

Polymorphic Type Inference

COMP3610 – Principles of Programming Languages

Ben Lippmeier

Australian National University
Semester 2, 2009

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

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

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

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

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

Reminder: The Polymorphic Lambda Calculus

x → (value variables)
 a → (type variables)

Types

$\sigma ::= \forall \bar{a}. \tau$ (type schemes)
 $\tau ::= a$
| Int | Bool | Char (base types)
| $\tau_1 \rightarrow \tau_2$ (function type)

Expressions

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

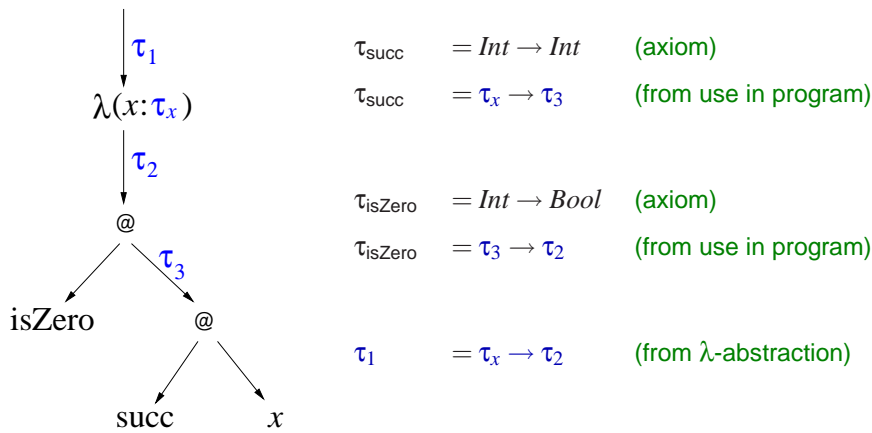
(Monomorphic) Type Inference

Example: $\lambda x. \text{isZero} (\text{succ } x)$

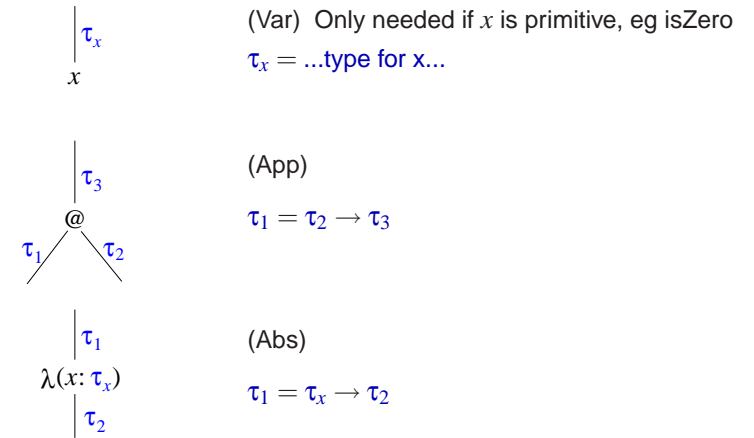
Step 1: Add fresh type variables to λ -abstractions in the program.

$\lambda x. \text{isZero} (\text{succ } x)$
 $\Rightarrow \lambda (x : \tau_x). \text{isZero} (\text{succ } x)$

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



Constraint Generation Rules



Step 3: Solve constraints.

1. $\tau_{\text{succ}} = \text{Int} \rightarrow \text{Int}$
2. $\tau_{\text{succ}} = \tau_x \rightarrow \tau_3$
3. $\tau_{\text{isZero}} = \text{Int} \rightarrow \text{Bool}$
4. $\tau_{\text{isZero}} = \tau_3 \rightarrow \tau_2$
5. $\tau_1 = \tau_x \rightarrow \tau_2$
6. $\tau_x = \text{Int}$ (Unify 1 2)
7. $\tau_3 = \text{Int}$ (Unify 1 2)
8. $\tau_2 = \text{Bool}$ (Unify 3 4)
9. $\tau_1 = \text{Int} \rightarrow \text{Bool}$ (Substitute 5 6 8)

Step 4: Substitute solution into original program.

Polymorphic type inference

Here is the rule for generalisation:

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

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

Let-polymorphism / aka the Hindley-Milner system

```
let id = λx. x
in Pair (id 5) (id "hello")
```

- 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.

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

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	(Cat(<i>kitty</i>) → HasFur(<i>kitty</i>)) ∧ Cat(<i>kitty</i>)	
2	Cat(<i>kitty</i>) → HasFur(<i>kitty</i>)	∧-E, 1
3	Cat(<i>kitty</i>)	∧-E, 1
4	HasFur(<i>kitty</i>)	→-E, 2, 3
5	∀x. HasFur(<i>x</i>)	WRONG

Not all members of the domain have fur.

The arbitrariness requirement

Recall the following rule of natural deduction.

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

n	a	⋮
⋮	⋮	⋮
m	Cat(<i>a</i>) → EatsFish(<i>a</i>)	
m + 1	∀x. Cat(<i>x</i>) → EatsFish(<i>x</i>)	

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.

Same problem, different presentation

```
let id = λ(x : τx). let y = x
in y
in succ (id "oh noes")
```

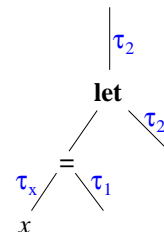
- When typing the inner **let**-expression, we assume that *x* has type τ_x . (Remember from Curry-Howard, abstraction \sim assumption)
- We must use the type τ_x for *y* as well, which gives $id :: \forall a. a \rightarrow a$.

$$\frac{\dots y : \tau_x, x : \tau_x \vdash x :: \tau_x \quad \dots y : \tau_x \vdash y :: \tau_x}{\dots x : \tau_x \vdash \text{let } y = x \text{ in } y :: \tau_x}$$

⋮

Don't generalise variables free in the type environment

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



τ_x | ... constraints for right of binding ...
 | ... constraints for right of binding ...
 $\tau_x^{\text{scheme}} = \text{Gen}(\Sigma, \tau_x)$

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

Side Condition

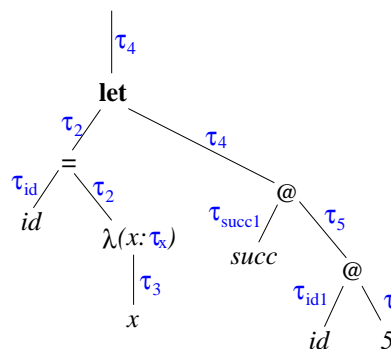
Before generalising the type of a **let**-bound variable, we must first solve all the constraints from its body, that is, in the preceding guard.

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

Constraint Generation Rules (for let-polymorphism)

$\begin{array}{c} \\ \tau_1 \\ x \end{array}$	$\tau_1 = \text{Inst}(\dots \text{type for } x \dots)$ (when x is primitive, eg isZero) $\tau_1 = \text{Inst}(\tau_x^{\text{scheme}})$ (when x is let -bound) $\tau_1 = \tau_x$ (when τ_x is not arbitrary / constraint is inside a guard for τ_x)
$\begin{array}{c} \\ \tau_3 \\ @ \\ \tau_1 \quad \tau_2 \end{array}$	$\tau_1 = \tau_2 \rightarrow \tau_3$
$\begin{array}{c} \\ \tau_1 \\ \lambda(x: \tau_x) \\ \\ \tau_2 \end{array}$	τ_x ... constraints for body constraints for body ... $\tau_1 = \tau_x \rightarrow \tau_2$

Example



$\tau_{id} \quad \tau_x$ | $\tau_3 = \tau_x$
 | $\tau_2 = \tau_x \rightarrow \tau_3$
 | $\tau_{id} = \tau_2$
 $\tau_{id}^{\text{scheme}} = \text{Gen}(\emptyset, \tau_{id})$
 $\tau_{succ1} = \text{Inst}(\text{Int} \rightarrow \text{Int})$
 $\tau_{succ1} = \tau_5 \rightarrow \tau_4$
 $\tau_{id1} = \text{Inst}(\tau_{id}^{\text{scheme}})$
 $\tau_{id1} = \tau_6 \rightarrow \tau_5$
 $\tau_6 = \text{Int}$

Exercises

From the lectures on Church Encoding we have:

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

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

```
let id =  $\lambda x. x$   
in Pair (id 5) (id "hello")
```

but not:

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

what's gone wrong?