Impredicativity in GHC, plan 2015

Alejandro Serrano, Jurriaan Hage, Dimitrios Vytiniotis, and Simon PJ

The *goal* is to build a better story for impredicative and higher-rank polymorphism in GHC. For that aim we introduce a new type of constraint, $\sigma_1 \leq \sigma_2$, which expresses that type σ_2 is an instance of σ_1. This new type of constraint is inspired on ideas from MLF and HML.

Notation: the grammar for types and constraints can be found in Figure 1.

Basic facts about $\leq$:

- The kind of $\leq$ is $* \to * \to \mathit{Constraint}$.
- The evidence for $\sigma_1 \leq \sigma_2$ is a function $\sigma_1 \to \sigma_2$.
- The canonical forms are $\sigma \leq \alpha$, where α is not a Skolem.
- In Haskell code, $\sigma_1 \leq \sigma_2$ is written as `sigma1 <~ sigma2`.

Type variables	α, β, γ		
Type constuctors	T, S		
Type families	F		
Type classes	$\mathtt{C}$		
Monomorphic types	μ	::=	$\alpha \mid a \mid \mu_1 \to \mu_2 \mid \mathsf{T}\ \overline{\mu} \mid \mathsf{F}\ \overline{\mu}$
Types without top-level $\forall$	τ	::=	$\alpha \mid a \mid \sigma_1 \to \sigma_2 \mid \mathsf{T}\ \overline{\sigma} \mid \mathsf{F}\ \overline{\mu}$
Polymorphic types	σ	::=	$\forall \overline{a}.Q \Rightarrow \sigma \mid \tau$
Constraints	Q	::=	$\epsilon \mid Q_1 \wedge Q_2 \mid \sigma_1 \sim \sigma_2 \mid \sigma_1 \leq \sigma_2 \mid \mathtt{C}\ \overline{\mu}$
Implications	I	::=	$\epsilon \mid I_1 \wedge I_2 \mid Q \mid \forall \overline{a}.\,(Q \supset I)$
Canonical constraints	Q^*	::=	$\alpha \sim \sigma \mid \mathsf{F}\ \overline{\mu} \sim \sigma \mid \sigma \leq \alpha \mid \mathtt{C}\ \overline{\mu}$
Family-free types	ξ	::=	$\alpha \mid a \mid \xi_1 \to \xi_2 \mid \mathsf{T}\ \overline{\xi}$
Instantiatiable types	ψ	::=	$a \mid \sigma_1 \to \sigma_2 \mid \mathsf{T}\ \overline{\sigma} \mid \mathsf{F}\ \overline{\mu}$

Figure 1: Grammar

$$
\begin{array}{lrcll}
[\sim\text{REFL}] & \mathit{canon}\,[\sigma \sim \sigma] & = & \epsilon & \\
[\sim\text{ORIENT}] & \mathit{canon}\,[\sigma_1 \sim \sigma_2] & = & \sigma_2 \sim \sigma_1 & \text{where } \sigma_2 \prec \sigma_1 \\
[\sim\text{TDEC}] & \mathit{canon}\,[(\mathsf{T}\ \overline{\sigma_1}) \sim (\mathsf{T}\ \overline{\sigma_2})] & = & \overline{\sigma_1 \sim \sigma_2} & \\
[\sim\text{FAILDEC}] & \mathit{canon}\,[(\mathsf{T}\ \overline{\sigma_1}) \sim (\mathsf{S}\ \overline{\sigma_2})] & = & \bot & \text{where } \mathsf{T} \neq \mathsf{S} \\
[\sim\text{OCC}] & \mathit{canon}\,[tv \sim \xi] & = & \bot & \text{where } tv \in \xi, \xi \neq tv \\
[\sim\forall\text{DEC}] & \mathit{canon}\,[(\forall \overline{a}.Q \Rightarrow \sigma_1) \sim (\forall \overline{a}.Q \Rightarrow \sigma_2)] & = & \forall \overline{a}.\,(Q \supset \sigma_1 \sim \sigma_2) & \\
[\sim\forall\text{FAIL}] & \mathit{canon}\,[(\mathsf{T}\ \overline{\sigma_1}) \sim (\forall \overline{a}.Q_2 \Rightarrow \sigma_2)] & = & \bot &
\end{array}
$$

Figure 2: Canonicalization rules for $\sim$

$$
\begin{array}{lrcl}
[\leq\text{REFL}] & \mathit{canon}\,[\sigma \leq \sigma] & = & \epsilon \\
[\leq\text{LCON}] & \mathit{canon}\,[\sigma_1 \leq \sigma_2] & = & \sigma_1 \sim \sigma_2 \\
 & & & \text{where } \sigma_1 \not\equiv \sigma_2, \sigma_1 \not\equiv \forall \overline{a}.Q \Rightarrow \sigma_1' \\
[\leq\text{L}\forall] & \mathit{canon}\,[(\forall \overline{a}.Q_1 \Rightarrow \sigma_1) \leq \psi_2] & = & [\overline{a \mapsto \alpha}]\sigma_1 \leq \sigma_2 \ \wedge\ [\overline{a \mapsto \alpha}]Q_1 \\
[\leq\text{R}\forall] & \mathit{canon}\,[\sigma_1 \leq (\forall \overline{a}.Q_2 \Rightarrow \sigma_2)] & = & \forall \overline{a}.\,(Q_2 \supset \sigma_1 \leq \sigma_2)
\end{array}
$$

Figure 3: Canonicalization rules for $\leq$

Changes to constraint solving

A subset of the canonicalization rules is given in Figures 2 and 3. The only missing ones are those related to flattening of constraints.

Notes on the [≤LCON] and [≤L∀] rules

- We disallow applying these rules in the case of unification variables in the right-hand side. If we did so, and later that variable was substituted by some other type, we would need to remember the instantiation done by this rule and apply it to the substituted value. Instead, we prefer to defer the instantiation of the constraint until the variable is changed to another type. The same reasoning applies to disallow application of the rule when the right-hand side is a type family.

  Prior, we disallowed application of the rules to any variable in the right-hand side, unification or Skolem. However, when one tries to type check the following code:

  ```
  g :: Monoid b => b
  g = mempty
  ```

  the only constraint being generated is $(\forall\, a.\, \mathit{Monoid}\, a \Rightarrow a) \leq b$. To go further, we need to instantiate. In this case it is safe to do so, since b is never going to be unified.

- We also disallow applying the rules to quantified types. In that case, we want to apply [≤R∀] first, which may in turn lead to an application of the other rules inside the implication.

- Previously, these rules along with generation and propagation worked very hard to ensure that no constraint of the form $\alpha \leq \sigma$, with α a unification variable, was ever produced.

  Nonetheless, there was still the chance that $\alpha \leq \sigma$ is produced from a constraint involving a type family $\mathsf{F}\ \alpha \leq \sigma$ whose family resolves to a variable $\mathsf{F}\ \alpha \sim \alpha$. Since we do not want to loose confluence, we ensure that this resolves to $\alpha \sim \sigma$ regardless of the order of application of rule [≤LCON] and type family rewriting.

  Furthermore, rule [≤LCON] is key in not getting stuck in some programs. Consider the following code, taken from the `GHC.List` module:

```
head :: [a] -> a
head (x:xs) = x
head []     = badHead

badHead :: b
badHead = error "..."
```

  When type checking the second branch, we generate a constraint of the form $\forall b.b \leq a$. When we apply rule [≤L∀], we get a constraint $\beta \leq a$. If we could not apply [∀LCON] here, we would be stuck. In the current system, we resolve the constraint to $\beta \sim a$.

  The rule [≤LCON] applied to type variables in the left-hand side is also key in being able to type check the `g = mempty` example above. Without it, we would rewrite $(\forall\, a.\, \mathit{Monoid}\ a \Rightarrow a) \leq b$ to $\mathit{Monoid}\ \alpha \wedge \alpha \leq b$. But this disallows progress, we want $\alpha \sim b$ instead.

Design choice: rules for →

Additionally, we may have these special rules for →, based on the covariance and contravariance of each of the argument positions:

$$\begin{array}{lrcl} [\leq\!\!\rightarrow\!\text{ALT1}] & \mathit{canon}\,[(\sigma_1 \rightarrow \sigma_2) \leq (\sigma_3 \rightarrow \sigma_4)] & = & \sigma_1 \leq \sigma_3 \wedge \sigma_2 \leq \sigma_4 \\ [\leq\!\!\rightarrow\!\text{ALT2}] & \mathit{canon}\,[(\sigma_1 \rightarrow \sigma_2) \leq (\sigma_3 \rightarrow \sigma_4)] & = & \sigma_1 \sim \sigma_3 \wedge \sigma_2 \leq \sigma_4 \end{array}$$

But it seems that we lose the ability to infer the type for `runST $ e`.

Design choice: rules for type families

There is important design choice regarding type families: whether they are allowed to resolve to σ-types or just to monomorphic types.

At first sight, it seems that there is no reason to restrict type families. We could think of a family such that $\mathsf{F}\ a \sim a \rightarrow a$, which applied to $\forall b.\, b$, gives rise to the constraint $\mathsf{F}, (\forall b.b) \sim (\forall b.\, b) \rightarrow (\forall b.\, b)$. However, this means that we should not apply rule [≤L∀] when

faced with a type family application, since we do not yet know whether it may resolve to a σ-type. We need to wait, as we do with unification variables.

Not applying [$\leq$L$\forall$] rule means that we cannot discharge some constraints that we would like to, though. Take the following code:

```
type family F a

f :: a -> F a
f x = undefined
```

The only constraint generated is $\forall b.\, b \leq \mathsf{F}\ a$. Given that we cannot canonicalize further, the code will be refused by the compiler with an `Undischarged forall b.b <~ F a` error.

Our solution is to apply [$\leq$L$\forall$] also in the presence of type families, but restrict the shape of what a type family can resolve to so that a σ-type is never produced from type family rewriting. GHC already disallows writing declarations such as `type instance G Bool = forall a.a -> a`, where quantified types appear in the right-hand side. The only missing part is restricting arguments to type families to be monomorphic types, something that we do in the grammar via a new syntactic category μ.

Note: there is still an unclear edge. We might have $\sigma \leq \mathsf{F}\ \alpha$ with $\mathsf{F}\ \alpha \sim \alpha$. In this case we might apply [$\leq$L$\forall$] or not depending on whether we resolve type families before or after the canonicalization phase.

Evidence generation for $\leq$

In the constraint solving process we do not only need to find a solution for the constraints, but also generate evidence of the solving. Remember that the evidence for a constraint $\sigma_1 \leq \sigma_2$ is a function $\sigma_1 \to \sigma_2$.

Rule [$\leq$REFL]**.** We need to build $W :: \sigma \to \sigma$. This is easy:

$$W = \lambda x.\ x$$

Rule [$\leq$LCON]**.** We need to build $W_1 :: \sigma_1 \to \sigma_2$. For this, we can use the evidence $W_2 :: \sigma_1 \sim \sigma_2$ generated by later solving steps. In this case the solution is to make:

$$W_1 = \lambda(x :: \sigma_1).\ x \rhd W_2$$

where $\rhd$ is the cast operator which applies a coercion $\sigma_a \sim \sigma_b$ to a value of type σ_a to produce the same value, but typed as σ_b.

Rule [$\leq$L$\forall$]**.** We need to build $W_1 :: (\forall \overline{a}.Q_1 \Rightarrow \sigma_1) \to \psi_2$ given $W_2 :: [\overline{a \mapsto \alpha}]\sigma_1 \to \psi_2$ and $W_3 :: [\overline{a \mapsto \alpha}]Q_1$. The first step is to get $[\overline{a \mapsto \alpha}]\sigma_1$ from $\forall \overline{a}.Q_1 \Rightarrow \sigma_1$, to do that we need to make a type application and afterwards apply the witness for $Q_1 \Rightarrow \sigma_1$:

$$\lambda(x :: \forall \overline{a}.Q_1 \Rightarrow \sigma_1).\ x\ \overline{\alpha}\ W_3 :: (\forall \overline{a}.Q_1 \Rightarrow \sigma_1) \to [\overline{a \mapsto \alpha}]\sigma_1$$

The last step is then applying W_2 to convert it to our desired type:

$$W_1 = \lambda(x :: \forall \overline{a}.Q_1 \Rightarrow \sigma_1).\ W_2\ (x\ \overline{\alpha}\ W_3)$$

Rule $[\leq_{\mathrm{R}}\forall]$**.** This is the most complicated rule for which to generate evidence. As usual, we want to generate evidence $W_1 :: \sigma_1 \to (\forall \overline{a}.Q_2 \Rightarrow \sigma_2)$. In order to do so, we can use the evidence generated by solving $\forall \overline{a}.\ (Q_2 \supset \sigma_1 \leq \sigma_2)$. In GHC, this implication generates two pieces of information: evidence $W_2 :: \sigma_1 \to \sigma_2$, and a series of bindings which might use the data in Q_2 and which make W_2 correct. We shall denote those bindings by $\square$.

In this case, we need to generate something whose type is $\forall \overline{a}.Q_2 \Rightarrow \ldots$. Thus, we need first a series of type abstractions and evidence abstractions. Then, we can just apply W_2 remembering to bring the bindings into scope.

$$W_1 = \lambda(x :: \sigma_1).\ \overline{\Lambda a}.\ \lambda(d :: Q_2).\ \mathsf{let}\ \square\ \mathsf{in}\ W_2\ x$$

Instantiation of arguments of type families and classes

Suppose we want to typecheck the following piece of code:

```
p = [] == []
```

The canonical constraints for this case are:

$$\mathsf{Eq}\ \alpha\ \wedge\ \forall a.[a] \leq \alpha$$

A similar scenario comes along when working with type families:

```
type family    F  a
type instance  F  [a] = Bool

f :: a -> F a

g :: Bool
g = f []
```

In this case these are the constraints to be solved:

$$\mathsf{F}\ \alpha \sim \mathsf{Bool}\ \wedge\ \forall a.[a] \leq \alpha$$

In both cases we are stuck, since we cannot substitute α by the polymorphic type.[1] The only way to go further is to *instantiate* some variables.

We have with two ways to deal with the problem, which are the two extremes of a solution:

[1]The grammar mandates for arguments of type classes and families to be monomorphic types μ.

1. Force all unification variables appearing in type families or type classes to be monomorphic. This monomorphism restriction needs to "infect" other variables. However, this poses its own problems, which we can realize by considering the following type family:

   ```
   type family    F  a    b
   type instance  F  [a]  b = b -> b
   ```

   Using the rule of always instantiating, the result of $\gamma \sim \mathsf{F}\ [Int]\ \beta$, $(\forall a.a \to a) \leq b$ is $\gamma \sim (\delta \to \delta) \to (\delta \to \delta)$. We have lost polymorphism in a way which was not expected. What we hoped is to get $\gamma \sim (\forall a.a \to a) \to (\forall a.a \to a)$.

2. Thus, we need to have a way to instantiate variables appearing in type classes and families, but only as necessary. We do this by temporarily instantiating variables when checking for axiom application, and returning extra constraints which make this instantiation possible if the match is successful.

   For example, in the previous case we want to apply the axiom $\forall e.\texttt{Eq}\ e \Rightarrow \texttt{Eq}\ [e]$, and thus we need to instantiate a. We return as residual constraints $\texttt{Eq}\ \xi \wedge \texttt{Eq}\ \alpha \sim \texttt{Eq}\ [\xi]$, and the solver takes care of the rest, that is, $\forall a.[a] \leq [\xi]$.

Generalization and defaulting

One nasty side-effect of this approach to impredicativity is that the solver may produce non-Haskell 2010 types. For example, when inferring the type of `singleton id`, where $singleton :: \forall a.\ a \to [a]$ and $id :: \forall a.\ a \to a$, the result would be $\forall a.(\forall b.b \to b) \leq a \Rightarrow [a]$. In short, we want to get rid of the $\leq$ constraints once a binding has been found to type check. This process is part of a larger one which in GHC is known as *approximation*.

Another related problem is that at the end of type checking (*not* inference) some $\leq$ constraints are left over. For example, take the following code:

```
unsafeCoerce :: a -> b  -- from libraries

data T a = T1

g :: Bool
g = unsafeCoerce T1
```

The following constraint is generated to relate the type of the constructor `T1` with its use as argument to `unsafeCoerce`: $\forall a.T\ a \leq \alpha$, where α is a new type variable coming from instantiating the type of `unsafeCoerce`. No more constraints are set on α, and thus that constraint remains at the end of the solving process. Our aim, though, is to make the set of residual constraints empty, for the check to be valid. In order to solve this problem, we hook in the *defaulting* mechanism of GHC.

There are two main procedures to move to types without $\leq$ constraints:

- *Convert $\leq$ constraints into type equality.* In the previous case, the type of `singleton id` is $\forall a.\, a \sim (\forall b.\, b \to b) \Rightarrow [a]$, or equivalently, $[\forall b.\, b \to b]$.

$$[\leq\text{GDEQ}] \quad (\forall \overline{a}.Q \Rightarrow \sigma) \leq \beta \quad \rightsquigarrow \quad (\forall \overline{a}.Q \Rightarrow \sigma) \sim \beta$$

  We prefer this option for defaulting, since it retains the most polymorphism.

- *Generate a type with the less possible polymorphism* by instantiation, which moves quantifiers out of the $\leq$ constraints to top-level. In this case, the type given to `singleton id` is $\forall b.\, [b \to b]$.

$$[\leq\text{GDINST}] \quad (\forall \overline{a}.Q \Rightarrow \sigma) \leq \beta \quad \rightsquigarrow \quad [\overline{a \mapsto \alpha}]\sigma \sim \beta \wedge [\overline{a \mapsto \alpha}]Q$$

  We prefer this option for the approximation phase in inference, since it leads to types which are more similar to those already inferred by GHC. Note that this approximation only applies to unannotated top-level bindings: the user can always ask to give $[\forall a.\, a \to a]$ as a type for `singleton id` via an annotation.

Constraint generation

Without propagation

In the $\Gamma \vdash e : \sigma \rightsquigarrow C$ judgement, Γ and e are inputs, whereas σ and C are outputs. The highlighted parts are changes with respect to the constraint generation judgement in the original OUTSIDEIN(X) system.

$$\frac{x :_\lambda \tau \in \Gamma}{\Gamma \vdash x : \tau \rightsquigarrow \epsilon}\ \text{VARCON}_\lambda$$

$$\frac{\alpha \text{ fresh} \qquad x : \sigma \in \Gamma}{\Gamma \vdash x : \alpha \rightsquigarrow \sigma \leq \alpha}\ \text{VARCON}$$

$$\frac{\alpha \text{ fresh} \qquad \Gamma, x :_\lambda \alpha \vdash e : \tau \rightsquigarrow C}{\Gamma \vdash \lambda x.e : \alpha \to \tau \rightsquigarrow C}\ \text{ABS}$$

$$\frac{\Gamma, x : \sigma_1 \vdash e : \tau_2 \rightsquigarrow C}{\Gamma \vdash \lambda(x :: \sigma_1).e : \sigma_1 \to \tau_2 \rightsquigarrow C}\ \text{ABSA}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \rightsquigarrow C_1 \qquad \Gamma \vdash e_2 : \tau_2 \rightsquigarrow C_2 \qquad \alpha \text{ fresh}}{\Gamma \vdash e_1\, e_2 : \alpha \rightsquigarrow C_1 \wedge C_2 \wedge \tau_1 \sim \tau_2 \to \alpha}\ \text{APP}$$

$$\frac{\Gamma \vdash e : \tau_2 \rightsquigarrow C}{\Gamma \vdash (e :: \sigma_1) : \sigma_1 \rightsquigarrow C \wedge \tau_2 \leq \sigma_1}\ \textsc{Annot}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \rightsquigarrow C_1 \qquad \Gamma,\ x :_\lambda \tau_1 \vdash e_2 : \tau_2 \rightsquigarrow C_2}{\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \tau_2 \rightsquigarrow C_1 \wedge C_2}\ \textsc{Let}$$

$$\frac{\Gamma, x : \sigma_1 \vdash e_1 : \sigma_1 \rightsquigarrow C_1 \qquad \Gamma, x : \sigma_1 \vdash e_2 : \tau_2 \rightsquigarrow C_2}{\Gamma \vdash \mathsf{let}\ x :: \sigma_1 = e_1\ \mathsf{in}\ e_2 : \tau_2 \rightsquigarrow C_1 \wedge C_2}\ \textsc{LetA}$$

$$\frac{\begin{array}{c}\Gamma \vdash e : \tau \rightsquigarrow C \\ \text{for each branch } K_i\,\overline{x_i} \to u_i \text{ do} \\ K_i : \forall \overline{a}\,\overline{b_i}.Q_i \Rightarrow \overline{v_i} \to \mathsf{T}\,\overline{a} \in \Gamma \qquad \overline{b_i} \text{ fresh} \\ \Gamma, \overline{x_i : [\overline{a \mapsto \gamma}]v_i} \vdash u_i : \tau_i \rightsquigarrow C_i \\ \overline{\delta_i} = fuv(\tau_i, C_i) - fuv(\Gamma, \overline{\gamma}) \\ C_i' = \begin{cases} C_i \wedge \tau_i \sim \beta & \text{if } \overline{b_i} \text{ and } Q_i \text{ empty} \\ \forall \overline{\delta_i}.\,([\overline{a \mapsto \gamma}])Q_i \supset C_i \wedge \tau_i \sim \beta \end{cases}\end{array}}{\Gamma \vdash \mathsf{case}\ e\ \mathsf{of}\ \{\overline{K_i\,\overline{x_i} \to u_i}\} : \beta \rightsquigarrow C \wedge \mathsf{T}\,\overline{\gamma} \sim \tau \wedge \bigwedge C_i'}\ \textsc{Case}$$

Why $:_\lambda$?

In principle, the only rule that wuld need to change is that of variables in the term level, which is the point in which instantiation may happen:

$$\frac{\alpha \text{ fresh} \qquad x : \sigma \in \Gamma}{\Gamma \vdash x : \alpha \rightsquigarrow \sigma \leq \alpha}\ \textsc{VarCon}$$

Unfortunately, this is not enough. Suppose we have the following piece of code:

```
(\f -> (f 1, f True)) (if ... then id else not)
```

We want to typecheck it, and we give the argument f a type variable α, and each of its appearances the types variables β and γ. The constraints that are generated are:

- $\alpha \leq \beta$ (from the usage in `f 1`)
- $\alpha \leq \gamma$ (from the usage in `f True`)
- $(\forall a.\, a \to a) \leq \alpha$ (from `id`)
- $(Bool \to Bool) \leq \alpha$ (from `not`)

At this point we are stuck, since we have no rule that could be applied. One might think about applying some kind transitivity of $\leq$, but this is just calling trouble, because it is not clear how to do this without losing information.

Our solution is to make this situation impossible by generating $\alpha \sim \beta$ and $\alpha \sim \gamma$ instead of their $\leq$ counterparts. We do this by splitting the [VarCon] rule in such a way that $\sim$ is generated when the variable comes from an unannotated abstraction or unannotated let. The environment is responsible for keeping track of this fact for each binding, by a small tag, which we denote by $:_\lambda$ in the type rules.

With this change, our initial example leads to an error (`f cannot be applied to both Bool and Int`), from which one can recover by adding an extra annotation. This is a better situation, though, that getting stuck in the middle of the solving process.

With propagation

We use propagation to cover two main scenarios:

- Propagating information from signatures to λ-bound variables. For example:

```
f :: (forall a. a -> a) -> (Int, Bool)
f = \x. -> (x 1, x True)
```

- Propagating information from known types of functions to arguments. Without this propagation, given the previous definition of `f`, then `f (\x -> x)` would not typecheck, but `f id` would.

In the $\Gamma \vdash_\Downarrow e : \sigma \rightsquigarrow C$ judgement, Γ, e and σ are inputs, and only C is an output.

$$\frac{x :_\lambda \tau \in \Gamma}{\Gamma \vdash_\Downarrow x : \sigma \rightsquigarrow \tau \sim \sigma} \textsc{VarCon}_\lambda$$

$$\frac{x : \sigma_1 \in \Gamma}{\Gamma \vdash_\Downarrow x : \sigma_2 \rightsquigarrow \sigma_1 \leq \sigma_2} \textsc{VarCon}$$

$$\frac{\Gamma \vdash_\Downarrow e : \tau \rightsquigarrow C}{\Gamma \vdash_\Downarrow e : (\forall \overline{a}.\, Q \Rightarrow \tau) \rightsquigarrow \forall \overline{a}.(Q \supset C)} \textsc{Prop}\forall$$

$$\frac{\alpha,\, \beta \text{ fresh} \qquad \Gamma, x :_\lambda \alpha \vdash_\Downarrow e : \beta \rightsquigarrow C}{\Gamma \vdash_\Downarrow \lambda x.e : \gamma \rightsquigarrow C \wedge \gamma \sim \alpha \to \beta} \textsc{AbsVar}$$

$$\frac{\Gamma, x :_\lambda \alpha \vdash_\Downarrow e : \sigma_2 \rightsquigarrow C}{\Gamma \vdash_\Downarrow \lambda x.e : \alpha \to \sigma_2 \rightsquigarrow C} \textsc{AbsArrowVar}$$

$$\frac{\Gamma, x : \forall a.\sigma_1 \vdash_{\Downarrow} e : \sigma_2 \rightsquigarrow C}{\Gamma \vdash_{\Downarrow} \lambda x.e : (\forall a.\sigma_1) \to \sigma_2 \rightsquigarrow C}\ \textsc{AbsArrow}\forall$$

$$\frac{\sigma_1 \not\equiv \alpha \qquad \sigma_1 \not\equiv \forall a.Q \Rightarrow \sigma \qquad \Gamma, x :_{\{\lambda,\text{-}\}} \sigma_1 \vdash_{\Downarrow} e : \sigma_2 \rightsquigarrow C}{\Gamma \vdash_{\Downarrow} \lambda x.e : \sigma_1 \to \sigma_2 \rightsquigarrow C}\ \textsc{AbsArrowRest}$$

$$\frac{\alpha \text{ fresh} \qquad \Gamma, x : \sigma_1 \vdash_{\Downarrow} e : \alpha \rightsquigarrow C}{\Gamma \vdash_{\Downarrow} \lambda(x :: \sigma_1).e : \gamma \rightsquigarrow C \wedge \gamma \sim \sigma_1 \to \alpha}\ \textsc{AbsAVar}$$

$$\frac{\Gamma, x : \sigma_1 \vdash_{\Downarrow} e : \sigma_3 \rightsquigarrow C}{\Gamma \vdash_{\Downarrow} \lambda(x :: \sigma_1).e : \sigma_2 \to \sigma_3 \rightsquigarrow C \wedge \sigma_2 \sim \sigma_1}\ \textsc{AbsAArrow}$$

$$\frac{\begin{array}{c} f \in \Gamma \qquad f : \forall \overline{a}.\, \sigma_1 \to \cdots \to \sigma_n \to \sigma_r \in \Gamma \\ \theta = [\overline{a \to \alpha}] \qquad \Gamma \vdash_{\Downarrow} e_i : \theta(\sigma_i) \rightsquigarrow C_i \qquad \theta(\sigma_r) \equiv \beta \end{array}}{\Gamma \vdash_{\Downarrow} f\, e_1\, \ldots\, e_n : \sigma \rightsquigarrow \bigwedge_i C_i \wedge \theta(Q) \wedge \theta(\sigma_r) \sim \sigma}\ \textsc{AppFunVar}$$

$$\frac{\begin{array}{c} f \in \Gamma \qquad f : \forall \overline{a}.\, \sigma_1 \to \cdots \to \sigma_n \to \sigma_r \in \Gamma \\ \theta = [\overline{a \to \alpha}] \qquad \Gamma \vdash_{\Downarrow} e_i : \theta(\sigma_i) \rightsquigarrow C_i \qquad \theta(\sigma_r) \not\equiv \beta \end{array}}{\Gamma \vdash_{\Downarrow} f\, e_1\, \ldots\, e_n : \sigma \rightsquigarrow \bigwedge_i C_i \wedge \theta(Q) \wedge \theta(\sigma_r) \leq \sigma}\ \textsc{AppFunNonVar}$$

$$\frac{\Gamma \vdash_{\Downarrow} e_1 : \alpha \to \sigma \rightsquigarrow C_1 \qquad \Gamma \vdash_{\Downarrow} e_2 : \alpha \rightsquigarrow C_2 \qquad \alpha \text{ fresh}}{\Gamma \vdash_{\Downarrow} e_1\, e_2 : \sigma \rightsquigarrow C_1 \wedge C_2}\ \textsc{App}$$

$$\frac{\Gamma \vdash_{\Downarrow} e : \sigma_1 \rightsquigarrow C}{\Gamma \vdash_{\Downarrow} (e :: \sigma_1) : \sigma_2 \rightsquigarrow C \wedge \sigma_1 \leq \sigma_2}\ \textsc{Annot}$$

$$\frac{\alpha \text{ fresh} \qquad \Gamma, x :_{\lambda} \alpha \vdash_{\Downarrow} e_1 : \alpha \rightsquigarrow C_1 \qquad \Gamma, x :_{\lambda} \alpha \vdash_{\Downarrow} e_2 : \sigma \rightsquigarrow C_2}{\Gamma \vdash_{\Downarrow} \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2 : \sigma \rightsquigarrow C_1 \wedge C_2}\ \textsc{Let}$$

$$\frac{\Gamma, x : \sigma_1 \vdash_{\Downarrow} e_1 : \sigma_1 \rightsquigarrow C_1 \qquad \Gamma, x : \sigma_1 \vdash_{\Downarrow} e_2 : \sigma_2 \rightsquigarrow C_2}{\Gamma \vdash_{\Downarrow} \mathsf{let}\ x :: \sigma_1 = e_1\ \mathsf{in}\ e_2 : \sigma_2 \rightsquigarrow C_1 \wedge C_2}\ \textsc{LetA}$$

$$\frac{\begin{array}{l} \overline{\gamma} \text{ fresh} \qquad \Gamma \vdash_{\Downarrow} e : \mathsf{T}\,\overline{\gamma} \rightsquigarrow C \\ \text{for each branch } K_i\,\overline{x_i} \to u_i \text{ do} \\ \quad K_i : \forall \overline{a}\,\overline{b_i}.Q_i \Rightarrow \overline{v_i} \to \mathsf{T}\,\overline{a} \in \Gamma \qquad \overline{b_i} \text{ fresh} \\ \quad \Gamma, \overline{x_i : [\overline{a \mapsto \gamma}]v_i} \vdash_{\Downarrow} u_i : \sigma \rightsquigarrow C_i \\ \quad \overline{\delta_i} = fuv(\sigma, C_i) - fuv(\Gamma, \overline{\gamma}) \\ \quad C_i' = \begin{cases} C_i & \text{if } \overline{b_i} \text{ and } Q_i \text{ empty} \\ \forall \overline{\delta_i}.\, ([\overline{a \mapsto \gamma}])Q_i \supset C_i) \end{cases} \end{array}}{\Gamma \vdash_{\Downarrow} \mathsf{case}\ e\ \mathsf{of}\ \{\overline{K_i\,\overline{x_i} \to u_i}\} : \sigma \rightsquigarrow C \wedge \bigwedge C_i'}\ \textsc{Case}$$

The most surprising rules are [AppFunVar] and [AppFunNonVar], which apply when we have a known expression f whose type can be recovered from the environment followed by some other freely-shaped expressions. For example, the case of `f (
x -> x)` above, where `f` is in the environment. In that case, we compute the type that the first block ought to have, and propagate it to the rest of arguments.

Of course, there is a reason for distinguishing between the cases of the return type being a variable or not, and generating equalities or instantiation constraints. In short, there is a reason for having both [AppFunVar] and [AppFunNonVar]. Consider the following:

```
data S a = S a

f :: [Char] -> S a
f x = S (error x)
```

If we apply [AppFunNonVar] directly, we instantiate the type of $error :: \forall b.[Char] \to b$ to $[Char] \to \beta$. Since we are pushing down a unification variable α because of the previous application of [AppFunNonVar] to $S :: \forall a.\, a \to S\ a$, we obtain a constraint $\beta \leq \alpha$. Since there are no more restrictions to either α or β, we are not stuck in solving.

Examples

`runST e`

$$(\$)^{\alpha \to \beta \to \gamma}\ runST^{\alpha}\ (e\ ::\ \forall s.ST\ s\ Int)^{\beta}$$

$$\begin{array}{rcl} \forall a, b.(a \to b) \to a \to b & \leq & \alpha \to \beta \to \gamma \\ \forall a.\,(\forall s.ST\ s\ a) \to a & \leq & \alpha \\ \forall s.ST\ s\ Int & \leq & \beta \end{array}$$

$$\downarrow$$

$$\begin{array}{rcl} (\forall s.ST\ s\ \epsilon) \to \epsilon & \leq & \beta \to \gamma \\ \forall s.ST\ s\ Int & \leq & \beta \end{array}$$

$$\downarrow$$

$$\begin{array}{rcl} \forall s.ST\ s\ \gamma & \sim & \beta \\ \forall s.ST\ s\ Int & \leq & \beta \end{array}$$

$$\downarrow$$

$$\forall s.ST\ s\ Int \ \leq\ \forall s.ST\ s\ \gamma$$

$$\downarrow$$

$$\forall s.\ (\epsilon \supset \forall s.ST\ s\ Int \leq ST\ s\ \gamma)$$

$$\downarrow$$

$$\forall s.\ (\epsilon \supset ST\ \pi\ Int \sim ST\ s\ \gamma)$$

$$\downarrow$$

$$\forall s.\ (\epsilon \supset \pi \sim s \wedge Int \sim \gamma)$$

$$\downarrow$$

$$\gamma \sim Int$$

η-expansion

$$f :: (\forall a.a \to a) \to Int$$
$$g_1 = f^{\alpha}$$
$$g_2 = \left(\lambda x.f^{\beta}\ x^{\gamma}\right)^{\delta} :: \forall \gamma\, \delta.\ (\tau_f \leq \gamma \to \delta) \Rightarrow \gamma \to \delta$$

$$(\forall a.a \to a) \to Int \leq \gamma \to \delta$$

$$\downarrow$$

$$(\forall a.a \to a) \to Int \sim \gamma \to \delta$$

$$\downarrow$$

$$\begin{array}{rcl} \gamma & \sim & \forall a.a \to a \\ \delta & \sim & Int \end{array}$$

So we lose nothing by η-expanding!