Polymorphic Type Inference

COMP3610 – Principles of Programming Languages

Ben Lippmeier

Australian National University

Semester 2, 2009

Reminder: The Polymorphic Lambda Calculus

$x \longrightarrow$ (value variables)

$a \longrightarrow$ (type variables)

Types

σ	$::=$	$\forall \overline{a}.\ \tau$	(type schemes)
τ	$::=$	a	
	$\mid$	$\mathrm{Int} \mid \mathrm{Bool} \mid \mathrm{Char}$	(base types)
	$\mid$	$\tau_1 \longrightarrow \tau_2$	(function type)

Expressions

e	$::=$	x	
	$\mid$	$\lambda x.\ e$	(function abstraction)
	$\mid$	$e_1\ \ e_2$	(function application)

$$\frac{x : \tau \ \in \ \Gamma}{\Gamma \vdash x :: \tau} \text{ (Var)}$$

$$\frac{\Gamma \vdash e :: \sigma \qquad a \notin \mathsf{fv}(\Gamma)}{\Gamma \vdash e :: \forall a.\sigma} \text{ (Gen)}$$

$$\frac{\Gamma,\ x : \tau_1 \vdash e_2 :: \tau_2}{\Gamma \vdash \lambda\ (x : \tau_1).\ e_2 :: \tau_1 \to \tau_2} \text{ (Abs)}$$

$$\frac{\Gamma \vdash e :: \forall a.\sigma}{\Gamma \vdash e :: \sigma[a := \tau]} \text{ (Inst)}$$

$$\frac{\Gamma \vdash e_1 :: \tau_{11} \to \tau_{12} \qquad \Gamma \vdash e_2 :: \tau_{11}}{\Gamma \vdash e_1\ e_2 :: \tau_{12}} \text{ (App)}$$

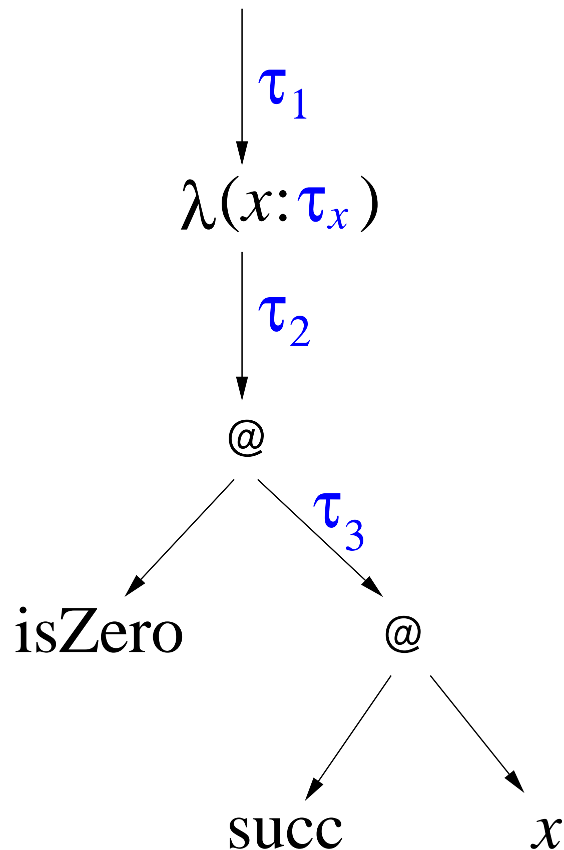
(Monomorphic) Type Inference

Example: $\lambda\, x.\ isZero\ (succ\ x)$

Step 1: Add fresh type variables to λ-abstractions in the program.

$$\begin{array}{ll} & \lambda\, x.\ isZero\ (succ\ x) \\ \Rightarrow & \lambda\, (x : \tau_x).\ isZero\ (succ\ x) \end{array}$$

Step 2: Extract type constraints from this annotated syntax tree.



$\tau_{\text{succ}} = Int \rightarrow Int$ (axiom)

$\tau_{\text{succ}} = \tau_x \rightarrow \tau_3$ (from use in program)

$\tau_{\text{isZero}} = Int \rightarrow Bool$ (axiom)

$\tau_{\text{isZero}} = \tau_3 \rightarrow \tau_2$ (from use in program)

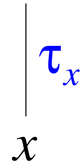
$\tau_1 = \tau_x \rightarrow \tau_2$ (from λ-abstraction)

Step 3: Solve constraints.

1.	τ_{succ}	$= Int \to Int$	
2.	τ_{succ}	$= \tau_x \to \tau_3$	
3.	τ_{isZero}	$= Int \to Bool$	
4.	τ_{isZero}	$= \tau_3 \to \tau_2$	
5.	τ_1	$= \tau_x \to \tau_2$	
6.	τ_x	$= Int$	(Unify 1 2)
7.	τ_3	$= Int$	(Unify 1 2)
8.	τ_2	$= Bool$	(Unify 3 4)
9.	τ_1	$= Int \to Bool$	(Substitute 5 6 8)

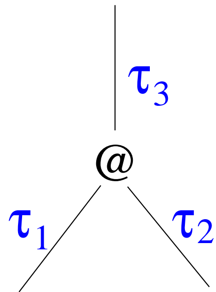
Step 4: Substitute solution into original program.

Constraint Generation Rules



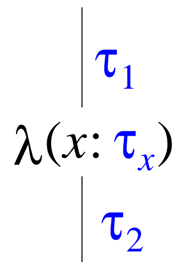
(Var) Only needed if x is primitive, eg isZero

$\tau_x =$ ...type for x...



(App)

$\tau_1 = \tau_2 \longrightarrow \tau_3$



(Abs)

$\tau_1 = \tau_x \longrightarrow \tau_2$

Polymorphic type inference

Here is the rule for generalisation:

$$\frac{\Gamma \vdash e :: \sigma \qquad a \notin \mathsf{fv}(\Gamma)}{\Gamma \vdash e :: \forall a.\sigma} \text{ (Gen)}$$

- The *typing rule* for generalisation is applicable anywhere in the program. We can always generalise the type of an expression, provided we meet the arbitraryness requirement $a \notin fv(\Gamma)$.
- However, only generalising $\mathbf{let}$-bound variables is usually enough for the programmer, and makes type inference easier.
- Type systems which tie generalisation to $\mathbf{let}$ are called “let-polymorphic”. eg: Haskell 98, ML, O’Caml.

Let-polymorphism / aka the Hindley-Milner system

$$\begin{array}{ll}\textbf{let} & id = \lambda x.\ x \\ \textbf{in} & Pair\ (id\ 5)\ (id\ \text{``}\texttt{hello}\text{''})\end{array}$$

- To type-check this example we need to build a polytype for id.
- Our inference algorithm will build a monotype for the right of the binding, generalise it into a type scheme, then use that scheme in the body, ie:
- Note that recursive calls of the bound variable use the monotype.

$$\frac{\Gamma,\ x : \tau_1 \vdash e_1 :: \tau_1 \qquad \Gamma,\ x : \mathrm{Gen}(\Gamma, \tau_1) \vdash e_2 :: \tau_2}{\Gamma \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 :: \tau_2}\ \text{(Let)}$$

The arbitrariness requirement

Recall the following rule of natural deduction.

$$\frac{P(a) \qquad (a \text{ arbitrary})}{\forall x.\ P(x)}$$

$$\begin{array}{l|l|l}
n & a & \vdots \\
\vdots & & \vdots \\
m & & \mathrm{Cat}(a) \to \mathrm{EatsFish}(a) \\
\hline
m+1 & \multicolumn{2}{l}{\forall x.\ \mathrm{Cat}(x) \to \mathrm{EatsFish}(x)}
\end{array}$$

The a on the left of the bar is a *guard* which reminds us that this variable is local to the inner derivation, and it cannot be free in an assumption.

Breaching the arbitrariness requirement

When we generalise for a variable a, the same proof steps must be possible for all members of the domain.

1	$(\text{Cat}(kitty) \rightarrow \text{HasFur}(\text{kitty})) \wedge \text{Cat}(\text{kitty})$	
2	$\text{Cat}(kitty) \rightarrow \text{HasFur}(\text{kitty})$	$\wedge$-E, 1
3	$\text{Cat}(\text{kitty})$	$\wedge$-E, 1
4	$\text{HasFur}(\text{kitty})$	$\rightarrow$-E, 2, 3
5	$\forall x.\ \text{HasFur}(x)$ *WRONG*	

Not all members of the doman have fur.

Same problem, different presentation

$$\begin{array}{lll} \textbf{let} & id & = \lambda(x:\tau_x).\ \ \begin{array}{ll}\textbf{let} & y = x \\ \textbf{in} & y\end{array} \\ \textbf{in} & succ\ (id\ \text{“oh noes”}) & \end{array}$$

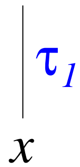
- When typing the inner **let**-expression, we assume that x has type τ_x. (Remember from Curry-Howard, abstraction $\sim$ assumption)
- We must use the type τ_x for y as well, which gives $id :: \forall a.\ a \rightarrow a$.

$$\dfrac{\dfrac{\ldots\ y:\tau_x,\ x:\tau_x \vdash x :: \tau_x \qquad \ldots\ y:\tau_x \vdash y :: \tau_x}{\ldots\ x:\tau_x \vdash \textbf{let}\ \ y = x\ \ \textbf{in}\ y :: \tau_x}}{\vdots}$$

Don’t generalise variables free in the type environment

$$\mathrm{Gen}(\Gamma, \tau) \stackrel{\mathrm{def}}{=} \forall \overline{a}.\, \tau \quad \text{where} \quad \overline{a} \notin fv(\Gamma)$$

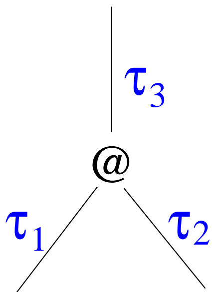
Constraint Generation Rules (for let-polymorphism)



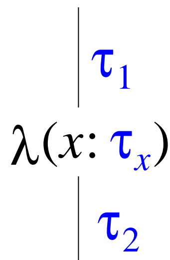
$\tau_1 = \mathrm{Inst}(\text{...type for x...})$ (when x is primitive, eg isZero)

$\tau_1 = \mathrm{Inst}(\tau_x^{\text{scheme}})$ (when x is **let**-bound)

$\tau_1 = \tau_x$ (when τ_x is not arbitrary / constraint is inside a guard for τ_x)



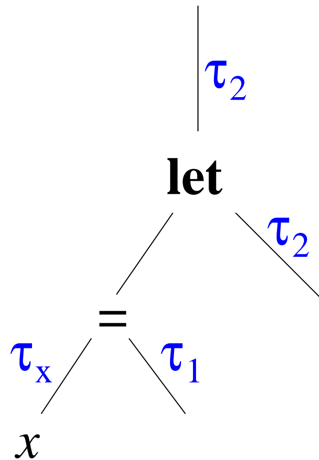
$\tau_1 = \tau_2 \rightarrow \tau_3$



τ_x | ... constraints for body ...

... constraints for body ...

$\tau_1 = \tau_x \rightarrow \tau_2$



τ_x | ... constraints for right of binding ...

| ... constraints for right of binding ...

$$\tau_x^{\text{scheme}} = \mathrm{Gen}(\Sigma, \tau_x)$$

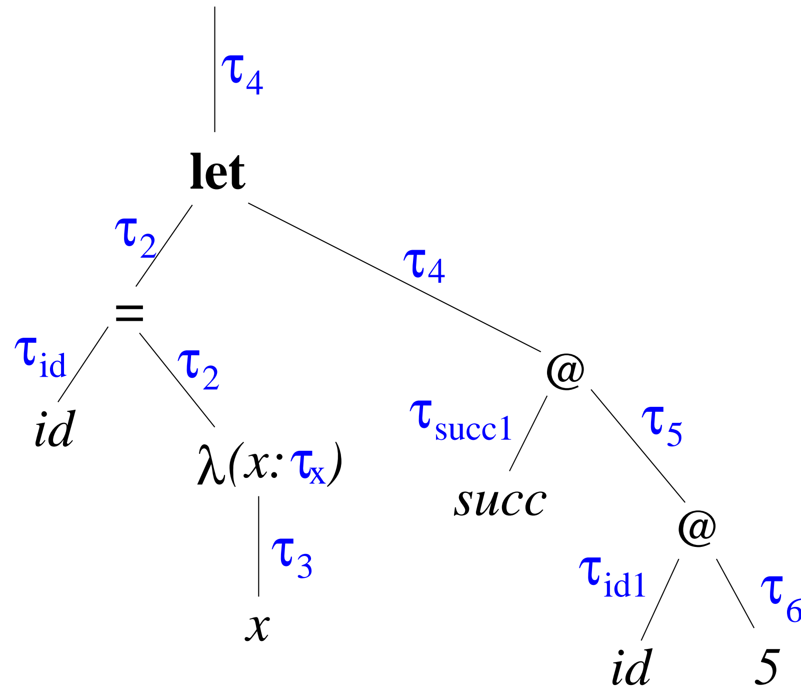
Note: Σ is the set of enclosing guard variables.
This models what the type environment is at that point in the program.

Side Condition

Before generalising the type of a **let**-bound variable, we must first solve all the constraints from its body, that is, in the preceeding guard.

This ensures that the type of the binding is “done” before we use it somewhere else in the program.

Example



$$\tau_{id} \;\Big|\; \tau_x \;\Big|\; \tau_3 = \tau_x$$

$$\tau_2 = \tau_x \rightarrow \tau_3$$

$$\tau_{id} = \tau_2$$

$$\tau_{id}^{\text{scheme}} = \text{Gen}(\emptyset,\ \tau_{id})$$

$$\tau_{succ1} = \text{Inst}(Int \rightarrow Int)$$

$$\tau_{succ1} = \tau_5 \rightarrow \tau_4$$

$$\tau_{id1} = \text{Inst}(\tau_{id}^{\text{scheme}})$$

$$\tau_{id1} = \tau_6 \rightarrow \tau_5$$

$$\tau_6 = Int$$

Exercises

From the lectures on Church Encoding we have:

$$\textbf{let}\ x = e_1\ \textbf{in}\ e_2 \stackrel{\text{def}}{=} (\lambda x.\ e_2)\ e_1$$

However, with let-polymorphism we can infer a type for:

$$\begin{array}{ll}\textbf{let} & id = \lambda x.\ x \\ \textbf{in} & Pair\ (id\ 5)\ (id\ \text{``}\texttt{hello}\text{''})\end{array}$$

but not:

$$(\lambda\ id\ .\ Pair\ (id\ 5)\ (id\ \text{``}\texttt{hello}\text{''}))\ (\lambda x.\ x)$$

what's gone wrong?