# System FC, as implemented in GHC[1]

July 30, 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:

| | |
|---|---|
| $x$ | Term-level variable names |
| $\alpha$, $\beta$ | Type-level variable names |
| $N$ | Type-level constructor names |
| $K$ | Term-level data constructor 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:

$\mathsf{lit}$ ::= 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:

| | | | | |
|---|---|---|---|---|
| $z$ | ::= | | Term or type name | |
| | $\vert$ | $\alpha$ | Type-level name | foo |
| | $\vert$ | $x$ | Term-level name | |

| | | | |
|---|---|---|---|
| $n$, $m$ | ::= | | Variable names, *basicTypes/Var.lhs*:`Var` |
| | $\vert$ | $z^\tau$ | Name, labeled with type/kind |

### 2.4 Expressions

The datatype that represents expressions:

[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.

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

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

Case alternatives:

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

Constructors as used in patterns:

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

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

| | | |
|---|---|---|
| *tick* | ::= | Internal notes, *coreSyn/CoreSyn.lhs*:`Tickish` |

A program is just a list of bindings:

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

## 2.5 Types

| | | | |
|---|---|---|---|
| $\tau,\ \kappa,\ \sigma$ | $::=$ | | Types/kinds, *types/TypeRep.lhs*:`Type` |
| | $\mid$ | $n$ | Variable |
| | $\mid$ | $\tau_1\,\tau_2$ | Application |
| | $\mid$ | $T\,\overline{\tau_i}^{\,i}$ | Application of type constructor |
| | $\mid$ | $\tau_1 \to \tau_2$ | Function |
| | $\mid$ | $\forall n.\tau$ | Polymorphism |
| | $\mid$ | $\mathsf{lit}$ | Type-level literal |

There are some invariants on types:

- 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.

## 2.6 Coercions

| | | | |
|---|---|---|---|
| $\gamma$ | $::=$ | | Coercions, *types/Coercion.lhs*:`Coercion` |
| | $\mid$ | $\langle\tau\rangle_\rho$ | Reflexivity |
| | $\mid$ | $T_\rho\,\overline{\gamma_i}^{\,i}$ | Type constructor application |
| | $\mid$ | $\gamma_1\,\gamma_2$ | Application |
| | $\mid$ | $\forall n.\gamma$ | Polymorphism |
| | $\mid$ | $n$ | Variable |
| | $\mid$ | $C\ \mathit{ind}\,\overline{\gamma_j}^{\,j}$ | Axiom application |
| | $\mid$ | $\tau_1 \overset{!}{\twoheadrightarrow}_\rho \tau_2$ | Universal coercion |
| | $\mid$ | $\mathsf{sym}\,\gamma$ | Symmetry |
| | $\mid$ | $\gamma_1 \mathbin{;} \gamma_2$ | Transitivity |
| | $\mid$ | $\mathsf{nth}_i\,\gamma$ | Projection (0-indexed) |
| | $\mid$ | $\mathit{LorR}\,\gamma$ | Left/right projection |
| | $\mid$ | $\gamma\,\tau$ | Type application |

Invariants on coercions:

- $\langle\tau_1\,\tau_2\rangle_\rho$ is used; never $\langle\tau_1\rangle_\rho\,\langle\tau_2\rangle_{\mathsf{N}}$.
- If $\langle T\rangle_\rho$ is applied to some coercions, at least one of which is not reflexive, use $T_\rho\,\overline{\gamma_i}^{\,i}$, never $\langle T\rangle_\rho\,\gamma_1\,\gamma_2 \ldots$.
- The $T$ in $T_\rho\,\overline{\gamma_i}^{\,i}$ is never a type synonym, though it could be a type function.

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` for more background.

| | | |
|---|---|---|
| $\rho$ | ::= | Roles, *types/CoAxiom.lhs*:`Role` |
| \| | $\mathsf{N}$ | Nominal |
| \| | $\mathsf{R}$ | Representational |
| \| | $\mathsf{P}$ | Phantom |

Is it a left projection or a right projection?

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

Axioms:

| | | | |
|---|---|---|---|
| $C$ | ::= | | Axioms, *types/TyCon.lhs*:`CoAxiom` |
| | \| | $T_\rho\ \overline{axBranch_i}^{\,i}$ | Axiom |
| $axBranch,\ b$ | ::= | | Axiom branches, *types/TyCon.lhs*:`CoAxBranch` |
| | \| | $\forall \overline{n_{\rho_i}}^{\,i}.(\overline{\tau_j}^{\,j} \rightsquigarrow \sigma)$ | Axiom branch |

The definition for *axBranch* above does not include the list of incompatible branches (field `cab_incomps` of `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 $\mathsf{no\_conflict}$. See Section 4.16.

## 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` |
| \| | $(\rightarrow)$ | Arrow |
| \| | $N^\kappa$ | Named tycon: algebraic, tuples, and synonyms |
| \| | $H$ | Primitive tycon |
| \| | $'K$ | Promoted data constructor |
| \| | $'T$ | Promoted type 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 |
| \| | $(\sim_{\mathsf{R}\#})$ | Unboxed representational equality |
| \| | $\Box$ | Sort of kinds |
| \| | $*$ | Kind of lifted types |
| \| | $\#$ | Kind of unlifted types |
| \| | $\mathsf{OpenKind}$ | Either $*$ or $\#$ |
| \| | $\mathsf{Constraint}$ | Constraint |

# 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.

$$
\begin{array}{llll}
\Gamma & ::= & & \text{List of bindings, } \textit{coreSyn/CoreLint.lhs}\text{:}\texttt{LintM} \\
 & | & n & \text{Single binding} \\
 & | & \overline{\Gamma_i}^{\,i} & \text{Context concatenation}
\end{array}
$$

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`

$$
\frac{\begin{array}{l}\Gamma = \overline{\mathsf{vars\_of}\ \mathit{binding}_i}^{\,i} \\ \mathsf{no\_duplicates}\ \overline{\mathit{binding}_i}^{\,i} \\ \overline{\Gamma \vdash_{\mathsf{bind}} \mathit{binding}_i}^{\,i}\end{array}}{\vdash_{\mathsf{prog}} \overline{\mathit{binding}_i}^{\,i}} \quad \textsc{Prog\_CoreBindings}
$$

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

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

$$
\frac{\Gamma \vdash_{\mathsf{sbind}} n \leftarrow e}{\Gamma \vdash_{\mathsf{bind}} n = e} \quad \textsc{Binding\_NonRec}
$$

$$
\frac{\overline{\Gamma \vdash_{\mathsf{sbind}} n_i \leftarrow e_i}^{\,i}}{\Gamma \vdash_{\mathsf{bind}} \mathbf{rec}\ \overline{n_i = e_i}^{\,i}} \quad \textsc{Binding\_Rec}
$$

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

$$\frac{\begin{array}{l}\Gamma \vdash_{\mathsf{tm}} e : \tau \\ \Gamma \vdash_{\mathsf{n}} z^{\tau}\ \mathsf{ok} \\ \overline{m_i}^{\,i} = \mathit{fv}(\tau) \\ \overline{m_i \in \Gamma}^{\,i}\end{array}}{\Gamma \vdash_{\mathsf{sbind}} z^{\tau} \leftarrow e} \quad \textsc{SBinding\_SingleBinding}$$

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`

$$\frac{\begin{array}{l}x^{\tau} \in \Gamma \\ \neg\,(\exists \tau_1, \tau_2, \kappa \text{ s.t. } \tau = \tau_1 \sim_{\#}^{\kappa} \tau_2)\end{array}}{\Gamma \vdash_{\mathsf{tm}} x^{\tau} : \tau} \quad \textsc{Tm\_Var}$$

$$\frac{\tau = \mathsf{literalType\,lit}}{\Gamma \vdash_{\mathsf{tm}} \mathsf{lit} : \tau} \quad \textsc{Tm\_Lit}$$

$$\frac{\begin{array}{l}\Gamma \vdash_{\mathsf{tm}} e : \sigma \\ \Gamma \vdash_{\mathsf{co}} \gamma : \sigma \sim_{\mathsf{R}}^{\kappa} \tau\end{array}}{\Gamma \vdash_{\mathsf{tm}} e \triangleright \gamma : \tau} \quad \textsc{Tm\_Cast}$$

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

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

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

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

$$\dfrac{\begin{array}{c}\Gamma \vdash_{\mathsf{tm}} e_1 : \forall \alpha^\kappa . \tau\\ \Gamma \vdash_{\mathsf{subst}} \alpha^\kappa \mapsto \sigma \;\mathsf{ok}\end{array}}{\Gamma \vdash_{\mathsf{tm}} e_1\, \sigma : \tau[\alpha^\kappa \mapsto \sigma]} \quad \textsc{Tm\_AppType}$$

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

$$\dfrac{\begin{array}{c}\Gamma \vdash_{\mathsf{ty}} \tau : \kappa\\ \Gamma, x^\tau \vdash_{\mathsf{tm}} e : \sigma\end{array}}{\Gamma \vdash_{\mathsf{tm}} \lambda x^\tau . e : \tau \to \sigma} \quad \textsc{Tm\_LamId}$$

$$\dfrac{\begin{array}{l}\Gamma' = \Gamma, \alpha^\kappa\\ \Gamma \vdash_{\mathsf{k}} \kappa \;\mathsf{ok}\\ \Gamma' \vdash_{\mathsf{tm}} e : \tau\end{array}}{\Gamma \vdash_{\mathsf{tm}} \lambda \alpha^\kappa . e : \forall \alpha^\kappa . \tau} \quad \textsc{Tm\_LamTy}$$

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

$$\dfrac{\Gamma \vdash_{\mathsf{co}} \gamma : \tau_1 \sim^\kappa_{\mathsf{N}} \tau_2}{\Gamma \vdash_{\mathsf{tm}} \gamma : \tau_1 \sim^\kappa_{\#} \tau_2} \quad \textsc{Tm\_Coercion}$$

- 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`.

- There is one more case for $\Gamma \vdash_{\mathsf{tm}} e : \tau$, for type expressions. This is included in the GHC code but is elided here because the case is never used in practice. Type expressions can only appear in arguments to functions, and these are handled in TM_APPTYPE.
- 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`

$$\frac{z^{\kappa} \in \Gamma}{\Gamma \vdash_{\mathsf{ty}} z^{\kappa} : \kappa} \quad \text{TY\_TYVARTY}$$

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

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathsf{ty}} \tau_1 : \kappa_1 \\ \Gamma \vdash_{\mathsf{ty}} \tau_2 : \kappa_2 \\ \Gamma \vdash_{\rightarrow} \kappa_1 \rightarrow \kappa_2 : \kappa\end{array}}{\Gamma \vdash_{\mathsf{ty}} \tau_1 \rightarrow \tau_2 : \kappa} \quad \text{TY\_FUNTY}$$

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

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathsf{k}} \kappa_1\ \mathsf{ok} \\ \Gamma, z^{\kappa_1} \vdash_{\mathsf{ty}} \tau : \kappa_2\end{array}}{\Gamma \vdash_{\mathsf{ty}} \forall z^{\kappa_1}.\tau : \kappa_2} \quad \text{TY\_FORALLTY}$$

$$\frac{\Gamma \vdash_{\mathsf{tylit}} \mathsf{lit} : \kappa}{\Gamma \vdash_{\mathsf{ty}} \mathsf{lit} : \kappa} \quad \text{TY\_LITTY}$$

## 4.5 Kind validity

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

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

## 4.6 Coercion typing

$\boxed{\Gamma \vdash_{\mathsf{co}} \gamma : \tau_1 \sim_{\rho}^{\kappa} \tau_2}$ Coercion typing, *coreSyn/CoreLint.lhs*:`lintCoercion`

$$\frac{\Gamma \vdash_{\mathsf{ty}} \tau : \kappa}{\Gamma \vdash_{\mathsf{co}} \langle \tau \rangle_{\rho} : \tau \sim_{\rho}^{\kappa} \tau} \quad \textsc{Co\_Refl}$$

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

$$\frac{\begin{array}{l}T \neq (\rightarrow) \\ \overline{\rho_i}^{\,i} = \mathsf{tyConRolesX}\ \rho\ T \\ \overline{\Gamma \vdash_{\mathsf{co}} \gamma_i : \sigma_i \sim_{\rho_i}^{\kappa_i} \tau_i}^{\,i} \\ \Gamma \vdash_{\mathsf{app}} \overline{(\sigma_i : \kappa_i)}^{\,i} : \mathsf{tyConKind}\ T \rightsquigarrow \kappa\end{array}}{\Gamma \vdash_{\mathsf{co}} T_{\rho}\, \overline{\gamma_i}^{\,i} : T\, \overline{\sigma_i}^{\,i} \sim_{\rho}^{\kappa} T\, \overline{\tau_i}^{\,i}} \quad \textsc{Co\_TyConAppCo}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathsf{co}} \gamma_1 : \sigma_1 \sim_{\rho}^{\kappa_1} \tau_1 \\ \Gamma \vdash_{\mathsf{co}} \gamma_2 : \sigma_2 \sim_{\mathsf{N}}^{\kappa_2} \tau_2 \\ \Gamma \vdash_{\mathsf{app}} (\sigma_2 : \kappa_2) : \kappa_1 \rightsquigarrow \kappa\end{array}}{\Gamma \vdash_{\mathsf{co}} \gamma_1\, \gamma_2 : (\sigma_1\, \sigma_2) \sim_{\rho}^{\kappa} (\tau_1\, \tau_2)} \quad \textsc{Co\_AppCo}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathsf{co}} \gamma_1 : \sigma_1 \sim_{\mathsf{P}}^{\kappa_1} \tau_1 \\ \Gamma \vdash_{\mathsf{co}} \gamma_2 : \sigma_2 \sim_{\mathsf{P}}^{\kappa_2} \tau_2 \\ \Gamma \vdash_{\mathsf{app}} (\sigma_2 : \kappa_2) : \kappa_1 \rightsquigarrow \kappa\end{array}}{\Gamma \vdash_{\mathsf{co}} \gamma_1\, \gamma_2 : (\sigma_1\, \sigma_2) \sim_{\mathsf{P}}^{\kappa} (\tau_1\, \tau_2)} \quad \textsc{Co\_AppCoPhantom}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathsf{k}} \kappa_1\ \mathsf{ok} \\ \Gamma, z^{\kappa_1} \vdash_{\mathsf{co}} \gamma : \sigma \sim_{\rho}^{\kappa_2} \tau\end{array}}{\Gamma \vdash_{\mathsf{co}} \forall z^{\kappa_1}.\gamma : (\forall z^{\kappa_1}.\sigma) \sim_{\rho}^{\kappa_2} (\forall z^{\kappa_1}.\tau)} \quad \textsc{Co\_ForAllCo}$$

$$\frac{z^{(\tau \sim_{\#}^{\Box} \tau)} \in \Gamma}{\Gamma \vdash_{\mathsf{co}} z^{(\tau \sim_{\#}^{\Box} \tau)} : \tau \sim_{\mathsf{N}}^{\Box} \tau} \quad \textsc{Co\_CoVarCoBox}$$

$$\frac{\begin{array}{l} z^{(\sigma \sim_{\#}^{\kappa} \tau)} \in \Gamma \\ \kappa \neq \Box \end{array}}{\Gamma \vdash_{\mathsf{co}} z^{(\sigma \sim_{\#}^{\kappa} \tau)} : \sigma \sim_{\mathsf{N}}^{\kappa} \tau} \quad \textsc{Co\_CoVarCoNom}$$

$$\frac{\begin{array}{l} z^{(\sigma \sim_{\mathsf{R}\#}^{\kappa} \tau)} \in \Gamma \\ \kappa \neq \Box \end{array}}{\Gamma \vdash_{\mathsf{co}} z^{(\sigma \sim_{\mathsf{R}\#}^{\kappa} \tau)} : \sigma \sim_{\mathsf{R}}^{\kappa} \tau} \quad \textsc{Co\_CoVarCoRepr}$$

$$\frac{\Gamma \vdash_{\mathsf{ty}} \tau_1 : \kappa}{\Gamma \vdash_{\mathsf{co}} \tau_1 \mathrel{!\!\!\twoheadrightarrow}\ _{\rho} \tau_2 : \tau_1 \sim_{\rho}^{\kappa} \tau_2} \quad \textsc{Co\_UnivCo}$$

$$\frac{\Gamma \vdash_{\mathsf{co}} \gamma : \tau_1 \sim_{\rho}^{\kappa} \tau_2}{\Gamma \vdash_{\mathsf{co}} \mathsf{sym}\,\gamma : \tau_2 \sim_{\rho}^{\kappa} \tau_1} \quad \textsc{Co\_SymCo}$$

$$\frac{\begin{array}{l} \Gamma \vdash_{\mathsf{co}} \gamma_1 : \tau_1 \sim_{\rho}^{\kappa} \tau_2 \\ \Gamma \vdash_{\mathsf{co}} \gamma_2 : \tau_2 \sim_{\rho}^{\kappa} \tau_3 \end{array}}{\Gamma \vdash_{\mathsf{co}} \gamma_1 \mathbin{;} \gamma_2 : \tau_1 \sim_{\rho}^{\kappa} \tau_3} \quad \textsc{Co\_TransCo}$$

$$\frac{\begin{array}{l} \Gamma \vdash_{\mathsf{co}} \gamma : (T\, \overline{\sigma_j}^{\,j}) \sim_{\rho}^{\kappa} (T\, \overline{\tau_j}^{\,j}) \\ \mathsf{length}\, \overline{\sigma_j}^{\,j} = \mathsf{length}\, \overline{\tau_j}^{\,j} \\ i < \mathsf{length}\, \overline{\sigma_j}^{\,j} \\ \Gamma \vdash_{\mathsf{ty}} \sigma_i : \kappa \\ \rho' = (\mathsf{tyConRolesX}\, \rho\, T)[i] \end{array}}{\Gamma \vdash_{\mathsf{co}} \mathsf{nth}_i\, \gamma : \sigma_i \sim_{\rho'}^{\kappa} \tau_i} \quad \textsc{Co\_NthCo}$$

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

$$\frac{\begin{array}{l} \Gamma \vdash_{\mathsf{co}} \gamma : (\sigma_1\, \sigma_2) \sim_{\mathsf{N}}^{\kappa'} (\tau_1\, \tau_2) \\ \Gamma \vdash_{\mathsf{ty}} \sigma_2 : \kappa \end{array}}{\Gamma \vdash_{\mathsf{co}} \mathsf{right}\, \gamma : \sigma_2 \sim_{\mathsf{N}}^{\kappa} \tau_2} \quad \textsc{Co\_LRCoRight}$$

$$\frac{\begin{array}{l} \Gamma \vdash_{\mathsf{co}} \gamma : \forall m.\sigma \sim_{\rho}^{\kappa} \forall n.\tau \\ \Gamma \vdash_{\mathsf{ty}} \tau_0 : \kappa_0 \\ m = z^{\kappa_1} \\ \kappa_0 <: \kappa_1 \end{array}}{\Gamma \vdash_{\mathsf{co}} \gamma\, \tau_0 : \sigma[m \mapsto \tau_0] \sim_{\rho}^{\kappa} \tau[n \mapsto \tau_0]} \quad \textsc{Co\_InstCo}$$

$$\frac{\begin{array}{l} C = T_{\rho_0}\ \overline{axBranch_k}^{\,k} \\ 0 \leq ind < \mathsf{length}\ \overline{axBranch_k}^{\,k} \\ \forall \overline{n_{i\ \rho_i}}^{\,i}.(\overline{\sigma_{1\,j}}^{\,j} \rightsquigarrow \tau_1) = (\ \overline{axBranch_k}^{\,k}\ )[ind] \\ \overline{\Gamma \vdash_{\mathsf{co}} \gamma_i : \sigma'_i \sim_{\rho_i}^{\kappa'_i} \tau'_i}^{\,i} \\ \overline{subst_i}^{\,i} = \mathsf{inits}\,(\ \overline{[n_i \mapsto \sigma'_i]}^{\,i}\ ) \\ \overline{n_i = z_i{}^{\kappa_i}}^{\,i} \\ \overline{\kappa'_i <: subst_i(\kappa_i)}^{\,i} \\ \mathsf{no\_conflict}(C, \overline{\sigma_{2\,j}}^{\,j}, ind, ind - 1) \\ \overline{\sigma_{2\,j} = \sigma_{1\,j}\ \overline{[n_i \mapsto \sigma'_i]}^{\,i}}^{\,j} \\ \tau_2 = \tau_1\ \overline{[n_i \mapsto \tau'_i]}^{\,i} \\ \Gamma \vdash_{\mathsf{ty}} \tau_2 : \kappa \end{array}}{\Gamma \vdash_{\mathsf{co}} C\ ind\ \overline{\gamma_i}^{\,i} : T\ \overline{\sigma_{2\,j}}^{\,j} \sim_{\rho_0}^{\kappa} \tau_2} \quad \textsc{Co\_AxiomInstCo}$$

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.

See Section 4.15 for more information about tyConRolesX.

## 4.7 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`

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

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

## 4.8 Substitution consistency

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

$$\frac{\Gamma \vdash_{\mathsf{k}} \kappa \ \mathsf{ok}}{\Gamma \vdash_{\mathsf{subst}} z^{\Box} \mapsto \kappa \ \mathsf{ok}} \quad \textsc{Subst\_Kind}$$

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

## 4.9 Case alternative consistency

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

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

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

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

## 4.10 Telescope substitution

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

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

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

## 4.11 Case alternative binding consistency

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

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

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

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

## 4.12 Arrow kinding

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

$$\frac{}{\Gamma \vdash_{\to} \Box \to \kappa_2 : \Box}\quad \textsc{Arrow\_Box}$$

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

## 4.13 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`

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

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

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

### 4.14 Sub-kinding

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

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

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

$$\frac{}{* <: \mathsf{OpenKind}} \quad \textsc{SubKind\_LiftedTypeKind}$$

$$\frac{}{\mathsf{Constraint} <: \mathsf{OpenKind}} \quad \textsc{SubKind\_Constraint}$$

$$\frac{}{\mathsf{Constraint} <: *} \quad \textsc{SubKind\_ConstraintLifted}$$

$$\frac{}{* <: \mathsf{Constraint}} \quad \textsc{SubKind\_LiftedConstraint}$$

### 4.15 Role inference

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

The checks encoded in the following judgments are run from *typecheck/TcTyClsDecls.lhs*:`checkValidTyCon` when `-dcore-lint` is set.

$\boxed{\mathsf{validRoles}\ T}$ Type constructor role validity, *typecheck/TcTyClsDecls.lhs*:`checkValidRoles`

$$\dfrac{\begin{array}{c}\overline{K_i}^{\,i} \;=\; \mathsf{tyConDataCons}\; T \\ \overline{\rho_j}^{\,j} = \mathsf{tyConRoles}\; T \\ \overline{\mathsf{validDcRoles}\,\overline{\rho_j}^{\,j}\; K_i}^{\,i}\end{array}}{\mathsf{validRoles}\; T} \quad \textsc{Cvr\_DataCons}$$

$\boxed{\mathsf{validDcRoles}\,\overline{\rho_a}^{\,a}\; K}$ Data constructor role validity, *typecheck/TcTyClsDecls.lhs*:`check_dc_roles`

$$\dfrac{\begin{array}{c}\forall\overline{n_a}^{\,a}.\forall\overline{m_b}^{\,b}.\overline{\tau_c}^{\,c} \;\to\; T\;\overline{n_a}^{\,a} = \mathsf{dataConRepType}\; K \\ \overline{\overline{n_a : \rho_a}^{\,a}, \overline{m_b : \mathsf{N}}^{\,b} \vdash_{\mathsf{ctr}} \tau_c : \mathsf{R}}^{\,c}\end{array}}{\mathsf{validDcRoles}\,\overline{\rho_a}^{\,a}\; K} \quad \textsc{Cdr\_Args}$$

$\boxed{\Omega \vdash_{\mathsf{ctr}} \tau : \rho}$ Type role validity, *typecheck/TcTyClsDecls.lhs*:`check_ty_roles`

$$\dfrac{\begin{array}{c}\Omega(n) = \rho' \\ \rho' \leq \rho\end{array}}{\Omega \vdash_{\mathsf{ctr}} n : \rho} \quad \textsc{Ctr\_TyVarTy}$$

$$\dfrac{\begin{array}{c}\overline{\rho_i}^{\,i} = \mathsf{tyConRoles}\; T \\ \overline{\rho_i \;\in\; \{\mathsf{N},\mathsf{R}\} \;\Longrightarrow\; \Omega \vdash_{\mathsf{ctr}} \tau_i : \rho_i}^{\,i}\end{array}}{\Omega \vdash_{\mathsf{ctr}}\; T\,\overline{\tau_i}^{\,i} : \mathsf{R}} \quad \textsc{Ctr\_TyConAppRep}$$

$$\dfrac{\overline{\Omega \vdash_{\mathsf{ctr}} \tau_i : \mathsf{N}}^{\,i}}{\Omega \vdash_{\mathsf{ctr}}\; T\,\overline{\tau_i}^{\,i} : \mathsf{N}} \quad \textsc{Ctr\_TyConAppNom}$$

$$\dfrac{\begin{array}{c}\Omega \vdash_{\mathsf{ctr}} \tau_1 : \rho \\ \Omega \vdash_{\mathsf{ctr}} \tau_2 : \mathsf{N}\end{array}}{\Omega \vdash_{\mathsf{ctr}} \tau_1\,\tau_2 : \rho} \quad \textsc{Ctr\_AppTy}$$

$$\dfrac{\begin{array}{c}\Omega \vdash_{\mathsf{ctr}} \tau_1 : \rho \\ \Omega \vdash_{\mathsf{ctr}} \tau_2 : \rho\end{array}}{\Omega \vdash_{\mathsf{ctr}} \tau_1 \to \tau_2 : \rho} \quad \textsc{Ctr\_FunTy}$$

$$\dfrac{\Omega, n : \mathsf{N} \vdash_{\mathsf{ctr}} \tau : \rho}{\Omega \vdash_{\mathsf{ctr}} \forall n.\tau : \rho} \quad \textsc{Ctr\_ForAllTy}$$

$$\dfrac{}{\Omega \vdash_{\mathsf{ctr}} \mathsf{lit} : \rho} \quad \textsc{Ctr\_LitTy}$$

These judgments depend on a sub-role relation:

$\boxed{\rho_1 \leq \rho_2}$ Sub-role relation, *types/Coercion.lhs*:`ltRole`

$$\frac{}{\mathsf{N} \leq \rho} \quad \textsc{Rlt\_Nominal}$$

$$\frac{}{\rho \leq \mathsf{P}} \quad \textsc{Rlt\_Phantom}$$

$$\frac{}{\rho \leq \rho} \quad \textsc{Rlt\_Refl}$$

## 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 *axBranch*. However, it is cleaner in this presentation to simply check for compatibility here.

$\boxed{\mathsf{no\_conflict}(C, \overline{\sigma_j}^{\,j}, ind_1, ind_2)}$ Branched axiom conflict checking, *types/OptCoercion.lhs*:`checkAxInstCo` and *types/FamInstEnv.lhs*:`compatibleBranches`

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

$$\frac{\begin{array}{l} C = T_{\rho}\ \overline{axBranch_k}^{\,k} \\ \forall \overline{n_{i\ \rho_i}}^{\,i}.(\overline{\tau_j}^{\,j} \rightsquigarrow \tau') = (\overline{axBranch_k}^{\,k})[ind_2] \\ \mathsf{apart}\,(\overline{\sigma_j}^{\,j}, \overline{\tau_j}^{\,j}) \\ \mathsf{no\_conflict}(C, \overline{\sigma_j}^{\,j}, ind_1, ind_2 - 1) \end{array}}{\mathsf{no\_conflict}(C, \overline{\sigma_j}^{\,j}, ind_1, ind_2)} \quad \textsc{NoConflict\_Incompat}$$

$$\frac{\begin{array}{l} C = T_{\rho}\ \overline{axBranch_k}^{\,k} \\ \forall \overline{n_{i\ \rho_i}}^{\,i}.(\overline{\tau_j}^{\,j} \rightsquigarrow \sigma) = (\overline{axBranch_k}^{\,k})[ind_1] \\ \forall \overline{n'_{i\ \rho'_i}}^{\,i}.(\overline{\tau'_j}^{\,j} \rightsquigarrow \sigma') = (\overline{axBranch_k}^{\,k})[ind_2] \\ \mathsf{apart}\,(\overline{\tau_j}^{\,j}, \overline{\tau'_j}^{\,j}) \\ \mathsf{no\_conflict}(C, \overline{\sigma_j}^{\,j}, ind_1, ind_2 - 1) \end{array}}{\mathsf{no\_conflict}(C, \overline{\sigma_j}^{\,j}, ind_1, ind_2)} \quad \textsc{NoConflict\_CompatApart}$$

$$\frac{\begin{array}{l} C = T_{\rho}\ \overline{axBranch_k}^{\,k} \\ \forall \overline{n_{i\ \rho_i}}^{\,i}.(\overline{\tau_j}^{\,j} \rightsquigarrow \sigma) = (\overline{axBranch_k}^{\,k})[ind_1] \\ \forall \overline{n'_{i\ \rho'_i}}^{\,i}.(\overline{\tau'_j}^{\,j} \rightsquigarrow \sigma') = (\overline{axBranch_k}^{\,k})[ind_2] \\ \mathsf{unify}\,(\overline{\tau_j}^{\,j}, \overline{\tau'_j}^{\,j}) = subst \\ subst(\sigma) = subst(\sigma') \end{array}}{\mathsf{no\_conflict}(C, \overline{\sigma_j}^{\,j}, ind_1, ind_2)} \quad \textsc{NoConflict\_CompatCoincident}$$

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.

The algorithm `unify` is implemented in *types/Unify.lhs*:`tcUnifyTys`. It performs a standard unification, returning a substitution upon success.