1 Introduction

XXX write something general about the goals of the system here

The system sketched here is an extension of the version of System FC described in "Practical aspects of evidence-based compilation in System FC" (hereafter referred to as "Practical aspects").

2 Syntax

2.1 Kinds and kind polymorphism

The most interesting thing about the system as compared to System FC is the extended kind system (Figure 1).

$\kappa,\ \eta$	$::=$			Kinds
	$\mid$	$\mathcal{X}$		Kind variables
	$\mid$	$\star$		Star
	$\mid$	$\kappa_1 \Rightarrow \kappa_2$		Arrow kinds
	$\mid$	$\forall \mathcal{X}.\, \kappa$		Kind polymorphism
	$\mid$	$'T\, \kappa_1 \,..\, \kappa_n$		Lifted type application
	$\mid$	$\star^n \Rightarrow \star$	S	Abbreviation for $\underbrace{\star \Rightarrow \cdots \Rightarrow \star}_{n} \Rightarrow \star$
	$\mid$	(κ)	S	

Figure 1: Syntax of kinds

There are several things to note:

- We allow *polymorphic* kinds; for example, $\forall \mathcal{X}.\, \mathcal{X} \Rightarrow \mathcal{X}$ is the kind of an identity type operator which can be applied to another type of any kind.
- The kind system is *first-order*; we do not allow "kind constructors" (so, *e.g.*, arrow kinds are built-in syntax, not the application of an arrow constructor to two kinds). This obviates the need for a system of sorts/flavors/what-have-you to classify kinds.
- We allow the "lifting" of (fully applied) type constructors into kinds, $'T\, \kappa_1 \,..\, \kappa_n$. As we will see later, the kinding rules for this construction restrict which type constructors T can be lifted in this way.

Note that the system explained in "Practcial aspects" has a special kind # used to distinguish boxed from unboxed types; for simplicity we do not include it here but it should not be too hard to incorporate later.

2.2 Types

The type system is relatively straightforward (Figure 2). We have to add a kind instantiation form $\tau\ [\kappa]$ for types with polymorphic kinds. We also add

a case for lifting data constructors K into types $'K$. Otherwise it is essentially the same as the old System FC. Since the type system is higher-order, we can treat $(\rightarrow)$ and $(\sim)$ as special type constructors instead of baking them into the syntax. Figure 2 also lists a number of abbreviations we will make use of in expressing the typing rules.

H	$::=$		Type constants
	$\mid \quad T$		Datatypes
	$\mid \quad (\rightarrow)$		Arrow
	$\mid \quad (\sim)$		Equality
$\sigma,\ \tau,\ \varphi$	$::=$		Types
	$\mid \quad a$		Variables
	$\mid \quad H$		Constants
	$\mid \quad F$		Type functions
	$\mid \quad 'K$		Lifted data constructors
	$\mid \quad \forall\, a{:}\,\kappa.\,\tau$	$\mathsf{bind}\ a\ \mathsf{in}\ \tau$	Polymorphic types
	$\mid \quad \forall \overline{a{:}\,\kappa}.\,\tau$	S	Abbreviation for multiple $\forall$s
	$\mid \quad \tau_1\,\tau_2$		Type application
	$\mid \quad \tau\ [\kappa]$		Kind instantiation
	$\mid \quad \tau_1 \rightarrow \tau_2$	S	Abbreviation for $(\rightarrow)\ \tau_1\ \tau_2$
	$\mid \quad \overline{\sigma} \rightarrow \tau$	S	Abbreviation for $\sigma_1 \rightarrow \cdots \rightarrow \sigma_n \rightarrow \tau$
	$\mid \quad \tau_1 \sim \tau_2$	S	Abbreviation for $(\sim)\ [\kappa]\ \tau_1\ \tau_2$
	$\mid \quad \tau\,\xi_1\,..\,\xi_n$	S	Left-nested application to types and kinds
	$\mid \quad \tau\ \overline{\sigma}$	S	Left-nested application to multiple types
	$\mid \quad (\tau)$	S	

Figure 2: Syntax of types

2.3 Coercions and expressions

The syntax of coercions is shown in Figure 3. The only addition from "Practical aspects" is to add a kind instantiation form, $\gamma\ [\kappa]$, which allows us to build coercions between applications of types to kinds: if $\Gamma \vdash_{\mathsf{co}} \gamma\ :\ \tau_1 \sim \tau_2$ then $\Gamma \vdash_{\mathsf{co}} \gamma\ [\kappa]\ :\ \tau_1\ [\kappa] \sim \tau_2\ [\kappa]$.

The syntax of expressions is exactly the same as in "Practical aspects".

3 Typing rules

3.1 Kinds

Figure 4 shows the rules for determining which kinds are valid, and the kinding rules for types. The Δ context in the kind validity judgement is simply a set of type variables to ensure that valid kinds are well-scoped.

$\gamma,\ \delta$	::=			Coercions
	|	x		Variables
	|	$C\,\gamma_1\,..\,\gamma_n$		Axiom application
	|	$\langle\tau\rangle$		Reflexivity
	|	$\mathbf{sym}\,\gamma$		Symmetry
	|	$\gamma_1 \fatsemi \gamma_2$		Transitivity
	|	$\gamma_1\,\gamma_2$		Congruence
	|	$\forall\, a{:}\,\kappa.\,\gamma$	bind a in γ	Type polymorphism
	|	$\gamma\,\tau$		Type instantiation
	|	$\gamma\,[\kappa]$		Kind instantiation
	|	$\mathbf{nth}^i\ \gamma$		Nth argument projection
	|	(γ)	S	

Figure 3: Syntax of coercions

The interesting rule of the kind validity judgement is KV_Lift, which states that $'T\,\kappa_1\,..\,\kappa_n$ is only a valid kind if T is a fully applied type constructor of kind $\star^n \Rightarrow \star$. It's worth unpacking the consequences and motivations of this rule a bit:

- First, the kind language is not higher-order, so if T took any higher-order arguments there would be no appropriate kinds to which it could be applied. Hence, we cannot lift types of kind (say) $(\star \Rightarrow \star) \Rightarrow \star$.
- Besides higher-order types, there are also types whose kinds themselves involve lifted types, for example, $T : \star \Rightarrow {'}\mathsf{Nat} \Rightarrow \star$. We don't want to allow lifting such types. In this example, the second argument to $'T$ would have to be a kind classified by ${'}\mathsf{Nat}$; the only possibility would be something like a "doubly lifted" natural number (taking a natural number and lifting it all the way to the kind level). For now, at least, we only want to allow lifting a single level.
- Most importantly, we do not allow lifting types with polymorphic kinds. XXX explain why this would be bad

XXX some examples here

As for the kinding judgement, K_Lift shows how to treat a data constructor K as a type, by appropriately lifting its type τ into a kind κ. Figure 5 shows the details of this lifting translation.

L_Var, L_Arr, and L_Abs straightforwardly translate type variables into kind variables, arrow types into arrow kinds, and type polymorphism into kind polymorphism, respectively. The lifting of type constructor applications is restricted to those constructors with an appropriate kind by L_App. We also ensure that $T \neq (\rightarrow)$ for the simple reason that we lift arrow types specially in L_Arr. (However, if we were feeling particularly masochistic we could do away with arrow kinds and simply use a lifted type arrow...)

$\boxed{\Delta \vdash_{\mathsf{k}} \kappa}$ Kind validity

$$\frac{\mathcal{X} \in \Delta}{\Delta \vdash_{\mathsf{k}} \mathcal{X}} \quad \text{KV\_Var}$$

$$\frac{}{\Delta \vdash_{\mathsf{k}} \star} \quad \text{KV\_Star}$$

$$\frac{\Delta \vdash_{\mathsf{k}} \kappa_1 \quad \Delta \vdash_{\mathsf{k}} \kappa_2}{\Delta \vdash_{\mathsf{k}} \kappa_1 \Rightarrow \kappa_2} \quad \text{KV\_Arr}$$

$$\frac{\Delta, \mathcal{X} \vdash_{\mathsf{k}} \kappa}{\Delta \vdash_{\mathsf{k}} \forall \mathcal{X}. \kappa} \quad \text{KV\_Abs}$$

$$\frac{\Delta \vdash_{\mathsf{k}} \kappa_1 \quad .. \quad \Delta \vdash_{\mathsf{k}} \kappa_n \qquad \varnothing \vdash_{\mathsf{ty}} T \ : \ \star^n \Rightarrow \star}{\Delta \vdash_{\mathsf{k}} {}'T \, \kappa_1 \, .. \, \kappa_n} \quad \text{KV\_Lift}$$

$\boxed{\Gamma \vdash_{\mathsf{ty}} \tau \ : \ \kappa}$ Kinding

$$\frac{\vdash \Gamma \quad w{:}\kappa \in \Gamma}{\Gamma \vdash_{\mathsf{ty}} w \ : \ \kappa} \quad \text{K\_Var}$$

$$\frac{\vdash \Gamma \quad K{:}\tau \in \Gamma \quad \varnothing \vdash \tau \rightsquigarrow \kappa}{\Gamma \vdash_{\mathsf{ty}} {}'K \ : \ \kappa} \quad \text{K\_Lift}$$

$$\frac{\varnothing \vdash_{\mathsf{k}} \kappa \quad \Gamma, \, a{:}\kappa \vdash_{\mathsf{ty}} \tau \ : \ \star}{\Gamma \vdash_{\mathsf{ty}} \forall \, a{:}\kappa. \tau \ : \ \star} \quad \text{K\_Abs}$$

$$\frac{\Gamma \vdash_{\mathsf{ty}} \tau_1 \ : \ \kappa_1 \Rightarrow \kappa_2 \quad \Gamma \vdash_{\mathsf{ty}} \tau_2 \ : \ \kappa_1}{\Gamma \vdash_{\mathsf{ty}} \tau_1 \, \tau_2 \ : \ \kappa_2} \quad \text{K\_App}$$

$$\frac{\Gamma \vdash_{\mathsf{ty}} \tau \ : \ \forall \mathcal{X}. \kappa \quad \varnothing \vdash_{\mathsf{k}} \kappa_1}{\Gamma \vdash_{\mathsf{ty}} \tau \ [\kappa_1] \ : \ \kappa[\kappa_1/\mathcal{X}]} \quad \text{K\_KInst}$$

$$\frac{\vdash \Gamma}{\Gamma \vdash_{\mathsf{ty}} (\rightarrow) \ : \ \star \Rightarrow \star \Rightarrow \star} \quad \text{K\_Arr}$$

$$\frac{\vdash \Gamma}{\Gamma \vdash_{\mathsf{ty}} (\sim) \ : \ \forall \mathcal{X}. \mathcal{X} \Rightarrow \mathcal{X} \Rightarrow \star} \quad \text{K\_Eq}$$

Figure 4: Kinding rules

$\boxed{\Theta \vdash \tau \rightsquigarrow \kappa}$ Type lifting

$$\frac{a \mapsto \mathcal{X} \in \Theta}{\Theta \vdash a \rightsquigarrow \mathcal{X}} \text{ L\_Var}$$

$$\frac{\Theta \vdash \tau_1 \rightsquigarrow \kappa_1 \quad \Theta \vdash \tau_2 \rightsquigarrow \kappa_2}{\Theta \vdash \tau_1 \to \tau_2 \rightsquigarrow \kappa_1 \Rightarrow \kappa_2} \text{ L\_Arr}$$

$$\frac{\Theta, a \mapsto \mathcal{X} \vdash \tau \rightsquigarrow \kappa}{\Theta \vdash \forall\, a{:}\, \star\,.\, \tau \rightsquigarrow \forall\, \mathcal{X}.\, \kappa} \text{ L\_Abs}$$

$$\frac{T \neq (\to) \quad \varnothing \vdash_{\mathsf{ty}} T \,:\, \star^n \Rightarrow \star \quad \overline{\Theta \vdash \tau_i \rightsquigarrow \kappa_i}^{\,n}}{\Theta \vdash \tau\, \tau_1 \,..\, \tau_n \rightsquigarrow {}'T\, \kappa_1 \,..\, \kappa_n} \text{ L\_App}$$

Figure 5: Type to kind translation

3.2 Expression and coercion typing rules

The typing rules for expressions and coercions are straightforward and are included in the complete reproduction of the system at the end of this document. Note that we use $\Gamma \vdash \tau_1, \tau_2 \,:\, \kappa$ as an abbreviation for $\Gamma \vdash_{\mathsf{ty}} \tau_1 \,:\, \kappa \wedge \Gamma \vdash_{\mathsf{ty}} \tau_2 \,:\, \kappa$.

4 Metatheory

4.1 Preliminaries

Lemma 4.1 (Liftable types are closed)**.** *If $\varnothing \vdash \tau \rightsquigarrow \kappa$ then τ is closed.*

Proof. To get a strong enough induction hypothesis, we must generalize to the statement: if $\Theta \vdash \tau \rightsquigarrow \kappa$, then $\mathit{ftv}\, \tau \subseteq \mathit{dom}\, \Theta$. The proof is straightforward. □

Lemma 4.2 (Substitution preserves context validity)**.** *If $\vdash \Gamma,\, a{:}\, \eta, \Gamma'$ and $\Gamma \vdash_{\mathsf{ty}} \sigma \,:\, \eta$ then $\vdash \Gamma, \Gamma'[\sigma/a]$.*

Lemma 4.3 (Substitution lemma for kinding)**.** *If $\Gamma,\, a{:}\, \eta, \Gamma' \vdash_{\mathsf{ty}} \tau \,:\, \kappa$ and $\Gamma \vdash_{\mathsf{ty}} \sigma \,:\, \eta$ then $\Gamma, \Gamma'[\sigma/a] \vdash_{\mathsf{ty}} \tau[\sigma/a] \,:\, \kappa$.*

Proof. The previous two lemmas are proved by mutual induction. XXX finish □

Lemma 4.4. *If $\Gamma \vdash H\, \xi_1 \,..\, \xi_n\, \xi, H\, \xi'_1\, \xi'_n\, \xi' \,:\, \kappa$, then*

- *$H\, \xi_1 \,..\, \xi_n$ and $H\, \xi'_1 \,..\, \xi'_n$ have the same kind (under Γ), and*
- *either*
  - *ξ and ξ' are both kinds, or*

– ξ *and* ξ' *are both types with the same kind.*

Remark. If ξ and ξ' are both kinds, it does *not* follow that they must be identical: the kind of $H\,\xi_1\,..\,\xi_n$ could be $\forall\,\mathcal{X}.\,\kappa'$ with $\mathcal{X}$ not appearing free in κ'.

Proof. Simple induction on n. □

4.2 Invariants

Lemma 4.5 (Equality is homogeneous)**.** *If* $\Gamma \vdash_{\mathsf{ty}} \tau_1 \sim \tau_2 : \kappa$ *then there is some* κ' *such that* $\Gamma \vdash \tau_1, \tau_2 : \kappa'$.

Proof. By inspection of the kinding rules for application and $(\sim)$. □

Theorem 4.6 (Coercions are homogeneous)**.** *If* $\Gamma \vdash_{\mathsf{co}} \gamma : \tau_1 \sim \tau_2$*, then* $\Gamma \vdash \tau_1, \tau_2 : \kappa$ *and* $\vdash \Gamma$.

Proof. By induction on a derivation of $\Gamma \vdash_{\mathsf{co}} \gamma : \tau_1 \sim \tau_2$. Establishing $\vdash \Gamma$ is easy, so we concentrate on establishing $\Gamma \vdash \tau_1, \tau_2 : \kappa$.

- C_Var. This case follows from inversion on GWF_Type, homogeneity of equality (Lemma 4.5), and weakening.
- C_Ax. XXX finish
- C_Inst. IH + substitution lemma for kinding (Lemma 4.3).
- C_Nth. Induction on n + Lemma 4.4.

The remaining cases are applications of the induction hypothesis combined with simple local reasoning. □

5 Surface Syntax

The foregoing is good and well as a core language, but of course the average Haskell programmer will never see the core language and wants higher-level constructs to program with. In this section we discuss, with examples, the modifications to Haskell's surface syntax allowing programmers to make use of the new features.

5.1 Automatic Lifting

One of the most basic things a Haskell programmer will want to do is have names of types automatically interpreted as kinds, and names of data constructors automatically interpreted as types, when appropriate. However, we have to be careful since this introduces opportunities for ambiguity.

For example, suppose we are parsing a Haskell program, and see something like the following:

```
f :: Int -> [Bar]
```

What is `Bar`? (Of course, we might also very well ask about `Int` and `[]`, but for now let's stick with `Bar`!) Currently, GHC simply notes that it is in a type context and looks for a type constructor named `Bar`. If it finds one, great; if not, it emits an error complaining that there is no type constructor `Bar` in scope. The presence or absence of a *data* constructor named `Bar`, of course, makes no difference. For example, there might be a declaration like

```
data Bar = Bar Char
```

Thus, `Bar` is both a data and a type constructor, but since data and type constructors inhabit different namespaces, there is no ambiguity.

However, what if we now allow data constructors to be automatically lifted to types? Clearly, the occurrence of `Bar` in the type signature of `f` is now ambiguous. How should GHC decide which `Bar` is meant?

One possible solution is to require the programmer to give explicit annotations marking each place where she wishes to lift a data constructor into a type constructor:

```
f :: Int -> ['Bar]
```

However, putting simple quotes everywhere quickly gets rather annoying, especially in cases where there is no actual ambiguity. Therefore, we propose a "relaxed" version of the syntax where simple quotes can be omitted in most circumstances, which works as follows:

- When encountering a name in context C (either a kind, type or term context), GHC first checks whether the name is bound in the namespace corresponding to the current context (there is no kind namespace, so we necessarily fail this step when in kind context). If so, that name is chosen.
- Otherwise, GHC checks whether the name is bound in the *next* namespace (namespaces are ordered as follows: kind, type and term). If so, that name is chosen.
- Otherwise, GHC emits a not-in-scope error.

Simple quotes allow to skip one namespace.

5.2 Examples

A few examples are in order. First, Figure 6 shows a standard type of length-indexed vectors. The code above the line shows one way to encode length-indexed vectors in current GHC Haskell; the code below the line shows how it could be done using our proposed extensions.

There are a few things to note. First of all, in the "Before" code we are forced to declare two empty types `Zero` and `Succ` to serve as "type-level values"; if we

```
-- Before

data Zero
data Succ n

data Vec :: * -> * -> * where
  Nil  :: Vec a Zero
  Cons :: a -> Vec a n -> Vec a (Succ n)

-- After

data Nat = Zero | Succ Nat

data Vec :: * -> Nat -> * where
  Nil  :: Vec a Zero
  Cons :: a -> Vec a n -> Vec a (Succ n)
```

Figure 6: Length-indexed vectors, before and after

wanted value-level natural numbers as well, we would have to declare them separately. The "After" code, on the other hand, lets us reuse the normal `Nat` data type as an index to the `Vec` type. The `Nat` in the kind of `Vec` is the type `Nat`, automatically lifted to a kind; `Zero` and `Succ` are used as automatically-lifted type indices to `Vec` in the types of `Vec`'s constructors.

Another thing to note is that in the "Before" code, there is nothing preventing us from accidentally writing a type like

```
sum :: Vec n Int -> Int
```

where we intended to indicate that `sum` takes a vector of any length containing `Int`s, and returns their sum. Unfortunately, we got the parameters to `Vec` the wrong way around, and since both parameters to `Vec` have kind `*`, there is nothing inherently wrong with this type itself. However, we will most certainly have trouble implementing a function of this type, and will likely get rather confusing error messages from the body of `sum`. On the other hand, in the "After" code, the second type parameter to `Vec` has a much more descriptive kind, and the above type signature (before we have even written any code for `sum`) will generate a helpful kind mismatch error, informing us that the second parameter to `Vec` was expected to have kind `Nat`, but `Int` has kind `*`.

Here are some examples about naturals:

```
data Nat = Z | S Nat
data T :: Nat -> *
f :: T (S (S (S (S a)))) -> T (S a)
f' :: T ('S ('S (S (S a)))) -> T (S a)
data P :: (Nat -> *) -> Nat -> Nat -> *
```

```
gz :: P T Z n -> T n
gs :: P T (S m) n -> P T m (S n)
```

Here are some examples about other type constructors:

```
data T :: (*,*) -> *
f :: T '(Int,Bool) -> Int
data P :: [*] -> *
g :: P [Int,Bool,Char] -> (Int,Bool)
h :: T ('(,) Char [Int]) -> P '[] -> P (Bool ': a) -> P a
```

5.3 Warning

Implicit lifting can sometimes be confusing. Consider the following examples at the type level.

```
Usual types:
 (Int,Bool) :: *
 []         :: * -> *
 [Int]      :: *
New types:
'(Int,Bool) :: (*,*)  -- which is also '(*,*)
'[Int]      :: [*]  -- which is also '[*]
 [Int,Bool] :: [*]
'[]         :: [*]
```

5.4 Typeable

5.4.1 Poly-kinded Typeable

Sometimes we might want the value representation of a type, for example to implement dynamic types. Currently there are a family of Typeable type classes which provide a typeOf function. With poly-kinded type classes, we could have the following definition which would factor all the current Typeable type classes.

```
-- Current version
class Typeable  a where typeOf  :: a -> TypeRep
class Typeable1 t where typeOf1 :: t a -> TypeRep
class Typeable2 t where typeOf2 :: t a b -> TypeRep

-- New version with poly-kinded type classes
class Typeable (a :: k) where typeOf :: proxy a -> TypeRep

-- What it means in FC
Typeable : forall k. k -> *
typeOf : forall k. forall (a:k). forall (proxy:k -> *).
         Typeable k a -> proxy a -> TypeRep
```

```
-- We can also make Proxy a built-in type
Proxy :: forall k. k -> *
proxy :: forall k. forall (a:k). Proxy a
```

The surface syntax for kind polymorphism is the same as type polymorphism at term level. Free kind variables are bound at top level of the currently defined kind. Kind variables are implicitly instantiated.

5.4.2 Indexed TypeRep with old kinds

In addition to the Typeable type class, we have to define the representation of a type at the value level. This is done in the TypeRep data type. We might want to have it indexed by the type it represents. This could be used to have proxies to do explicit type instantiation defined at function definition site (and not call site). But this will have to be baked-in since it is an open GADT. Or it can be used to have type-safe dynamics.

```
-- Current version
data TypeRep = TypeRep !Key TyCon [TypeRep]
data TyCon = TyCon !Key String
newtype Key = Key Int deriving( Eq )

-- New version (FC notation)
TypeRep : * -> *
TInt : TypeRep Int

Rep : forall k. k -> *
Rep * a = TypeRep a
Rep (k1 -> k2) a = forall (b::k1). Rep k1 b -> Rep k2 (a b)

TInt : Rep * Int
TMaybe : Rep (* -> *) Maybe = forall (a:*). Rep * a -> Rep * (Maybe a)
TMaybe TInt : Rep * (Maybe Int)

-- Dynamic
data Dynamic where
  Pack :: TypeRep a -> a -> Dynamic
```

5.4.3 Indexed TypeRep with new kinds

Finally we could use this mechanism to build singleton types for our promoted data types, so that we can pattern-match on a promoted type.

```
-- Peano style
data N = Z | S N

data Vec a n where
```

```
  Nil :: Vec a 'Z
  Cons :: a -> Vec a n -> Vec a ('S n)

replicate :: %N n -> a -> Vec a n
  -- ^ @%N n@ is a notation for @Rep n@ which is @Rep 'N n@ in FC.
replicate %Z _ = Nil
replicate (%S n) x = Cons x (replicate n x)
```

The meaning of % is defined in singleton.tex. It mainly builds a GADT mimicing the associated data type in a way that the new version data constructors are indexed by their promoted old version. So that we have a family of singleton types, one for each value of the type we are promoting.

```
-- Int style (needs Iavor stuff)
data Vec a n where
  Nil :: Vec a 0
  Cons :: a -> Vec a n -> Vec a (1 + n)

replicate :: %Int n -> a -> Vec a n
replicate n _ | unsingleBool (n %< %0) = error "negative argument"
replicate %0 _ = Nil
replicate n x = Cons x (replicate (n %- %1) x)
  -- here the solver shows that 1 + (n_22 - 1) = n_22
-- or
replicate :: (0 <= n) => %Int n -> a -> Vec a n

-- These functions could be built-in in a generic way
-- unsingle : forall k (a:k). Rep k a -> Unlift k
-- unsingle : forall k (a:k). Rep (Lift t) a -> t
unsingleBool :: %Bool b -> Bool
unsingleBool %True = True
unsingleBool %False = False

-- Does Iavor have the followings?
-- type level built-ins
(<) :: 'Int -> 'Int -> 'Bool
(-) :: 'Int -> 'Int -> 'Int
-- term level built-ins
(%<) :: %Int m -> %Int n -> %Bool (m < n)
(%-) :: %Int m -> %Int n -> %Int (m - n)
```

6 Ye Compleat System

termvar, x, y, z	term variable
tyvar, a, b	type variable
covar, c, C	coercion variable or constant
kvar, $\mathcal{X}$, $\mathcal{Y}$	kind variable
datacon, K	data constructor
tycon, T	type constructor
tyfunvar, F	type function
index, i, j, m, n	indices

$\kappa,\ \eta$	$::=$			Kinds
	$\mid$	$\mathcal{X}$		Kind variables
	$\mid$	$\star$		Star
	$\mid$	$\kappa_1 \Rightarrow \kappa_2$		Arrow kinds
	$\mid$	$\forall\, \mathcal{X}.\, \kappa$		Kind polymorphism
	$\mid$	$'T\, \kappa_1\, ..\, \kappa_n$		Lifted type application
	$\mid$	$\star^n \Rightarrow \star$	S	Abbreviation for $\underbrace{\star \Rightarrow \cdots \Rightarrow \star}_{n} \Rightarrow \star$
	$\mid$	(κ)	S	
H	$::=$			Type constants
	$\mid$	T		Datatypes
	$\mid$	$(\rightarrow)$		Arrow
	$\mid$	$(\sim)$		Equality
$\sigma,\ \tau,\ \varphi$	$::=$			Types
	$\mid$	a		Variables
	$\mid$	H		Constants
	$\mid$	F		Type functions
	$\mid$	$'K$		Lifted data constructors
	$\mid$	$\forall\, a{:}\, \kappa.\, \tau$	bind a in τ	Polymorphic types
	$\mid$	$\forall \overline{a{:}\, \kappa}.\, \tau$	S	Abbreviation for multiple $\forall$s
	$\mid$	$\tau_1\, \tau_2$		Type application
	$\mid$	$\tau\ [\kappa]$		Kind instantiation
	$\mid$	$\tau_1 \rightarrow \tau_2$	S	Abbreviation for $(\rightarrow)\ \tau_1\ \tau_2$
	$\mid$	$\overline{\sigma} \rightarrow \tau$	S	Abbreviation for $\sigma_1 \rightarrow \cdots \rightarrow \sigma_n \rightarrow \tau$
	$\mid$	$\tau_1 \sim \tau_2$	S	Abbreviation for $(\sim)\ [\kappa]\ \tau_1\ \tau_2$
	$\mid$	$\tau\, \xi_1\, ..\, \xi_n$	S	Left-nested application to types and kinds
	$\mid$	$\tau\ \overline{\sigma}$	S	Left-nested application to multiple types
	$\mid$	(τ)	S	
w	$::=$			Type-level things (type variables, constants, families)
	$\mid$	a		
	$\mid$	H		
	$\mid$	F		
v	$::=$			Term-level things (term variables, constructors, axioms)
	$\mid$	x		
	$\mid$	K		

	$\mid$	C	
ξ	$::=$		Things which can be kinds or types
	$\mid$	κ	
	$\mid$	σ	
$\gamma,\ \delta$	$::=$		Coercions
	$\mid$	x	Variables
	$\mid$	$C\,\gamma_1 \mathinner{..} \gamma_n$	Axiom application
	$\mid$	$\langle \tau \rangle$	Reflexivity
	$\mid$	$\mathbf{sym}\,\gamma$	Symmetry
	$\mid$	$\gamma_1 \fatsemi \gamma_2$	Transitivity
	$\mid$	$\gamma_1\,\gamma_2$	Congruence
	$\mid$	$\forall\, a{:}\,\kappa.\,\gamma$	$\mathsf{bind}\ a\ \mathsf{in}\ \gamma$ Type polymorphism
	$\mid$	$\gamma\,\tau$	Type instantiation
	$\mid$	$\gamma\,[\kappa]$	Kind instantiation
	$\mid$	$\mathbf{nth}^i\ \gamma$	Nth argument projection
	$\mid$	(γ)	S
$e,\ u$	$::=$		Expressions
	$\mid$	x	Variables
	$\mid$	$\lambda x{:}\,\tau.\,e$	Abstraction
	$\mid$	$e_1\ e_2$	Application
	$\mid$	$\Lambda a{:}\,\kappa.\,e$	Type abstraction
	$\mid$	$e\,\tau$	Type instantiation
	$\mid$	K	Data constructors
	$\mid$	$\mathbf{case}\ e\ \mathbf{of}\ \overline{p \to u}$	Case analysis
	$\mid$	$\mathbf{let}\ x{:}\,\tau = u\ \mathbf{in}\ e$	Let binding
	$\mid$	$e \triangleright \gamma$	Casting
	$\mid$	$\lfloor \gamma \rfloor$	Embedded coercion
p	$::=$		Patterns
	$\mid$	$K\ \overline{b{:}\,\kappa}\ \overline{x{:}\,\tau}$	b are existential type args; x are expression args
bnd	$::=$		Bindings
	$\mid$	$v{:}\,\tau$	Term variables and constants
	$\mid$	$w{:}\,\kappa$	Type variables and constants
Γ	$::=$		Contexts
	$\mid$	$\varnothing$	Empty context

	bnd	S	Single binding
	$\Gamma,\, bnd$		Binding
	Γ_1, Γ_2	S	Append two contexts
	$\Gamma, \overline{bnd}$	S	List of bindings
	(Γ)	S	

Δ	::=		Kind variable sets
	$\mid$	$\varnothing$	
	$\mid$	$\Delta, \mathcal{X}$	

Θ	::=		Type to kind variable mappings
	$\mid$	$\varnothing$	
	$\mid$	$\Theta, a \mapsto \mathcal{X}$	

$\boxed{\vdash \Gamma}$ Context well-formedness

$$\frac{}{\vdash \varnothing} \quad \text{GWF\_Empty}$$

$$\frac{\vdash \Gamma \quad \varnothing \vdash_{\mathsf{k}} \kappa \quad w \# \Gamma}{\vdash \Gamma,\, w{:}\kappa} \quad \text{GWF\_Kind}$$

$$\frac{\vdash \Gamma \quad \Gamma \vdash_{\mathsf{ty}} \tau \,:\, \kappa \quad v \# \Gamma}{\vdash \Gamma,\, v{:}\tau} \quad \text{GWF\_Type}$$

$\boxed{\Delta \vdash_{\mathsf{k}} \kappa}$ Kind validity

$$\frac{\mathcal{X} \in \Delta}{\Delta \vdash_{\mathsf{k}} \mathcal{X}} \quad \text{KV\_Var}$$

$$\frac{}{\Delta \vdash_{\mathsf{k}} \star} \quad \text{KV\_Star}$$

$$\frac{\Delta \vdash_{\mathsf{k}} \kappa_1 \quad \Delta \vdash_{\mathsf{k}} \kappa_2}{\Delta \vdash_{\mathsf{k}} \kappa_1 \Rightarrow \kappa_2} \quad \text{KV\_Arr}$$

$$\frac{\Delta, \mathcal{X} \vdash_{\mathsf{k}} \kappa}{\Delta \vdash_{\mathsf{k}} \forall \mathcal{X}.\, \kappa} \quad \text{KV\_Abs}$$

$$\frac{\Delta \vdash_{\mathsf{k}} \kappa_1 \quad .. \quad \Delta \vdash_{\mathsf{k}} \kappa_n \qquad \varnothing \vdash_{\mathsf{ty}} T \,:\, \star^n \Rightarrow \star}{\Delta \vdash_{\mathsf{k}} {'T}\, \kappa_1 \,..\, \kappa_n} \quad \text{KV\_Lift}$$

$\boxed{\Gamma \vdash_{\mathsf{ty}} \tau \,:\, \kappa}$ Kinding

$$\frac{\vdash \Gamma \quad w{:}\kappa \in \Gamma}{\Gamma \vdash_{\mathsf{ty}} w \,:\, \kappa} \quad \text{K\_Var}$$

$$\frac{\vdash \Gamma \quad K{:}\tau \in \Gamma \quad \varnothing \vdash \tau \rightsquigarrow \kappa}{\Gamma \vdash_{\mathsf{ty}} {'K} \,:\, \kappa} \quad \text{K\_Lift}$$

$$\frac{\varnothing \vdash_{\mathsf{k}} \kappa \quad \Gamma,\, a{:}\kappa \vdash_{\mathsf{ty}} \tau \,:\, \star}{\Gamma \vdash_{\mathsf{ty}} \forall\, a{:}\kappa.\, \tau \,:\, \star} \quad \text{K\_Abs}$$

$$\frac{\Gamma \vdash_{\mathsf{ty}} \tau_1 \,:\, \kappa_1 \Rightarrow \kappa_2 \quad \Gamma \vdash_{\mathsf{ty}} \tau_2 \,:\, \kappa_1}{\Gamma \vdash_{\mathsf{ty}} \tau_1\, \tau_2 \,:\, \kappa_2} \quad \text{K\_App}$$

$$\frac{\Gamma \vdash_{\mathsf{ty}} \tau \,:\, \forall\, \mathcal{X}.\, \kappa \quad \varnothing \vdash_{\mathsf{k}} \kappa_1}{\Gamma \vdash_{\mathsf{ty}} \tau\ [\kappa_1] \,:\, \kappa[\kappa_1/\mathcal{X}]} \quad \text{K\_KInst}$$

$$\frac{\vdash \Gamma}{\Gamma \vdash_{\mathsf{ty}} (\rightarrow) \,:\, \star \Rightarrow \star \Rightarrow \star} \quad \text{K\_Arr}$$

$$\frac{\vdash \Gamma}{\Gamma \vdash_{\mathsf{ty}} (\sim) \,:\, \forall\, \mathcal{X}.\, \mathcal{X} \Rightarrow \mathcal{X} \Rightarrow \star} \quad \text{K\_Eq}$$

$\boxed{\Theta \vdash \tau \rightsquigarrow \kappa}$ Type lifting

$$\frac{a \mapsto \mathcal{X} \,\in\, \Theta}{\Theta \vdash a \rightsquigarrow \mathcal{X}} \quad \text{L\_Var}$$

$$\frac{\Theta \vdash \tau_1 \rightsquigarrow \kappa_1 \quad \Theta \vdash \tau_2 \rightsquigarrow \kappa_2}{\Theta \vdash \tau_1 \rightarrow \tau_2 \rightsquigarrow \kappa_1 \Rightarrow \kappa_2} \quad \text{L\_Arr}$$

$$\frac{\Theta, a \mapsto \mathcal{X} \vdash \tau \rightsquigarrow \kappa}{\Theta \vdash \forall\, a{:}\,\star\,.\, \tau \rightsquigarrow \forall\, \mathcal{X}.\, \kappa} \quad \text{L\_Abs}$$

$$\frac{\begin{array}{l} T \neq (\rightarrow) \\ \varnothing \vdash_{\mathsf{ty}} T \,:\, \star^n \Rightarrow \star \quad \overline{\Theta \vdash \tau_i \rightsquigarrow \kappa_i}^{\,n} \end{array}}{\Theta \vdash \tau\, \tau_1\, ..\, \tau_n \rightsquigarrow {'T}\, \kappa_1\, ..\, \kappa_n} \quad \text{L\_App}$$

$\boxed{\Gamma \vdash_{\mathsf{tm}} e \,:\, \tau}$ Expression typing

$$\frac{x{:}\tau \,\in\, \Gamma \quad \tau \neq \sigma_1 \sim \sigma_2}{\Gamma \vdash_{\mathsf{tm}} x \,:\, \tau} \quad \text{T\_Var}$$

$$\frac{\Gamma,\, x{:}\sigma \vdash_{\mathsf{tm}} e \,:\, \tau \quad \Gamma \vdash_{\mathsf{ty}} \sigma \,:\, \kappa}{\Gamma \vdash_{\mathsf{tm}} \lambda x{:}\sigma.\, e \,:\, \sigma \rightarrow \tau} \quad \text{T\_Abs}$$

$$\frac{\Gamma \vdash_{\mathsf{tm}} e \,:\, \sigma \rightarrow \tau \quad \Gamma \vdash_{\mathsf{tm}} u \,:\, \sigma}{\Gamma \vdash_{\mathsf{tm}} e\, u \,:\, \tau} \quad \text{T\_App}$$

$$\frac{\Gamma,\, a{:}\kappa \vdash_{\mathsf{tm}} e \,:\, \tau}{\Gamma \vdash_{\mathsf{tm}} \Lambda a{:}\kappa.\, e \,:\, \forall\, a{:}\kappa.\, \tau} \quad \text{T\_TAbs}$$

$$\frac{\Gamma \vdash_{\mathsf{tm}} e \,:\, \forall\, a{:}\kappa.\, \tau \quad \Gamma \vdash_{\mathsf{ty}} \sigma \,:\, \kappa}{\Gamma \vdash_{\mathsf{tm}} e\, \sigma \,:\, \tau[\sigma/a]} \quad \text{T\_TApp}$$

$$\frac{\Gamma \vdash_{\mathsf{ty}} \sigma \,:\, \star \quad \Gamma \vdash_{\mathsf{tm}} u \,:\, \sigma \quad \Gamma,\, x{:}\sigma \vdash_{\mathsf{tm}} e \,:\, \tau}{\Gamma \vdash_{\mathsf{tm}} \mathbf{let}\, x{:}\sigma = u\, \mathbf{in}\, e \,:\, \tau} \quad \text{T\_Let}$$

$$\frac{\Gamma \vdash_{\mathsf{tm}} e \,:\, \tau_1 \quad \Gamma \vdash_{\mathsf{co}} \gamma \,:\, \tau_1 \sim \tau_2}{\Gamma \vdash_{\mathsf{tm}} e \rhd \gamma \,:\, \tau_2} \quad \text{T\_Cast}$$

$$\frac{\Gamma \vdash_{\mathsf{co}} \gamma \ : \ \sigma_1 \sim \sigma_2}{\Gamma \vdash_{\mathsf{tm}} \lfloor \gamma \rfloor \ : \ \sigma_1 \sim \sigma_2} \quad \text{T\_Coercion}$$

$$\frac{K{:}\tau \ \in \ \Gamma_0}{\Gamma \vdash_{\mathsf{tm}} K \ : \ \tau} \quad \text{T\_Con}$$

$$\frac{\begin{array}{l}\Gamma \vdash_{\mathsf{tm}} e \ : \ T \ \overline{\sigma} \\ \overline{K_j{:}\forall \overline{a{:}\,\eta}.\, \forall \overline{b_j{:}\,\kappa}.\, \overline{\varphi_j} \to (T \ \overline{a}) \ \in \ \Gamma_0}^{\,j} \\ \overline{\tau_{j\,i} = \varphi_{j\,i}\,[\overline{\sigma}/\overline{a}]}^{\,j} \\ \overline{\Gamma, \overline{b_j{:}\,\kappa}, \overline{x_j{:}\,\tau_j} \vdash_{\mathsf{tm}} u_j \ : \ \sigma}^{\,j}\end{array}}{\Gamma \vdash_{\mathsf{tm}} \mathbf{case}\ e\ \mathbf{of}\ \overline{K_j \ \overline{b_j{:}\,\kappa} \ \overline{x_j{:}\,\tau_j} \to u_j} \ : \ \sigma} \quad \text{T\_Case}$$

$\boxed{\Gamma \vdash_{\mathsf{co}} \gamma \ : \ \tau}$ Coercion typing

$$\frac{\vdash \Gamma \quad x{:}\,\tau_1 \sim \tau_2 \ \in \ \Gamma}{\Gamma \vdash_{\mathsf{co}} x \ : \ \tau_1 \sim \tau_2} \quad \text{C\_Var}$$

$$\frac{C{:}\forall \overline{a{:}\,\kappa}.\, (\tau_1 \sim \tau_2) \ \in \ \Gamma \qquad \overline{\Gamma \vdash \sigma_i, \varphi_i \ : \ \kappa_i}^{\,n} \qquad \overline{\Gamma \vdash_{\mathsf{co}} \gamma_i \ : \ \sigma_i \sim \varphi_i}^{\,n}}{\Gamma \vdash_{\mathsf{co}} C\,\gamma_1\,..\,\gamma_n \ : \ \tau_1\,[\overline{\sigma}/\overline{a}] \sim \tau_2\,[\overline{\varphi}/\overline{a}]} \quad \text{C\_Ax}$$

$$\frac{\Gamma \vdash_{\mathsf{ty}} \tau \ : \ \kappa}{\Gamma \vdash_{\mathsf{co}} \langle \tau \rangle \ : \ \tau \sim \tau} \quad \text{C\_Refl}$$

$$\frac{\Gamma \vdash_{\mathsf{co}} \gamma \ : \ \tau_1 \sim \tau_2}{\Gamma \vdash_{\mathsf{co}} \mathbf{sym}\,\gamma \ : \ \tau_2 \sim \tau_1} \quad \text{C\_Sym}$$

$$\frac{\Gamma \vdash_{\mathsf{co}} \gamma_1 \ : \ \tau_1 \sim \tau_2 \quad \Gamma \vdash_{\mathsf{co}} \gamma_2 \ : \ \tau_2 \sim \tau_3}{\Gamma \vdash_{\mathsf{co}} \gamma_1 \fatsemi \gamma_2 \ : \ \tau_1 \sim \tau_3} \quad \text{C\_Trans}$$

$$\frac{\begin{array}{c}\Gamma \vdash \sigma_1, \sigma_2 \ : \ \kappa_1 \Rightarrow \kappa_2 \quad \Gamma \vdash \tau_1, \tau_2 \ : \ \kappa_1 \\ \Gamma \vdash_{\mathsf{co}} \gamma_1 \ : \ \sigma_1 \sim \sigma_2 \quad \Gamma \vdash_{\mathsf{co}} \gamma_2 \ : \ \tau_1 \sim \tau_2\end{array}}{\Gamma \vdash_{\mathsf{co}} \gamma_1\,\gamma_2 \ : \ \sigma_1\,\tau_1 \sim \sigma_2\,\tau_2} \quad \text{C\_App}$$

$$\frac{\begin{array}{c}\Gamma,\ a{:}\,\kappa \vdash \tau_1, \tau_2 \ : \ \star \\ \Gamma,\ a{:}\,\kappa \vdash_{\mathsf{co}} \gamma \ : \ \tau_1 \sim \tau_2\end{array}}{\Gamma \vdash_{\mathsf{co}} \forall\, a{:}\,\kappa.\, \gamma \ : \ \forall\, a{:}\,\kappa.\, \tau_1 \sim \forall\, a{:}\,\kappa.\, \tau_2} \quad \text{C\_Abs}$$

$$\frac{\Gamma \vdash_{\mathsf{co}} \gamma \ : \ \forall\, a{:}\,\kappa.\, \tau_1 \sim \forall\, a{:}\,\kappa.\, \tau_2 \quad \Gamma \vdash_{\mathsf{ty}} \sigma \ : \ \kappa}{\Gamma \vdash_{\mathsf{co}} \gamma\,\sigma \ : \ \tau_1[\sigma/a] \sim \tau_2[\sigma/a]} \quad \text{C\_Inst}$$

$$\frac{\begin{array}{c}\Gamma \vdash \tau_1, \tau_2 \ : \ \forall\, \mathcal{X}.\, \kappa' \\ \Gamma \vdash_{\mathsf{co}} \gamma \ : \ \tau_1 \sim \tau_2 \quad \varnothing \vdash_{\mathsf{k}} \kappa\end{array}}{\Gamma \vdash_{\mathsf{co}} \gamma\ [\kappa] \ : \ \tau_1\ [\kappa] \sim \tau_2\ [\kappa]} \quad \text{C\_KInst}$$

$$\frac{\Gamma \vdash_{\mathsf{co}} \gamma \ : \ H\,\xi_1\,..\,\xi_n \sim H\,\xi'_1\,..\,\xi'_n}{\Gamma \vdash_{\mathsf{co}} \mathbf{nth}^j\ \gamma \ : \ \tau_j \sim \tau'_j} \quad \text{C\_Nth}$$

References