System FC, as implemented in GHC[1]

8 July, 2013

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	Term-level variable names
α, β	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:

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:

[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.

z	::=		Term or type name
	|	α	Type-level name
	|	x	Term-level name

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

2.4 Expressions

The datatype that represents expressions:

$e,\ u$	::=		Expressions, *coreSyn/CoreSyn.lhs*:`Expr`
	|	n	Variable
	|	lit	Literal
	|	$e_1\ e_2$	Application
	|	$\lambda n.e$	Abstraction
	|	**let** *binding* **in** e	Variable binding
	|	**case** e **as** n **return** τ **of** $\overline{alt_i}^{\,i}$	Pattern match
	|	$e \triangleright \gamma$	Cast
	|	$e_{\{tick\}}$	Internal note
	|	τ	Type
	|	γ	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`
	|	$n = e$	Non-recursive binding
	|	**rec** $\overline{n_i = e_i}^{\,i}$	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	Data constructor
	|	lit	Literal (such as an integer or character)
	|	$\_$	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`
	|	$\overline{binding_i}^{\,i}$	List of bindings

2.5 Types

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

There are some invariants on types:

- 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 `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

γ	::=		Coercions, *types/Coercion.lhs*:`Coercion`
	|	$\langle\tau\rangle$	Reflexivity
	|	$T\,\overline{\gamma_i}^{\,i}$	Type constructor application
	|	$\gamma_1\,\gamma_2$	Application
	|	$\forall n.\gamma$	Polymorphism
	|	n	Variable
	|	$C\ ind\,\overline{\gamma_j}^{\,j}$	Axiom application
	|	$\tau_1 \mathrel{\not\!\!\rightarrow} \tau_2$	Unsafe coercion
	|	$\mathsf{sym}\,\gamma$	Symmetry

	$\gamma_1 \fatsemi \gamma_2$	Transitivity
|	$\mathsf{nth}_i\,\gamma$	Projection (0-indexed)
|	$\mathit{LorR}\,\gamma$	Left/right projection
|	$\gamma\,\tau$	Type application

Invariants on coercions:

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

Is it a left projection or a right projection?

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

Axioms:

C	::=		Axioms, *types/TyCon.lhs*:`CoAxiom`
	|	$T\, \overline{\mathit{axBranch}_i}^{\,i}$	Axiom

axBranch, b	::=		Axiom branches, *types/TyCon.lhs*:`CoAxBranch`
	|	$\forall \overline{n_i}^{\,i}.(\overline{\tau_j}^{\,j} \rightsquigarrow \sigma)$	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 no_conflict. See Section 4.15.

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^{κ}	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
	|	$\Box$	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 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`

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

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`

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

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

$$\frac{\begin{array}{l}\Gamma \vdash_{\mathsf{tm}} e : \sigma \\ \Gamma \vdash_{\mathsf{co}} \gamma : \sigma \sim^{\kappa}_{\#} \tau\end{array}}{\Gamma \vdash_{\mathsf{tm}} e \rhd \gamma : \tau} \quad \textsc{Tm\_Cast}$$

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

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

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

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

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

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

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

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

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

$$\frac{\Gamma \vdash_{\mathsf{co}} \gamma : \tau_1 \sim^{\kappa}_{\#} \tau_2}{\Gamma \vdash_{\mathsf{tm}} \gamma : \tau_1 \sim^{\kappa}_{\#} \tau_2}\ \textsc{Tm\_Coercion}$$

- 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 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`.
- 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`

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

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

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

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

$$\frac{\Gamma \vdash_{\mathsf{k}} \kappa_1 \ \mathsf{ok} \quad \Gamma, z^{\kappa_1} \vdash_{\mathsf{ty}} \tau : \kappa_2}{\Gamma \vdash_{\mathsf{ty}} \forall z^{\kappa_1}.\tau : \kappa_2} \quad \textsc{Ty\_ForAllTy}$$

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

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

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

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

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

$$\frac{\begin{array}{c}T \neq (\rightarrow) \\ \overline{\Gamma \vdash_{\mathsf{co}} \gamma_i : \sigma_i \sim^{\kappa_i}_{\#} \tau_i}^{\,i} \\ \Gamma \vdash_{\mathsf{app}} \overline{(\sigma_i : \kappa_i)}^{\,i} : \mathsf{tyConKind}\ T \rightsquigarrow \kappa\end{array}}{\Gamma \vdash_{\mathsf{co}} T\, \overline{\gamma_i}^{\,i} : T\, \overline{\sigma_i}^{\,i} \sim^{\kappa}_{\#} T\, \overline{\tau_i}^{\,i}} \quad \textsc{Co\_TyConAppCo}$$

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

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

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

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

$$\frac{\Gamma \vdash_{\mathsf{ty}} \tau_1 : \kappa}{\Gamma \vdash_{\mathsf{co}} \tau_1 \mathrel{!\!\!\twoheadrightarrow} \tau_2 : \tau_1 \sim^{\kappa}_{\#} \tau_2} \quad \textsc{Co\_UnsafeCo}$$

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

$$\frac{\begin{array}{l} \Gamma \vdash_{\mathsf{co}} \gamma_1 : \tau_1 \sim^{\kappa}_{\#} \tau_2 \\ \Gamma \vdash_{\mathsf{co}} \gamma_2 : \tau_2 \sim^{\kappa}_{\#} \tau_3 \end{array}}{\Gamma \vdash_{\mathsf{co}} \gamma_1 \fatsemi \gamma_2 : \tau_1 \sim^{\kappa}_{\#} \tau_3} \quad \textsc{Co\_TransCo}$$

$$\frac{\begin{array}{l} \Gamma \vdash_{\mathsf{co}} \gamma : (T\, \overline{\sigma_j}^{\,j}) \sim^{\kappa}_{\#} (T\, \overline{\tau_j}^{\,j}) \\ \mathsf{length}\, \overline{\sigma_j}^{\,j} = \mathsf{length}\, \overline{\tau_j}^{\,j} \\ i < \mathsf{length}\, \overline{\sigma_j}^{\,j} \\ \Gamma \vdash_{\mathsf{ty}} \sigma_i : \kappa \end{array}}{\Gamma \vdash_{\mathsf{co}} \mathsf{nth}_i\, \gamma : \sigma_i \sim^{\kappa}_{\#} \tau_i} \quad \textsc{Co\_NthCo}$$

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

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

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

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

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.

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`

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

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

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

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

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

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`

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

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

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

4.10 Telescope substitution

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

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

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

4.11 Case alternative binding consistency

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

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

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

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

4.12 Arrow kinding

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

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

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

4.13 Type application kinding

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

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

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

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

4.14 Sub-kinding

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

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

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

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

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

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

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

4.15 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`

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

$$\frac{\begin{array}{l} C = T\ \overline{axBranch_k}^{\,k} \\ \forall \overline{n_i}^{\,i}.(\overline{\tau_j}^{\,j} \rightsquigarrow \tau') = (\overline{axBranch_k}^{\,k})[ind_2] \\ \mathsf{apart}\,(\overline{\sigma_j}^{\,j}, \overline{\tau_j}^{\,j}) \\ \mathsf{no\_conflict}(C, \overline{\sigma_j}^{\,j}, ind_1, ind_2 - 1) \end{array}}{\mathsf{no\_conflict}(C, \overline{\sigma_j}^{\,j}, ind_1, ind_2)}\ \textsc{NoConflict\_Incompat}$$

$$\frac{\begin{array}{l} C = T\ \overline{axBranch_k}^{\,k} \\ \forall \overline{n_i}^{\,i}.(\,\overline{\tau_j}^{\,j} \rightsquigarrow \sigma) = (\,\overline{axBranch_k}^{\,k})[ind_1] \\ \forall \overline{n'_i}^{\,i}.(\,\overline{\tau'_j}^{\,j} \rightsquigarrow \sigma') = (\,\overline{axBranch_k}^{\,k})[ind_2] \\ \mathsf{apart}\,(\,\overline{\tau_j}^{\,j}\,,\,\overline{\tau'_j}^{\,j}\,) \\ \mathsf{no\_conflict}(C,\,\overline{\sigma_j}^{\,j}\,,\,ind_1,\,ind_2-1) \end{array}}{\mathsf{no\_conflict}(C,\,\overline{\sigma_j}^{\,j}\,,\,ind_1,\,ind_2)}\ \textsc{NoConflict\_CompatApart}$$

$$\frac{\begin{array}{l} C = T\ \overline{axBranch_k}^{\,k} \\ \forall \overline{n_i}^{\,i}.(\,\overline{\tau_j}^{\,j} \rightsquigarrow \sigma) = (\,\overline{axBranch_k}^{\,k})[ind_1] \\ \forall \overline{n'_i}^{\,i}.(\,\overline{\tau'_j}^{\,j} \rightsquigarrow \sigma') = (\,\overline{axBranch_k}^{\,k})[ind_2] \\ \mathsf{unify}\,(\,\overline{\tau_j}^{\,j}\,,\,\overline{\tau'_j}^{\,j}\,) = subst \\ subst(\sigma) = subst(\sigma') \end{array}}{\mathsf{no\_conflict}(C,\,\overline{\sigma_j}^{\,j}\,,\,ind_1,\,ind_2)}\ \textsc{NoConflict\_CompatCoincident}$$

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

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

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

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

$$\frac{\begin{array}{c}\gamma_0 = \mathsf{sym}\,(\mathsf{nth}_0\,\gamma) \\ \gamma_1 = \mathsf{nth}_1\,\gamma \\ \neg\,\exists \tau \text{ s.t. } e_2 = \tau \\ \neg\,\exists \gamma \text{ s.t. } e_2 = \gamma\end{array}}{\Sigma \vdash_{\mathsf{op}} ((\lambda n.e_1) \triangleright \gamma)\ e_2 \longrightarrow (\lambda n.e_1 \triangleright \gamma_1)\ (e_2 \triangleright \gamma_0)} \quad \text{S\_Push}$$

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

$$\frac{\begin{array}{c}\gamma_0 = \mathsf{nth}_1\,(\mathsf{nth}_0\,\gamma) \\ \gamma_1 = \mathsf{sym}\,(\mathsf{nth}_2\,(\mathsf{nth}_0\,\gamma)) \\ \gamma_2 = \mathsf{nth}_1\,\gamma\end{array}}{\Sigma \vdash_{\mathsf{op}} ((\lambda n.e) \triangleright \gamma)\ \gamma' \longrightarrow (\lambda n.e \triangleright \gamma_2)\ (\gamma_0 \fatsemi \gamma' \fatsemi \gamma_1)} \quad \text{S\_CPush}$$

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

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

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

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

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

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

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

$$\frac{\begin{array}{l} T\,\overline{\tau_a}^{\,a} \sim^{\kappa}_{\#} T\,\overline{\tau'_a}^{\,a} = \mathsf{coercionKind}\,\gamma \\ \forall \overline{\alpha_a{}^{\kappa_a}}^{\,a}.\forall \overline{\beta_b{}^{\kappa'_b}}^{\,b}.\overline{\tau_1}^{\,c} \to T\,\overline{\alpha_a{}^{\kappa_a}}^{\,a} = \mathsf{dataConRepType}\,K \\ e'_c = e_c \rhd (\tau_1\, \overline{[\alpha_a{}^{\kappa_a} \mapsto \mathsf{nth}_a\,\gamma]}^{\,a}\, \overline{[\beta_b{}^{\kappa'_b} \mapsto \langle \sigma_b \rangle]}^{\,b})^{\,c} \end{array}}{\Sigma \vdash_{\mathsf{op}} \mathbf{case}\,(K\,\overline{\tau_a}^{\,a}\,\overline{\sigma_b}^{\,b}\,\overline{e_c}^{\,c}) \rhd \gamma\, \mathbf{as}\, n\, \mathbf{return}\, \tau_2\, \mathbf{of}\, \overline{alt_i}^{\,i} \longrightarrow \mathbf{case}\, K\,\overline{\tau'_a}^{\,a}\,\overline{\sigma_b}^{\,b}\,\overline{e'_c}^{\,c}\, \mathbf{as}\, n\, \mathbf{return}\, \tau_2\, \mathbf{of}\, \overline{alt_i}^{\,i}} \quad \text{S\_CasePush}$$

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

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

5.4 Notes

- The S_LetRec and S_LetRecReturn 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 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 kind) 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.

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