2015

System FC, as implemented in GHC

Richard A. Eisenberg

Bryn Mawr College, rae@cs.brynmawr.edu

Custom Citation

R. A. Eisenberg. System FC, as implemented in GHC. University of Pennsylvania Technical Report MS-CIS-15-09, 2015.

System FC, as implemented in GHC[1]

23 October, 2015

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
α, β	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:

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
  | α 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.

| x	Term-level name	

n, m, α, x	::=	Variable names, *basicTypes/Var.lhs*:`Var`
	| z^{τ}	Name, labeled with type/kind

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

2.4 Expressions

The datatype that represents expressions:

e, u	::=		Expressions, *coreSyn/CoreSyn.lhs*:`Expr`
	|	n	`Var`: Variable
	|	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
	|	τ	`Type`: Type
	|	γ	`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.
- The τ form of an expression must not then turn out to be a coercion. In other words, the payload inside of a `Type` constructor must not turn out to be built with `CoercionTy`.

Bindings for **let** statements:

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:

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
	|	lit	`LitAlt`: Literal (such as an integer or character)

$\mid$	$\_$	`DEFAULT`: 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`
	$\mid$	$\overline{binding_i}^{\,i}$	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
	$\mid$	$\tau_1 \to \tau_2$	`ForAllTy (Anon ...)  ...`: Function
	$\mid$	$\forall n.\tau$	`ForAllTy (Named ...)  ...`: Type and coercion polymorphism
	$\mid$	lit	`LitTy`: Type-level literal
	$\mid$	$\tau \rhd \gamma$	`CastTy`: Kind cast
	$\mid$	γ	`CoercionTy`: Coercion used in type

`ForAllTy`s are represented in two different ways, depending on whether the `ForAllTy` is anonymous (written $\tau_1 \to \tau_2$) 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 τ_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 `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 $T\,\overline{\tau_i}^{\,i}$ 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.

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

2.6 Coercions

$\gamma,\ \eta$	::=		Coercions, *types/TyCoRep.lhs*:`Coercion`
	$\mid$	$\langle\tau\rangle_\rho$	`Refl`: Reflexivity

$\mid \quad T_\rho\,\overline{\gamma_i}^{\,i}$	`TyConAppCo`: Type constructor application
$\mid \quad \gamma_1\,\gamma_2$	`AppCo`: Application
$\mid \quad \forall z{:}\eta.\gamma$	`ForAllCo`: Polymorphism
$\mid \quad n$	`CoVarCo`: Variable
$\mid \quad C\ ind\,\overline{\gamma_i}^{\,i}$	`AxiomInstCo`: Axiom application
$\mid \quad {}_{prov}\langle\tau_1,\tau_2\rangle_\rho^\eta$	`UnivCo`: Universal coercion
$\mid \quad \mathsf{sym}\,\gamma$	`SymCo`: Symmetry
$\mid \quad \gamma_1 \fatsemi \gamma_2$	`TransCo`: Transitivity
$\mid \quad \mu\,\overline{\tau_i}^{\,i}\ \overline{\gamma_j}^{\,j}$	`AxiomRuleCo`: Axiom-rule application (for type-nats)
$\mid \quad \mathsf{nth}^i\,\gamma$	`NthCo`: Projection (0-indexed)
$\mid \quad LorR\,\gamma$	`LRCo`: Left/right projection
$\mid \quad \gamma@\eta$	`InstCo`: Instantiation
$\mid \quad \gamma \rhd \eta$	`CoherenceCo`: Coherence
$\mid \quad \mathsf{kind}\,\gamma$	`KindCo`: Kind extraction
$\mid \quad \mathsf{sub}\,\gamma$	`SubCo`: Sub-role — convert nominal to representational

Invariants on coercions:

- $\langle\tau_1\,\tau_2\rangle_\rho$ is used; never $\langle\tau_1\rangle_\rho\,\langle\tau_2\rangle_{\mathsf{N}}$.
- 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.
- Every non-reflexive coercion coerces between two distinct types.
- The name in a coercion must be a term-level name (`Id`).
- The contents of $\langle\tau\rangle_\rho$ must not be a coercion. In other words, the payload in a `Refl` must not be built with `CoercionTy`.

The `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:

prov	::=		`UnivCo` provenance, *types/TyCoRep.lhs*:`UnivCoProvenance`
	|	unsafe	From `unsafeCoerce#`
	|	phant	From the need for a phantom coercion
	|	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` and `http://research.microsoft.com/en-us/um/people/simonpj/papers/ext-f/coercible.pdf` for more background.

ρ	::=		Roles, *types/CoAxiom.lhs*:`Role`
	|	N	Nominal
	|	R	Representational
	|	P	Phantom

Is it a left projection or a right projection?

LorR	::=		left or right deconstructor, *types/TyCoRep.lhs*:`LeftOrRight`
	|	left	`CLeft`: Left projection
	|	right	`CRight`: Right projection

Axioms:

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

The left-hand sides $\overline{\tau_j}^{\,j}$ 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 `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 no_conflict. See Section 4.16.

Axiom rules, produced by the type-nats solver:

μ	::=		CoAxiomRules, *types/CoAxiom.lhs*:`CoAxiomRule`
	|	$M_{(i,\overline{\rho_j}^{\,j},\rho')}$	Named rule, with parameter info

An axiom rule $\mu = M_{(i,\overline{\rho_j}^{\,j},\rho')}$ is an axiom name M, with a type arity i, a list of roles $\overline{\rho_j}^{\,j}$ for its coercion parameters, and an output role ρ'. The definition within GHC also includes a field named 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 *typecheck/TcTypeNats.lhs*:`mkBinAxiom` and *typecheck/TcTypeNats.lhs*:`mkAxiom1`.

In Co_UnivCo, function 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 *coreSyn/CoreLint.lhs*:`checkTypes`. For futher discussion see `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	::=		Type constructors, *types/TyCon.lhs*:`TyCon`
	|	$(\to)$	`FunTyCon`: Arrow
	|	N^κ	`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 (`intPrimTyCon`)
	|	$(\sim_\#)$	Unboxed equality (`eqPrimTyCon`)

$\mid$	$(\sim_{\mathsf{R}\#})$	Unboxed representational equality (`eqReprPrimTyCon`)
$\mid$	$\star$	Kind of lifted types (`liftedTypeKindTyCon`)
$\mid$	$\#$	Kind of unlifted types (`unliftedTypeKindTyCon`)
$\mid$	OpenKind	Either $*$ or $\#$ (`openTypeKindTyCon`)
$\mid$	Constraint	Constraint (`constraintTyCon`)
$\mid$	TYPE	TYPE (`tYPETyCon`)
$\mid$	Levity	Levity (`LevityTyCon`)

Note that although GHC contains distinct type constructors $\star$ and Constraint, this formalism treats only $\star$. These two type constructors are considered wholly equivalent. In particular the function `eqType` returns `True` when comparing $\star$ and 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 `data Levity = Lifted | Unlifted`. The type of TYPE is Levity $\to$ TYPE $'$Lifted. The idea is that TYPE$'$Lifted classifies lifted types and TYPE$'$Unlifted classifies unlifted types. Indeed $\star$ 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 *coreSyn/CoreLint.lhs* use the `LintM` monad. This monad contains a context with a set of bound variables Γ. The formalism treats Γ as an ordered list, but GHC uses a set as its representation.

Γ	::=		List of bindings, *coreSyn/CoreLint.lhs*:`LintM`
	$\mid$	n	Single binding
	$\mid$	$\overline{\Gamma_i}^{\,i}$	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 Typing 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`

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

Here is the definition of vars_of , taken from *coreSyn/CoreSyn.lhs*:`bindersOf`:

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

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`

$$\frac{\begin{array}{l} x^{\tau} \in \Gamma \\ \neg\,(\exists \tau_1, \tau_2, \kappa_1, \kappa_2 \text{ s.t. } \tau = \tau_1\, {}^{\kappa_1}{\sim}^{\kappa_2}_{\#}\, \tau_2) \end{array}}{\Gamma \vdash_{\mathsf{tm}} x^{\tau} : \tau} \quad \textsc{Tm\_Var}$$

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

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

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

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

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

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

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

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

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

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

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

$$\frac{\begin{array}{l}\Gamma \vdash_{\mathsf{tm}} e : \sigma \\ \sigma = \star \vee \sigma = \# \\ \Gamma \vdash_{\mathsf{ty}} \tau : \mathsf{TYPE}\,\sigma_2 \\ \overline{\Gamma, z^{\sigma}; \sigma \vdash_{\mathsf{alt}} alt_i : \tau}^{\,i} \end{array}}{\Gamma \vdash_{\mathsf{tm}} \mathbf{case}\; e \;\mathbf{as}\; z^{\sigma} \;\mathbf{return}\; \tau \;\mathbf{of}\; \overline{alt_i}^{\,i} : \tau} \quad \textsc{Tm\_Case}$$

$$\frac{\Gamma \vdash_{\mathsf{co}} \gamma : \tau_1 \; {}^{\kappa_1}{\sim}^{\kappa_2}_{\mathsf{N}} \; \tau_2}{\Gamma \vdash_{\mathsf{tm}} \gamma : \tau_1 \; {}^{\kappa_1}{\sim}^{\kappa_2}_{\#} \; \tau_2} \quad \textsc{Tm\_Coercion}$$

$$\frac{\Gamma \vdash_{\mathsf{co}} \gamma : \tau_1 \; {}^{\kappa_1}{\sim}^{\kappa_2}_{\mathsf{R}} \; \tau_2}{\Gamma \vdash_{\mathsf{tm}} \gamma : ({\sim}_{\mathsf{R}\#})\; \kappa_1 \; \kappa_2 \; \tau_1 \; \tau_2} \quad \textsc{Tm\_CoercionRep}$$

- 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 σ'_i in a context containing all the types that come before it in the list of bindings. The Γ'_i are contexts containing the names and kinds of all type variables (and term variables, for that matter) up to the ith 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`

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

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

$$\frac{\Gamma \vdash_{\mathsf{ty}} \tau_1 : \kappa_1 \quad \Gamma \vdash_{\mathsf{ty}} \tau_2 : \kappa_2 \quad \Gamma \vdash_{\rightarrow} \kappa_1 \rightarrow \kappa_2 : \kappa}{\Gamma \vdash_{\mathsf{ty}} \tau_1 \rightarrow \tau_2 : \kappa} \quad \textsc{Ty\_FunTy}$$

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

$$\frac{\Gamma \vdash_{\mathsf{k}} \kappa_1\ \mathsf{ok} \quad \Gamma, z^{\kappa_1} \vdash_{\mathsf{ty}} \tau : \mathsf{TYPE}\,\sigma \quad \neg\,(z \in \mathit{fv}(\sigma))}{\Gamma \vdash_{\mathsf{ty}} \forall z^{\kappa_1}.\tau : \mathsf{TYPE}\,\sigma} \quad \textsc{Ty\_ForAllTy}$$

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

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

$$\frac{\Gamma \vdash_{\mathsf{co}} \gamma : \tau_1 \,{}^{\kappa_1}\!\sim_{\mathsf{N}}^{\kappa_2} \tau_2}{\Gamma \vdash_{\mathsf{ty}} \gamma : \tau_1 \,{}^{\kappa_1}\!\sim_{\#}^{\kappa_2} \tau_2} \quad \textsc{Ty\_CoercionTy\_Nom}$$

$$\frac{\Gamma \vdash_{\mathsf{co}} \gamma : \tau_1 \,{}^{\kappa_1}\!\sim_{\mathsf{R}}^{\kappa_2} \tau_2}{\Gamma \vdash_{\mathsf{ty}} \gamma : (\sim_{\mathsf{R}\#})\, \kappa_1\, \kappa_2\, \tau_1\, \tau_2} \quad \textsc{Ty\_CoercionTy\_Repr}$$

4.5 Kind validity

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

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

$$\frac{\Gamma \vdash_{\mathsf{ty}} \kappa : \#}{\Gamma \vdash_{\mathsf{k}} \kappa\ \mathsf{ok}} \quad \text{K\_HASH}$$

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 $(\sim_{\mathsf{R}\#})$. Note that the difference between these two forms of equality is interpreted in the rules Co_CoVarCoNom and Co_CoVarCoRepr.

$\boxed{\Gamma \vdash_{\mathsf{co}} \gamma : \tau_1\ {}^{\kappa_1}{\sim}_{\rho}^{\kappa_2}\ \tau_2}$ Coercion typing, *coreSyn/CoreLint.lhs*:`lintCoercion`

$$\frac{\Gamma \vdash_{\mathsf{ty}} \tau : \kappa}{\Gamma \vdash_{\mathsf{co}} \langle \tau \rangle_{\rho} : \tau\ {}^{\kappa}{\sim}_{\rho}^{\kappa}\ \tau} \quad \text{CO\_REFL}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathsf{co}} \gamma_1 : \sigma_1\ {}^{\kappa_1}{\sim}_{\rho}^{\kappa_1'}\ \tau_1 \\ \Gamma \vdash_{\mathsf{co}} \gamma_2 : \sigma_2\ {}^{\kappa_2}{\sim}_{\rho}^{\kappa_2'}\ \tau_2 \\ \Gamma \vdash_{\to} \kappa_1 \to \kappa_2 : \kappa \\ \Gamma \vdash_{\to} \kappa_1' \to \kappa_2' : \kappa'\end{array}}{\Gamma \vdash_{\mathsf{co}} (\to)_{\rho}\ \gamma_1\ \gamma_2 : (\sigma_1 \to \sigma_2)\ {}^{\kappa}{\sim}_{\rho}^{\kappa'} (\tau_1 \to \tau_2)} \quad \text{CO\_TYCONAPPCOFUNTY}$$

$$\frac{\begin{array}{l}T \neq (\to) \\ \overline{\rho_i}^{\,i} = \mathsf{take}(\mathsf{length}\,\overline{\gamma_i}^{\,i}, \mathsf{tyConRolesX}\,\rho\ T) \\ \overline{\Gamma \vdash_{\mathsf{co}} \gamma_i : \sigma_i\ {}^{\kappa_i'}{\sim}_{\rho_i}^{\kappa_i}\ \tau_i}^{\,i} \\ \Gamma \vdash_{\mathsf{app}} \overline{(\sigma_i : \kappa_i')}^{\,i} : \mathsf{tyConKind}\ T \rightsquigarrow \kappa' \\ \Gamma \vdash_{\mathsf{app}} \overline{(\tau_i : \kappa_i)}^{\,i} : \mathsf{tyConKind}\ T \rightsquigarrow \kappa\end{array}}{\Gamma \vdash_{\mathsf{co}} T_{\rho}\ \overline{\gamma_i}^{\,i} : T\ \overline{\sigma_i}^{\,i}\ {}^{\kappa'}{\sim}_{\rho}^{\kappa}\ T\ \overline{\tau_i}^{\,i}} \quad \text{CO\_TYCONAPPCO}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathsf{co}} \gamma_1 : \sigma_1\ {}^{\kappa_1}{\sim}_{\rho}^{\kappa_2}\ \sigma_2 \\ \Gamma \vdash_{\mathsf{co}} \gamma_2 : \tau_1\ {}^{\kappa_1'}{\sim}_{\mathsf{N}}^{\kappa_2'}\ \tau_2 \\ \Gamma \vdash_{\mathsf{app}} (\tau_1 : \kappa_1') : \kappa_1 \rightsquigarrow \kappa_3 \\ \Gamma \vdash_{\mathsf{app}} (\tau_2 : \kappa_2') : \kappa_2 \rightsquigarrow \kappa_4\end{array}}{\Gamma \vdash_{\mathsf{co}} \gamma_1\ \gamma_2 : (\sigma_1\ \tau_1)\ {}^{\kappa_3}{\sim}_{\rho}^{\kappa_4} (\sigma_2\ \tau_2)} \quad \text{CO\_APPCO}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathsf{co}} \gamma_1 : \sigma_1\ {}^{\kappa_1}{\sim}_{\mathsf{P}}^{\kappa_2}\ \sigma_2 \\ \Gamma \vdash_{\mathsf{co}} \gamma_2 : \tau_1\ {}^{\kappa_1'}{\sim}_{\mathsf{P}}^{\kappa_2'}\ \tau_2 \\ \Gamma \vdash_{\mathsf{app}} (\tau_1 : \kappa_1') : \kappa_1 \rightsquigarrow \kappa_3 \\ \Gamma \vdash_{\mathsf{app}} (\tau_2 : \kappa_2') : \kappa_2 \rightsquigarrow \kappa_4\end{array}}{\Gamma \vdash_{\mathsf{co}} \gamma_1\ \gamma_2 : (\sigma_1\ \tau_1)\ {}^{\kappa_3}{\sim}_{\mathsf{P}}^{\kappa_4} (\sigma_2\ \tau_2)} \quad \text{CO\_APPCOPHANTOM}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathsf{co}} \eta : \kappa_1 \;{}^{\star}\!\sim^{\star}_{\mathsf{N}} \kappa_2 \\ \Gamma, z^{\kappa_1} \vdash_{\mathsf{co}} \gamma : \tau_1 \;{}^{\kappa_3}\!\sim^{\kappa_4}_{\rho} \tau_2\end{array}}{\Gamma \vdash_{\mathsf{co}} \forall z{:}\eta.\gamma : (\forall z^{\kappa_1}.\tau_1) \;{}^{\kappa_3}\!\sim^{\kappa_4}_{\rho} (\forall z^{\kappa_2}.(\tau_2 \,[z \mapsto z^{\kappa_2} \triangleright \mathsf{sym}\,\eta]))} \quad \text{CO\_FORALLCO}$$

$$\frac{\begin{array}{l}z^{\phi} \in \Gamma \\ \phi = \tau_1 \;{}^{\kappa_1}\!\sim^{\kappa_2}_{\#} \tau_2\end{array}}{\Gamma \vdash_{\mathsf{co}} z^{\phi} : \tau_1 \;{}^{\kappa_1}\!\sim^{\kappa_2}_{\mathsf{N}} \tau_2} \quad \text{CO\_COVARCONOM}$$

$$\frac{\begin{array}{l}z^{\phi} \in \Gamma \\ \phi = \tau_1 \;{}^{\kappa_1}\!\sim^{\kappa_2}_{\mathsf{R}\#} \tau_2\end{array}}{\Gamma \vdash_{\mathsf{co}} z^{\phi} : \tau_1 \;{}^{\kappa_1}\!\sim^{\kappa_2}_{\mathsf{R}} \tau_2} \quad \text{CO\_COVARCOREPR}$$

$$\frac{\begin{array}{l}\Gamma \vdash_{\mathsf{co}} \eta : \kappa_1 \;{}^{\star}\!\sim^{\star}_{\mathsf{N}} \kappa_2 \\ \Gamma \vdash_{\mathsf{ty}} \tau_1 : \kappa_1 \\ \Gamma \vdash_{\mathsf{ty}} \tau_2 : \kappa_2 \\ \rho \leq \mathsf{P} \vee \neg\,(\mathsf{classifiesTypeWithValues}\,\kappa_1) \vee \\ \quad \neg\,(\mathsf{classifiesTypeWithValues}\,\kappa_2) \vee \mathsf{compatibleUnBoxedTys}\,\tau_1\,\tau_2\end{array}}{\Gamma \vdash_{\mathsf{co}} {}_{\mathsf{unsafe}}\langle \tau_1, \tau_2 \rangle^{\eta}_{\rho} : \tau_1 \;{}^{\kappa_1}\!\sim^{\kappa_2}_{\rho} \tau_2} \quad \text{CO\_UNIVCOUNSAFE}$$

$$\frac{\begin{array}{l}\Gamma \vdash_{\mathsf{co}} \eta : \kappa_1 \;{}^{\star}\!\sim^{\star}_{\mathsf{N}} \kappa_2 \\ \Gamma \vdash_{\mathsf{ty}} \tau_1 : \kappa_1 \\ \Gamma \vdash_{\mathsf{ty}} \tau_2 : \kappa_2\end{array}}{\Gamma \vdash_{\mathsf{co}} {}_{\mathsf{phant}}\langle \tau_1, \tau_2 \rangle^{\eta}_{\mathsf{P}} : \tau_1 \;{}^{\kappa_1}\!\sim^{\kappa_2}_{\mathsf{P}} \tau_2} \quad \text{CO\_UNIVCOPHANTOM}$$

$$\frac{\begin{array}{l}\Gamma \vdash_{\mathsf{co}} \eta : \phi_1 \;{}^{\star}\!\sim^{\star}_{\mathsf{N}} \phi_2 \\ \Gamma \vdash_{\mathsf{ty}} \gamma_1 : \phi_1 \\ \Gamma \vdash_{\mathsf{ty}} \gamma_2 : \phi_2\end{array}}{\Gamma \vdash_{\mathsf{co}} {}_{\mathsf{irrel}}\langle \gamma_1, \gamma_2 \rangle^{\eta}_{\rho} : \gamma_1 \;{}^{\phi_1}\!\sim^{\phi_2}_{\rho} \gamma_2} \quad \text{CO\_UNIVCOPROOFIRREL}$$

$$\frac{\Gamma \vdash_{\mathsf{co}} \gamma : \tau_1 \;{}^{\kappa_1}\!\sim^{\kappa_2}_{\rho} \tau_2}{\Gamma \vdash_{\mathsf{co}} \mathsf{sym}\,\gamma : \tau_2 \;{}^{\kappa_2}\!\sim^{\kappa_1}_{\rho} \tau_1} \quad \text{CO\_SYMCO}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathsf{co}} \gamma_1 : \tau_1 \;{}^{\kappa_1}\!\sim^{\kappa_2}_{\rho} \tau_2 \\ \Gamma \vdash_{\mathsf{co}} \gamma_2 : \tau_2 \;{}^{\kappa_2}\!\sim^{\kappa_3}_{\rho} \tau_3\end{array}}{\Gamma \vdash_{\mathsf{co}} \gamma_1 \mathbin{;} \gamma_2 : \tau_1 \;{}^{\kappa_1}\!\sim^{\kappa_3}_{\rho} \tau_3} \quad \text{CO\_TRANSCO}$$

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

$$\frac{\Gamma \vdash_{\mathsf{co}} \gamma : (\forall z_1{}^{\kappa_1}.\tau_1)\ {}^{\kappa_3}{\sim}_{\rho}^{\kappa_4}\,(\forall z_2{}^{\kappa_2}.\tau_2)}{\Gamma \vdash_{\mathsf{co}} \mathsf{nth}^{0}\,\gamma : \kappa_1\ {}^{\star}{\sim}_{\mathsf{N}}^{\star}\,\kappa_2}\ \textsc{Co\_NthCoForAll}$$

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

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

$$\frac{\begin{array}{l}\Gamma \vdash_{\mathsf{co}} \gamma : (\forall z_1{}^{\kappa_1}.\tau_1)\ {}^{\kappa_3}{\sim}_{\rho}^{\kappa_4}\,(\forall z_2{}^{\kappa_2}.\tau_2) \\ \Gamma \vdash_{\mathsf{co}} \eta : \sigma_1\ {}^{\kappa_1}{\sim}_{\mathsf{N}}^{\kappa_2}\,\sigma_2\end{array}}{\Gamma \vdash_{\mathsf{co}} \gamma@\eta : (\tau_1[z_1{}^{\kappa_1} \mapsto \sigma_1])\ {}^{\kappa_3}{\sim}_{\rho}^{\kappa_4}\,(\tau_2[z_2{}^{\kappa_2} \mapsto \sigma_2])}\ \textsc{Co\_InstCo}$$

$$\frac{\begin{array}{l}C = T_{\rho_0}\,\overline{axBranch_k}^{\,k} \\ 0 \le ind < \mathsf{length}\,\overline{axBranch_k}^{\,k} \\ \forall \overline{n_{i\,\rho_i}}^{\,i}.(\overline{\sigma_{1\,j}}^{\,j} \rightsquigarrow \tau_1) = (\,\overline{axBranch_k}^{\,k}\,)[ind] \\ \Gamma \vdash_{\mathsf{axk}} [\overline{n_{i\rho_i}}^{\,i} \mapsto \overline{\gamma_i}^{\,i}] \rightsquigarrow (subst_1, subst_2) \\ \overline{\sigma_{2\,j} = subst_1(\sigma_{1\,j})}^{\,j} \\ \mathsf{no\_conflict}(C, \overline{\sigma_{2\,j}}^{\,j}, ind, ind - 1) \\ \tau_2 = subst_2(\tau_1) \\ \sigma_2 = T\,\overline{\sigma_{2\,j}}^{\,j} \\ \Gamma \vdash_{\mathsf{ty}} \sigma_2 : \kappa \\ \Gamma \vdash_{\mathsf{ty}} \tau_2 : \kappa'\end{array}}{\Gamma \vdash_{\mathsf{co}} C\ ind\ \overline{\gamma_i}^{\,i} : \sigma_2\ {}^{\kappa}{\sim}_{\rho_0}^{\kappa'}\,\tau_2}\ \textsc{Co\_AxiomInstCo}$$

$$\frac{\Gamma \vdash_{\mathsf{co}} \gamma : \tau_1 \; {}^{\kappa_1}{\sim}_{\rho}^{\kappa_2} \; \tau_2 \qquad \Gamma \vdash_{\mathsf{ty}} \tau_1 \triangleright \eta : \kappa_1'}{\Gamma \vdash_{\mathsf{co}} \gamma \triangleright \eta : \tau_1 \triangleright \eta \; {}^{\kappa_1'}{\sim}_{\rho}^{\kappa_2} \; \tau_2} \quad \textsc{Co\_CoherenceCo}$$

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

$$\frac{\Gamma \vdash_{\mathsf{co}} \gamma : \sigma \; {}^{\kappa'}{\sim}_{\mathsf{N}}^{\kappa} \; \tau}{\Gamma \vdash_{\mathsf{co}} \mathsf{sub}\, \gamma : \sigma \; {}^{\kappa'}{\sim}_{\mathsf{R}}^{\kappa} \; \tau} \quad \textsc{Co\_SubCo}$$

$$\frac{\begin{array}{l} \mu = M_{(i, \overline{\rho_j}^{\,j}, \rho')} \\ \overline{\Gamma \vdash_{\mathsf{ty}} \tau_i : \kappa_i}^{\,i} \\ \overline{\Gamma \vdash_{\mathsf{co}} \gamma_j : \sigma_j \; {}^{\kappa_j''}{\sim}_{\rho_j}^{\kappa_j'} \; \sigma_j'}^{\,j} \\ \mathsf{Just}\,(\tau_1', \tau_2') = \mathsf{coaxrProves}\, \mu \, \overline{\tau_i}^{\,i} \, \overline{(\sigma_j, \sigma_j')}^{\,j} \\ \Gamma \vdash_{\mathsf{ty}} \tau_1' : \kappa_0 \\ \Gamma \vdash_{\mathsf{ty}} \tau_2' : \kappa_0' \end{array}}{\Gamma \vdash_{\mathsf{co}} \mu \, \overline{\tau_i}^{\,i} \; \overline{\gamma_j}^{\,j} : \tau_1' \; {}^{\kappa_0}{\sim}_{\rho'}^{\kappa_0'} \; \tau_2'} \quad \textsc{Co\_AxiomRuleCo}$$

See Section 4.15 for more information about tyConRolesX, and see Section 2.6 for more information about coaxrProves.

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`

$$\frac{\Gamma \vdash_{\mathsf{ty}} \tau : \kappa \qquad \kappa = \star \vee \kappa = \#}{\Gamma \vdash_{\mathsf{n}} x^{\tau} \; \mathsf{ok}} \quad \textsc{Name\_Id}$$

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

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

$$\frac{\Gamma \vdash_{\mathsf{ty}} \tau : \kappa \qquad \kappa = \star \vee \kappa = \#}{\Gamma \vdash_{\mathsf{bnd}} x^{\tau} \; \mathsf{ok}} \quad \textsc{Binding\_Id}$$

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

4.8 Substitution consistency

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

$$\frac{\Gamma \vdash_{\mathsf{ty}} \tau : \kappa}{\Gamma \vdash_{\mathsf{subst}} z^{\kappa} \mapsto \tau\ \mathsf{ok}} \quad \textsc{Subst\_Type}$$

4.9 Case alternative consistency

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

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

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

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

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`

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

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

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

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

4.12 Arrow kinding

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

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

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`

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

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

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

4.14 Axiom argument kinding

$\boxed{\Gamma \vdash_{\mathsf{axk}} [\overline{n_{\hat{\rho}_i}}^{\,i} \mapsto \overline{\gamma}] \rightsquigarrow (subst_1, subst_2)}$ Axiom argument kinding, *coreSyn/CoreLint.lhs*:`check_ki`

$$\frac{}{\Gamma \vdash_{\mathsf{axk}} [\cdot \mapsto \cdot] \rightsquigarrow (\cdot, \cdot)} \quad \text{AxiomKind\_Empty}$$

$$\frac{\begin{array}{l}\Gamma \vdash_{\mathsf{axk}} [\overline{n_{\hat{\rho}_i}}^{\,i} \mapsto \overline{\gamma}] \rightsquigarrow (subst_1, subst_2) \\ n = z^{\kappa} \\ \Gamma \vdash_{\mathsf{co}} \gamma_0 : \tau_1 \;{}^{subst_1(\kappa)}\!\sim_{\rho}^{subst_2(\kappa)}\; \tau_2\end{array}}{\Gamma \vdash_{\mathsf{axk}} [\overline{n_{\hat{\rho}_i}}^{\,i}, n_{\rho} \mapsto \overline{\gamma}, \gamma_0] \rightsquigarrow (subst_1\,[n \mapsto \tau_1], subst_2\,[n \mapsto \tau_2])} \quad \text{AxiomKind\_Arg}$$

4.15 Roles

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 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`

$$\frac{\begin{array}{l}\overline{K_i}^{\,i} = \mathsf{tyConDataCons}\ T \\ \overline{\rho_j}^{\,j} = \mathsf{tyConRoles}\ T \\ \overline{\mathsf{validDcRoles}\,\overline{\rho_j}^{\,j}\ K_i}^{\,i}\end{array}}{\mathsf{validRoles}\ T} \quad \text{Cvr\_DataCons}$$

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

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

In the following judgment, the role ρ is an *input*, not an output. The metavariable Ω denotes a *role context*, as shown here:

Ω	$::=$		Mapping from type variables to roles
	$\vert$	$\overline{n_i : \rho_i}^{\,i}$	List of bindings

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

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

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

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

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

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

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

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

$$\frac{\Omega \vdash_{\mathsf{ctr}} \tau : \rho}{\Omega \vdash_{\mathsf{ctr}} \tau \triangleright \gamma : \rho} \quad \textsc{Ctr\_CastTy}$$

$$\frac{}{\Omega \vdash_{\mathsf{ctr}} \gamma : \mathsf{P}} \quad \textsc{Ctr\_CoercionTy}$$

These judgments depend on a sub-role relation:

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

$$\frac{}{\mathsf{N} \leq \rho} \quad \text{RLT\_NOMINAL}$$

$$\frac{}{\rho \leq \mathsf{P}} \quad \text{RLT\_PHANTOM}$$

$$\frac{}{\rho \leq \rho} \quad \text{RLT\_REFL}$$

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`

$$\frac{}{\mathsf{no\_conflict}(C, \overline{\sigma_i}^{\,i}, ind, -1)} \quad \text{NOCONFLICT\_NOBRANCH}$$

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

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

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

The judgment apart checks to see whether two lists of types are surely apart. $\mathsf{apart}(\overline{\tau_i}^{\,i}, \overline{\sigma_i}^{\,i})$, where $\overline{\tau_i}^{\,i}$ is a list of types and $\overline{\sigma_i}^{\,i}$ is a list of type *patterns* (as in a type family equation), first flattens the $\overline{\tau_i}^{\,i}$ using *types/FamInstEnv.lhs*:`flattenTys` and then checks to see if *types/Unify.lhs*:`tcUnifyTysFG` returns `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 unify is implemented in *types/Unify.lhs*:`tcUnifyTys`. 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 `CoreLint.lhs`. Nevertheless, these rules are included in this document to help the reader understand System FC.

5.2 The context Σ

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

$$\Sigma \quad ::= \quad \cdot \mid \Sigma, [n \mapsto e]$$

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

5.3 Operational semantics rules

$\boxed{\Sigma \vdash_{\mathsf{op}} e \longrightarrow e'}$ Single step semantics

$$\frac{\Sigma(n) = e}{\Sigma \vdash_{\mathsf{op}} n \longrightarrow e} \quad \text{S\_Var}$$

$$\frac{\Sigma \vdash_{\mathsf{op}} e_1 \longrightarrow e_1'}{\Sigma \vdash_{\mathsf{op}} e_1\, e_2 \longrightarrow e_1'\, e_2} \quad \text{S\_App}$$

$$\frac{}{\Sigma \vdash_{\mathsf{op}} (\lambda n.e_1)\, e_2 \longrightarrow e_1\, [n \mapsto e_2]} \quad \text{S\_Beta}$$

$$\frac{\begin{array}{l}\gamma_0 = \mathsf{sym}\,(\mathsf{nth}^0\,\gamma)\\ \gamma_1 = \mathsf{nth}^1\,\gamma\\ \neg\,\exists\tau \text{ s.t. } e_2 = \tau\\ \neg\,\exists\gamma \text{ s.t. } e_2 = \gamma\end{array}}{\Sigma \vdash_{\mathsf{op}} ((\lambda n.e_1) \triangleright \gamma)\, e_2 \longrightarrow (\lambda n.e_1 \triangleright \gamma_1)\,(e_2 \triangleright \gamma_0)} \quad \text{S\_Push}$$

$$\frac{}{\Sigma \vdash_{\mathsf{op}} ((\lambda n.e) \triangleright \gamma)\, \tau \longrightarrow (\lambda n.(e \triangleright \gamma\, n))\, \tau} \quad \text{S\_TPush}$$

$$\frac{\begin{array}{l} \gamma_0 = \mathsf{nth}^1\, (\mathsf{nth}^0\, \gamma) \\ \gamma_1 = \mathsf{sym}\, (\mathsf{nth}^2\, (\mathsf{nth}^0\, \gamma)) \\ \gamma_2 = \mathsf{nth}^1\, \gamma \end{array}}{\Sigma \vdash_{\mathsf{op}} ((\lambda n.e) \triangleright \gamma)\, \gamma' \longrightarrow (\lambda n.e \triangleright \gamma_2)\, (\gamma_0 \fatsemi \gamma' \fatsemi \gamma_1)} \quad \text{S\_CPush}$$

$$\frac{}{\Sigma \vdash_{\mathsf{op}} (e \triangleright \gamma_1) \triangleright \gamma_2 \longrightarrow e \triangleright (\gamma_1 \fatsemi \gamma_2)} \quad \text{S\_Trans}$$

$$\frac{\Sigma \vdash_{\mathsf{op}} e \longrightarrow e'}{\Sigma \vdash_{\mathsf{op}} e \triangleright \gamma \longrightarrow e' \triangleright \gamma} \quad \text{S\_Cast}$$

$$\frac{\Sigma \vdash_{\mathsf{op}} e \longrightarrow e'}{\Sigma \vdash_{\mathsf{op}} e_{\{tick\}} \longrightarrow e'_{\{tick\}}} \quad \text{S\_Tick}$$

$$\frac{\Sigma \vdash_{\mathsf{op}} e \longrightarrow e'}{\Sigma \vdash_{\mathsf{op}} \mathbf{case}\, e\, \mathbf{as}\, n\, \mathbf{return}\, \tau\, \mathbf{of}\, \overline{alt_i}^{\,i} \longrightarrow \mathbf{case}\, e'\, \mathbf{as}\, n\, \mathbf{return}\, \tau\, \mathbf{of}\, \overline{alt_i}^{\,i}} \quad \text{S\_Case}$$

$$\frac{\begin{array}{l} alt_j = K\, \overline{\alpha_b{}^{\kappa_b}}^{\,b}\, \overline{x_c{}^{\tau_c}}^{\,c} \to u \\ u' = u\, [n \mapsto e]\, \overline{[\alpha_b{}^{\kappa_b} \mapsto \sigma_b]}^{\,b}\, \overline{[x_c{}^{\tau_c} \mapsto e_c]}^{\,c} \end{array}}{\Sigma \vdash_{\mathsf{op}} \mathbf{case}\, K\, \overline{\tau'_a}^{\,a}\, \overline{\sigma_b}^{\,b}\, \overline{e_c}^{\,c}\, \mathbf{as}\, n\, \mathbf{return}\, \tau\, \mathbf{of}\, \overline{alt_i}^{\,i} \longrightarrow u'} \quad \text{S\_MatchData}$$

$$\frac{alt_j = \mathsf{lit} \to u}{\Sigma \vdash_{\mathsf{op}} \mathbf{case}\, \mathsf{lit}\, \mathbf{as}\, n\, \mathbf{return}\, \tau\, \mathbf{of}\, \overline{alt_i}^{\,i} \longrightarrow u\, [n \mapsto \mathsf{lit}]} \quad \text{S\_MatchLit}$$

$$\frac{\begin{array}{l} alt_j = \_ \to u \\ \text{no other case matches} \end{array}}{\Sigma \vdash_{\mathsf{op}} \mathbf{case}\, e\, \mathbf{as}\, n\, \mathbf{return}\, \tau\, \mathbf{of}\, \overline{alt_i}^{\,i} \longrightarrow u\, [n \mapsto e]} \quad \text{S\_MatchDefault}$$

$$\frac{\begin{array}{l} T\, \overline{\tau_a}^{\,a}\, {}^{\kappa'}\!\sim_{\#}^{\kappa}\, T\, \overline{\tau'_a}^{\,a} = \mathsf{coercionKind}\, \gamma \\ \forall \overline{\alpha_a{}^{\kappa_a}}^{\,a}. \forall \overline{\beta_b{}^{\kappa'_b}}^{\,b}. \overline{\tau_{1\,c}}^{\,c} \to T\, \overline{\alpha_a{}^{\kappa_a}}^{\,a} = \mathsf{dataConRepType}\, K \\ \hline \overline{e'_c = e_c \triangleright (\tau_{1\,c} \overline{[\alpha_a{}^{\kappa_a} \mapsto \mathsf{nth}^a\, \gamma]}^{\,a}\, \overline{[\beta_b{}^{\kappa'_b} \mapsto \langle \sigma_b \rangle_{\mathsf{N}}]}^{\,b})}^{\,c} \end{array}}{\begin{array}{l} \Sigma \vdash_{\mathsf{op}} \mathbf{case}\, (K\, \overline{\tau_a}^{\,a}\, \overline{\sigma_b}^{\,b}\, \overline{e_c}^{\,c}) \triangleright \gamma\, \mathbf{as}\, n\, \mathbf{return}\, \tau_2\, \mathbf{of}\, \overline{alt_i}^{\,i} \longrightarrow \\ \qquad \mathbf{case}\, K\, \overline{\tau'_a}^{\,a}\, \overline{\sigma_b}^{\,b}\, \overline{e'_c}^{\,c}\, \mathbf{as}\, n\, \mathbf{return}\, \tau_2\, \mathbf{of}\, \overline{alt_i}^{\,i} \end{array}} \quad \text{S\_CasePush}$$

$$\frac{}{\Sigma \vdash_{\mathsf{op}} \textbf{let}\ n = e_1\ \textbf{in}\ e_2 \longrightarrow e_2\,[n \mapsto e_1]} \quad \textsc{S\_LetNonRec}$$

$$\frac{\Sigma,\ \overline{[n_i \mapsto e_i]}^{\,i} \vdash_{\mathsf{op}} u \longrightarrow u'}{\Sigma \vdash_{\mathsf{op}} \textbf{let rec}\ \overline{n_i = e_i}^{\,i}\ \textbf{in}\ u \longrightarrow \textbf{let rec}\ \overline{n_i = e_i}^{\,i}\ \textbf{in}\ u'} \quad \textsc{S\_LetRec}$$

$$\frac{}{\Sigma \vdash_{\mathsf{op}} (\textbf{let rec}\ \overline{n_i = e_i}^{\,i}\ \textbf{in}\ u)\ e' \longrightarrow \textbf{let rec}\ \overline{n_i = e_i}^{\,i}\ \textbf{in}\ (u\ e')} \quad \textsc{S\_LetRecApp}$$

$$\frac{}{\Sigma \vdash_{\mathsf{op}} (\textbf{let rec}\ \overline{n_i = e_i}^{\,i}\ \textbf{in}\ u) \triangleright \gamma \longrightarrow \textbf{let rec}\ \overline{n_i = e_i}^{\,i}\ \textbf{in}\ (u \triangleright \gamma)} \quad \textsc{S\_LetRecCast}$$

$$\frac{}{\begin{array}{l}\Sigma \vdash_{\mathsf{op}} \textbf{case}\ (\textbf{let rec}\ \overline{n_i = e_i}^{\,i}\ \textbf{in}\ u)\ \textbf{as}\ n_0\ \textbf{return}\ \tau\ \textbf{of}\ \overline{alt_j}^{\,j} \longrightarrow \\ \qquad \textbf{let rec}\ \overline{n_i = e_i}^{\,i}\ \textbf{in}\ (\textbf{case}\ u\ \textbf{as}\ n_0\ \textbf{return}\ \tau\ \textbf{of}\ \overline{alt_j}^{\,j})\end{array}} \quad \textsc{S\_LetRecCase}$$

$$\frac{}{\Sigma \vdash_{\mathsf{op}} \textbf{let rec}\ \overline{n_i = e_i}^{\,i}\ \textbf{in}\ (\textbf{let rec}\ \overline{n'_j = e'_j}^{\,j}\ \textbf{in}\ u) \longrightarrow \textbf{let rec}\ \overline{n_i = e_i}^{\,i};\ \overline{n'_j = e'_j}^{\,j}\ \textbf{in}\ u} \quad \textsc{S\_LetRecFlat}$$

$$\frac{fv(u) \cap \overline{n_i}^{\,i} = \cdot}{\Sigma \vdash_{\mathsf{op}} \textbf{let rec}\ \overline{n_i = e_i}^{\,i}\ \textbf{in}\ u \longrightarrow u} \quad \textsc{S\_LetRecReturn}$$

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 $\overline{\sigma_b}^{\,b}$—the existentially quantified types—do not change during this step.