Compositional Type Checking
for Hindley-Milner Type Systems with Ad-hoc Polymorphism

Dr. Gergő Érdi
http://gergo.erdhi.hu/
Supervised by: Péter Diviánszky

Budapest, January 24, 2011.
Typing $\lambda$ calculus with Hindley-Milner

### Syntax

| Expression: $E$ | $=$ | $v$
|-----------------|-----|-----|
|                 |     | $E \ E$
|                 |     | $\lambda v \mapsto E$

| Variable: $v$ | $=$ | $f \ | \ x \ | \ y \ | \ldots$

$(x :: \tau) \in \Gamma$  
$\vdash x :: \tau$  
$\vdash E :: \tau'$  
$\vdash F :: \tau'$  
$(\text{App}) \Gamma \vdash E \ F :: \tau$  
$\Gamma; (x :: \tau') \vdash E :: \tau'$  
$(\text{Abs}) \Gamma \vdash \lambda x \mapsto E :: \tau'$
Typing $\lambda$ calculus with Hindley-Milner

Syntax

Expression: $E =$  

$\mid v$

$\mid EE$

$\mid \lambda v \mapsto E$

Variable: $v =$  

$f \mid x \mid y \mid \ldots$

\[
\frac{(x :: \tau) \in \Gamma}{\Gamma \vdash x :: \tau} \quad \text{(MONOVAR)}
\]

\[
\frac{\Gamma \vdash E :: \tau' \rightarrow \tau \quad \Gamma \vdash F :: \tau'}{\Gamma \vdash EF :: \tau} \quad \text{(APP)}
\]

\[
\frac{\Gamma; (x :: \tau') \vdash E :: \tau}{\Gamma \vdash \lambda x \mapsto E :: \tau' \rightarrow \tau} \quad \text{(ABS)}
\]
Type inference algorithms

\[ \mathcal{W} \]

\[ \mathcal{W}(\Gamma, E) = (\Psi, \tau) \]

where

- \( \Gamma \): a type context, mapping variables to types
- \( E \): the expression whose type we are to infer
- \( \Psi \): a substitution, mapping type variables to types
- \( \tau \): the inferred type of \( E \)
Type inference algorithms

\[ \mathcal{W} \]
\[ \mathcal{W}(\Gamma, E) = (\Psi, \tau) \]
where
- \( \Gamma \): a type context, mapping variables to types
- \( E \): the expression whose type we are to infer
- \( \Psi \): a substitution, mapping type variables to types
- \( \tau \): the inferred type of \( E \)

\[ \mathcal{M} \]
\[ \mathcal{M}(\Gamma, E, \tau) = \Psi \]
where
- \( \Gamma \): a type context, mapping variables to types
- \( E \): the expression to typecheck
- \( \tau \): the expected type of \( E \)
- \( \Psi \): a substitution, mapping type variables to types
\[ \mathcal{W}(\Gamma, E F) = (\Psi \circ \Psi_2 \circ \Psi_1, \Psi \beta) \]

where
\[
(\Psi_1, \tau_1) = \mathcal{W}(\Gamma, E) \\
(\Psi_2, \tau_2) = \mathcal{W}(\Psi_1 \Gamma, F) \\
\Psi = \mathcal{U}(\Psi_2 \tau_1 \sim \tau_2 \rightarrow \beta)
\]

\(\beta\) new
\[ \mathcal{W}(\Gamma, E F) = (\psi \circ \psi_2 \circ \psi_1, \psi \beta) \]

where

\[ (\psi_1, \tau_1) = \mathcal{W}(\Gamma, E) \]
\[ (\psi_2, \tau_2) = \mathcal{W}(\psi_1 \Gamma, F) \]
\[ \psi = \mathcal{U}(\psi_2 \tau_1 \sim \tau_2 \rightarrow \beta) \]

\( \beta \) new

\[ \Gamma \]

\[ E \quad F \]
\[ \mathcal{W}(\Gamma, E F) = (\Psi \circ \Psi_2 \circ \Psi_1, \Psi_\beta) \]

where

\[
\begin{align*}
(\Psi_1, \tau_1) &= \mathcal{W}(\Gamma, E) \\
(\Psi_2, \tau_2) &= \mathcal{W}(\Psi_1 \Gamma, F) \\
\Psi &= \mathcal{U}(\Psi_2 \tau_1 \sim \tau_2 \rightarrow \beta) \\
\beta &= \text{new}
\end{align*}
\]
\( \mathcal{W} \) for application

\[
\mathcal{W}(\Gamma, E F) = (\Psi \circ \Psi_2 \circ \Psi_1, \Psi \beta)
\]

where

\[
(\Psi_1, \tau_1) = \mathcal{W}(\Gamma, E)
\]

\[
(\Psi_2, \tau_2) = \mathcal{W}(\Psi_1 \Gamma, F)
\]

\[
\Psi = \mathcal{U}(\Psi_2 \tau_1 \sim \tau_2 \rightarrow \beta)
\]

\( \beta \) new
Linearity

\[ \mathcal{W}(\Gamma, EF) = (\psi \circ \psi_2 \circ \psi_1, \psi_\beta) \]

where

\[ (\psi_1, \tau_1) = \mathcal{W}(\Gamma, E) \]
\[ (\psi_2, \tau_2) = \mathcal{W}(\psi_1 \Gamma, F) \]
\[ \psi = \mathcal{U}(\psi_2 \tau_1 \sim \tau_2 \rightarrow \beta) \]

\( \beta \) new
Error messages from \( \mathcal{W} \)

**Input**

\[
\begin{align*}
toUpper & :: \text{Char} \rightarrow \text{Char} \\
not & :: \text{Bool} \rightarrow \text{Bool} \\
foo \ x & = (\text{toUpper} \ x, \not \ x)
\end{align*}
\]
Input

toUpper :: Char -> Char
not :: Bool -> Bool
foo x = (toUpper x, not x)

Output from GHC 6.12

foo.hs:1:24:
  Couldn’t match expected type ‘Bool’
    against inferred type ‘Char’
  In the first argument of ‘not’, namely ‘x’
  In the expression: not x
  In the expression: (toUpper x, not x)
Error messages from \( \mathcal{W} \)

Input

toUpper :: Char -> Char
not :: Bool -> Bool
foo x = (toUpper x, not x)

Output from GHC 6.12

\texttt{foo.hs:1:24:}
Couldn’t match expected type ‘Bool’
against inferred type ‘Char’
In the first argument of ‘not’, namely ‘x’
In the expression: \texttt{not x}
In the expression: (toUpper x, not x)
Error messages from \texttt{W}

Input

toupper :: Char -> Char
not :: Bool -> Bool
foo x = (toupper x, not x)

Output from Hugs 98

\texttt{ERROR "foo.hs":1 - Type error in application}
*** Expression : toupper x
*** Term : x
*** Type : Bool
*** Does not match : Char
Input

toUpper :: Char -> Char
not :: Bool -> Bool
foo x = (toUpper x, not x)

Output from Hugs 98

ERROR "foo.hs":1 - Type error in application
*** Expression : toUpper x
*** Term : x
*** Type : Bool
*** Does not match : Char
Error messages from \texttt{W}

Input

toUpper :: Char -> Char
not :: Bool -> Bool
foo x = (toUpper x, not x)

So where \textit{is} the error?
Typing λ calculus compositionally

\[
\frac{x \notin \text{dom } \Gamma \quad \alpha \text{ new}}{\Gamma; \{x :: \alpha\} \vdash x :: \alpha} \quad (\text{MONOVAR})
\]

\[
\frac{\Gamma; \Delta_1 \vdash E :: \tau' \quad \Gamma; \Delta_2 \vdash F :: \tau''}{\Gamma; \Delta \vdash E \ F :: \tau} \quad (\text{APP})
\]

where \( \alpha \text{ new} \)

\[
\psi = \mathcal{U}(\{\Delta_1, \Delta_2\}, \{\tau' \sim \tau'' \to \alpha\})
\]

\[
\Delta = \psi \Delta_1 \cup \psi \Delta_2
\]

\[
\tau = \psi \alpha
\]

\[
\frac{\Gamma; \Delta \vdash E :: \tau \quad (x :: \tau') \in \Delta}{\Gamma; \Delta \setminus x \vdash \lambda x \mapsto E :: \tau' \to \tau} \quad (\text{ABS})
\]
Typing $\lambda$ calculus compositionally

\[
\frac{
  x \notin \text{dom } \Gamma
}{
  \Gamma; \{x :: \alpha\} \vdash x :: \alpha
} \quad \text{(MONOVAR)}
\]

\[
\frac{
  \Gamma; \Delta_1 \vdash E :: \tau' \\
  \Gamma; \Delta_2 \vdash F :: \tau''
}{
  \Gamma; \Delta \vdash E \ F :: \tau
} \quad \text{(APP)}
\]

\[
\frac{
  \Gamma; \Delta \vdash E :: \tau \\
  (x :: \tau') \in \Delta
}{
  \Gamma; \Delta \setminus x \vdash \lambda x \mapsto E :: \tau' \rightarrow \tau
} \quad \text{(ABS)}
\]

Not just an inference system, but also an algorithm:

\[
C(\Gamma, E) = \Delta \vdash \tau
\]

where

- $\Gamma$: a type context, mapping variables to types
- $E$: the expression whose type we are to infer
- $\Delta$: a typing environment, mapping type variables to types
- $\tau$: the inferred type of $E$, provided $\Delta$ holds
Not linear, compositional!

C for application

\[
\begin{align*}
\frac{
\Gamma; \Delta_1 \vdash E :: \tau_1 & \quad \Gamma; \Delta_2 \vdash F :: \tau_2
}{
\Gamma; \Delta \vdash E \ F :: \tau}
\end{align*}
\]  

(APP)

where \( \Psi = \mathcal{U}(\{\Delta_1, \Delta_2\}, \{\tau_1 \sim \tau_2 \to \alpha\}) \)

\( \Delta = \Psi \Delta_1 \cup \Psi \Delta_2 \)

\( \tau = \Psi \alpha \)

\( E \) \hspace{1cm} \( F \)
Not linear, compositional!

C for application

\[
\frac{\Gamma; \Delta_1 \vdash E :: \tau_1 \quad \Gamma; \Delta_2 \vdash F :: \tau_2}{\Gamma; \Delta \vdash E \ F :: \tau}
\]  \hspace{1cm}  (\text{APP})

where \( \Psi = \mathcal{U}(\{\Delta_1,\Delta_2\},\{\tau_1 \sim \tau_2 \rightarrow \alpha\}) \)

\( \Delta = \Psi \Delta_1 \cup \Psi \Delta_2 \)

\( \tau = \Psi \alpha \)
Not linear, compositional!

C for application

\[
\frac{\Gamma; \Delta_1 \vdash E :: \tau_1 \quad \Gamma; \Delta_2 \vdash F :: \tau_2}{\Gamma; \Delta \vdash E \ F :: \tau}
\]  \hspace{1cm} \text{(APP)}

where \( \Psi = \mathcal{U}(\{\Delta_1, \Delta_2\}, \{\tau_1 \sim \tau_2 \to \alpha\}) \)
\[
\Delta = \Psi \Delta_1 \cup \Psi \Delta_2
\]
\[
\tau = \Psi \alpha
\]
Not linear, compositional!

\[ \frac{\Gamma; \Delta_1 \vdash E :: \tau_1 \quad \Gamma; \Delta_2 \vdash F :: \tau_2}{\Gamma; \Delta \vdash E \ F :: \tau} \quad (\text{APP}) \]

where \( \Psi = \mathcal{U}(\{\Delta_1, \Delta_2\}, \{\tau_1 \sim \tau_2 \rightarrow \alpha\}) \)

\[ \Delta = \Psi \Delta_1 \cup \Psi \Delta_2 \]

\[ \tau = \Psi \alpha \]
toUpper :: Char -> Char
not :: Bool -> Bool
foo x = (toUpper x, not x)

foo.hs:1:8-25:
(toUpper x, not x)
Cannot unify ‘Char’ with ‘Bool’ when unifying ‘x’:
    toUpper x    not x
    Char        Bool
x :: Char    Bool
Errors from C

Input

toupper :: Char -> Char
not :: Bool -> Bool
foo x = (toupper x, not x)

Output from Tandoori

foo.hs:1:8-25:
(toUpper x, not x)
Cannot unify ‘Char’ with ‘Bool’ when unifying ‘x’:
    toUpper x not x
    Char Bool
x :: Char Bool
Input

```
toupper :: Char -> Char
not :: Bool -> Bool
foo x = (toupper x, not x)
```

Output from Tandoori

```
foo.hs:1:8-25:
(toUpper x, not x)
Cannot unify ‘Char’ with ‘Bool’ when unifying ‘x’:
   toUpper x not x
   Char Bool
x :: Char Bool
```
Haskell 98 is more than just $\lambda$ calculus

- Algebraic data types
- Pattern matching
- Let-polymorphism
- Recursive definitions
- Type declarations
- Type class polymorphism
- Record data types
- Do-notation

Accounted for in Olaf Chitil's 2001 paper
Haskell 98 is more than just $\lambda$ calculus

- Algebraic data types
- Pattern matching
- Let-polymorphism
- Recursive definitions
- Type declarations
- Type class polymorphism
- Record data types
- Do-notation

Accounted for in Olaf Chitil’s 2001 paper
Haskell 98 is more than just $\lambda$ calculus

- Algebraic data types
- Pattern matching
- Let-polymorphism
- Recursive definitions

Accounted for in Olaf Chitil’s 2001 paper

- Type declarations
- Type class polymorphism

Our contribution

- Record data types
- Do-notation
Haskell 98 is more than just $\lambda$ calculus

- Algebraic data types
- Pattern matching
- Let-polymorphism
- Recursive definitions
- Type declarations
- Type class polymorphism
- Record data types
- Do-notation

\{ Accounted for in Olaf Chitil’s 2001 paper \\
\{ Our contribution \\
\{ Future work
Ad-hoc polymorphism

Motivating example: equality testing

elem x [] = False
elem x (y:ys) = (x == y) || (elem x ys)
Motivating example: equality testing

\[
\text{elem } x \ [ ] = \text{False} \\
\text{elem } x \ (y:ys) = (x == y) || (\text{elem } x \ ys)
\]

Equality testing has

- the same signature for all types: \( \alpha \to \alpha \to \text{BOOL} \)
**Ad-hoc polymorphism**

Motivating example: equality testing

\[
\text{elem } x \; [] = \text{False} \\
\text{elem } x \; (y:ys) = (x == y) \; || \; (\text{elem } x \; ys)
\]

Equality testing has

- the same signature for all types: \(\alpha \rightarrow \alpha \rightarrow \text{BOOL}\)
- different definition for different types
Ad-hoc polymorphism

Motivating example: equality testing

```
elem x []     = False
elem x (y:ys) = (x == y) || (elem x ys)
```

Equality testing has

- the same signature for all types: \( \alpha \to \alpha \to \text{BOOL} \)
- different definition for different types

Type classes

Ad-hoc polymorphic variables grouped into type classes

Type of \( \text{elem} \): \( \forall \alpha. \text{Eq}\ \alpha \Rightarrow \alpha \to [\alpha] \to \text{BOOL} \)
Ad-hoc polymorphism

Motivating example: equality testing

elem x [] = False
elem x (y:ys) = (x == y) || (elem x ys)

Equality testing has

- the same signature for all types: \( \alpha \rightarrow \alpha \rightarrow \text{BOOL} \)
- different definition for different types

Type classes

Ad-hoc polymorphic variables grouped into type classes

Type of elem: \( \forall \alpha.\text{Eq } \alpha \Rightarrow \alpha \rightarrow [\alpha] \rightarrow \text{BOOL} \)
\( C \) with type classes: \( C^k \)

\[ \frac{x \not\in \text{dom } \Gamma \quad \alpha \text{ new}}{\Gamma; \{x :: \alpha\} \vdash x :: \alpha} \quad \text{(MONOVAR)} \]

\[ \frac{\Gamma; \Delta_1 \vdash E :: \tau' \quad \Gamma; \Delta_2 \vdash F :: \tau''}{\Gamma; \Delta \vdash E F :: \tau} \quad \text{(APP)} \]

where \( \alpha \text{ new} \)

\[ \Psi = \mathcal{U}(\{\Delta_1, \Delta_2\}, \{\tau' \sim \tau'' \rightarrow \alpha\}) \]

\[ \Delta = \Psi\Delta_1 \cup \Psi\Delta_2 \]

\[ \tau = \Psi\alpha \]

\[ \frac{\Gamma; \Delta \vdash E :: \tau \quad (x :: \tau') \in \Delta}{\Gamma; \Delta \setminus x \vdash \lambda x \mapsto E :: \tau' \rightarrow \tau} \quad \text{(ABS)} \]
C with type classes: \( C^\kappa \)

\[
\frac{x \not\in \text{dom } \Gamma \quad \alpha \text{ new}}{
\Gamma; \{x :: \alpha\}; \emptyset \vdash x :: \alpha} \quad (\text{MONOVAR})
\]

\[
\frac{
\Gamma; \Delta_1; \Theta_1 \vdash E :: \tau' \quad \Gamma; \Delta_2; \Theta_2 \vdash F :: \tau''
}{
\Gamma; \Delta; \Theta \vdash EF :: \tau}
\quad (\text{APP})
\]

where \( \alpha \text{ new} \)
\[
\Psi = \mathcal{U}(\{\Delta_1, \Delta_2\}, \{\tau' \sim \tau'' \rightarrow \alpha\})
\]
\[
\Delta = \Psi \Delta_1 \cup \Psi \Delta_2
\]
\[
\Theta = \Psi \Theta_1 + \Psi \Theta_2
\]
\[
\tau = \Psi \alpha
\]

\[
\frac{
\Gamma; \Delta; \Theta \vdash E :: \tau \quad (x :: \tau') \in \Delta
}{
\Gamma; \Delta \setminus x; \Theta \vdash \lambda x \mapsto E :: \tau' \rightarrow \tau}
\quad (\text{ABS})
\]
Tandoori is the implementation of $C^\kappa$ for a reasonable subset of Haskell 98

- Based on GHC 6.12’s parser and renamer front-end
- Get it from [http://gergo.erdih.hu/projects/tandoori/](http://gergo.erdih.hu/projects/tandoori/), available under a BSD license
Questions?