Compositional Type Checking

Gergő Érdi
http://gergo.erdi.hu/

Haskell.SG, July 2016.

The Hindley-Milner type system

Hindley-Milner type system: Syntax

$$
\begin{array}{lcl}
\langle \textit{term} \rangle & ::= & \langle \textit{var} \rangle \\
& | & \langle \textit{term} \rangle \ \langle \textit{term} \rangle \\
& | & `\lambda\text{'} \ \langle \textit{var} \rangle \ `\mapsto\text{'} \ \langle \textit{term} \rangle \\
& | & `\textbf{let}\text{'} \ \langle \textit{definition} \rangle \ \ldots \ \langle \textit{definition} \rangle \ `\textbf{in}\text{'} \ \langle \textit{term} \rangle
\end{array}
$$

$$
\begin{array}{lcl}
\langle \textit{var} \rangle & ::= & `x\text{'} \ | \ \ldots \\
\langle \textit{definition} \rangle & ::= & \langle \textit{var} \rangle \ `=\text{'} \ \langle \textit{term} \rangle
\end{array}
$$

Hindley-Milner type system: Syntax

$$
\begin{array}{lcl}
\langle \textit{term} \rangle & ::= & \langle \textit{var} \rangle \\
 & | & \langle \textit{term} \rangle\ \langle \textit{term} \rangle \\
 & | & `\lambda\text{'}\ \langle \textit{var} \rangle\ `\mapsto\text{'}\ \langle \textit{term} \rangle \\
 & | & `\textbf{let}\text{'}\ \langle \textit{definition} \rangle\ \ldots\ \langle \textit{definition} \rangle\ `\textbf{in}\text{'}\ \langle \textit{term} \rangle \\
 & | & \langle \textit{data-con} \rangle \\
 & | & `\textbf{case}\text{'}\ \langle \textit{term} \rangle\ `\textbf{of}\text{'}\ \langle \textit{alternative} \rangle\ \ldots\ \langle \textit{alternative} \rangle \\
 & & \\
\langle \textit{var} \rangle & ::= & `x\text{'}\ |\ \ldots \\
 & & \\
\langle \textit{definition} \rangle & ::= & \langle \textit{var} \rangle\ `=\text{'}\ \langle \textit{term} \rangle \\
 & & \\
\langle \textit{data-con} \rangle & ::= & `K\text{'}\ |\ \ldots \\
 & & \\
\langle \textit{alternative} \rangle & ::= & \langle \textit{pat} \rangle\ `\mapsto\text{'}\ \langle \textit{term} \rangle \\
 & & \\
\langle \textit{pat} \rangle & ::= & \langle \textit{data-con} \rangle\ \langle \textit{pat} \rangle\ \ldots\ \langle \textit{pat} \rangle \\
 & | & \langle \textit{var} \rangle\ |\ `\_\text{'}
\end{array}
$$

Hindley-Milner type system: Types

$$
\begin{array}{lcl}
\langle \sigma\textit{-type} \rangle & ::= & `\forall\text{'}\ \langle \textit{ty-var} \rangle \ \ldots \ \langle \textit{ty-var} \rangle \ `\texttt{.}\text{'}\ \langle \tau\textit{-type} \rangle \\
\langle \tau\textit{-type} \rangle & ::= & \langle \textit{ty-var} \rangle \\
 & | & \langle \tau\textit{-type} \rangle \ `\to\text{'}\ \langle \tau\textit{-type} \rangle \\
\langle \textit{ty-var} \rangle & ::= & `\alpha\text{'} \mid \ldots
\end{array}
$$

Hindley-Milner type system: Types

$$
\begin{array}{rcl}
\langle \sigma\textit{-type} \rangle & ::= & `\forall\text{'}\ \langle \textit{ty-var} \rangle \ldots \langle \textit{ty-var} \rangle\ `\mathtt{.}\text{'}\ \langle \tau\textit{-type} \rangle \\[1em]
\langle \tau\textit{-type} \rangle & ::= & \langle \textit{ty-var} \rangle \\
 & | & \langle \tau\textit{-type} \rangle\ `\!\rightarrow\!\text{'}\ \langle \tau\textit{-type} \rangle \\
 & | & \langle \textit{ty-con} \rangle\ \langle \tau\textit{-type} \rangle \ldots \langle \tau\textit{-type} \rangle \\[1em]
\langle \textit{ty-var} \rangle & ::= & `\alpha\text{'} \mid \ldots \\[1em]
\langle \textit{ty-con} \rangle & ::= & `T\text{'} \mid \ldots
\end{array}
$$

Hindley-Milner type system: Derivation rules

$$\frac{x :: \sigma \in \Gamma \qquad \tau \in \mathit{Inst}(\sigma)}{\Gamma \vdash x :: \tau} \quad (\textsc{Var})$$

$$\frac{\Gamma \vdash F :: \tau_1 \to \tau_2 \qquad \Gamma \vdash E :: \tau_1}{\Gamma \vdash F\ E :: \tau_2} \quad (\textsc{App})$$

$$\frac{\Gamma, x :: \tau_1 \vdash E :: \tau_2}{\Gamma \vdash \lambda x \mapsto E :: \tau_1 \to \tau_2} \quad (\textsc{Lam})$$

$$\frac{\Gamma, x :: \tau_0 \vdash E_0 :: \tau_0 \qquad \sigma = \mathit{Gen}(\Gamma, \tau_0) \qquad \Gamma, x :: \sigma \vdash E :: \tau}{\Gamma \vdash \mathbf{let}\ x = E_0\ \mathbf{in}\ E :: \tau} \quad (\textsc{Let})$$

Hindley-Milner type system: Derivation rules

$$\frac{x :: \sigma \in \Gamma \qquad \tau \in \mathit{Inst}(\sigma)}{\Gamma \vdash x :: \tau} \quad (\textsc{Var})$$

$$\frac{\Gamma \vdash F :: \tau_1 \to \tau_2 \qquad \Gamma \vdash E :: \tau_1}{\Gamma \vdash F\,E :: \tau_2} \quad (\textsc{App})$$

$$\frac{\Gamma, x :: \tau_1 \vdash E :: \tau_2}{\Gamma \vdash \lambda x \mapsto E :: \tau_1 \to \tau_2} \quad (\textsc{Lam})$$

$$\frac{\Gamma, x :: \tau_0 \vdash E_0 :: \tau_0 \qquad \sigma = \mathit{Gen}(\Gamma, \tau_0) \qquad \Gamma, x :: \sigma \vdash E :: \tau}{\Gamma \vdash \mathbf{let}\ x = E_0\ \mathbf{in}\ E :: \tau} \quad (\textsc{Let})$$

τ in $\textsc{Var}$?
τ_1 in $\textsc{Lam}$?
τ_1 in $\textsc{Let}$?

HM type inference algorithms

$\mathcal{W}$

$\mathcal{W}(\Gamma, E) = (\Sigma, \tau)$

where

Γ	:	a type context, mapping variables to types
E	:	the expression whose type we are to infer
Σ	:	a substitution, mapping type variables to types
τ	:	the inferred type of E

HM type inference algorithms

$\mathcal{W}$

$\mathcal{W}(\Gamma, E) = (\Sigma, \tau)$

where

Γ	:	a type context, mapping variables to types
E	:	the expression whose type we are to infer
Σ	:	a substitution, mapping type variables to types
τ	:	the inferred type of E

$\mathcal{M}$

$\mathcal{M}(\Gamma, E, \tau) = \Sigma$

where

Γ	:	a type context, mapping variables to types
E	:	the expression to typecheck
τ	:	the expected type of E
Σ	:	a substitution, mapping type variables to types

Hindley-Milner is linear

Linearity

$\mathcal{W}$ for application

$\mathcal{W}(\Gamma, E\ F) = (\Sigma \circ \Sigma_2 \circ \Sigma_1, \Sigma\beta)$

where

$$\begin{aligned}
(\Sigma_1, \tau_1) &= \mathcal{W}(\Gamma, E) \\
(\Sigma_2, \tau_2) &= \mathcal{W}(\Sigma_1\Gamma, F) \\
\Sigma &= \mathcal{U}(\Sigma_2\tau_1 \sim \tau_2 \to \beta)
\end{aligned}$$

β fresh

E F

Linearity

$\mathcal{W}$ for application

$\mathcal{W}(\Gamma, E\ F) = (\Sigma \circ \Sigma_2 \circ \Sigma_1, \Sigma\beta)$

where

$$
\begin{aligned}
(\Sigma_1, \tau_1) &= \mathcal{W}(\Gamma, E) \\
(\Sigma_2, \tau_2) &= \mathcal{W}(\Sigma_1\Gamma, F) \\
\Sigma &= \mathcal{U}(\Sigma_2\tau_1 \sim \tau_2 \to \beta)
\end{aligned}
$$

β fresh

Linearity

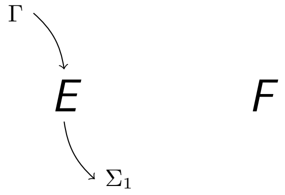
$\mathcal{W}$ for application

$\mathcal{W}(\Gamma, E\ F) = (\Sigma \circ \Sigma_2 \circ \Sigma_1, \Sigma\beta)$

where

$$
\begin{aligned}
(\Sigma_1, \tau_1) &= \mathcal{W}(\Gamma, E) \\
(\Sigma_2, \tau_2) &= \mathcal{W}(\Sigma_1\Gamma, F) \\
\Sigma &= \mathcal{U}(\Sigma_2\tau_1 \sim \tau_2 \to \beta)
\end{aligned}
$$

β fresh

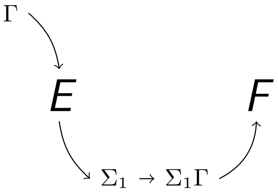


Linearity

$\mathcal{W}$ for application

$\mathcal{W}(\Gamma, E\ F) = (\Sigma \circ \Sigma_2 \circ \Sigma_1, \Sigma\beta)$
where

$(\Sigma_1, \tau_1) \quad = \mathcal{W}(\Gamma, E)$
$(\Sigma_2, \tau_2) \quad = \mathcal{W}(\Sigma_1\Gamma, F)$
$\Sigma \quad = \mathcal{U}(\Sigma_2\tau_1 \sim \tau_2 \to \beta)$
β fresh



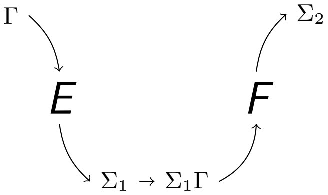
Linearity

$\mathcal{W}$ for application

$\mathcal{W}(\Gamma, E\ F) = (\Sigma \circ \Sigma_2 \circ \Sigma_1, \Sigma\beta)$
where

$$\begin{aligned}
(\Sigma_1, \tau_1) &= \mathcal{W}(\Gamma, E) \\
(\Sigma_2, \tau_2) &= \mathcal{W}(\Sigma_1\Gamma, F) \\
\Sigma &= \mathcal{U}(\Sigma_2\tau_1 \sim \tau_2 \to \beta)
\end{aligned}$$

β fresh



Error messages

Input

```
isJust :: Maybe a -> Bool
not :: Bool -> Bool
foo x = (isJust x, not x)
```

Error messages

Input

```
isJust :: Maybe a -> Bool
not :: Bool -> Bool
foo x = (isJust x, not x)
```

Output from GHC (7.10.3)

```
foo.hs:1:24:
    Couldn't match expected type `Bool'
           with actual type `Maybe a'
    In the first argument of `not', namely `x'
    In the expression: not x
```

Error messages

Input

```
isJust :: Maybe a -> Bool
not :: Bool -> Bool
foo x = (isJust x, not x)
```

Output from GHC (7.10.3)

```
foo.hs:1:24:
    Couldn't match expected type `Bool'
            with actual type `Maybe a'
    In the first argument of `not', namely `x'
    In the expression: not x
```

Error messages

Input

```
isJust :: Maybe a -> Bool
not :: Bool -> Bool
foo x = (isJust x, not x)
```

Output from Hugs 98 (September 2006)

```
ERROR "foo.hs":1 - Type error in application
*** Expression     : isJust x
*** Term           : x
*** Type           : Bool
*** Does not match : Maybe a
```

Error messages

Input

```
isJust :: Maybe a -> Bool
not :: Bool -> Bool
foo x = (isJust x, not x)
```

Output from Hugs 98 (September 2006)

```
ERROR "foo.hs":1 - Type error in application
*** Expression     : isJust x
*** Term           : x
*** Type           : Bool
*** Does not match : Maybe a
```

Error messages

Input

```
isJust :: Maybe a -> Bool
not :: Bool -> Bool
foo x = (isJust x, not x)
```

So where *is* the error?

A compositional type system for HM

Typings

- To implement a compositional type system with the same behaviour as HM, we need to track more intermediate results than just the types of subexpressions
- The context of a variable occurrence can affect the type of some encolsing scope

```
foo x = (isJust x, not x)
```

Typings

- To implement a compositional type system with the same behaviour as HM, we need to track more intermediate results than just the types of subexpressions
- The context of a variable occurrence can affect the type of some encolsing scope

```
foo x = (isJust x, not x)
```

$isJust\ x :: Bool$

$x :: Maybe\ \alpha$

Typings

- To implement a compositional type system with the same behaviour as HM, we need to track more intermediate results than just the types of subexpressions
- The context of a variable occurrence can affect the type of some encolsing scope

```
foo x = (isJust x, not x)
```

$\textit{not}\ x :: Bool$

$x :: Bool$

Typings

- To implement a compositional type system with the same behaviour as HM, we need to track more intermediate results than just the types of subexpressions
- The context of a variable occurrence can affect the type of some encolsing scope

```
foo x = (isJust x, not x)
```

$$\begin{array}{lcl} \mathit{isJust}\; x :: \mathit{Bool} & & \textsf{not}\; x :: \mathit{Bool} \\ x :: \mathit{Maybe}\; \alpha & \Rightarrow\Leftarrow & x :: \mathit{Bool} \end{array}$$

Typings

- To implement a compositional type system with the same behaviour as HM, we need to track more intermediate results than just the types of subexpressions
- The context of a variable occurrence can affect the type of some encolsing scope

  ```
  foo x = (isJust x, not x)
  ```

  $$\begin{array}{lcl} \mathit{isJust}\ x :: \mathit{Bool} & & \textsf{\textit{not}}\ x :: \mathit{Bool} \\ x :: \mathit{Maybe}\ \alpha & \Rightarrow\Leftarrow & x :: \mathit{Bool} \end{array}$$

- So we will assign to subexpresisons, instead of *types*, something called *typings*:
  $\mathit{isJust}\ x :: \{x :: \mathit{Maybe}\ \alpha\} \vdash \mathit{Bool}$
  $\textsf{\textit{not}}\ x :: \{x :: \mathit{Bool}\} \vdash \mathit{Bool}$

Compositional derivation rules

$$\frac{(x :: \Delta_0 \vdash \tau_0) \in \Gamma \qquad \Delta \vdash \tau = \mathit{Freshen}(\Delta_0 \vdash \tau_0)}{\Gamma \vdash x :: \Delta \vdash \tau} \quad (\textsc{Var})$$

$$\frac{\begin{array}{ll}\Gamma, (x :: \{x :: \alpha\} \vdash \alpha) \vdash E :: \Delta \vdash \tau_2 & \alpha \text{ fresh} \\ (x :: \tau_1) \in \Delta \vee (x \notin \Delta \wedge \tau_1 = \alpha) & \end{array}}{\Gamma \vdash \lambda x \mapsto E :: \Delta \backslash x \vdash \tau_1 \to \tau_2} \quad (\textsc{Lam})$$

$$\frac{\begin{array}{c}\Gamma \vdash F :: \Delta_1 \vdash \tau_1 \\ \Gamma \vdash E :: \Delta_2 \vdash \tau_2\end{array}}{\Gamma \vdash F\, E :: \Delta \vdash \tau} \quad (\textsc{App})$$

where α fresh

$(\Delta, \Sigma) = \mathcal{U}(\Delta_1, \Delta_2, \tau_1 \sim \tau_2 \to \alpha)$

$\tau = \Sigma\alpha$

Compositional derivation rules: **let**

$$\frac{\begin{array}{lll}\Gamma,(x :: \{x :: \alpha\} \vdash \alpha) & \vdash E_0 :: \Delta_0 \vdash \tau_0 & \alpha \text{ fresh} \\ \Gamma,(x :: \Delta_0'' \vdash \Sigma_0\tau_0) & \vdash E :: \Delta \vdash \tau & \end{array}}{\Gamma \vdash \mathbf{let}\ x = E_0\ \mathbf{in}\ E :: \Delta' \vdash \Sigma\tau}\ (\textsc{Let})$$

where
$$\begin{aligned}(\Delta_0', \Sigma_0) &= \mathcal{U}(\Delta_0, \tau_0 \sim \Delta_0(x)) \\ \Delta_0'' &= \Delta_0' \backslash x \\ (\Delta', \Sigma) &= \mathcal{U}(\Delta_0'', \Delta)\end{aligned}$$

Where is $\mathbf{let}$-polymorphism?

- If $(x :: \Delta_0 \vdash \tau_0) \in \Gamma$, then x is polymorphic iff $x \notin \Delta_0$:

  $$\frac{(x :: \Delta_0 \vdash \tau_0) \in \Gamma \qquad \Delta \vdash \tau \in \mathit{Freshen}(\Delta_0 \vdash \tau_0)}{\Gamma \vdash x :: \Delta \vdash \tau}$$

  This results in two occurrences of x to yield a constraint that their types match only if $x \in \Delta$ ($\Leftrightarrow x \in \Delta_0$)
- $\lambda x \mapsto E$ introduces $x :: \{x :: \alpha\} \vdash \alpha$ to Γ, i.e. x *is monomorphic*
- $\mathbf{let}\ x = E_0\ \mathbf{in}\ E$ introduces $x :: \Delta \vdash \tau$ to Γ after removing x from the typing of E_0, i.e. x *is polymorphic in* E

Implementation

Implementation: `hm-compo`

Both linear and compositional type checking implemented for our model language:

- Concrete syntax (parser & pretty printer)
  - Indentation-based parsing is a nightmare
  - `haskell-src-exts` to the rescue!
- `unification-fd`-based representation
  - Immediate rewriting of type-meta-variables: no delayed occurs checks
  - Explicit zonking

$$
\begin{aligned}
&\mathbf{class}\ (Unifiable\ t, Variable\ v, Monad\ m) \Rightarrow MonadTC\ t\ v\ m \\
&\qquad\quad \mid m\ t \to v, m\ v \to t\ \mathbf{where} \\
&\quad freshVar :: m\ v \\
&\quad readVar\ :: v \to m\ (Maybe\ (UTerm\ t\ v)) \\
&\quad writeVar :: v \to UTerm\ t\ v \to m\ ()
\end{aligned}
$$

$$
\begin{aligned}
&zonk :: (Traversable\ t, MonadTC\ t\ v\ m) \\
&\qquad \Rightarrow UTerm\ t\ v \to m\ (UTerm\ t\ v)
\end{aligned}
$$

Implementation: `hm-compo`

Both linear and compositional type checking implemented for our model language:

- Code mostly shared between the two typecheckers

$$\textbf{data}\ \mathit{TC}\ \mathit{ctx}\ \mathit{err}\ s\ \mathit{loc}\ a$$

$$\begin{array}{l}\textbf{instance}\ \mathit{MonadReader}\ \mathit{ctx}\ (\mathit{TC}\ \mathit{ctx}\ \mathit{err}\ s\ \mathit{loc}) \\ \textbf{instance}\ \mathit{MonadError}\ \mathit{err}\ (\mathit{TC}\ \mathit{ctx}\ \mathit{err}\ s\ \mathit{loc}) \\ \textbf{instance}\ \mathit{MonadTC}\ \mathit{Ty0}\ (\mathit{MVar}\ s)\ (\mathit{TC}\ \mathit{ctx}\ \mathit{err}\ s\ \mathit{loc})\end{array}$$

$$\mathit{freshTVar} :: \mathit{TC}\ \mathit{ctx}\ \mathit{err}\ s\ \mathit{loc}\ \mathit{TVar}$$

- Representation of Γ is different: there are no σ-types in the compositional type system.

Demo time

Motivating example

Input

```
isJust :: Maybe a -> Bool
not :: Bool -> Bool
foo x = MkPair (isJust x) (not x)
```

Output of `hm-compo`

```
demo/pair.hm (13,8):
    MkPair (not x) (isJust x)

Cannot unify 'Bool' with 'Maybe a' when unifying 'x':
Cannot unify 'Bool' with 'Maybe a' in the following context:
        MkPair (not x)              isJust x
        Bool → Pair Bool Bool       Bool
x ::    Bool                        Maybe a
```

Types agree for well-typed terms

```
id :: a → a
const :: a → b → a
fix :: (a → a) → a
flip :: (a → b → c) → b → a → c
foldr :: (a → b → b) → b → List a → b
map :: (a → b) → List a → List b
undefined :: a
undefined1 :: a
undefined2 :: a
```

For further information

- *Compositional Explanation of Types and Algorithmic Debugging of Type Errors*, Olaf Chitil (2001)
- *Compositional Type Checking for Hindley-Milner Type Systems with* Ad-hoc *Polymorphism*, Gergő Érdi (2011)