text{ return } \tau \text{ of } \text{alt}_i \]

\[ \sum \vdash_{op} \text{case } \text{alt}_j = \omega \rightarrow u \quad \text{S\_MatchDefault} \]

\[ T \tau_\alpha \triangleright \delta \sim^\gamma_{\tau_\alpha} T \tau_\alpha \triangleright \delta = \text{coercionKind } \gamma \]

\[ \forall \alpha \triangleright \delta \triangleright \gamma \quad \forall \beta \triangleright \delta \triangleright \gamma \]

\[ \sum \vdash_{op} \text{case } (K \tau_\alpha \triangleright \delta \triangleright \gamma) \triangleright \gamma \text{ as } n \text{ return } \tau_2 \text{ of } \text{alt}_i \rightarrow \text{case } (K \tau_\alpha \triangleright \delta \triangleright \gamma) \triangleright \gamma \text{ as } n \text{ return } \tau_2 \text{ of } \text{alt}_i \]

\[ \sum \vdash_{op} \text{case } (K \tau_\alpha \triangleright \delta \triangleright \gamma) \triangleright \gamma \text{ as } n \text{ return } \tau_2 \text{ of } \text{alt}_i \rightarrow \text{case } (K \tau_\alpha \triangleright \delta \triangleright \gamma) \triangleright \gamma \text{ as } n \text{ return } \tau_2 \text{ of } \text{alt}_i \]
5.4 Notes

- The S_LetRec rules implement recursion. S_LetRec adds to the context Σ bindings for all of the mutually recursive equations. Then, after perhaps many steps, when the body of the let rec contains no variables that are bound in the let rec, the context is popped in S_LetRecReturn. The other S_LetRecXXX rules are there to prevent reduction from getting stuck.

- In the case rules, a constructor K is written taking three lists of arguments: two lists of types and a list of terms. The types passed in are the universally and, respectively, existentially quantified type variables to the constructor. The terms are the regular term arguments stored in an algebraic datatype. Coercions (say, in a GADT) are considered term arguments.

- The rule S_CasePush is the most complex rule.
  - The logic in this rule is implemented in coreSyn/CoreSubst.lhs:exprIsConApp_maybe.
  - The coercionKind function (types/Coercion.lhs:coercionKind) extracts the two types (and their kinds) from a coercion. It does not require a typing context, as it does not check the coercion, just extracts its types.
– The `dataConRepType` function (`basicTypes/DataCon.lhs:dataConRepType`) extracts the full type of a data constructor. Following the notation for constructor expressions, the parameters to the constructor are broken into three groups: universally quantified types, existentially quantified types, and terms.

– The substitutions in the last premise to the rule are unusual: they replace type variables with coercions. This substitution is called lifting and is implemented in `types/Coercion.lhs:liftCoSubst`. The notation is essentially a pun on the fact that types and coercions have such similar structure. This operation is quite non-trivial. Please see *System FC with Explicit Kind Equality* for details.

– Note that the types $\sigma^b$—the existentially quantified types—do not change during this step.