(note this is a draft,I plan to change the actual source codes to psuedo-source, which hides the implementation choices like constructor names. And also apply proper typesetting and replace the images with actual latex code. I'll also move to generating documentation like UHC does)

1 HML

The type system implemented in this thesis is called HML, this section explains the finer details of the type system which is needed to understand the rest of this thesis.

1.1 Types

Types	$\sigma ::= \forall\alpha.\,\sigma \mid \alpha \mid c\ \sigma_1 \ldots \sigma_n$
Type schemes	$\varphi ::= \forall(\alpha \geqslant \varphi_1).\,\varphi_2 \mid \sigma \mid \bot$
Prefix	$Q ::= \alpha_1 \geqslant \hat{\varphi}_1, \ldots, \alpha_n \geqslant \hat{\varphi}_n$
Mono types	$\tau ::= \alpha \mid c\ \tau_1 \ldots \tau_n$
Unquantified types	$\rho ::= \alpha \mid c\ \sigma_1 \ldots \sigma_n$
Quantified schemes	$\hat{\varphi} ::= \forall(\alpha \geqslant \hat{\varphi}_1).\,\varphi_2$ with $\alpha \in ftv(\varphi_2) \mid \bot$
Syntactic sugar	$\forall\alpha = \forall(\alpha \geqslant \bot)$
	$\forall Q.\,\varphi = \forall(\alpha_1 \geqslant \hat{\varphi}_1).\ \ldots \forall(\alpha_n \geqslant \hat{\varphi}_n).\,\varphi$

Figure 1: HML Types

The types in HML are relatively simple. The first types in the figure are the SystemF (σ) types. You have the case for Quantification $\forall\alpha.\sigma$ which binds α in σ. The second case is the case for variable α and the last case is the case

for constructors $c\sigma_1 \ldots \sigma_n$. The construct $\rightarrow$ is not treated any specially in this type system.

Type Schemes (φ) are the most important types in HML. It consist of the bottom $\perp$ type, which signifies undefined values. A type scheme can also consist of a systemF type σ or be a quantified type. The form of the quantification is different than that of SystemF types. The quantification $\forall(\alpha \geq \varphi_1).\varphi_2$ should be interpreted as *in the type φ_2 the variable α is bound by the quantification. The value of α can be any instantiation of the type φ_1*. These quantifications can be arbitrarily deep or long. e.g. $\forall(\alpha \geq \forall(\beta \geq \varphi_1).\varphi_3).\varphi_2$ which contains a deep nested type.

The next type is called the *Prefix*. This is essentially a type environment where every variable a_i is bound to the quantified type scheme $\hat{\varphi}_i$. The *Prefix* cannot contain any *trivial* types, which is to say, It can now contain any σ types as a bound. Instead of creating a binding for such types inside the prefix a substitution is created instead. The subtitution environment will contain atleast those types that are not in bound in the prefix, or in other words, the domain of the subsitution environment θ and of the *Prefix* are disjoint. Prefixes have a implicit ordering on them, the order of the elements in the prefix should always be the order in which they were introduced. This is important for a number of functions but also important for the right ordering of quantifications.

To express this limitation the Quantified schemes $\hat{\varphi}$ were created. However this shouldn't be viewed as a *concrete* type, but rather a type synonym. The invariant on the Prefix is maintained by the functions that can extend the prefix.

The Mono type τ and the Unquantified type ρ types should be seen as a testable constraint and not real types. A type can be tested to see if it's a mono type, but no function actually generated a mono type. The following two types are purely syntactical sugar and don't actually exist. The first one simply states that the systemF quantification $\forall\alpha$ is equivallent to the type scheme quantification $\forall(\alpha \geq \perp)$, this is called an unconstrained bound. The second states that quantifying a type scheme with a Prefix means concatinating the types in the Prefix together. It's also important to note that every SystemF type can be represented as a flexible type, and vice-versa.

1.2 Normal Form

In order to easily compare types, they are usually converted to normal form first. This convertion si specified by:

$$
\begin{array}{lll}
nf(\sigma) & = \sigma & \\
nf(\bot) & = \bot & \\
nf(\forall(\alpha \geqslant \varphi_1).\, \varphi_2) & = nf(\varphi_2) & \text{iff } \alpha \notin ftv(\varphi_2) \\
nf(\forall(\alpha \geqslant \varphi_1).\, \varphi_2) & = nf(\varphi_1) & \text{iff } nf(\varphi_2) = \alpha \\
nf(\forall(\alpha \geqslant \varphi_1).\, \varphi_2) & = nf([\alpha := \rho]\varphi_2) & \text{iff } nf(\varphi_1) = \rho \\
nf(\forall(\alpha \geqslant \varphi_1).\, \varphi_2) & = \forall(\alpha \geqslant nf(\varphi_1)).\, nf(\varphi_2) &
\end{array}
$$

Figure 2: Normal form conversion

1.3 Utility functions

There are a number of simple utility functions to make things easier. These all operate on the Prefix Q and all keep the invariant placed on Q.

1.3.1 update function

The update functions updates a prefix with a new binding or updates an existing binding. $update(Q, \alpha \geq \varphi_2)$ updates the binding for α in Q. First Q is split based on the free variables of φ_2. This is a form of scoping and in the case that $\alpha \in ftv(\varphi_2)$ is prevents a possible infinite expansion(by repeated calls to update). The existing binding to α in the Prefix is removed leaving 3 sub prefixes. The prefixes Q_0 and Q_0 have no further influence on the operations, but the Prefix Q_2 can contain references to α. Remember the invariant on Q and that prefixed when desugared follow eachother. For example $[(\alpha \geq \bot), (\beta \geq (\forall(\gamma \geq \bot).\alpha \rightarrow \gamma))]$ is a valid type and would be unfolded to $\forall(\alpha \geq \bot).\forall(\beta \geq (\forall(\gamma \geq \bot).\alpha \rightarrow \gamma)).\beta$ for instance.

The first check is to see if the normal form of φ_2 is a unquantified type. This check preserves the invariant that no trivial bounds (e.g. unquantified type) can enter the environment. If the type is unquantified, the substitution $[\alpha := \sigma]$ is applied to Q_2 since there is no longer any binding for α in the prefix, the result of the substitution is concatinated with Q_0 and Q_1

(update the prefix)

$update(Q, \alpha \geqslant \varphi_2)$

let $(Q_0, (Q_1, \alpha \geqslant \hat{\varphi}_1, Q_2)) = split(Q, ftv(\varphi_2))$

if $(nf(\varphi_2) = \rho)$

then return $((Q_0, Q_1, [\alpha := \rho]Q_2), [\alpha := \rho])$

else return $((Q_0, Q_1, \alpha \geqslant \varphi_2, Q_2), [])$

$update(Q, \alpha := \sigma)$

let $(Q_0, (Q_1, \alpha \geqslant \varphi, Q_2)) = split(Q, ftv(\sigma))$

return $((Q_0, Q_1, [\alpha := \sigma]Q_2), [\alpha := \sigma])$

(extend a prefix

$extend(Q, \alpha \geqslant \varphi) =$

if $(nf(\varphi) = \rho)$

then return $(Q, [\alpha := \rho])$

else return $((Q, \alpha \geqslant \varphi), [])$

(split a prefix)

$split :: (Q, \overline{\alpha}) \rightarrow (Q, Q)$

$split(\varnothing, \overline{\alpha}) =$

return $(\varnothing, \varnothing)$

$split((Q, \alpha \geqslant \hat{\varphi}), \overline{\alpha}) =$ with $\alpha \in \overline{\alpha}$

let $(Q_1, Q_2) = split(Q, ((\overline{\alpha} - \alpha) \cup ftv(\varphi)))$

return $((Q_1, \alpha \geqslant \hat{\varphi}), Q_2)$

$split((Q, \alpha \geqslant \hat{\varphi}), \overline{\alpha}) =$ with $\alpha \notin \overline{\alpha}$

let $(Q_1, Q_2) = split(Q, \overline{\alpha})$

return $(Q_1, (Q_2, \alpha \geqslant \hat{\varphi}))$

Figure 3: Utility functions

in the order specified to retain the invariant. Along with the new Prefix the substitution is returned so that any other type can also be updated. If the type is not a unquantified type a new binding $\alpha \geq \varphi_2$ is put in the environment in the same place as the old one was removed. This function also illustrates that φ and $\hat{\varphi}$ are the same types.

$update(Q, \alpha := \sigma)$ is the same as $update(Q, \alpha \geq \varphi_2)$ when $nf(\varphi_2)$ is a unquantified type.

1.3.2 extend

Extends attemps to extend the Prefix with a new entry, just as update is does so only if the normal form of the bound is not a unquantified type. If that is not the case a substitution is returned instead.

1.3.3 split

The split function is the most trivial of the three, It just splits the prefix in 2. The first containing all the bindings where the variable is an element of the variable list given and the second with the those that are not. The ordering is broken, but inside the two Prefixes the invariant still holds. Because of the way split is used (e.g. always on the ftv list of a binding we're updating) the invariant would still hold after concatinating the two prefixes back together.

1.4 Inferrence

The inferrence algorithem is pretty self explanatory. The difference with the standard Hindley-Milner inferrence is that an extra environment Q (the Prefix) is also passed along. The extend calls are how we insert a new binding inside the Prefix. A type is always quantified, either in the Prefix Q or in the type itself. It will be elaborated on later when the EH implementation of it is presented.

It is important to note that the result of the inferrence will always be the desugared types, since the syntactical sugar is not an actual type.

1.5 Unification

The unification function will be elaborated on in the implementation section, just as the inferrence algorithm. Here we would like to point out that it uses a mix of SystemF types σ and flexible types φ. In particulair we beleive this is not needed and it would be much nicer to only use one of the two typee. Since a systemF type can always be expressed in a flexible type we will show a version of unify that uses only flexible types.

$$
\begin{array}{l}
\mathit{infer} :: (Q, \Gamma, e) \to (Q, \theta, \varphi) \\
\\
\mathit{infer}(Q, \Gamma, x) = \\
\quad \text{return } (Q,\ [],\ \Gamma(x)) \\
\\
\mathit{infer}(Q, \Gamma,\ \textbf{let } x = e_1 \textbf{ in } e_2) \\
\quad \text{let } (Q_1, \theta_1, \varphi_1) = \mathit{infer}(Q, \Gamma, e_1) \\
\quad \text{let } (Q_2, \theta_2, \varphi_2) = \mathit{infer}(Q_1, (\theta_1\Gamma, x : \varphi_1), e_2) \\
\quad \text{return } (Q_2,\ \theta_2 \circ \theta_1,\ \varphi_2) \\
\\
\mathit{infer}(Q, \Gamma, \lambda x.\ e) = \\
\quad \text{assume } \alpha, \beta \text{ are fresh} \\
\quad \text{let } (Q_1, \theta_1, \varphi_1) = \mathit{infer}((Q, \alpha \geqslant \bot), (\Gamma, x : \alpha), e) \\
\quad \text{fail if not } (\theta_1\alpha = \tau) \text{ for some } \tau \\
\quad \text{let } (Q_2,\ Q_3) \quad\ = \mathit{split}(Q_1, \mathit{dom}(Q)) \\
\quad \text{let } (Q_3', \theta_3') \quad\ = \mathit{extend}(Q_3, \beta \geqslant \varphi_1) \\
\quad \text{return } (Q_2,\ \theta_1,\ \forall Q_3'.\ \theta_1\alpha \to \theta_3'\beta) \\
\\
\mathit{infer}(Q, \Gamma, \lambda(x :: \sigma).\ e) = \\
\quad \text{assume } \beta \text{ is fresh, } \sigma \text{ is closed} \\
\quad \text{let } (Q_1, \theta_1, \varphi_1) = \mathit{infer}(Q, (\Gamma, x : \sigma), e) \\
\quad \text{let } (Q_2,\ Q_3) \quad\ = \mathit{split}(Q_1, \mathit{dom}(Q)) \\
\quad \text{let } (Q_3', \theta_3') \quad\ = \mathit{extend}(Q_3, \beta \geqslant \varphi_1) \\
\quad \text{return } (Q_2,\ \theta_1,\ \forall Q_3'.\ \sigma \to \theta_3'\beta) \\
\\
\mathit{infer}(Q, \Gamma, e_1\ e_2) = \\
\quad \text{assume } \alpha_1, \alpha_2, \beta \text{ are fresh} \\
\quad \text{let } (Q_1, \varphi_1, \theta_1) = \mathit{infer}(Q, \Gamma, e_1) \\
\quad \text{let } (Q_2, \varphi_2, \theta_2) = \mathit{infer}(Q_1, \theta_1\Gamma, e_2) \\
\quad \text{let } (Q_2', \theta_2') \ = \mathit{extend}(Q_2, \alpha_1 \geqslant \theta_2\varphi_1, \alpha_2 \geqslant \varphi_2, \beta \geqslant \bot) \\
\quad \text{let } (Q_3, \theta_3) \ = \mathit{unify}(Q_2', \theta_2'\alpha_1, \theta_2'\alpha_2 \to \beta) \\
\quad \text{let } (Q_4,\ Q_5) = \mathit{split}(Q_3, \mathit{dom}(Q)) \\
\quad \text{return } (Q_4,\ \theta_3 \circ \theta_2 \circ \theta_1,\ \forall Q_5.\ \theta_3\beta)
\end{array}
$$

Figure 4: Inferrence algorithm

The last case if never used during normal unification. It's only used when you specifically want to unifify two quantified systemF types. for instance when checking a type signature given by a user with a inferred type.

$\alpha \in \mathit{dom}(Q/\hat{\varphi})$ should be seen as a reachability test. It means formally that $\alpha \in \mathit{dom}(Q/\hat{\varphi})$ if and only if $Q = (Q_1, a \geq \hat{\varphi}_1, Q_2)$ and $\alpha \in \mathit{ftv}(\forall Q_2.\varphi)$. As an example $\mathit{dom}((\gamma \geq \bot, \beta \geq \forall\delta.\delta \to \gamma)/(\forall\alpha.\alpha \to \beta))$ is $\{\beta, \gamma\}$ because even

$unify :: (Q, \sigma_1, \sigma_2) \rightarrow (Q, \theta)$
where σ_1 and σ_2 are in normal form
and $(\alpha := \varphi) \in \theta \;\Rightarrow\; nf(\varphi) = \sigma$

$unify(Q, \alpha, \alpha) =$
return $(Q, [\,])$

$unify(Q_1, c\ \sigma_1 \ldots \sigma_n, c\ \sigma'_1 \ldots \sigma'_n) =$
let $\theta_1 \quad = [\,]$
let $\theta_{i+1} = \theta'_i \circ \theta_i$
let $(Q_{i+1}, \theta'_i) = unify(Q_i, \theta_i\sigma_i, \theta_i\sigma'_i)$
return $(Q_{n+1},\ \theta_{n+1})$

$unify(Q, \alpha, \sigma)$ or
$unify(Q, \sigma, \alpha)$ with $(\alpha \geqslant \hat{\varphi}) \in Q \wedge \sigma \notin \mathcal{V}$
fail if $(\alpha \in dom(Q/\sigma))$ ('occurs' check)
let $(Q_1, \theta_1) = subsume(Q, \sigma, \hat{\varphi})$
let $(Q_2, \theta_2) = update(Q_1, \alpha := \theta_1\sigma)$
return $(Q_2,\ \theta_2 \circ \theta_1)$

$unify(Q, \alpha_1, \alpha_2)$ with $(\alpha_1 \geqslant \hat{\varphi}_1) \in Q \wedge (\alpha_2 \geqslant \hat{\varphi}_2) \in Q$
fail if $(\alpha_1 \in dom(Q/\hat{\varphi}_2) \vee \alpha_2 \in dom(Q/\hat{\varphi}_1))$
let $(Q_1, \theta_1, \varphi) = unifyScheme(Q, \hat{\varphi}_1, \hat{\varphi}_2)$
let $(Q_2, \theta_2) \quad = update(Q_1, \alpha_1 := \alpha_2)$
let $(Q_3, \theta_3) \quad = update(Q_2, \alpha_2 \geqslant \varphi)$
return $(Q_3,\ \theta_3 \circ \theta_2 \circ \theta_1)$

$unify(Q, \forall\alpha.\, \sigma_1, \forall\beta.\, \sigma_2) =$
assume c is a fresh (skolem) constant
let $(Q_1, \theta_1) = unify(Q, [\alpha := c]\sigma_1, [\beta := c]\sigma_2)$
fail if $(c \in (con(\theta_1) \cup con(Q_1)))$
return $(Q_1,\ \theta_1)$

Figure 5: Unification algorithm

though γ is not directly reachable from the type is is reachable by β.

1.6 Instantiation

The next two functions are instantiation and scheme unification.

Instantiation is just your standard instantiation, We're tying to instantiate

$subsume :: (Q, \sigma, \varphi) \rightarrow (Q, \theta)$
where σ and φ are in normal form

$subsume(Q, \forall \overline{\alpha}.\, \rho_1, \forall Q_2.\, \rho_2)$
assume $dom(Q) \# dom(Q_2)$ and $\overline{c}$ are fresh constants
let $(Q_1, \theta_1) = unify(QQ_2, [\overline{\alpha} := \overline{c}]\rho_1, \rho_2)$
let $(Q_2, Q_3) = split(Q_1, dom(Q))$
let $\theta_2 = \theta_1 - dom(Q_3)$
fail if $(\overline{c} \in (con(\theta_2) \cup con(Q_2)))$
return (Q_2, θ_2)

$unifyScheme :: (Q, \varphi_1, \varphi_2) \rightarrow (Q, \theta, \varphi)$
where φ_1 and φ_2 are in normal form

$unifyScheme(Q, \bot, \varphi)$ or $unifyScheme(Q, \varphi, \bot)$
return $(Q, [], \varphi)$

$unifyScheme(Q, \forall Q_1.\, \rho_1, \forall Q_2.\, \rho_2)$
assume the domains of Q, Q_1, and Q_2 are disjoint
let $(Q_3, \theta_3) = unify(QQ_1Q_2, \rho_1, \rho_2)$
let $(Q_4, Q_5) = split(Q_3, dom(Q))$
return $(Q_4, \theta_3, \forall Q_5.\, \theta_3\rho_1)$

Figure 6: subsume & unifyScheme

the first type with the second type. In order to assure that we don't instantiate quantified variables we skolemize them and call unify again. Afterwards we return the a the updated bindings for the entried that were originally already in the Prefixed passed to instantiate, they now contain the bindings needed to instantiate ρ_1 to ρ_2. But since simple bounds can't be in the prefix, we also need to return the scoped substitutions. We also scope the substitutions so that skolem variables don't escape from instantiation since they hold no intrinsic value and would prevent further unification. This function is what handles higher-rank types.

However a very important case is missing from the given specification, which is when we try to instantiate a type with $\bot$. Since instantiation always succeeds when instantiating to $\bot$ we should always return $(Q, [])$ for calls to $subsume(Q, \sigma, \bot)$.

UnifyScheme is used to unify two flexible types. It returns aside from a new Prefix and substitution environment also the most general type φ

that describes both ρ_1 and ρ_2. In other words, using the Prefix Q and the substitutions θ, ρ_1 and ρ_2 can be instantiated to φ.

1.7 Typing rules

The type rules in HML are almost the same as the ones in Hindley-Milner, except that in the presence of *flexible* types the Instantiation rule is slightly different and every rule now gets an explicit *Prefix*.

$$\text{Var:} \quad \frac{x : \varphi \in \Gamma}{Q, \Gamma \vdash x : \varphi}$$

The var rule is still very straight forward, if there is a binding for x in the environment Γ with the type φ then return the type of φ for x under the same environment Γ and prefix Q.

$$\text{Inst:} \quad \frac{Q, \Gamma \vdash e : \varphi_1 \quad Q \vdash \varphi_1 \sqsubset \varphi_2}{Q, \Gamma \vdash e : \varphi_2}$$

The instantiation rule states that we can always use a instance of a type under the prefix Q as the type of the expression. This instance of relation is denote by the $\sqsubset$ relation above which states that φ_2 is an instance of φ_1 under the prefix Q

$$\text{Gen:} \quad \frac{(Q, \alpha \geq \hat{\varphi}_1), \Gamma \vdash e : \varphi_2 \quad \alpha \notin ftv(\Gamma)}{Q, \Gamma \vdash e : \forall(\alpha \geq \hat{\varphi}_1).\varphi_2}$$

The generalization rule generalizes a type by moving it's bounds out from the prefix Q into the type φ.

$$\text{App:} \quad \frac{Q, \Gamma \vdash e_1 : \sigma_2 \to \sigma \quad Q, \Gamma \vdash e_2 : \sigma_2}{Q, \Gamma \vdash e_1\ e_2 : \sigma}$$

The application rule is the standard application rule over types. It requires that the type of e_2 be equal to the type of the argument of e_1. Note that this ranges over type and not typeschemes.

$$\text{Let:} \quad \frac{Q, \Gamma \vdash e_1 : \varphi_1 \quad Q, (\Gamma, x : \varphi_1) \vdash e_2 : \varphi_2}{Q, \Gamma \vdash \textbf{Let}\ x = e_1\ \textbf{in}\ e_2 : \varphi_2}$$

The Let rule is pretty straight forward, given an expression e_1 with type φ_1 and a variable x which in the environment Γ has the same type φ_1 and an expression e_2 with type φ_2 we can create a let binding **Let** x = e_1 **in** e_2.

$$\text{Fun:} \quad \frac{Q, (\Gamma, x : \tau) \vdash e : \sigma}{Q, \Gamma \vdash \lambda x.e : \tau \to \sigma}$$

The Fun rule restricts the type of the parameter to a monomorphic type τ in order to avoid having to guess polymorphic types. This does not mean that a polymorphic type can not be infered since the type schemes are hidden in the prefix Q.

$$\text{Fun-Ann:} \quad \frac{Q, (\Gamma, x : \sigma_1) \vdash e : \sigma_2}{Q, \Gamma \vdash \lambda(x :: \sigma_1).e : \sigma_1 \to \sigma_2}$$

The Function Annotation rule is used to provide a mechanism to explicitly annotate abstraction variables with a (possibly) polymorphic type. Like annotation of the *push* example above with the needed higher-rank type.

In particular the *FUN* rule for lambda expressions restrict the type of the parameters to be a mono type τ since any scheme on the mono type is in the Prefix Q, this is the secrete behind HML.

2 Modifications to HML

The implementation of HML differes somewhat compared to textbook HML[**?**]. During implementation of the algorithm we realised that internally it used both flexible types and SystemF types. This introduces a number of differences between the textbook implementation and the version we implemented.

<insert figure containing full algorithm here>

2.1 Promotions

We wanted to limit the amount that SystemF types were being used and promote more usage of flexible types during unification. This is accomplished by modifying the *extend* function, which is the only way to introduce a new type binding in the *Prefix*. This function as mentioned before also enforces the invariant that no *simple* bounds can enter the environment. These are instead returned as a substitution;

```
extend :: (Prefix, Scheme) -> (Prefix, Env)
extend (q, (Scheme_Simple var phi))
```

```
 = let p = nf phi
   in case isUnQualTy p of
        True  -> (q, [(var, p)])
        _     -> (q++[TyIndex_Group var (promote phi)], [])
```

The only difference between this and the described function in the HML specification is that upon adding the type to the *Prefix* it is first *promoted* by the **Promote** function.

```
promote :: TyScheme -> TyQuantifiedScheme
promote TyScheme_Bottom
  = TyScheme_Bottom
promote (TyScheme_Quant (Scheme_Simple nm ty) ty')
  = TyScheme_Quant (Scheme_Simple nm (promote ty)) ty'
promote (TyScheme_SystemF x)
  = embedF x
```

Implementing promote is pretty straight forward. If we find a $\perp$ value we just ignore it, if we find a type scheme we just have to check in it's bounds to make sure it doesn't contain a SystemF type, and if we find a SystemF type we convert it to a flexible type using *embedF*.

```
embedF :: TyExpr -> TyScheme
embedF = embed id
 where embed :: (TyScheme -> TyScheme) -> TyExpr -> TyScheme
       embed val (TyExpr_Parens     s) = embed val s
       embed val (TyExpr_Quant  _ a t) = let e = TyScheme_Quant (Scheme_Simple ↩
           a TyScheme_Bottom)
                                           in  embed (val . e) t
       embed val e                     = val (TyScheme_SystemF e)
```

embedF works by stripping every outter $\forall$-quantifiers and replacing them with type schemes instead. We only replace the outter quantifiers which means that the only $\forall$s still in the type would be those of higher rank types.

When confronted during unification with two quantified SystemF types the *classic* implementation it proceeded by skolemizing both types and recursively call unification. If we were to remove the usage of SystemF types by unification this case would no longer be needed. The actual comparison is already being done by *instantiation* (subsume) where we would still atmost have one quantified SystemF type.

This SystemF type in *subsume* can also be removed, but that would complicate both *embedF* and *subsume*. If we were to do it the required changes would be to change embedF to:

```
embedF :: TyExpr -> Int -> TyScheme
```

```
embedF e = uncurry ($) . second (TyScheme_SystemF . fst) . embed False e
 where embed :: Bool -> TyExpr -> Int -> (TyScheme -> TyScheme, (TyExpr, Int))
       embed b (TyExpr_Parens     s) i = if b
                                         then let (f1, (e, i')) = embed False ↩
                                             s i
                                                  (var, ires)   = freshT "t" i↩
                                                      '
                                                  f2            = ↩
                                                      TyScheme_Quant (↩
                                                      Scheme_Simple var (f1 $ ↩
                                                      TyScheme_SystemF e))
                                              in (f2, (TyExpr_Var var, ires))
                                         else second (first TyExpr_Parens) (↩
                                             embed b s i)
       embed b (TyExpr_Quant  _ a t) i = let e = TyScheme_Quant (Scheme_Simple ↩
           a TyScheme_Bottom)
                                         in  first (e.) $ embed b t i
       embed b (TyExpr_Forall _ v t) i = let e = foldl' (.) id [ TyScheme_Quant↩
            (Scheme_Simple a TyScheme_Bottom) | a <- v ]
                                         in  first (e.) $ embed b t i
       embed b (TyExpr_AppTop     x) i = let (f, (e, i')) = embed b x i
                                         in  (f, (TyExpr_AppTop e, i'))
       embed _ (TyExpr_App      f x) i = let (f1, (e1, i1)) = embed True f i
                                             (f2, (e2, i2)) = embed True x i1
                                         in  (f2 . f1, (TyExpr_App e1 e2, i2))
       embed _ e                   i = (id, (e, i))
```

where *first* and *second* are arrow functions. In this context *first f* applies f to the first element in the tuple and *second* to the second element respectively. *freshT* generates a fresh variable using the given template *"t"* in this case and a starting point.

The reason we need to generate a fresh variable is that a type scheme is not valid within a systemF type. As an example, the type $\forall a.\forall b.(\forall c.c \rightarrow Int) \rightarrow a \rightarrow b$ would become the following flexible type: $(a \geq \bot).(b \geq \bot).(t1 \geq (c >= \bot).c \rightarrow Int).t1 \rightarrow a \rightarrow b$.

The next objective is to modify *subsume* to take two flexibly types as argument and to skolemize the variables of the first type where they're bound to a $\bot$. e.g. in the example above only a and b should be skolemized. This is because skolemization is done on bounded quantifiers, and *t1* is not a $\forall$ in systemF but a type.

2.2 Renaming

When typechecking an expression such as (id id) the environment would return the same type for both occurences of id, this is logical since they are the same values. However because they use the same type variables unification will get confused and accidently apply substitution to both

types.

Both types are entered into the Prefix, however, eventually when both are inspected the same binding (e.g. $(a \geq \bot)$) will be put in the prefix. While there are two occurences, there's nothing that distinguishes between them. Because of the ordering inside the Prefix, any actions executed which was intended to be executed on the second entry is only applied to the first.

To prevent this, we explicitly α-rename types before we add them to the Prefix. this also ensures that all type variables are unique. We must also store the α-renaming substitution and include it in the final substitution mappings returned from unification. This becomes apparent if we look at the simple example of

```
op :: (Int -> t) -> t
op = \x -> x 1
```

When the variable x is introduced in the λ-abstraction it is given a type a. During unification the type of x is renamed to b and the type of b is later inferred to be (Int $\rightarrow$ t) but in the λ x is still bound to a, which is why the renaming substitution (a, b) should be returned.

2.3 Prefix scoping

Because we now produce very deeply embedded types from the type checker, in order to get scoping of the type variables correct a few changes had to be made.

2.3.1 Prefix Domain

The domain of a prefix Q is $\{a_1 \ldots a_n\}$ where $\{(a_1 \geq \ldots), \ldots, (a_n \geq \ldots)\}$. A consequence of having mostly flexible types during unification is that deeply nested type schemes can be created (most are only 2 levels deep however). This is a problem for the correct scoping of types. Consider the example *map id* where $id :: (a \geq (b \geq \bot).b \rightarrow b).a$

The type returned would be $[b] \rightarrow [b]$ with in the prefix the binding for b $(b \geq \bot)$ and not $[a] \rightarrow [a]$. When we generalise on the result of unification we do so based on the *domain* of the original prefix passed to the call which in this case would be just a.

What we end up determining for *map id* is thus improperly scoped since b was never in the original domain. For this reason we change the definition of *domain* to also take into account the recursive nature of prefixes;

```
instance Domain TyIndex where
  domain   (TyIndex_Group a b) = a : binds b
    where binds :: TyScheme -> [HsName]
          binds (TyScheme_Forall a _) = a
          binds (TyScheme_Quant a b ) = domain a ++ binds b
          binds                     _ = []
```

The domain for *id* now returns a, b which now allows us to properly apply generalisation.

2.3.2 Type split

When typing types such as that of the compose function *(f (g x))* the type of f depends on the type of g which in turn depends on the type of x alot of information is ”hidden” in the recursive prefixes returned by infer. To properly account for these we introduce the type rule **Scope**

$$\text{Scope:} \quad \frac{Q, \Gamma \vdash e : \forall Q_1.\varphi}{QQ_1, \Gamma \vdash e : \varphi}$$

This rule moves the bounds from the type itself into the environment. Because a type is either fully quantified or bound by the environment this rule is sound and thus safe to perform.

2.4 Desugaring

At the end of every *infer* step, we generalise the type and return it. This is done using the syntax $\forall Q.\varphi$ which is syntactical sugar for the unfolding introduced in the previous chapter.

However we take this opportunity to simplify the type a bit. Instead of just returning the unrolled type we redefine the *desugar* step to also perform a partial convertion.

$$\begin{aligned} &\text{ptype}(\varphi) = \text{ft}(\varphi) \\ &\qquad \text{where} \\ &\qquad\quad \text{ft}(\forall(\alpha \geq \forall Q.\rho).\varphi) = \text{ft}(\forall Q.[\alpha ::= \rho]\varphi) \end{aligned}$$

$$\text{ft}(x) \qquad\qquad = x$$

This will only be done at the very end, and is done to remove superfluous bindings such as the one seen in the *id* example and to simplify the type we return to the user. The resulting type has the added benefit of also being in normal form.

While this isn't strictly needed, the alternative would be to just use the classical desugaring method, but then we'd have to complicate lookups of bindings in the Prefix and also the removal of a binding;

Given a Prefix $(a \geq (b \geq \bot).b \to b)$ we end up at the situation where if we were to lookup a, we'd find a flexible type which is bound by a variable b. We continue using this result until at somepoint we lookup what b is supposed to be and we remove it. But removing b in this case would also mean removing a and any parent of

2.5 Normal Form Checks

In *unifyScheme* and *subsume* there are checks to see whether the types given as an argument to the functions are in normal form. This is now no longer the case. Because the types are now more recursive than before they are now no longer always in normal form, for instance $(a \geq \bot).a$ whose normal form is $\bot$ is not a valid argument to that function.

We now relax these restriction and allow any type to be passed to the functions, but because these functions *compare* types we now perform explicitly convert the types to normal forms instead. This keeps these functions simple. If we were to not force the types to normal form we would have to perform extra lookups. Instantiation of b and $(a \geq \bot).a$ would have to lookup what a is before determening that a is bound by $\bot$ and is thus fully polymorphic, in which case instantiation of b is complete.

3 Algorithm in terms of EH

The algorithm detailed by the HML paper was given in terms of a very small λ-calculus which includes Let bindings. In order to implement it had to be *scaled* up in terms of EH.

References

[1] Utrecht Haskell Compiler Structure *Structure Documentation* `http://www.cs.uu.nl/wiki/bin/view/Ehc/EhcStructureDocumentation`

[2] UUAGC *"Utrecht Attribute Grammar System"* `http://www.cs.uu.nl/wiki/bin/view/HUT/AttributeGrammarSystem`

[3] Arie Middelkoop, Atze Dijkstra, S.Doatse Swierstra *Iterative Type Inference with Attribute Grammars* Utrecht University

[4] Arie Middelkoop, Atze Dijkstra, S.Doatse Swierstra, Lucilia Camarao de Figueiredo *Controlling Non-Determinism in Type rules using First-Class Guessing* Utrecht University, Universidade Federal de Ouro Preto

[5] Atze Dijkstra, S. Doaitse Swierstra *Ruler: Programming Type Rules* LNCS FLOPS 2006 proceedings of FLOPS 2006

[6] Arie Middelkoop, Atze Dijkstra, S.Doatse Swierstra *Factoring Type Rules Into Aspects* Utrecht University

[7] Atze Dijkstra *Stepping through Haskell* ISBN 90-393-4070-6

[8] Dimitrios Vytiniotis, Simon Peyton-Jones, Tom Schrijvers, Martin Sulzann *Modular type inference with local assumptions* Microsoft Research, Katholieke Universiteit Leuven, Informatik Consulting Systems AG. Submitted to Journal of Functional Programming.

[9] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Welrich and Mark Shields *Practical type inference for arbitrary-rank types* Microsoft Research, University of Pennsylvania, Microsoft

[10] Dimitrios Vytiniotis, Stephanie Weirich and Simon Peyton Jones *FPH: First-class Polymorphism for Haskell* University of Pennsylvania and Microsoft Research

[11] Daan Leijen *Robust type inference for first-class polymorphism* Microsoft Research

[12] Martin Odersky and Konstantin Laufer *Putting Type Annotations to Work* University of Karlsruhe and Loyola University Chicago

[13] Benjamin C. Pierce and David N. Turner *Local Type Inference* Indiana University and Technology Transfer Center