

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 $* \rightarrow * \rightarrow \textit{Constraint}$.
- The evidence for $\sigma_1 \leq \sigma_2$ is a function $\sigma_1 \rightarrow \sigma_2$.
- 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 constructors	\mathbf{T}, \mathbf{S}
Type families	\mathbf{F}
Type classes	\mathbf{C}
Family-free types	$\xi ::= \alpha \mid a \mid \xi_1 \rightarrow \xi_2 \mid \mathbf{T} \bar{\xi}$
Monomorphic types	$\mu ::= \alpha \mid a \mid \mu_1 \rightarrow \mu_2 \mid \mathbf{T} \bar{\mu} \mid \mathbf{F} \bar{\mu}$
Types without top-level \forall	$\tau ::= \alpha \mid a \mid \sigma_1 \rightarrow \sigma_2 \mid \mathbf{T} \bar{\sigma} \mid \mathbf{F} \bar{\mu}$
Polymorphic types	$\sigma ::= \forall \bar{a}. Q \Rightarrow \sigma \mid \tau$
Constraints	$Q ::= \epsilon \mid Q_1 \wedge Q_2 \mid \sigma_1 \sim \sigma_2 \mid \sigma_1 \leq \sigma_2 \mid \mathbf{C} \bar{\mu}$
Implications	$I ::= \epsilon \mid I_1 \wedge I_2 \mid Q \mid \forall \bar{a}. (Q \supset I)$
Canonical constraints	$Q^* ::= \alpha \sim \sigma \mid \mathbf{F} \bar{\mu} \sim \sigma \mid \sigma \leq \alpha \mid \mathbf{C} \bar{\mu}$

Figure 1: Grammar

[\sim REFL]	$canon [\sigma \sim \sigma]$	=	ϵ	
[\sim ORIENT]	$canon [\sigma_1 \sim \sigma_2]$	=	$\sigma_2 \sim \sigma_1$	where $\sigma_2 \prec \sigma_1$
[\sim TDEC]	$canon [(\mathbf{T} \bar{\sigma}_1) \sim (\mathbf{T} \bar{\sigma}_2)]$	=	$\overline{\sigma_1 \sim \sigma_2}$	
[\sim FAILDEC]	$canon [(\mathbf{T} \bar{\sigma}_1) \sim (\mathbf{S} \bar{\sigma}_2)]$	=	\perp	where $\mathbf{T} \neq \mathbf{S}$
[\sim OCC]	$canon [tv \sim \xi]$	=	\perp	where $tv \in \xi, \xi \neq tv$
[\sim \forall DEC]	$canon [(\forall \bar{a}.Q \Rightarrow \sigma_1) \sim (\forall \bar{a}.Q \Rightarrow \sigma_2)]$	=	$\forall \bar{a}. (Q \supset \sigma_1 \sim \sigma_2)$	
[\sim \forall FAIL]	$canon [(\mathbf{T} \bar{\sigma}_1) \sim (\forall \bar{a}.Q_2 \Rightarrow \sigma_2)]$	=	\perp	

Figure 2: Canonicalization rules for \sim

[\leq REFL]	$canon [\sigma \leq \sigma]$	=	ϵ	
[\leq LCON]	$canon [\psi_1 \leq \sigma_2]$	=	$\psi_1 \sim \sigma_2$	
[\leq L \forall VAR]	$canon [(\forall \bar{a}.Q_1 \Rightarrow v) \leq \psi_2]$	=	$[\bar{a} \mapsto \bar{\alpha}]v \sim \psi_2 \wedge [\bar{a} \mapsto \bar{\alpha}]Q_1$	where v is a unification variable and $v \in \bar{a}$
[\leq L \forall]	$canon [(\forall \bar{a}.Q_1 \Rightarrow \sigma_1) \leq \psi_2]$	=	$[\bar{a} \mapsto \bar{\alpha}]\sigma_1 \leq \sigma_2 \wedge [\bar{a} \mapsto \bar{\alpha}]Q_1$	where v is not a unification variable or $v \notin \bar{a}$
[\leq R \forall]	$canon [\sigma_1 \leq (\forall \bar{a}.Q_2 \Rightarrow \sigma_2)]$	=	$\forall \bar{a}. (Q_2 \supset \sigma_1 \leq \sigma_2)$	

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.

We represent by ψ a type with one of the following forms:

- Constructor applied to zero or more types, $\mathbf{T} \bar{\sigma}$
- An application headed by a type variable, $\alpha \bar{\sigma}$ or $a \bar{\sigma}$
- A Skolem variable, a

Notes on the [\leq L \forall VAR] and [\leq L \forall] 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 $(\forall a. \text{Monoid } a \Rightarrow a) \leq b$. 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 $[\leq R\forall]$ first, which may in turn lead to an application of the other rules inside the implication.
- 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 $\forall b. b \leq a$. If we were to apply rule $[\leq L\forall]$, we would get a constraint $\beta \leq a$, getting stuck. Instead, we have the rule $[\leq L\forall\text{VAR}]$, which ensures that the generated constraint is $\beta \sim a$.

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

- Note that these rules along with the generation and propagation ones ensure that no constraint of the form $\alpha \leq \sigma$, with α a unification variable, is even produced.

Design choice: rules for \rightarrow

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

$$\begin{aligned} [\leq \rightarrow \text{ALT1}] \quad \text{canon} [(\sigma_1 \rightarrow \sigma_2) \leq (\sigma_3 \rightarrow \sigma_4)] &= \sigma_1 \leq \sigma_3 \wedge \sigma_2 \leq \sigma_4 \\ [\leq \rightarrow \text{ALT2}] \quad \text{canon} [(\sigma_1 \rightarrow \sigma_2) \leq (\sigma_3 \rightarrow \sigma_4)] &= \sigma_1 \sim \sigma_3 \wedge \sigma_2 \leq \sigma_4 \end{aligned}$$

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

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 $\sigma_1 \rightarrow \sigma_2$.

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

$$W = \lambda x. x$$

Rule [\leq LCON]. We need to build $W_1 :: \psi_1 \rightarrow \sigma_2$. For this, we can use the evidence $W_2 :: \psi_1 \sim \sigma_2$ generated by later solving steps. In this case the solution is to make:

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

where \triangleright 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 $W_1 :: (\forall \bar{a}. Q_1 \Rightarrow \sigma_1) \rightarrow \psi_2$ given $W_2 :: [\bar{a} \mapsto \bar{\alpha}] \sigma_1 \rightarrow \psi_2$ and $W_3 :: [\bar{a} \mapsto \bar{\alpha}] Q_1$. The first step is to get $[\bar{a} \mapsto \bar{\alpha}] \sigma_1$ from $\forall \bar{a}. Q_1 \Rightarrow \sigma_1$, to do that we need to make a type application and afterwards apply the witness for $Q_1 \Rightarrow \sigma_1$:

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

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

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

Rule [\leq L \forall VAR]. This case is similar to the previous one, but now $W_2 :: [\bar{a} \mapsto \bar{\alpha}] v \sim \psi_2$. Instead of using application, we use a cast:

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

Rule [\leq R \forall]. This is the most complicated rule for which to generate evidence. As usual, we want to generate evidence $W_1 :: \sigma_1 \rightarrow (\forall \bar{a}. Q_2 \Rightarrow \sigma_2)$. In order to do so, we can use the evidence generated by solving $\forall \bar{a}. (Q_2 \supset \sigma_1 \leq \sigma_2)$. In GHC, this implication generates two pieces of information: evidence $W_2 :: \sigma_1 \rightarrow \sigma_2$, 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 $\forall \bar{a}. Q_2 \Rightarrow \dots$. 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.

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

Type families and type classes

Suppose we want to typecheck the following piece of code:

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

The canonical constraints for this case are:

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

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:

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

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. 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 $\gamma \sim F [Int] \beta, (\forall a.a \rightarrow a) \leq b$ is $\gamma \sim (\delta \rightarrow \delta) \rightarrow (\delta \rightarrow \delta)$. We have lost polymorphism in a way which was not expected. What we hoped is to get $\gamma \sim (\forall a.a \rightarrow a) \rightarrow (\forall a.a \rightarrow a)$.

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 $\forall e.\text{Eq } e \Rightarrow \text{Eq } [e]$, and thus we need to instantiate a . We return as residual constraints $\text{Eq } \xi \wedge \text{Eq } \alpha \sim \text{Eq } [\xi]$, and the solver takes care of the rest, that is, $\forall a.[a] \leq [\xi]$.

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

Generalization

One nasty side-effect of this approach to impredicativity is that the solver may produce non-Haskell 2010 types. For example, when type checking `singleton id`, where $singleton :: \forall a. a \rightarrow [a]$ and $id :: \forall a. a \rightarrow a$, the result would be $\forall a. (forall b. b \rightarrow b) \leq a \Rightarrow [a]$. 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.

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

- *Convert all \leq constraints into type equality.* In the previous case, the type of `singleton id` is $\forall a. a \sim (\forall b. b \rightarrow b) \Rightarrow [a]$, or equivalently, $[\forall b. b \rightarrow b]$.
- *Generate a type with the less possible polymorphism,* by moving quantifiers out of the \leq constraints to top-level. In this case, the type given to `singleton id` is $\forall b. [b \rightarrow b]$.

We prefer the second option, 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 $[\forall a. a \rightarrow a]$ as a type for `singleton id` via an annotation.

The conversion works by applying repeatedly the following rules to the residual constraints:

$$\begin{array}{l} [\leq\text{GEN1}] \quad (\forall \bar{a}. Q_1 \Rightarrow \sigma_1) \leq \sigma_2 \rightsquigarrow [\bar{a} \mapsto \bar{\alpha}] \sigma_1 \leq \sigma_2 \wedge [\bar{a} \mapsto \bar{\alpha}] Q_1 \\ [\leq\text{GEN2}] \quad \sigma_1 \leq (\forall \bar{a}. Q_2 \Rightarrow \sigma_2) \rightsquigarrow \sigma_1 \sim (\forall \bar{a}. Q_2 \Rightarrow \sigma_2) \end{array}$$

Note that the first rule is a variation of $[\leq\text{GEN2}]$ without any restriction on the shape of the left-hand side. This corresponds to generating the less amount of polymorphism.

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.

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

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

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

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

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

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

$$\frac{\Gamma \vdash e_1 : \tau_1 \rightsquigarrow C_1 \quad \Gamma, x : \lambda \tau_1 \vdash e_2 : \tau_2 \rightsquigarrow C_2}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau_2 \rightsquigarrow C_1 \wedge C_2} \text{ LET}$$

$$\frac{\Gamma, x : \sigma_1 \vdash e_1 : \sigma_1 \rightsquigarrow C_1 \quad \Gamma, x : \sigma_1 \vdash e_2 : \tau_2 \rightsquigarrow C_2}{\Gamma \vdash \text{let } x :: \sigma_1 = e_1 \text{ in } e_2 : \tau_2 \rightsquigarrow C_1 \wedge C_2} \text{ LETA}$$

$$\frac{\begin{array}{l} \Gamma \vdash e : \tau \rightsquigarrow C \\ \text{for each branch } K_i \bar{x}_i \rightarrow u_i \text{ do} \\ K_i : \forall \bar{a} \bar{b}_i. Q_i \Rightarrow \bar{v}_i \rightarrow \top \bar{a} \in \Gamma \quad \bar{b}_i \text{ fresh} \\ \Gamma, x_i : [\bar{a} \mapsto \bar{\gamma}] \bar{v}_i \vdash u_i : \tau_i \rightsquigarrow C_i \\ \bar{\delta}_i = fuv(\tau_i, C_i) - fuv(\Gamma, \bar{\gamma}) \\ C'_i = \begin{cases} C_i \wedge \tau_i \sim \beta & \text{if } \bar{b}_i \text{ and } Q_i \text{ empty} \\ \forall \bar{\delta}_i. ([\bar{a} \mapsto \bar{\gamma}]) Q_i \supset C_i \wedge \tau_i \sim \beta \end{cases} \end{array}}{\Gamma \vdash \text{case } e \text{ of } \{K_i \bar{x}_i \rightarrow u_i\} : \beta \rightsquigarrow C \wedge \top \bar{\gamma} \sim \tau \wedge \bigwedge C'_i} \text{ CASE}$$

Why $:\lambda$?

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

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

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`)
- $(\forall a. a \rightarrow a) \leq \alpha$ (from `id`)
- $(Bool \rightarrow Bool) \leq \alpha$ (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, than 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.

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

$$\frac{x : \sigma_1 \in \Gamma}{\Gamma \vdash_{\Downarrow} x : \sigma_2 \rightsquigarrow \sigma_1 \leq \sigma_2} \text{VARCON}$$

$$\frac{\Gamma \vdash_{\Downarrow} e : \tau \rightsquigarrow C}{\Gamma \vdash_{\Downarrow} e : (\forall \bar{a}. Q \Rightarrow \tau) \rightsquigarrow \forall \bar{a}. (Q \supset C)} \text{PROP}\forall$$

$$\begin{array}{c}
\frac{\alpha, \beta \text{ fresh} \quad \Gamma, x : \lambda \alpha \vdash_{\Downarrow} e : \beta \rightsquigarrow C}{\Gamma \vdash_{\Downarrow} \lambda x. e : \gamma \rightsquigarrow C \wedge \gamma \sim \alpha \rightarrow \beta} \text{ ABSVAR} \\
\\
\frac{\Gamma, x : \lambda \alpha \vdash_{\Downarrow} e : \sigma_2 \rightsquigarrow C}{\Gamma \vdash_{\Downarrow} \lambda x. e : \alpha \rightarrow \sigma_2 \rightsquigarrow C} \text{ 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) \rightarrow \sigma_2 \rightsquigarrow C} \text{ ABSARROWV} \\
\\
\frac{\sigma_1 \not\equiv \alpha \quad \sigma_1 \not\equiv \forall a. Q \Rightarrow \sigma \quad \Gamma, x : \{\lambda, \cdot\} \sigma_1 \vdash_{\Downarrow} e : \sigma_2 \rightsquigarrow C}{\Gamma \vdash_{\Downarrow} \lambda x. e : \sigma_1 \rightarrow \sigma_2 \rightsquigarrow C} \text{ ABSARROWREST} \\
\\
\frac{\alpha \text{ fresh} \quad \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 \rightarrow \alpha} \text{ ABSAVAR} \\
\\
\frac{\Gamma, x : \sigma_1 \vdash_{\Downarrow} e : \sigma_3 \rightsquigarrow C}{\Gamma \vdash_{\Downarrow} \lambda(x :: \sigma_1). e : \sigma_2 \rightarrow \sigma_3 \rightsquigarrow C \wedge \sigma_2 \sim \sigma_1} \text{ ABSAARROW} \\
\\
\frac{f \in \Gamma \quad f : \forall \bar{a}. \sigma_1 \rightarrow \dots \rightarrow \sigma_n \rightarrow \sigma_r \in \Gamma \quad \theta = [\bar{a} \rightarrow \bar{\alpha}] \quad \Gamma \vdash_{\Downarrow} e_i : \theta(\sigma_i) \rightsquigarrow C_i \quad \theta(\sigma_r) \equiv \beta}{\Gamma \vdash_{\Downarrow} f e_1 \dots e_n : \sigma \rightsquigarrow \bigwedge_i C_i \wedge \theta(Q) \wedge \theta(\sigma_r) \sim \sigma} \text{ APPFUNVAR} \\
\\
\frac{f \in \Gamma \quad f : \forall \bar{a}. \sigma_1 \rightarrow \dots \rightarrow \sigma_n \rightarrow \sigma_r \in \Gamma \quad \theta = [\bar{a} \rightarrow \bar{\alpha}] \quad \Gamma \vdash_{\Downarrow} e_i : \theta(\sigma_i) \rightsquigarrow C_i \quad \theta(\sigma_r) \not\equiv \beta}{\Gamma \vdash_{\Downarrow} f e_1 \dots e_n : \sigma \rightsquigarrow \bigwedge_i C_i \wedge \theta(Q) \wedge \theta(\sigma_r) \leq \sigma} \text{ APPFUNNONVAR} \\
\\
\frac{\Gamma \vdash_{\Downarrow} e_1 : \alpha \rightarrow \sigma \rightsquigarrow C_1 \quad \Gamma \vdash_{\Downarrow} e_2 : \alpha \rightsquigarrow C_2 \quad \alpha \text{ fresh}}{\Gamma \vdash_{\Downarrow} e_1 e_2 : \sigma \rightsquigarrow C_1 \wedge C_2} \text{ 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} \text{ ANNOT} \\
\\
\frac{\alpha \text{ fresh} \quad \Gamma, x : \lambda \alpha \vdash_{\Downarrow} e_1 : \alpha \rightsquigarrow C_1 \quad \Gamma, x : \lambda \alpha \vdash_{\Downarrow} e_2 : \sigma \rightsquigarrow C_2}{\Gamma \vdash_{\Downarrow} \text{let } x = e_1 \text{ in } e_2 : \sigma \rightsquigarrow C_1 \wedge C_2} \text{ LET} \\
\\
\frac{\Gamma, x : \sigma_1 \vdash_{\Downarrow} e_1 : \sigma_1 \rightsquigarrow C_1 \quad \Gamma, x : \sigma_1 \vdash_{\Downarrow} e_2 : \sigma_2 \rightsquigarrow C_2}{\Gamma \vdash_{\Downarrow} \text{let } x :: \sigma_1 = e_1 \text{ in } e_2 : \sigma_2 \rightsquigarrow C_1 \wedge C_2} \text{ LETA}
\end{array}$$

$$\begin{array}{c}
\bar{\gamma} \text{ fresh} \quad \Gamma \vdash_{\Downarrow} e : \top \bar{\gamma} \rightsquigarrow C \\
\text{for each branch } K_i \bar{x}_i \rightarrow u_i \text{ do} \\
\frac{K_i : \forall \bar{a} \bar{b}_i. Q_i \Rightarrow \bar{v}_i \rightarrow \top \bar{a} \in \Gamma \quad \bar{b}_i \text{ fresh}}{\Gamma, x_i : [\bar{a} \mapsto \bar{\gamma}] v_i \vdash_{\Downarrow} u_i : \sigma \rightsquigarrow C_i} \\
\bar{\delta}_i = fuv(\sigma, C_i) - fuv(\Gamma, \bar{\gamma}) \\
C'_i = \begin{cases} C_i & \text{if } \bar{b}_i \text{ and } Q_i \text{ empty} \\ \forall \bar{\delta}_i. ([\bar{a} \mapsto \bar{\gamma}]) Q_i \supset C_i \end{cases} \\
\hline
\Gamma \vdash_{\Downarrow} \text{case } e \text{ of } \{K_i \bar{x}_i \rightarrow u_i\} : \sigma \rightsquigarrow C \wedge \bigwedge C'_i \quad \text{CASE}
\end{array}$$

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 $\mathbf{f} \ (x \rightarrow x)$ above, where \mathbf{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 $error :: \forall b. [Char] \rightarrow b$ to $[Char] \rightarrow \beta$. Since we are pushing down a unification variable α because of the previous application of [APPFUNNONVAR] to $S :: \forall a. a \rightarrow S a$, 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
```

$$\begin{array}{c}
(\$)^{\alpha \rightarrow \beta \rightarrow \gamma} \text{runST}^{\alpha} (e :: \forall s. ST s Int)^{\beta} \\
\forall a, b. (a \rightarrow b) \rightarrow a \rightarrow b \leq \alpha \rightarrow \beta \rightarrow \gamma \\
\forall a. (\forall s. ST s a) \rightarrow a \leq \alpha \\
\forall s. ST s Int \leq \beta \\
\downarrow \\
(\forall s. ST s \epsilon) \rightarrow \epsilon \leq \beta \rightarrow \gamma \\
\forall s. ST s Int \leq \beta \\
\downarrow
\end{array}$$

$$\begin{aligned}
& \forall s. ST\ s\ \gamma \sim \beta \\
& \forall s. ST\ s\ Int \leq \beta \\
& \downarrow \\
& \forall s. ST\ s\ Int \leq \forall s. ST\ s\ \gamma \\
& \downarrow \\
& \forall s. (\epsilon \supset \forall s. ST\ s\ Int \leq ST\ s\ \gamma) \\
& \downarrow \\
& \forall s. (\epsilon \supset ST\ \pi\ Int \sim ST\ s\ \gamma) \\
& \downarrow \\
& \forall s. (\epsilon \supset \pi \sim s \wedge Int \sim \gamma) \\
& \downarrow \\
& \gamma \sim Int
\end{aligned}$$

η -expansion

$$\begin{aligned}
& f :: (\forall a. a \rightarrow a) \rightarrow Int \\
& g_1 = f^\alpha \\
& g_2 = (\lambda x. f^\beta\ x^\gamma)^\delta :: \forall \gamma \delta. (\tau_f \leq \gamma \rightarrow \delta) \Rightarrow \gamma \rightarrow \delta \\
& (\forall a. a \rightarrow a) \rightarrow Int \leq \gamma \rightarrow \delta \\
& \downarrow \\
& (\forall a. a \rightarrow a) \rightarrow Int \sim \gamma \rightarrow \delta \\
& \downarrow \\
& \gamma \sim \forall a. a \rightarrow a \\
& \delta \sim Int
\end{aligned}$$

So we lose nothing by η -expanding!