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 $* \to * \to Constraint$.
- The evidence for $\sigma_1 \leq \sigma_2$ is a function $\sigma_1 \to \sigma_2$.
- The canonical forms are $(\forall \overline{a}.Q \Rightarrow \sigma) \leq \upsilon$, where υ is either a unification variable or a type family application.
- In Haskell code, $\sigma_1 \leq \sigma_2$ is written as `sigma1 <~ sigma2`.

1 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.

1.1 Canonicalization, variables and type families

Unification variables. We disallow applying canonicalization 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.

For the left-hand side, we always look at a constraint $\alpha \leq \sigma$ as $\alpha \sim \sigma$. This conveys the following philosophy of impredicativity in our system:

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	$::=$	$\forall \overline{a}.Q \Rightarrow \sigma \mid \tau$
Instantiable types	ψ	$::=$	$a \mid \sigma_1 \to \sigma_2 \mid \mathsf{T}\ \overline{\sigma}$
Instantiable-or-polymorphic	ψ^*	$::=$	$\psi \mid \forall \overline{a}.Q \Rightarrow \sigma$
Non-instantiable types	υ	$::=$	$\alpha \mid \mathsf{F}\ \overline{\mu}$
Constraints	Q	$::=$	$\epsilon \mid Q_1 \wedge Q_2 \mid \sigma_1 \sim \sigma_2 \mid \psi_1^* \leq \sigma_2 \mid \mathtt{C}\ \overline{\mu}$
Implications	I	$::=$	$\epsilon \mid I_1 \wedge I_2 \mid Q \mid \forall \overline{a}.\,(Q \supset I)$
Canonical constraints	Q^*	$::=$	$\upsilon \sim \sigma \mid a \sim \sigma \mid (\forall \overline{a}.Q \Rightarrow \sigma) \leq \upsilon \mid \mathtt{C}\ \overline{\mu}$
Family-free types	ξ	$::=$	$\alpha \mid a \mid \xi_1 \to \xi_2 \mid \mathsf{T}\ \overline{\xi}$

Figure 1: Grammar

- The system is impredicative in the sense that a polymorphic function may be instantiated at a polytype. Or more operationally, a unification variable may be unified with a polytype.
- But it does not follow that after being filled in with a polytype, a unification variable can then be instantiated.

Instead of producing $\alpha \leq \sigma$ constraints which are later changed to $\alpha \sim \sigma$ and could be affected by the flow of unification, both canonicalization along with generation and propagation work very hard to ensure that no constraint $\alpha \leq \sigma$, with α a unification variable, is ever produced.

Skolem variables. 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 this 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.

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

Figure 2: Canonicalization rules for $\sim$

$$
\begin{array}{lrcl}
[\leq\text{L}\tau\text{R}\tau] & \mathit{canon}\,[\tau_1 \leq \tau_2] & = & \tau_1 \sim \tau_2 \\
[\leq\text{L}\forall\psi\text{R}\psi] & \mathit{canon}\,[(\forall\overline{a}.Q_1 \Rightarrow \psi_1^*) \leq \psi_2] & = & [\overline{a \mapsto \alpha}]\psi_1^* \leq \psi_2 \wedge [\overline{a \mapsto \alpha}]Q_1 \\
[\leq\text{L}\forall\upsilon\text{R}\psi] & \mathit{canon}\,[(\forall\overline{a}.Q_1 \Rightarrow \upsilon_1) \leq \psi_2] & = & [\overline{a \mapsto \alpha}]\upsilon_1 \sim \psi_2 \wedge [\overline{a \mapsto \alpha}]Q_1 \\
[\leq\text{L}\forall\upsilon\text{R}\upsilon] & \mathit{canon}\,[(\forall\overline{a}.\upsilon_1) \leq \upsilon_2] & = & [\overline{a \mapsto \alpha}]\upsilon_1 \sim \upsilon_2 \\
[\leq\text{R}\forall] & \mathit{canon}\,[\sigma_1 \leq (\forall\overline{a}.Q_2 \Rightarrow \sigma_2)] & = & \forall\overline{a}.\,(Q_2 \supset \sigma_1 \leq \sigma_2)
\end{array}
$$

Figure 3: Canonicalization rules for $\leq$

Type families. There is one caveat in our discussion of unification variables: there would still be possible to produce $\alpha \leq \sigma$ is produced from a constraint involving a type family $\mathsf{F}\ \alpha \leq \sigma$ whose family resolves to a variable $\mathsf{F}\ \alpha \sim \alpha$. Our solution is to treat type families as unification variables in this case. Thus, we do not generate $\mathsf{F}\ \overline{\mu} \leq \sigma$ either, we use $\sim$ directly.

There is another 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 $\mathsf{F}\ a \sim a \to a$, which applied to $\forall b.\, b$, gives rise to the constraint $\mathsf{F}, (\forall b.b) \sim (\forall b.\, b) \to (\forall b.\, b)$. This means that we should not instantiate when we find a type family on the right-hand side, either.

As a conclusion, we need to treat type families as unification variables in all circumstances involving $\leq$ constraints. We have introduced a new syntactic category, υ types, to recall this distinction in both the grammar and the rules.

Examples where υ types matter. The following examples show why it is important to generate $\sim$ constraints instead of $\leq$ ones when the left-hand side has a $\forall$ with a unification variable or type family. 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 $\forall b.b \leq a$. Suppose we would apply a canonicalization rule which would give $\beta \leq a$. We would be stuck, because β or a will not get any further unification. In the current system, we use the rule $[\leq \text{L} \forall \upsilon \text{R} \psi]$ and resolve the constraint to $\beta \sim a$.

The same operation is also key in being able to type check the following code:

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

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.

This rule is important in checking some programs with type families. Take the following:

```
type family F a

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

The only constraint generated is $\forall b.\, b \leq \mathsf{F}\ a$. Usually, υ types at the right-hand side do not canonicalize further. Thus, the code would be refused by the compiler with an `Undischarged forall b.b <~ F a` error. However, we have the rule $[\leq \text{L} \forall \upsilon \text{R} \upsilon]$, which rewrites that constraint to $\beta \sim \mathsf{F}\ a$.

1.2 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:

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

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

1.3 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 $[\leq \text{L}\tau\text{R}\tau]$. We need to build $W_1 :: \tau_1 \rightarrow \tau_2$. For this, we can use the evidence $W_2 :: \tau_1 \sim \tau_2$ 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 $\tau_a \sim \tau_b$ to a value of type τ_a to produce the same value, but typed as τ_b.

Rule $[\leq \text{L}\forall\psi\text{R}\psi]$**.** We need to build $W_1 :: (\forall\overline{a}.Q_1 \Rightarrow \psi_1^*) \to \psi_2$ given $W_2 :: [\overline{a \mapsto \alpha}]\psi_1^* \to \psi_2$ and $W_3 :: [\overline{a \mapsto \alpha}]Q_1$. The first step is to get $[\overline{a \mapsto \alpha}]\psi_1^*$ from $\forall\overline{a}.Q_1 \Rightarrow \psi_1^*$, to do that we need to make a type application and afterwards apply the witness for $Q_1 \Rightarrow \sigma_1$:

$$\lambda(x :: \forall\overline{a}.Q_1 \Rightarrow \psi_1^*).\ x\ \overline{\alpha}\ W_3 :: (\forall\overline{a}.Q_1 \Rightarrow \psi_1^*) \to [\overline{a \mapsto \alpha}]\psi_1^*$$

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

$$W_1 = \lambda(x :: \forall\overline{a}.Q_1 \Rightarrow \psi_1^*).\ W_2\ (x\ \overline{\alpha}\ W_3)$$

Rules $[\leq \text{L}\forall\upsilon\text{R}\psi]$ **and** $[\leq \text{L}\forall\upsilon\text{R}\upsilon]$**.** These cases are very similar to the previous one. The only difference is that instead of evidence for another $\leq$ constraint, W_2 is now a coercion. In the most general case we get:

$$W_1 = \lambda(x :: \forall\overline{a}.Q_1 \Rightarrow \upsilon_1).(x\ \overline{\alpha}\ W_3) \rhd W_2$$

where in the case $[\leq \text{L}\forall\upsilon\text{R}\upsilon]$ there is not application of W_3.

Rule $[\leq \text{R}\forall]$**.** 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.

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

1.4 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:

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

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

1.5 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
```

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

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.

2 Constraint generation

2.1 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.

As discussed in the canonicalization rules, we need to ensure that no constraint of the form $\alpha \leq \sigma$ or $\mathsf{F}\,\overline{\mu} \leq \sigma$ is produced. For that matter, we introduce the following operation:

$$\sigma_1 \preccurlyeq \sigma_2 = \begin{cases} \sigma_1 \sim \sigma_2 & \text{if } \sigma_1 \text{ is a } \upsilon \text{ type} \\ \sigma_1 \leq \sigma_2 & \text{otherwise} \end{cases}$$

With this operation and the canonicalization rules, we can guarantee that the $\leq$ constraints inside the system have only the form $\psi_1^* \leq \sigma$.

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

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

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

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

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

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

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

2.2 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.

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

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

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

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

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

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

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

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

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

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

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

The most surprising rule is [AppFun], which applies 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.

2.3 Why (not) $:_\lambda$?

In this section, we describe (part of) the thought process that led to unification variables in the left-hand side to always be regarded as $\sim$ constraints. At first, the conversion from $\leq$ to $\sim$ only occured when the left-hand side involved a type headed by a constructor.

At the very beginning, the only rule that seemed to be in need for a 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, if unification variables in the left-hand side cannot ever be changed into equalities, 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`)
- $(\mathit{Bool} \to \mathit{Bool}) \leq \alpha$ (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 goal turned into making this situation impossible by generating $\alpha \sim \beta$ and $\alpha \sim \gamma$ upfront instead of their $\leq$ counterparts. We did 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 (places where we knew that a fresh variable is generated). 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.

$$\frac{x :_\lambda \tau \in \Gamma}{\Gamma \vdash x : \tau \rightsquigarrow \epsilon} \textsc{VarCon}_\lambda$$

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

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

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.

This was not the only change that was needed to ensure that $\alpha \leq \sigma$ is not produced. You need also a special case of the rule [≤L∀] when the body of a σ-type is a single variable:

$$[\leq \text{L}\forall \text{VAR}] \quad \mathit{canon}\,[(\forall \overline{a}.Q_1 \Rightarrow v) \leq \psi_2] \quad = \quad [\overline{a \mapsto \alpha}]v \sim \psi_2 \wedge [\overline{a \mapsto \alpha}]Q_1 \quad \text{where } v \text{ is a unification variable and } v \in \overline{a}$$

And a special case for the AppFun rule of propagation:

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

For the case of the special AppFun rule, consider the following:

```
data S a = S a

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

If we apply [AppFun] 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 [AppFun] 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.

At the end, we came to the realization that we should never have $\alpha \leq \sigma$ constraints in the system as a general rule. Thus, we removed all those special cases and $:_\lambda$ annotations, and instead moved to an operation $\preccurlyeq$ which takes care of not generating $\leq$ if the left-hand side is a variable or type family.

3 Nice results

3.1 `f $ x` is equivalent to `f x`

One nice property of the system is that the rule AppFun applied to the case `f $ x` for `f` not in the environment is equivalent to the rule App applied to `f x`. In the first case we have:

$$\frac{\alpha, \beta \text{ fresh} \qquad \Gamma \vdash_{\Downarrow} f : \alpha \to \beta \rightsquigarrow C_1 \qquad \Gamma \vdash_{\Downarrow} x : \alpha \rightsquigarrow C_2}{\Gamma \vdash_{\Downarrow} f\ \$\ x : \sigma \rightsquigarrow C_1 \wedge C_2 \wedge \beta \leq \sigma}$$

Note that we know that $\beta \leq \sigma$ will be readily changed to $\beta \sim \sigma$. And by substitution of equals, we get the rule:

$$\frac{\alpha \text{ fresh} \qquad \Gamma \vdash_{\Downarrow} f : \alpha \to \sigma \rightsquigarrow C_1 \qquad \Gamma \vdash_{\Downarrow} x : \alpha \rightsquigarrow C_2}{\Gamma \vdash_{\Downarrow} f \$ x : \sigma \rightsquigarrow C_1 \wedge C_2}$$

Which is exactly the App rule applied to $f\ x$!

Note that this equivalence only holds in the case where App would be applied to the expression `f x`. In particular, if $f \in \Gamma$, the rule AppFun is chosen instead. If equivalence between `f $ x` and `f x` is desired in this case, an extra rule needs to be added:

$$\frac{f \in \Gamma \qquad f : \forall \overline{a}.\, \sigma_s \to \sigma_r \in \Gamma \qquad \theta = [\overline{a \to \alpha}] \qquad \Gamma \vdash_{\Downarrow} e : \theta(\sigma_s) \rightsquigarrow C}{\Gamma \vdash_{\Downarrow} f \,\$\, e : \sigma \rightsquigarrow C \ \wedge\ \theta(Q) \ \wedge\ \theta(\sigma_r) \preccurlyeq \sigma}\ \textsc{App\$Fun}$$

3.2 η-reduction and η-expansion

As usual in impredicative type systems, it is not always the case that you might reduce a λ-abstraction by η-reduction. The reason is that the λ-bound variable might have an annotation, which is lost when η-reducing. Our system, though, still has some nice properties when dealing with λ-abstractions.

Checking f is equivalent to checking $\lambda\, x.f\, x$. More formally, suppose $\Gamma \vdash_{\Downarrow} f :: \forall \overline{a}.Q \Rightarrow \sigma_1 \to \sigma_2$. Then $\Gamma \vdash_{\Downarrow} \lambda\, x.f\, x :: \forall \overline{a}.\, Q \Rightarrow \sigma_1 \to \sigma_2$. There are two cases, depending on $f \in \Gamma$:

- Suppose $f :: (\forall \overline{b}.\, Q^* \Rightarrow \sigma_1^* \to \sigma_2^*) \in \Gamma$. When checking $\Gamma \vdash_{\Downarrow} f :: \forall \overline{a}.Q \Rightarrow \sigma_1 \to \sigma_2$ we get just one constraint by the [VarCon] rule:

  $$(\forall \overline{b}.\, Q^* \Rightarrow \sigma_1^* \to \sigma_2^*) \leq (\forall \overline{a}.Q \Rightarrow \sigma_1 \to \sigma_2)$$

  After several steps of canonicalization, we get to the implication:

  $$\forall \overline{a}.\, (Q \supset \theta(Q^*) \wedge \theta(\sigma_1^*) \sim \sigma_1 \wedge \theta(\sigma_2^*) \sim \sigma_2)$$

  We know that implication is true since the type checking was successful.

  For the η-expanded version, we apply the generation and propagation rules:

  $$\dfrac{\dfrac{\dfrac{\Gamma, x : \sigma_1 \vdash_{\Downarrow} x : \sigma_1 \rightsquigarrow \sigma_1 \leq \theta'(\sigma_1^*)}{\Gamma, x : \sigma_1 \vdash_{\Downarrow} f\, x : \sigma_2 \rightsquigarrow \sigma_1 \leq \theta'(\sigma_1^*) \wedge \theta'(Q^*) \wedge \theta'(\sigma_2) \leq \sigma_2}\ \textsc{AppFun}}{\Gamma \vdash_{\Downarrow} \lambda x.f\, x : \sigma_1 \to \sigma_2 \rightsquigarrow \sigma_1 \leq \theta'(\sigma_1^*) \wedge \theta'(Q^*) \wedge \theta'(\sigma_2) \leq \sigma_2}\ \textsc{AbsArrow}}{\Gamma \vdash_{\Downarrow} \lambda x.f\, x : \forall \overline{a}.Q \Rightarrow \sigma_1 \to \sigma_2 \rightsquigarrow \forall \overline{a}.\, (Q \supset \sigma_1 \leq \theta'(\sigma_1^*) \wedge \theta'(Q^*) \wedge \theta'(\sigma_2) \leq \sigma_2)}\ \textsc{Prop}\forall$$

  It is the case that the instantiation in both the canonicalization rules and the AppFun propagation rule is done in the same way. Thus, we know that $\theta = \theta'$, modulo renaming of variables. Thus, the final constraint in the η-expanded case is equivalent to:

  $$(Q \supset \sigma_1 \leq \theta(\sigma_1^*) \wedge \theta(Q^*) \wedge \theta(\sigma_2) \leq \sigma_2)$$

  This last constraint is clearly implied by the one for type checking f, by taking as solution for $\leq$ constraints the corresponding equality.

- Suppose now that $f \notin \Gamma$ (maybe because it is an expression larger than a single variable). Since we know that it has been proven that $\Gamma \vdash_{\Downarrow} f :: \forall \overline{a}.Q \Rightarrow \sigma_1 \to \sigma_2$, it must have been the case that the last rule applied is PROP$\forall$.

$$\frac{f :: \sigma_1 \to \sigma_2 \rightsquigarrow C}{f :: \forall \overline{a}.\, Q \Rightarrow \sigma_1 \to \sigma_2 \rightsquigarrow \forall \overline{a}.\, (Q \supset C)}\ \textsc{Prop}\forall$$

  Now, for the η-expanded version, we get the following typing derivation:

$$\dfrac{\dfrac{\dfrac{f :: \sigma_1 \to \sigma_2 \rightsquigarrow C}{\Gamma, x : \sigma_1 \vdash_{\Downarrow} f : \alpha \to \sigma \rightsquigarrow C \wedge \alpha \sim \sigma_1} \quad \dfrac{}{\Gamma, x : \sigma_1 \vdash_{\Downarrow} x : \alpha \rightsquigarrow \sigma_1 \leq \alpha}\ \textsc{VarCon}}{\Gamma, x : \sigma_1 \vdash_{\Downarrow} f\, x : \sigma_2 \rightsquigarrow C \wedge \alpha \sim \sigma_1 \wedge \sigma_1 \leq \alpha}\ \textsc{App}}{\Gamma \vdash_{\Downarrow} \lambda x.f\, x : \sigma_1 \to \sigma_2 \rightsquigarrow C \wedge \alpha \sim \sigma_1 \wedge \sigma_1 \leq \alpha}\ \textsc{AbsArrow}}{\Gamma \vdash_{\Downarrow} \lambda x.f\, x : \forall \overline{a}.\, Q \Rightarrow \sigma_1 \to \sigma_2 \rightsquigarrow \forall \overline{a}.\, (Q \supset C \wedge \alpha \sim \sigma_1 \wedge \sigma_1 \leq \alpha)}\ \textsc{Prop}\forall$$

  Clearly, we can make $C \wedge \alpha \sim \sigma_1 \wedge \sigma_1 \leq \alpha$ equivalent to C by unification of the fresh variable α to σ_1. Since it is fresh, we know that it is not captured in any other part of the typing derivation. Thus, we get equivalent constraints from the η-reduced and η-expanded versions.

Inferring f is (almost) equivalent to inferring $\lambda x.f\, x$. When inferring, we apply the propagation rules starting with a fresh unification variable γ. In this case, the η-expanded version might not return the exact same type as the η-reduced version, but a less polymorphic version, depending on the generalization strategy taken by the compiler.

Let us focus on the case when $f : \forall \overline{a}.\, Q \Rightarrow \sigma_1 \to \sigma_2 \in \Gamma$. The η-expanded version has the following derivation tree:

$$\dfrac{\dfrac{\dfrac{}{\Gamma, x : \alpha \vdash_{\Downarrow} x : \theta(\sigma_1) \rightsquigarrow \alpha \sim \theta(\sigma_1)}\ \textsc{VarCon}}{\Gamma, x : \alpha \vdash_{\Downarrow} f\, x : \beta \rightsquigarrow \theta(Q) \wedge \alpha \sim \theta(\sigma_1) \wedge \theta(\sigma_2) \leq \beta}\ \textsc{AppFun}}{\Gamma \vdash_{\Downarrow} \lambda x.f\, x : \gamma \rightsquigarrow \gamma \sim \alpha \to \beta \wedge \theta(Q) \wedge \alpha \sim \theta(\sigma_1) \wedge \theta(\sigma_2) \leq \beta}\ \textsc{AbsVar}$$

If σ_2 has no $\forall$ inside it, or the approximation strategy involved changing $\leq$ to $\sim$ everywhere, we get as final set of constraints:

$$\gamma \sim (\theta(\sigma_1) \to \theta(\sigma_2)) \ \wedge\ \theta(Q)$$

Once we generalize, we shall get back the original type we started with. However, if we approximate by instantiation and σ_2 is headed by $\forall$, we might get back a less polymorphic type. For example, if we started with:

$$f :: (\forall a.a \to a) \to (\forall b.b \to b)$$

Then we would have:

$$\lambda x.f\, x :: \forall b.\, ((\forall a.a \to a) \to b \to b)$$

4 Examples

4.1 `runST e`

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

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

$$\downarrow$$

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

$$\downarrow$$

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

$$\downarrow$$

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

$$\downarrow$$

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

$$\downarrow$$

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

$$\downarrow$$

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

$$\downarrow$$

$$\gamma\ \sim\ Int$$

4.2 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!