# System FC, as implemented in GHC[1]
February 12, 2013

## 1 Introduction

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 |
| $K$ | Term-level data constructor names |
| $i$, $j$, $k$ | Indices to be used in lists |

### 2.2 Literals

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

$\mathsf{lit}$ ::= Literals, *basicTypes/Literal.lhs*:`Literal`

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

### 2.3 Variables

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

$$
\begin{array}{lcll}
z & ::= & & \text{Term or type name} \\
  & | & \alpha & \quad \text{Type-level name} \\
  & | & x & \quad \text{Term-level name}
\end{array}
$$

$$
\begin{array}{lcll}
n,\ m & ::= & & \text{Variable names, } \textit{basicTypes/Var.lhs}\text{:}\texttt{Var} \\
  & | & z^{\tau} & \quad \text{Name, labeled with type/kind}
\end{array}
$$

We sometimes omit the type/kind annotation to a variable when it is obvious from context.

[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.

## 2.4 Expressions

The datatype that represents expressions:

| | | | |
|---|---|---|---|
| $e,\ u$ | ::= | | Expressions, *coreSyn/CoreSyn.lhs*:`Expr` |
| | \| | $n$ | `Var`: Variable |
| | \| | $\mathsf{lit}$ | `Lit`: Literal |
| | \| | $e_1\, e_2$ | `App`: Application |
| | \| | $\lambda n.e$ | `Lam`: Abstraction |
| | \| | $\mathbf{let}\ \mathit{binding}\ \mathbf{in}\ e$ | `Let`: Variable binding |
| | \| | $\mathbf{case}\ e\ \mathbf{as}\ n\ \mathbf{return}\ \tau\ \mathbf{of}\ \overline{\mathit{alt}_i}^{\,i}$ | `Case`: Pattern match |
| | \| | $e \triangleright \gamma$ | `Cast`: Cast |
| | \| | $e_{\{\mathit{tick}\}}$ | `Tick`: Internal note |
| | \| | $\tau$ | `Type`: Type |
| | \| | $\gamma$ | `Coercion`: Coercion |

There are a few key invariants about expressions:

- The right-hand sides of all top-level and recursive **let**s must be of lifted type.
- The right-hand side of a non-recursive **let** and the argument of an application may be of unlifted type, but only if the expression is ok-for-speculation. See `#let_app_invariant#` in *coreSyn/CoreSyn.lhs*.
- We allow a non-recursive **let** for bind a type variable.
- The ␣ case for a **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.

Bindings for **let** statements:

| | | | |
|---|---|---|---|
| $\mathit{binding}$ | ::= | | Let-bindings, *coreSyn/CoreSyn.lhs*:`Bind` |
| | \| | $n = e$ | `NonRec`: Non-recursive binding |
| | \| | $\mathbf{rec}\ \overline{n_i = e_i}^{\,i}$ | `Rec`: Recursive binding |

Case alternatives:

| | | | |
|---|---|---|---|
| $\mathit{alt}$ | ::= | | Case alternative, *coreSyn/CoreSyn.lhs*:`Alt` |
| | \| | $\mathbb{K}\, \overline{n_i}^{\,i} \to e$ | Constructor applied to fresh names |

Constructors as used in patterns:

| | | | |
|---|---|---|---|
| $\mathbb{K}$ | ::= | | Constructors used in patterns, *coreSyn/CoreSyn.lhs*:`AltCon` |
| | \| | $K$ | `DataAlt`: Data constructor |
| | \| | $\mathsf{lit}$ | `LitAlt`: Literal (such as an integer or character) |
| | \| | ␣ | `DEFAULT`: Wildcard |

Notes that can be inserted into the AST. We leave these abstract:

| | | |
|---|---|---|
| $\mathit{tick}$ | ::= | Internal notes, *coreSyn/CoreSyn.lhs*:`Tickish` |

A program is just a list of bindings:

| | | | |
|---|---|---|---|
| $\mathit{program}$ | ::= | | A System FC program, *coreSyn/CoreSyn.lhs*:`CoreProgram` |
| | \| | $\overline{\mathit{binding}_i}^{\,i}$ | List of bindings |

## 2.5 Types

| | | | |
|---|---|---|---|
| $\tau,\ \kappa,\ \sigma,\ \phi$ | $::=$ | | Types/kinds, *types/TyCoRep.lhs*:`Type` |
| | $\mid$ | $n$ | `TyVarTy`: Variable |
| | $\mid$ | $\tau_1\,\tau_2$ | `AppTy`: Application |
| | $\mid$ | $T\,\overline{\tau_i}^{\,i}$ | `TyConApp`: Application of type constructor |
| | $\mid$ | $\tau_1 \to \tau_2$ | `FunTy`: Function |
| | $\mid$ | $\forall n.\tau$ | `ForAllTy`: Type and coercion polymorphism |
| | $\mid$ | $\mathsf{lit}$ | `LitTy`: Type-level literal |
| | $\mid$ | $\tau \triangleright \gamma$ | `CastTy`: Kind cast |
| | $\mid$ | $\gamma$ | `CoercionTy`: Coercion used in type |

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 $T\,\overline{\tau_i}^{\,i}$ (`TyConApp`) does *not* need to be saturated.
- A saturated application of $(\to)\,\tau_1\,\tau_2$ should be represented as $\tau_1 \to \tau_2$. This is a different point in the grammar, not just pretty-printing. The constructor for a saturated $(\to)$ is `FunTy`.
- 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.
- A kind cast should operate over only type variables, quantifications, and type literals. In application scenarios, the cast is pushed left. If casting a casted type, the coercions are combined using transitivity.

Note that the use of the $T\,\overline{\tau_i}^{\,i}$ form and the $\tau_1 \to \tau_2$ 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.

We use the notation $\tau_1 \;{}^{\kappa_1}{\sim}_{\#}^{\kappa_2}\; \tau_2$ to stand for $(\sim_{\#})\,\kappa_1\,\kappa_2\,\tau_1\,\tau_2$.

## 2.6 Coercions

| | | | |
|---|---|---|---|
| $\gamma,\ \eta$ | $::=$ | | Coercions, *types/TyCoRep.lhs*:`Coercion` |
| | $\mid$ | $\langle\tau\rangle$ | `Refl`: Reflexivity |
| | $\mid$ | $T\,\overline{\omega_i}^{\,i}$ | `TyConAppCo`: Type constructor application |
| | $\mid$ | $\gamma\,\omega$ | `AppCo`: Application |
| | $\mid$ | $\forall cobndr.\gamma$ | `ForAllCo`: Polymorphism |
| | $\mid$ | $n$ | `CoVarCo`: Variable |
| | $\mid$ | $C\ ind\,\overline{\omega_i}^{\,i}$ | `AxiomInstCo`: Axiom application |
| | $\mid$ | $\tau_1 \twoheadrightarrow\!\!\!\!\!\!+\ \tau_2$ | `UnsafeCo`: Unsafe coercion |
| | $\mid$ | $\mathsf{sym}\,\gamma$ | `SymCo`: Symmetry |

| | | | |
|---|---|---|---|
| | $\mid$ | $\gamma_1 \fatsemi \gamma_2$ | `TransCo`: Transitivity |
| | $\mid$ | $\mathsf{nth}^{ind}\,\gamma$ | `NthCo`: Projection (0-indexed) |
| | $\mid$ | $LorR\,\gamma$ | `LRCo`: Left/right projection |
| | $\mid$ | $\gamma@\omega$ | `InstCo`: Instantiation |
| | $\mid$ | $\gamma \triangleright \eta$ | `CoherenceCo`: Coherence |
| | $\mid$ | $\mathsf{kind}\,\gamma$ | `KindCo`: Kind extraction |

| | | | |
|---|---|---|---|
| $\omega$ | $::=$ | | Argument to a coercion application, *types/TyCoRep.lhs*:`CoercionArg` |
| | $\mid$ | $\gamma$ | `TyCoArg`: Coercion between types |
| | $\mid$ | $(\gamma_1, \gamma_2)$ | `CoCoArg`: "Coercion" between coercions |

| | | | |
|---|---|---|---|
| *cobndr* | $::=$ | | Binder to quantified coercion, *types/TyCoRep.lhs*:`ForAllCoBndr` |
| | $\mid$ | $n$ | `TyHomo`: Homogeneous type quantification |
| | $\mid$ | $_\gamma(n_1, n_2, n_3)$ | `TyHetero`: Heterogeneous type quantification |
| | $\mid$ | $n$ | `CoHomo`: Homogeneous coercion quantification |
| | $\mid$ | $_\gamma(n_1, n_2)$ | `CoHetero`: Heterogeneous coercion quantification |

Invariants on coercions:

- $\langle \tau_1\,\tau_2 \rangle$ is used; never $\langle \tau_1 \rangle\,\langle \tau_2 \rangle$.
- If $\langle T \rangle$ is applied to some coercions, at least one of which is not reflexive, use $T\,\overline{\gamma_i}^{\,i}$, never $\langle T \rangle\,\gamma_1\,\gamma_2 \ldots$.
- The $T$ in $T\,\overline{\gamma_i}^{\,i}$ 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 (`Id`).

Similarly to with types, the form $T\,\overline{\omega_i}^{\,i}$ is needed only to stay faithful to the implementation.

| | | | |
|---|---|---|---|
| *LorR* | $::=$ | | left or right deconstructor, *types/TyCoRep.lhs*:`LeftOrRight` |
| | $\mid$ | $\mathsf{left}$ | `CLeft`: Left projection |
| | $\mid$ | $\mathsf{right}$ | `CRight`: Right projection |

Axioms:

| | | | |
|---|---|---|---|
| $C$ | $::=$ | | Axioms, *types/TyCon.lhs*:`CoAxiom` |
| | $\mid$ | $T\,\overline{axBranch_i}^{\,i}$ | `CoAxiom`: Axiom |

| | | | |
|---|---|---|---|
| $axBranch,\ b$ | $::=$ | | Axiom branches, *types/TyCon.lhs*:`CoAxBranch` |
| | $\mid$ | $\forall \overline{n_i}^{\,i}.(\overline{\tau_j}^{\,j} \rightsquigarrow \sigma)$ | `CoAxBranch`: Axiom branch |

The left-hand sides $\overline{\tau_j}^{\,j}$ of different branches of one axiom must all have the same length.

## 2.7 Type constructors

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

| | | | |
|---|---|---|---|
| $T$ | $::=$ | | Type constructors, *types/TyCon.lhs*:`TyCon` |
| | $\mid$ | $(\rightarrow)$ | `FunTyCon`: Arrow |

| | | |
|---|---|---|
| \| | $N^\kappa$ | `AlgTyCon`, `TupleTyCon`, `SynTyCon`: algebraic, tuples, families, and synonyms |
| \| | $H$ | `PrimTyCon`: Primitive tycon |
| \| | $'K$ | `PromotedDataCon`: Promoted data constructor |

We include some representative primitive type constructors. There are many more in *prelude/TysPrim.lhs*.

| | | | |
|---|---|---|---|
| $H$ | ::= | | Primitive type constructors, *prelude/TysPrim.lhs*: |
| | \| | $\mathsf{Int}_{\#}$ | Unboxed Int |
| | \| | $(\sim_{\#})$ | Unboxed equality |
| | \| | $\star$ | Kind of lifted types |
| | \| | $\#$ | Kind of unlifted types |
| | \| | $\square$ | Kind of kinds |
| | \| | $\mathsf{OpenKind}$ | Either $\star$ or $\#$ |
| | \| | $\mathsf{Constraint}$ | Constraint |

Note that although GHC contains distinct type constructors $\star$, $\square$, and $\mathsf{Constraint}$, this formalism treats only $\star$. These three type constructors are considered wholly equivalent. In particular the function `eqType` returns `True` when comparing any two members of this group. We need all three because they have different roles in source Haskell.

# 3 Contexts

The functions in *coreSyn/CoreLint.lhs* use the `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.

| | | | |
|---|---|---|---|
| $\Gamma$ | ::= | | List of bindings, *coreSyn/CoreLint.lhs*:`LintM` |
| | \| | $n$ | Single binding |
| | \| | $\overline{\Gamma_i}^{\,i}$ | 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 *types/Type.lhs*:`substTyVarBndr` for details.

# 4 Judgments

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

$\boxed{\vdash_{\mathsf{prog}}\ \mathit{program}}$ Program typing, *coreSyn/CoreLint.lhs*:`lintCoreBindings`

$$\frac{\Gamma = \overline{\mathsf{vars\_of}\ binding_i}^{\,i} \quad \mathsf{no\_duplicates}\ \overline{binding_i}^{\,i} \quad \overline{\Gamma \vdash_{\mathsf{bind}} binding_i}^{\,i}}{\vdash_{\mathsf{prog}} \overline{binding_i}^{\,i}} \quad \text{PROG\_COREBINDINGS}$$

Here is the definition of $\mathsf{vars\_of}$ , taken from *coreSyn/CoreSyn.lhs*:`bindersOf`:

$$\begin{array}{ll} \mathsf{vars\_of}\ n = e & = n \\ \mathsf{vars\_of}\ \mathbf{rec}\ \overline{n_i = e_i}^{\,i} & = \overline{n_i}^{\,i} \end{array}$$

## 4.2 Binding consistency

$\boxed{\Gamma \vdash_{\mathsf{bind}} binding}$ Binding typing, *coreSyn/CoreLint.lhs*:`lint_bind`

$$\frac{\Gamma \vdash_{\mathsf{sbind}} n \leftarrow e}{\Gamma \vdash_{\mathsf{bind}} n = e} \quad \text{BINDING\_NONREC}$$

$$\frac{\overline{\Gamma \vdash_{\mathsf{sbind}} n_i \leftarrow e_i}^{\,i}}{\Gamma \vdash_{\mathsf{bind}} \mathbf{rec}\ \overline{n_i = e_i}^{\,i}} \quad \text{BINDING\_REC}$$

$\boxed{\Gamma \vdash_{\mathsf{sbind}} n \leftarrow e}$ Single binding typing, *coreSyn/CoreLint.lhs*:`lintSingleBinding`

$$\frac{\Gamma \vdash_{\mathsf{tm}} e : \tau \quad \Gamma \vdash_{\mathsf{n}} z^{\tau}\ \mathsf{ok} \quad \overline{m_i}^{\,i} = fv(\tau) \quad \overline{m_i \in \Gamma}^{\,i}}{\Gamma \vdash_{\mathsf{sbind}} z^{\tau} \leftarrow e} \quad \text{SBINDING\_SINGLEBINDING}$$

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.

## 4.3 Expression typing

$\boxed{\Gamma \vdash_{\mathsf{tm}} e : \tau}$ Expression typing, *coreSyn/CoreLint.lhs*:`lintCoreExpr`

$$\frac{x^{\tau} \in \Gamma \quad \neg\,(\exists \tau_1, \tau_2, \kappa_1, \kappa_2 \text{ s.t. } \tau = \tau_1\ {}^{\kappa_1}\!\!\sim_{\#}^{\kappa_2} \tau_2)}{\Gamma \vdash_{\mathsf{tm}} x^{\tau} : \tau} \quad \text{TM\_VAR}$$

$$\frac{\tau = \mathsf{literalType}\,\mathsf{lit}}{\Gamma \vdash_{\mathsf{tm}} \mathsf{lit} : \tau} \quad \text{TM\_LIT}$$

$$\frac{\begin{array}{l}\Gamma \vdash_{\mathsf{tm}} e : \sigma \\ \Gamma \vdash_{\mathsf{co}} \gamma : \sigma \mathbin{{}^{\kappa_1}{\sim}^{\kappa_2}_{\#}} \tau \\ \kappa_2 \in \{\star, \#\}\end{array}}{\Gamma \vdash_{\mathsf{tm}} e \triangleright \gamma : \tau} \quad \textsc{Tm\_Cast}$$

$$\frac{\Gamma \vdash_{\mathsf{tm}} e : \tau}{\Gamma \vdash_{\mathsf{tm}} e_{\{tick\}} : \tau} \quad \textsc{Tm\_Tick}$$

$$\frac{\begin{array}{l}\Gamma' = \Gamma, \alpha^\kappa \\ \Gamma \vdash_{\mathsf{k}} \kappa \ \mathsf{ok} \\ \Gamma' \vdash_{\mathsf{subst}} \alpha^\kappa \mapsto \sigma \ \mathsf{ok} \\ \Gamma' \vdash_{\mathsf{tm}} e[\alpha^\kappa \mapsto \sigma] : \tau\end{array}}{\Gamma \vdash_{\mathsf{tm}} \mathbf{let}\, \alpha^\kappa = \sigma \,\mathbf{in}\, e : \tau} \quad \textsc{Tm\_LetTyKi}$$

$$\frac{\begin{array}{l}\Gamma \vdash_{\mathsf{sbind}} x^\sigma \leftarrow u \\ \Gamma \vdash_{\mathsf{ty}} \sigma : \kappa \\ \Gamma, x^\sigma \vdash_{\mathsf{tm}} e : \tau\end{array}}{\Gamma \vdash_{\mathsf{tm}} \mathbf{let}\, x^\sigma = u \,\mathbf{in}\, e : \tau} \quad \textsc{Tm\_LetNonRec}$$

$$\frac{\begin{array}{l}\overline{\Gamma'_i}^{\,i} = \mathsf{inits}\,(\,\overline{z_i^{\sigma_i}}^{\,i}\,) \\ \overline{\Gamma, \Gamma'_i \vdash_{\mathsf{ty}} \sigma_i : \kappa_i}^{\,i} \\ \mathsf{no\_duplicates}\ \overline{z_i}^{\,i} \\ \Gamma' = \Gamma, \overline{z_i^{\sigma_i}}^{\,i} \\ \overline{\Gamma' \vdash_{\mathsf{sbind}} z_i^{\sigma_i} \leftarrow u_i}^{\,i} \\ \Gamma' \vdash_{\mathsf{tm}} e : \tau\end{array}}{\Gamma \vdash_{\mathsf{tm}} \mathbf{let\,rec}\, \overline{z_i^{\sigma_i} = u_i}^{\,i} \,\mathbf{in}\, e : \tau} \quad \textsc{Tm\_LetRec}$$

$$\frac{\begin{array}{l}\Gamma \vdash_{\mathsf{tm}} e : \forall \alpha^\kappa.\tau \\ \Gamma \vdash_{\mathsf{subst}} \alpha^\kappa \mapsto \sigma \ \mathsf{ok}\end{array}}{\Gamma \vdash_{\mathsf{tm}} e\,\sigma : \tau[\alpha^\kappa \mapsto \sigma]} \quad \textsc{Tm\_AppType}$$

$$\frac{\begin{array}{l}\Gamma \vdash_{\mathsf{tm}} e : \forall c^\phi.\tau \\ \Gamma \vdash_{\mathsf{co}} \gamma : \phi\end{array}}{\Gamma \vdash_{\mathsf{tm}} e\,\gamma : \tau[c^\phi \mapsto \gamma]} \quad \textsc{Tm\_AppCo}$$

$$\frac{\begin{array}{l}\neg\,(\exists \tau \text{ s.t. } e_2 = \tau) \\ \Gamma \vdash_{\mathsf{tm}} e_1 : \tau_1 \to \tau_2 \\ \Gamma \vdash_{\mathsf{tm}} e_2 : \tau_1\end{array}}{\Gamma \vdash_{\mathsf{tm}} e_1\, e_2 : \tau_2} \quad \textsc{Tm\_AppExpr}$$

$$\frac{\begin{array}{l}\neg\,(\exists \tau_1, \tau_2, \kappa_1, \kappa_2 \text{ s.t. } \kappa = \tau_1 \; {}^{\kappa_1}\!\!\sim^{\kappa_2}_{\#} \tau_2) \\ \Gamma \vdash_{\mathsf{ty}} \tau : \kappa \\ \Gamma, x^{\tau} \vdash_{\mathsf{tm}} e : \sigma\end{array}}{\Gamma \vdash_{\mathsf{tm}} \lambda x^{\tau}.e : \tau \to \sigma} \quad \text{TM\_LAMID}$$

$$\frac{\begin{array}{l}\Gamma \vdash_{\mathsf{k}} \kappa \;\mathsf{ok} \\ \Gamma, \alpha^{\kappa} \vdash_{\mathsf{tm}} e : \tau\end{array}}{\Gamma \vdash_{\mathsf{tm}} \lambda \alpha^{\kappa}.e : \forall \alpha^{\kappa}.\tau} \quad \text{TM\_LAMTY}$$

$$\frac{\begin{array}{l}\phi = \sigma_1 \; {}^{\kappa_1}\!\!\sim^{\kappa_2}_{\#} \sigma_2 \\ \Gamma \vdash_{\mathsf{k}} \phi \;\mathsf{ok} \\ \Gamma, c^{\phi} \vdash_{\mathsf{tm}} e : \tau\end{array}}{\Gamma \vdash_{\mathsf{tm}} \lambda c^{\phi}.e : \forall c^{\phi}.\tau} \quad \text{TM\_LAMCO}$$

$$\frac{\begin{array}{l}\Gamma \vdash_{\mathsf{tm}} e : \sigma \\ \Gamma \vdash_{\mathsf{ty}} \sigma : \kappa_1 \\ \Gamma \vdash_{\mathsf{ty}} \tau : \kappa_2 \\ \overline{\Gamma, z^{\sigma}; \sigma \vdash_{\mathsf{alt}} alt_i : \tau}^{\,i}\end{array}}{\Gamma \vdash_{\mathsf{tm}} \mathbf{case}\; e \;\mathbf{as}\; z^{\sigma} \;\mathbf{return}\; \tau \;\mathbf{of}\; \overline{alt_i}^{\,i} : \tau} \quad \text{TM\_CASE}$$

$$\frac{\Gamma \vdash_{\mathsf{co}} \gamma : \tau_1 \; {}^{\kappa_1}\!\!\sim^{\kappa_2}_{\#} \tau_2}{\Gamma \vdash_{\mathsf{tm}} \gamma : \tau_1 \; {}^{\kappa_1}\!\!\sim^{\kappa_2}_{\#} \tau_2} \quad \text{TM\_COERCION}$$

- Some explication of TM_LETREC is helpful: The idea behind the second premise ($\overline{\Gamma, \Gamma'_i \vdash_{\mathsf{ty}} \sigma_i : \kappa_i}^{\,i}$) is that we wish to check each substituted type $\sigma'_i$ in a context containing all the types that come before it in the list of bindings. The $\Gamma'_i$ 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*:`collectAr` 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

$\boxed{\Gamma \vdash_{\mathsf{ty}} \tau : \kappa}$ Kinding, *coreSyn/CoreLint.lhs*:`lintType`

$$\frac{z^{\kappa} \in \Gamma}{\Gamma \vdash_{\mathsf{ty}} z^{\kappa} : \kappa} \quad \textsc{Ty\_TyVarTy}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathsf{ty}} \tau_1 : \kappa_1 \\ \Gamma \vdash_{\mathsf{ty}} \tau_2 : \kappa_2 \\ \Gamma \vdash_{\mathsf{app}} (\tau_2 : \kappa_2) : \kappa_1 \rightsquigarrow \kappa\end{array}}{\Gamma \vdash_{\mathsf{ty}} \tau_1\, \tau_2 : \kappa} \quad \textsc{Ty\_AppTy}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathsf{ty}} \tau_1 : \kappa_1 \\ \Gamma \vdash_{\mathsf{ty}} \tau_2 : \kappa_2 \\ \Gamma \vdash_{\to} \kappa_1 \to \kappa_2 : \kappa\end{array}}{\Gamma \vdash_{\mathsf{ty}} \tau_1 \to \tau_2 : \kappa} \quad \textsc{Ty\_FunTy}$$

$$\frac{\begin{array}{l}\neg\,(\mathsf{isUnLiftedTyCon}\ T) \vee \mathsf{length}\,\overline{\tau_i}^{\,i} = \mathsf{tyConArity}\ T \\ \overline{\Gamma \vdash_{\mathsf{ty}} \tau_i : \kappa_i}^{\,i} \\ \Gamma \vdash_{\mathsf{app}} \overline{(\tau_i : \kappa_i)}^{\,i} : \mathsf{tyConKind}\ T \rightsquigarrow \kappa\end{array}}{\Gamma \vdash_{\mathsf{ty}} T\,\overline{\tau_i}^{\,i} : \kappa} \quad \textsc{Ty\_TyConApp}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathsf{k}} \kappa_1\ \mathsf{ok} \\ \Gamma, z^{\kappa_1} \vdash_{\mathsf{ty}} \tau : \star\end{array}}{\Gamma \vdash_{\mathsf{ty}} \forall z^{\kappa_1}.\tau : \star} \quad \textsc{Ty\_ForAllTy}$$

$$\frac{\Gamma \vdash_{\mathsf{tylit}} \mathsf{lit} : \kappa}{\Gamma \vdash_{\mathsf{ty}} \mathsf{lit} : \kappa} \quad \textsc{Ty\_LitTy}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathsf{ty}} \tau : \kappa_1 \\ \Gamma \vdash_{\mathsf{co}} \gamma : \kappa_1\, {}^{\star}\!\sim_{\#}^{\star} \kappa_2\end{array}}{\Gamma \vdash_{\mathsf{ty}} \tau \triangleright \gamma : \kappa_2} \quad \textsc{Ty\_CastTy}$$

$$\frac{\Gamma \vdash_{\mathsf{co}} \gamma : \phi}{\Gamma \vdash_{\mathsf{ty}} \gamma : \phi} \quad \textsc{Ty\_CoercionTy}$$

## 4.5 Kind validity

$\boxed{\Gamma \vdash_{\mathsf{k}} \kappa\ \mathsf{ok}}$ Kind validity, *coreSyn/CoreLint.lhs*:`lintKind`

$$\frac{\Gamma \vdash_{\mathsf{ty}} \kappa : \star}{\Gamma \vdash_{\mathsf{k}} \kappa\ \mathsf{ok}} \quad \textsc{K\_Star}$$

## 4.6 Coercion typing

$\boxed{\Gamma \vdash_{\mathsf{co}} \gamma : \phi}$ Coercion typing, *coreSyn/CoreLint.lhs*:`lintCoercion`

$$\frac{\Gamma \vdash_{\mathsf{ty}} \tau : \kappa}{\Gamma \vdash_{\mathsf{co}} \langle\tau\rangle : \tau\ {}^{\kappa}\!\sim^{\kappa}_{\#}\ \tau} \quad \textsc{Co\_Refl}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathsf{co}} \gamma_1 : \sigma_1\ {}^{\kappa_1}\!\sim^{\kappa_1'}_{\#}\ \tau_1 \\ \Gamma \vdash_{\mathsf{co}} \gamma_2 : \sigma_2\ {}^{\kappa_2}\!\sim^{\kappa_2'}_{\#}\ \tau_2 \\ \Gamma \vdash_{\to} \kappa_1 \to \kappa_2 : \kappa \\ \Gamma \vdash_{\to} \kappa_1' \to \kappa_2' : \kappa'\end{array}}{\Gamma \vdash_{\mathsf{co}} (\to)\,\gamma_1\,\gamma_2 : (\sigma_1 \to \sigma_2)\ {}^{\kappa}\!\sim^{\kappa'}_{\#}(\tau_1 \to \tau_2)} \quad \textsc{Co\_TyConAppCoFunTy}$$

$$\frac{\begin{array}{c}T\ \neq (\to) \\ \overline{\Gamma \vdash_{\mathsf{arg}} \omega_i : (\sigma_i : \kappa_i', \tau_i : \kappa_i)}^{\,i} \\ \Gamma \vdash_{\mathsf{app}} \overline{(\sigma_i : \kappa_i')}^{\,i} : \mathsf{tyConKind}\ T \leadsto \kappa' \\ \Gamma \vdash_{\mathsf{app}} \overline{(\tau_i : \kappa_i)}^{\,i} : \mathsf{tyConKind}\ T \leadsto \kappa\end{array}}{\Gamma \vdash_{\mathsf{co}} T\,\overline{\omega_i}^{\,i} : T\,\overline{\sigma_i}^{\,i}\ {}^{\kappa'}\!\sim^{\kappa}_{\#}\ T\,\overline{\tau_i}^{\,i}} \quad \textsc{Co\_TyConAppCo}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathsf{co}} \gamma : \sigma_1\ {}^{\kappa_1}\!\sim^{\kappa_2}_{\#}\ \sigma_2 \\ \Gamma \vdash_{\mathsf{arg}} \omega : (\tau_1 : \kappa_1', \tau_2 : \kappa_2') \\ \Gamma \vdash_{\mathsf{app}} (\tau_1 : \kappa_1') : \kappa_1 \leadsto \kappa_3 \\ \Gamma \vdash_{\mathsf{app}} (\tau_2 : \kappa_2') : \kappa_2 \leadsto \kappa_4\end{array}}{\Gamma \vdash_{\mathsf{co}} \gamma\,\omega : (\sigma_1\,\tau_1)\ {}^{\kappa_3}\!\sim^{\kappa_4}_{\#}(\sigma_2\,\tau_2)} \quad \textsc{Co\_AppCo}$$

$$\frac{\begin{array}{c}\Gamma, \alpha^{\kappa} \vdash_{\mathsf{co}} \gamma : \tau_1\ {}^{\star}\!\sim^{\star}_{\#}\ \tau_2 \\ \Gamma \vdash_{\mathsf{ty}} \forall \alpha^{\kappa}.\tau_1 : \star \\ \Gamma \vdash_{\mathsf{ty}} \forall \alpha^{\kappa}.\tau_2 : \star\end{array}}{\Gamma \vdash_{\mathsf{co}} \forall \alpha^{\kappa}.\gamma : (\forall \alpha^{\kappa}.\tau_1)\ {}^{\star}\!\sim^{\star}_{\#}(\forall \alpha^{\kappa}.\tau_2)} \quad \textsc{Co\_ForAllCo\_TyHomo}$$

$$\frac{\begin{array}{l}\Gamma \vdash_{\mathsf{co}} \eta : \kappa_1\ {}^{\star}\!\sim^{\star}_{\#}\ \kappa_2 \\ \Gamma, {\alpha_1}^{\kappa_1}, {\alpha_2}^{\kappa_2}, c^{(\alpha_1\ {}^{\kappa_1}\!\sim^{\kappa_2}_{\#}\ \alpha_2)} \vdash_{\mathsf{co}} \gamma : \tau_1\ {}^{\star}\!\sim^{\star}_{\#}\ \tau_2 \\ \Gamma \vdash_{\mathsf{ty}} \forall {\alpha_1}^{\kappa_1}.\tau_1 : \star \\ \Gamma \vdash_{\mathsf{ty}} \forall {\alpha_2}^{\kappa_2}.\tau_2 : \star\end{array}}{\Gamma \vdash_{\mathsf{co}} \forall_{\eta}({\alpha_1}^{\kappa_1}, {\alpha_2}^{\kappa_2}, c).\gamma : (\forall {\alpha_1}^{\kappa_1}.\tau_1)\ {}^{\star}\!\sim^{\star}_{\#}(\forall {\alpha_2}^{\kappa_2}.\tau_2)} \quad \textsc{Co\_ForAllCo\_TyHetero}$$

$$\frac{\begin{array}{l} c \mathrel{\#} |\gamma| \\ \Gamma, c^{\phi} \vdash_{\mathsf{co}} \gamma : \tau_1 \,{}^{\star}\!\!\sim_{\#}^{\star} \tau_2 \\ \Gamma \vdash_{\mathsf{ty}} \forall c^{\phi}.\tau_1 : \star \\ \Gamma \vdash_{\mathsf{ty}} \forall c^{\phi}.\tau_2 : \star \end{array}}{\Gamma \vdash_{\mathsf{co}} \forall c^{\phi}.\gamma : (\forall c^{\phi}.\tau_1) \,{}^{\star}\!\!\sim_{\#}^{\star} (\forall c^{\phi}.\tau_2)} \quad \textsc{Co\_ForAllCo\_CoHomo}$$

$$\frac{\begin{array}{l} c_1 \mathrel{\#} |\gamma| \\ c_2 \mathrel{\#} |\gamma| \\ \Gamma \vdash_{\mathsf{co}} \eta : \phi_1 \,{}^{\star}\!\!\sim_{\#}^{\star} \phi_2 \\ \phi_1 \neq \phi_2 \\ \Gamma, c_1{}^{\phi_1}, c_2{}^{\phi_2} \vdash_{\mathsf{co}} \gamma : \tau_1 \,{}^{\star}\!\!\sim_{\#}^{\star} \tau_2 \\ \Gamma \vdash_{\mathsf{ty}} \forall c_1{}^{\phi_1}.\tau_1 : \star \\ \Gamma \vdash_{\mathsf{ty}} \forall c_2{}^{\phi_2}.\tau_2 : \star \end{array}}{\Gamma \vdash_{\mathsf{co}} \forall_{\eta}(c_1{}^{\phi_1}, c_2{}^{\phi_2}).\gamma : (\forall c_1{}^{\phi_1}.\tau_1) \,{}^{\star}\!\!\sim_{\#}^{\star} (\forall c_2{}^{\phi_2}.\tau_2)} \quad \textsc{Co\_ForAllCo\_CoHetero}$$

$$\frac{\begin{array}{l} z^{\phi} \in \Gamma \\ \phi = \tau_1 \,{}^{\kappa_1}\!\!\sim_{\#}^{\kappa_2} \tau_2 \end{array}}{\Gamma \vdash_{\mathsf{co}} z^{\phi} : \phi} \quad \textsc{Co\_CoVarCo}$$

$$\frac{\begin{array}{l} \Gamma \vdash_{\mathsf{ty}} \tau_1 : \kappa_1 \\ \Gamma \vdash_{\mathsf{ty}} \tau_2 : \kappa_2 \end{array}}{\Gamma \vdash_{\mathsf{co}} \tau_1 \mathrel{!\!\twoheadrightarrow} \tau_2 : \tau_1 \,{}^{\kappa_1}\!\!\sim_{\#}^{\kappa_2} \tau_2} \quad \textsc{Co\_UnsafeCo}$$

$$\frac{\Gamma \vdash_{\mathsf{co}} \gamma : \tau_1 \,{}^{\kappa_1}\!\!\sim_{\#}^{\kappa_2} \tau_2}{\Gamma \vdash_{\mathsf{co}} \mathsf{sym}\,\gamma : \tau_2 \,{}^{\kappa_2}\!\!\sim_{\#}^{\kappa_1} \tau_1} \quad \textsc{Co\_SymCo}$$

$$\frac{\begin{array}{l} \Gamma \vdash_{\mathsf{co}} \gamma_1 : \tau_1 \,{}^{\kappa_1}\!\!\sim_{\#}^{\kappa_2} \tau_2 \\ \Gamma \vdash_{\mathsf{co}} \gamma_2 : \tau_2 \,{}^{\kappa_2}\!\!\sim_{\#}^{\kappa_3} \tau_3 \end{array}}{\Gamma \vdash_{\mathsf{co}} \gamma_1 \mathbin{⨾} \gamma_2 : \tau_1 \,{}^{\kappa_1}\!\!\sim_{\#}^{\kappa_3} \tau_3} \quad \textsc{Co\_TransCo}$$

$$\frac{\begin{array}{l} \Gamma \vdash_{\mathsf{co}} \gamma : (T\, \overline{\sigma_j}^{\,j}) \,{}^{\kappa_1}\!\!\sim_{\#}^{\kappa_1'} (T\, \overline{\tau_j}^{\,j}) \\ \mathsf{length}\, \overline{\sigma_j}^{\,j} = \mathsf{length}\, \overline{\tau_j}^{\,j} \\ i < \mathsf{length}\, \overline{\sigma_j}^{\,j} \\ \Gamma \vdash_{\mathsf{ty}} \sigma_i : \kappa_2 \\ \Gamma \vdash_{\mathsf{ty}} \tau_i : \kappa_2' \\ \neg\, (\exists \gamma \text{ s.t. } \sigma_i = \gamma) \\ \neg\, (\exists \gamma \text{ s.t. } \tau_i = \gamma) \end{array}}{\Gamma \vdash_{\mathsf{co}} \mathsf{nth}^{i}\, \gamma : \sigma_i \,{}^{\kappa_2}\!\!\sim_{\#}^{\kappa_2'} \tau_i} \quad \textsc{Co\_NthCoTyCon}$$

$$\frac{\Gamma \vdash_{\mathsf{co}} \gamma : (\forall {z_1}^{\kappa_1}.\tau_1)\ {}^{\star}\!\sim^{\star}_{\#} (\forall {z_2}^{\kappa_2}.\tau_2)}{\Gamma \vdash_{\mathsf{co}} \mathsf{nth}^0\,\gamma : \kappa_1\ {}^{\star}\!\sim^{\star}_{\#}\ \kappa_2} \quad \textsc{Co\_NthCoForAll}$$

$$\frac{\begin{array}{l}\Gamma \vdash_{\mathsf{co}} \gamma : (\sigma_1\,\sigma_2)\ {}^{\kappa}\!\sim^{\kappa'}_{\#} (\tau_1\,\tau_2) \\ \Gamma \vdash_{\mathsf{ty}} \sigma_1 : \kappa_1 \\ \Gamma \vdash_{\mathsf{ty}} \tau_1 : \kappa_1'\end{array}}{\Gamma \vdash_{\mathsf{co}} \mathsf{left}\,\gamma : \sigma_1\ {}^{\kappa_1}\!\sim^{\kappa_1'}_{\#}\ \tau_1} \quad \textsc{Co\_LRCoLeft}$$

$$\frac{\begin{array}{l}\Gamma \vdash_{\mathsf{co}} \gamma : (\sigma_1\,\sigma_2)\ {}^{\kappa}\!\sim^{\kappa'}_{\#} (\tau_1\,\tau_2) \\ \Gamma \vdash_{\mathsf{ty}} \sigma_2 : \kappa_2 \\ \Gamma \vdash_{\mathsf{ty}} \tau_2 : \kappa_2' \\ \neg\,(\exists \gamma \text{ s.t. } \sigma_2 = \gamma) \\ \neg\,(\exists \gamma \text{ s.t. } \tau_2 = \gamma)\end{array}}{\Gamma \vdash_{\mathsf{co}} \mathsf{right}\,\gamma : \sigma_2\ {}^{\kappa_2}\!\sim^{\kappa_2'}_{\#}\ \tau_2} \quad \textsc{Co\_LRCoRight}$$

$$\frac{\begin{array}{l}\Gamma \vdash_{\mathsf{co}} \gamma : (\forall {z_1}^{\kappa_1}.\tau_1)\ {}^{\star}\!\sim^{\star}_{\#} (\forall {z_2}^{\kappa_2}.\tau_2) \\ \Gamma \vdash_{\mathsf{arg}} \omega : (\sigma_1 : \kappa_1', \sigma_2 : \kappa_2') \\ \kappa_1' <: \kappa_1 \\ \kappa_2' <: \kappa_2\end{array}}{\Gamma \vdash_{\mathsf{co}} \gamma\,\omega : (\tau_1[{z_1}^{\kappa_1} \mapsto \sigma_1])\ {}^{\star}\!\sim^{\star}_{\#} (\tau_2[{z_2}^{\kappa_2} \mapsto \sigma_2])} \quad \textsc{Co\_InstCo}$$

$$\frac{\begin{array}{l}C = T\,\overline{axBranch_k}^{\,k} \\ 0 \le ind < \mathsf{length}\,\overline{axBranch_k}^{\,k} \\ \forall \overline{n_i}^{\,i}.(\,\overline{\sigma_{1\,j}}^{\,j} \rightsquigarrow \tau_1) = (\,\overline{axBranch_k}^{\,k}\,)[ind] \\ \overline{\Gamma \vdash_{\mathsf{arg}} \omega_i : (\sigma_i' : \kappa_i', \tau_i' : \kappa_i'')}^{\,i} \\ \overline{subst_i}^{\,i} = \mathsf{inits}\,(\,\overline{[n_i \mapsto \sigma_i']}^{\,i}\,) \\ \overline{subst_i'}^{\,i} = \mathsf{inits}\,(\,\overline{[n_i \mapsto \tau_i']}^{\,i}\,) \\ \overline{n_i = {z_i}^{\kappa_i}}^{\,i} \\ \overline{\kappa_i' <: subst_i(\kappa_i)}^{\,i} \\ \overline{\kappa_i'' <: subst_i'(\kappa_i)}^{\,i} \\ \overline{\sigma_{2\,j} = \sigma_{1\,j}\,\overline{[n_i \mapsto \sigma_i']}^{\,i}}^{\,j} \\ \mathsf{no\_conflict}(C, \overline{\sigma_{2\,j}}^{\,j}, ind - 1) \\ \tau_2 = \tau_1\,\overline{[n_i \mapsto \tau_i']}^{\,i} \\ \sigma_2 = T\,\overline{\sigma_{2\,j}}^{\,j} \\ \Gamma \vdash_{\mathsf{ty}} \sigma_2 : \kappa \\ \Gamma \vdash_{\mathsf{ty}} \tau_2 : \kappa'\end{array}}{\Gamma \vdash_{\mathsf{co}} C\ ind\,\overline{\omega_i}^{\,i} : \sigma_2\ {}^{\kappa}\!\sim^{\kappa'}_{\#}\ \tau_2} \quad \textsc{Co\_AxiomInstCo}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathsf{co}} \gamma : \tau_1\ {}^{\kappa_1}\!\sim^{\kappa_2}_{\#}\ \tau_2 \\ \Gamma \vdash_{\mathsf{ty}} \tau_1 \triangleright \eta : \kappa_1'\end{array}}{\Gamma \vdash_{\mathsf{co}} \gamma \triangleright \eta : \tau_1 \triangleright \eta\ {}^{\kappa_1'}\!\sim^{\kappa_2}_{\#}\ \tau_2} \quad \textsc{Co\_CoherenceCo}$$

$$\frac{\Gamma \vdash_{\mathsf{co}} \gamma : \tau_1 \ {}^{\kappa_1}{\sim}^{\kappa_2}_{\#} \ \tau_2}{\Gamma \vdash_{\mathsf{co}} \mathsf{kind}\, \gamma : \kappa_1 \ {}^{\star}{\sim}^{\star}_{\#} \ \kappa_2} \quad \textsc{Co\_KindCo}$$

The # checks in the rules Co_ForAllCo_Co... are freshness checks. The variable to the left of # may not appear in the *erased* version of the coercion on the right. See the *Down with kinds* paper for details.

In Co_AxiomInstCo, the use of inits creates substitutions from the first $i$ mappings in $\overline{[n_i \mapsto \sigma_i]}^{\,i}$. This has the effect of folding the substitution over the kinds for kind-checking.

### 4.7 Coercion argument typing

$\boxed{\Gamma \vdash_{\mathsf{arg}} \omega : (\tau_1 : \kappa_1, \tau_2 : \kappa_2)}$ Coercion argument kinding, *coreSyn/CoreLint.lhs*:`lintCoArg`

$$\frac{\Gamma \vdash_{\mathsf{co}} \gamma : \tau_1 \ {}^{\kappa_1}{\sim}^{\kappa_2}_{\#} \ \tau_2}{\Gamma \vdash_{\mathsf{arg}} \gamma : (\tau_1 : \kappa_1, \tau_2 : \kappa_2)} \quad \textsc{Arg\_TyCoArg}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathsf{co}} \gamma_1 : \phi_1 \\ \Gamma \vdash_{\mathsf{co}} \gamma_2 : \phi_2\end{array}}{\Gamma \vdash_{\mathsf{arg}} (\gamma_1, \gamma_2) : (\gamma_1 : \phi_1, \gamma_2 : \phi_2)} \quad \textsc{Arg\_CoCoArg}$$

### 4.8 Name consistency

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

$\boxed{\Gamma \vdash_{\mathsf{n}} n \ \mathsf{ok}}$ Name consistency check, *coreSyn/CoreLint.lhs*:`lintSingleBinding#lintBinder`

$$\frac{\Gamma \vdash_{\mathsf{ty}} \tau : \kappa}{\Gamma \vdash_{\mathsf{n}} x^{\tau} \ \mathsf{ok}} \quad \textsc{Name\_Id}$$

$$\frac{}{\Gamma \vdash_{\mathsf{n}} \alpha^{\kappa} \ \mathsf{ok}} \quad \textsc{Name\_TyVar}$$

$\boxed{\Gamma \vdash_{\mathsf{bnd}} n \ \mathsf{ok}}$ Binding consistency, *coreSyn/CoreLint.lhs*:`lintBinder`

$$\frac{\Gamma \vdash_{\mathsf{ty}} \tau : \kappa}{\Gamma \vdash_{\mathsf{bnd}} x^{\tau} \ \mathsf{ok}} \quad \textsc{Binding\_Id}$$

$$\frac{\Gamma \vdash_{\mathsf{k}} \kappa \ \mathsf{ok}}{\Gamma \vdash_{\mathsf{bnd}} \alpha^{\kappa} \ \mathsf{ok}} \quad \textsc{Binding\_TyVar}$$

## 4.9 Substitution consistency

$\boxed{\Gamma \vdash_{\mathsf{subst}} n \mapsto \tau\ \mathsf{ok}}$ Substitution consistency, *coreSyn/CoreLint.lhs*:`checkTyKind`

$$\frac{\begin{array}{l}\Gamma \vdash_{\mathsf{ty}} \tau : \kappa_2 \\ \kappa_2 <: \kappa_1\end{array}}{\Gamma \vdash_{\mathsf{subst}} z^{\kappa_1} \mapsto \tau\ \mathsf{ok}} \quad \textsc{Subst\_Type}$$

## 4.10 Case alternative consistency

$\boxed{\Gamma; \sigma \vdash_{\mathsf{alt}} alt : \tau}$ Case alternative consistency, *coreSyn/CoreLint.lhs*:`lintCoreAlt`

$$\frac{\Gamma \vdash_{\mathsf{tm}} e : \tau}{\Gamma; \sigma \vdash_{\mathsf{alt}} \_ \to e : \tau} \quad \textsc{Alt\_DEFAULT}$$

$$\frac{\begin{array}{l}\sigma = \mathsf{literalType}\, \mathsf{lit} \\ \Gamma \vdash_{\mathsf{tm}} e : \tau\end{array}}{\Gamma; \sigma \vdash_{\mathsf{alt}} \mathsf{lit} \to e : \tau} \quad \textsc{Alt\_LitAlt}$$

$$\frac{\begin{array}{l}T = \mathsf{dataConTyCon}\, K \\ \neg\, (\mathsf{isNewTyCon}\ T) \\ \tau_1 = \mathsf{dataConRepType}\, K \\ \tau_2 = \tau_1 \{\, \overline{\sigma_j}^{\,j}\, \} \\ \overline{\Gamma \vdash_{\mathsf{bnd}} n_i\ \mathsf{ok}}^{\,i} \\ \Gamma' = \Gamma, \overline{n_i}^{\,i} \\ \Gamma' \vdash_{\mathsf{altbnd}} \overline{n_i}^{\,i} : \tau_2 \leadsto T\, \overline{\sigma_j}^{\,j} \\ \Gamma' \vdash_{\mathsf{tm}} e : \tau\end{array}}{\Gamma; T\, \overline{\sigma_j}^{\,j} \vdash_{\mathsf{alt}} K\, \overline{n_i}^{\,i} \to e : \tau} \quad \textsc{Alt\_DataAlt}$$

## 4.11 Telescope substitution

$\boxed{\tau' = \tau\{\, \overline{\sigma_i}^{\,i}\, \}}$ Telescope substitution, *types/Type.lhs*:`applyTys`

$$\frac{}{\tau = \tau\{\,\}} \quad \textsc{ApplyTys\_Empty}$$

$$\frac{\begin{array}{l}\tau' = \tau\{\, \overline{\sigma_i}^{\,i}\, \} \\ \tau'' = \tau'[n \mapsto \sigma]\end{array}}{\tau'' = (\forall n.\tau)\{\sigma, \overline{\sigma_i}^{\,i}\}} \quad \textsc{ApplyTys\_Ty}$$

### 4.12 Case alternative binding consistency

$\boxed{\Gamma \vdash_{\mathsf{altbnd}} vars : \tau_1 \leadsto \tau_2}$ Case alternative binding consistency, *coreSyn/CoreLint.lhs*:`lintAltBinders`

$$\frac{}{\Gamma \vdash_{\mathsf{altbnd}} \cdot : \tau \leadsto \tau} \quad \textsc{AltBinders\_Empty}$$

$$\frac{\begin{array}{l}\Gamma \vdash_{\mathsf{subst}} \beta^{\kappa'} \mapsto \alpha^{\kappa}\ \mathsf{ok} \\ \Gamma \vdash_{\mathsf{altbnd}} \overline{n_i}^{\,i} : \tau[\beta^{\kappa'} \mapsto \alpha^{\kappa}] \leadsto \sigma\end{array}}{\Gamma \vdash_{\mathsf{altbnd}} \alpha^{\kappa}, \overline{n_i}^{\,i} : (\forall \beta^{\kappa'}.\tau) \leadsto \sigma} \quad \textsc{AltBinders\_TyVar}$$

$$\frac{\Gamma \vdash_{\mathsf{altbnd}} \overline{n_i}^{\,i} : \tau[z^{\phi} \mapsto c^{\phi}] \leadsto \sigma}{\Gamma \vdash_{\mathsf{altbnd}} c^{\phi}, \overline{n_i}^{\,i} : (\forall z^{\phi}.\tau) \leadsto \sigma} \quad \textsc{AltBinders\_IdCoercion}$$

$$\frac{\Gamma \vdash_{\mathsf{altbnd}} \overline{n_i}^{\,i} : \tau_2 \leadsto \sigma}{\Gamma \vdash_{\mathsf{altbnd}} x^{\tau_1}, \overline{n_i}^{\,i} : (\tau_1 \to \tau_2) \leadsto \sigma} \quad \textsc{AltBinders\_IdTerm}$$

### 4.13 Arrow kinding

$\boxed{\Gamma \vdash_{\to} \kappa_1 \to \kappa_2 : \kappa}$ Arrow kinding, *coreSyn/CoreLint.lhs*:`lintArrow`

$$\frac{\begin{array}{c}\kappa_1 \in \{\star, \#\} \\ \kappa_2 \in \{\star, \#\}\end{array}}{\Gamma \vdash_{\to} \kappa_1 \to \kappa_2 : \star} \quad \textsc{Arrow\_Kind}$$

### 4.14 Type application kinding

$\boxed{\Gamma \vdash_{\mathsf{app}} \overline{(\sigma_i : \kappa_i)}^{\,i} : \kappa_1 \leadsto \kappa_2}$ Type application kinding, *coreSyn/CoreLint.lhs*:`lint_app`

$$\frac{}{\Gamma \vdash_{\mathsf{app}} \cdot : \kappa \leadsto \kappa} \quad \textsc{App\_Empty}$$

$$\frac{\begin{array}{l}\kappa <: \kappa_1 \\ \Gamma \vdash_{\mathsf{app}} \overline{(\tau_i : \kappa_i)}^{\,i} : \kappa_2 \leadsto \kappa'\end{array}}{\Gamma \vdash_{\mathsf{app}} (\tau : \kappa), \overline{(\tau_i : \kappa_i)}^{\,i} : (\kappa_1 \to \kappa_2) \leadsto \kappa'} \quad \textsc{App\_FunTy}$$

$$\frac{\begin{array}{l}\kappa <: \kappa_1 \\ \Gamma \vdash_{\mathsf{app}} \overline{(\tau_i : \kappa_i)}^{\,i} : \kappa_2[z^{\kappa_1} \mapsto \tau] \leadsto \kappa'\end{array}}{\Gamma \vdash_{\mathsf{app}} (\tau : \kappa), \overline{(\tau_i : \kappa_i)}^{\,i} : (\forall z^{\kappa_1}.\kappa_2) \leadsto \kappa'} \quad \textsc{App\_ForAllTy}$$

### 4.15 Sub-kinding

$\boxed{\kappa_1 <: \kappa_2}$ Sub-kinding, *types/Kind.lhs*:`isSubKind`

$$\frac{}{\kappa <: \kappa} \quad \textsc{SubKind\_Refl}$$

$$\frac{}{\# <: \mathsf{OpenKind}} \quad \textsc{SubKind\_UnliftedTypeKindOpen}$$

$$\frac{}{\star <: \mathsf{OpenKind}} \quad \textsc{SubKind\_LiftedTypeKindOpen}$$

### 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.

$\boxed{\mathsf{no\_conflict}(C, \overline{\sigma_j}^{\,j}, ind)}$ Branched axiom conflict checking, *coreSyn/CoreLint.lhs*:`lintCoercion#check_no_conflict`

$$\frac{}{\mathsf{no\_conflict}(C, \overline{\sigma_i}^{\,i}, -1)} \quad \textsc{NoConflict\_NoBranch}$$

$$\frac{\begin{array}{l} C = T\ \overline{axBranch_k}^{\,k} \\ \forall \overline{n_i}^{\,i}.(\overline{\tau_j}^{\,j} \rightsquigarrow \tau') = (\overline{axBranch_k}^{\,k})[ind] \\ \mathsf{apart}\ (\overline{\sigma_j}^{\,j}, \overline{\tau_j}^{\,j}) \\ \mathsf{no\_conflict}(C, \overline{\sigma_j}^{\,j}, ind - 1) \end{array}}{\mathsf{no\_conflict}(C, \overline{\sigma_j}^{\,j}, ind)} \quad \textsc{NoConflict\_Branch}$$

The judgment apart checks to see whether two lists of types are surely apart. It checks to see if *types/Unify.lhs*:`tcApartTys` returns `SurelyApart`. Two types are apart if neither type is a type family application and if they do not unify.