Compositional Type Checking

for Hindley-Milner Type Systems with *Ad-hoc* Polymorphism

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

Budapest, January 24, 2011.

Typing λ calculus with Hindley-Milner

Syntax

Expression:	E	$=$	v
		$\mid$	$E\ E$
		$\mid$	$\lambda v \mapsto E$
Variable:	v	$=$	$f \mid x \mid y \mid \ldots$

Typing λ calculus with Hindley-Milner

Syntax

$$
\begin{array}{lrcl}
\text{Expression:} & E & = & v \\
 & & | & E\ E \\
 & & | & \lambda v \mapsto E \\
\text{Variable:} & v & = & f \mid x \mid y \mid \ldots
\end{array}
$$

$$\frac{(x :: \tau) \in \Gamma}{\Gamma \vdash x :: \tau} \quad (\textsc{MonoVar})$$

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

$$\frac{\Gamma; (x :: \tau') \vdash E :: \tau}{\Gamma \vdash \lambda x \mapsto E :: \tau' \to \tau} \quad (\textsc{Abs})$$

Type inference algorithms

$\mathcal{W}$

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

where

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

Type inference algorithms

$\mathcal{W}$

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

where

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

$\mathcal{M}$

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

where

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

Linearity

$\mathcal{W}$ for application

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

$$
\begin{aligned}
(\Psi_1, \tau_1) &= \mathcal{W}(\Gamma, E) \\
(\Psi_2, \tau_2) &= \mathcal{W}(\Psi_1\Gamma, F) \\
\Psi &= \mathcal{U}(\Psi_2\tau_1 \sim \tau_2 \rightarrow \beta)
\end{aligned}
$$

β new

$E \qquad\qquad F$

Linearity

$\mathcal{W}$ for application

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

where

$$
\begin{aligned}
(\Psi_1, \tau_1) &= \mathcal{W}(\Gamma, E) \\
(\Psi_2, \tau_2) &= \mathcal{W}(\Psi_1\Gamma, F) \\
\Psi &= \mathcal{U}(\Psi_2\tau_1 \sim \tau_2 \to \beta)
\end{aligned}
$$

β new

Γ

$E \qquad F$

Linearity

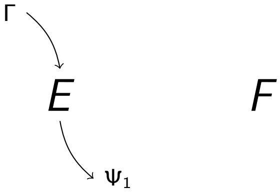
$\mathcal{W}$ for application

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

where

$$
\begin{aligned}
(\Psi_1, \tau_1) &= \mathcal{W}(\Gamma, E) \\
(\Psi_2, \tau_2) &= \mathcal{W}(\Psi_1\Gamma, F) \\
\Psi &= \mathcal{U}(\Psi_2\tau_1 \sim \tau_2 \to \beta)
\end{aligned}
$$

β new



Linearity

$\mathcal{W}$ for application

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

where

$$
\begin{aligned}
(\Psi_1, \tau_1) &= \mathcal{W}(\Gamma, E) \\
(\Psi_2, \tau_2) &= \mathcal{W}(\Psi_1\Gamma, F) \\
\Psi &= \mathcal{U}(\Psi_2\tau_1 \sim \tau_2 \to \beta)
\end{aligned}
$$

β new

$$
\begin{tikzcd}
\Gamma \arrow[d, bend left] & & & \\
E \arrow[dr, bend right] & & & F \\
& \Psi_1 \arrow[r] & \Psi_1\Gamma \arrow[ur, bend right] &
\end{tikzcd}
$$

Linearity

$\mathcal{W}$ for application

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

where

$$\begin{aligned}
(\Psi_1, \tau_1) &= \mathcal{W}(\Gamma, E) \\
(\Psi_2, \tau_2) &= \mathcal{W}(\Psi_1\Gamma, F) \\
\Psi &= \mathcal{U}(\Psi_2\tau_1 \sim \tau_2 \to \beta)
\end{aligned}$$

β new

$$\begin{tikzcd}
\Gamma \arrow[dr, bend left] & & & & & \Psi_2 \\
& E \arrow[dr, bend right] & & & F \arrow[ur, bend left] & \\
& & \Psi_1 \arrow[r] & \Psi_1\Gamma \arrow[ur, bend right] & &
\end{tikzcd}$$

Error messages from $\mathcal{W}$

Input

```
toUpper :: Char -> Char
not :: Bool -> Bool
foo x = (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

```
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

```
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 Hugs 98

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

Error messages from $\mathcal{W}$

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 $\mathcal{W}$

Input

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

So where *is* the error?

Typing λ calculus compositionally

$$\frac{x \notin \mathrm{dom}\,\Gamma \qquad \alpha \text{ new}}{\Gamma; \{x :: \alpha\} \vdash x :: \alpha} \quad (\textsc{MonoVar})$$

$$\frac{\Gamma; \Delta_1 \vdash E :: \tau' \qquad \Gamma; \Delta_2 \vdash F :: \tau''}{\Gamma; \Delta \vdash E\ F :: \tau} \quad (\textsc{App})$$

where α new

$$\Psi = \mathcal{U}(\{\Delta_1, \Delta_2\}, \{\tau' \sim \tau'' \to \alpha\})$$

$$\Delta = \Psi\Delta_1 \cup \Psi\Delta_2$$

$$\tau = \Psi\alpha$$

$$\frac{\Gamma; \Delta \vdash E :: \tau \qquad (x :: \tau') \in \Delta}{\Gamma; \Delta \setminus x \vdash \lambda x \mapsto E :: \tau' \to \tau} \quad (\textsc{Abs})$$

Typing λ calculus compositionally

$$\frac{x \notin \mathrm{dom}\,\Gamma \qquad \alpha \text{ new}}{\Gamma; \{x :: \alpha\} \vdash x :: \alpha} \quad (\textsc{MonoVar})$$

$$\frac{\Gamma; \Delta_1 \vdash E :: \tau' \qquad \Gamma; \Delta_2 \vdash F :: \tau''}{\Gamma; \Delta \vdash E\ F :: \tau} \quad (\textsc{App})$$

$$\frac{\Gamma; \Delta \vdash E :: \tau \qquad (x :: \tau') \in \Delta}{\Gamma; \Delta \setminus x \vdash \lambda x \mapsto E :: \tau' \to \tau} \quad (\textsc{Abs})$$

Not just an inference system, but also an algorithm:

C

$C(\Gamma, E) = \Delta \vdash \tau$

where

Γ:	a type context, mapping variables to types
E:	the expression whose type we are to infer
Δ:	a typing environment, mapping type variables to types
τ:	the inferred type of E, provided Δ holds

Not linear, compositional!

C for application

$$\frac{\Gamma; \Delta_1 \vdash E \ :: \ \tau_1 \qquad \Gamma; \Delta_2 \vdash F \ :: \ \tau_2}{\Gamma; \Delta \vdash E \ F \ :: \ \tau} \quad (\textsc{App})$$

where
$$\begin{aligned} \Psi &= \mathcal{U}(\{\Delta_1, \Delta_2\}, \{\tau_1 \sim \tau_2 \to \alpha\}) \\ \Delta &= \Psi\Delta_1 \cup \Psi\Delta_2 \\ \tau &= \Psi\alpha \end{aligned}$$

E F

Not linear, compositional!

C for application

$$\frac{\Gamma; \Delta_1 \vdash E \;::\; \tau_1 \qquad \Gamma; \Delta_2 \vdash F \;::\; \tau_2}{\Gamma; \Delta \vdash E\ F \;::\; \tau} \quad (\textsc{App})$$

where
$$\begin{aligned} \Psi &= \mathcal{U}(\{\Delta_1, \Delta_2\}, \{\tau_1 \sim \tau_2 \to \alpha\}) \\ \Delta &= \Psi\Delta_1 \cup \Psi\Delta_2 \\ \tau &= \Psi\alpha \end{aligned}$$

$$\begin{tikzcd} & \Gamma \arrow[dl, bend right] \arrow[dr, bend left] & \\ E & & F \end{tikzcd}$$

Not linear, compositional!

C for application

$$\frac{\Gamma; \Delta_1 \vdash E :: \tau_1 \qquad \Gamma; \Delta_2 \vdash F :: \tau_2}{\Gamma; \Delta \vdash E\ F :: \tau} \quad (\textsc{App})$$

where
$$\begin{aligned} \Psi &= \mathcal{U}(\{\Delta_1, \Delta_2\}, \{\tau_1 \sim \tau_2 \to \alpha\}) \\ \Delta &= \Psi\Delta_1 \cup \Psi\Delta_2 \\ \tau &= \Psi\alpha \end{aligned}$$

$$\begin{tikzcd} & \Gamma \arrow[dl, bend right] \arrow[dr, bend left] & \\ E \arrow[d] & & F \arrow[d] \\ \Delta_1 \vdash \tau_1 & & \Delta_2 \vdash \tau_2 \end{tikzcd}$$

Not linear, compositional!

C for application

$$\frac{\Gamma; \Delta_1 \vdash E \ :: \ \tau_1 \qquad \Gamma; \Delta_2 \vdash F \ :: \ \tau_2}{\Gamma; \Delta \vdash E \ F \ :: \ \tau} \quad (\textsc{App})$$

$$\begin{aligned} \text{where} \qquad \Psi &= \mathcal{U}(\{\Delta_1, \Delta_2\}, \{\tau_1 \sim \tau_2 \to \alpha\}) \\ \Delta &= \Psi\Delta_1 \cup \Psi\Delta_2 \\ \tau &= \Psi\alpha \end{aligned}$$

$$\begin{tikzcd}
 & \Gamma \arrow[dl, bend right] \arrow[dr, bend left] & \\
E \arrow[d] & & F \arrow[d] \\
\Delta_1 \vdash \tau_1 \arrow[dr, bend right] & & \Delta_2 \vdash \tau_2 \arrow[dl, bend left] \\
 & \Delta \vdash \tau &
\end{tikzcd}$$

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
```

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
```

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
```

Haskell 98 is more than just λ calculus

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

- Type declarations
- Type class polymorphism

- Record data types
- Do-notation

Haskell 98 is more than just λ calculus

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

Accounted for in Olaf Chitil's 2001 paper

- Type declarations
- Type class polymorphism

- Record data types
- Do-notation

Haskell 98 is more than just λ 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 λ 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

Future work

Ad-hoc polymorphism

Motivating example: equality testing

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

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: $\alpha \to \alpha \to \mathrm{BOOL}$

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: $\alpha \to \alpha \to \mathrm{BOOL}$
- 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: $\alpha \to \alpha \to \textsc{Bool}$
- different definition for different types

Type classes

Ad-hoc polymorphic variables grouped into type classes
Type of `elem`: $\forall \alpha . \mathit{Eq}\ \alpha \Rightarrow \alpha \to [\alpha] \to \textsc{Bool}$

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`: $\forall \alpha . \mathit{Eq}\ \alpha \Rightarrow \alpha \rightarrow [\alpha] \rightarrow \text{BOOL}$

C with type classes: C^κ

$$\frac{x \notin \operatorname{dom}\Gamma \qquad \alpha \text{ new}}{\Gamma; \{x :: \alpha\} \quad \vdash x :: \alpha} \quad (\textsc{MonoVar})$$

$$\frac{\Gamma; \Delta_1 \quad \vdash E :: \tau' \qquad \Gamma; \Delta_2 \quad \vdash F :: \tau''}{\Gamma; \Delta \quad \vdash E\ F :: \tau} \quad (\textsc{App})$$

where $\quad \alpha$ new

$$\Psi = \mathcal{U}(\{\Delta_1, \Delta_2\}, \{\tau' \sim \tau'' \to \alpha\})$$

$$\Delta = \Psi\Delta_1 \cup \Psi\Delta_2$$

$$\tau = \Psi\alpha$$

$$\frac{\Gamma; \Delta \quad \vdash E :: \tau \qquad (x :: \tau') \in \Delta}{\Gamma; \Delta \setminus x \quad \vdash \lambda x \mapsto E :: \tau' \to \tau} \quad (\textsc{Abs})$$

C with type classes: C^κ

$$\frac{x \notin \operatorname{dom}\Gamma \qquad \alpha \text{ new}}{\Gamma; \{x :: \alpha\}; \emptyset \vdash x :: \alpha} \quad (\textsc{MonoVar})$$

$$\frac{\Gamma; \Delta_1; \Theta_1 \vdash E :: \tau' \qquad \Gamma; \Delta_2; \Theta_2 \vdash F :: \tau''}{\Gamma; \Delta; \Theta \vdash E\ F :: \tau} \quad (\textsc{App})$$

where α new

$$\begin{aligned}
\Psi &= \mathcal{U}(\{\Delta_1, \Delta_2\}, \{\tau' \sim \tau'' \to \alpha\}) \\
\Delta &= \Psi\Delta_1 \cup \Psi\Delta_2 \\
\Theta &= \Psi\Theta_1 + \Psi\Theta_2 \\
\tau &= \Psi\alpha
\end{aligned}$$

$$\frac{\Gamma; \Delta; \Theta \vdash E :: \tau \qquad (x :: \tau') \in \Delta}{\Gamma; \Delta \setminus x; \Theta \vdash \lambda x \mapsto E :: \tau' \to \tau} \quad (\textsc{Abs})$$

Tandoori

- Tandoori is the implementation of C^{κ} for a reasonable subset of Haskell 98
- Based on GHC 6.12's parser and renamer front-end
- Get it from http://gergo.erdi.hu/projects/tandoori/, available under a BSD license

Questions?