Polymorphic Type Inference (II)

Sjaak Smetsers

6 March 2025

SL Expressions

$$
\begin{array}{rcl}
e & ::= & x \\
& | & \lambda x.e \\
& | & e_1 e_2 \\
& | & \textbf{if } e_c \textbf{ then } e_t \textbf{ else } e_e \\
& | & e_1 \; op \; e_2 \\
& | & i \\
& | & b \\
& | & \textbf{let } x = e_1 \textbf{ in } e_2 \\
& | & (e_1, e_2) \mid \textbf{fst} \mid \textbf{snd} \\
& | & [\,] \mid e_1 : e_2 \mid \textbf{null} \mid \textbf{head} \mid \textbf{tail} \\
op & ::= & + \mid \leq \mid \&\&
\end{array}
$$

SL Types

Types

$$
\begin{array}{rcl}
\sigma & ::= & \alpha \\
& | & \sigma_1 \to \sigma_2 \\
& | & (\sigma_1, \sigma_2) \\
& | & [\sigma] \\
& | & \mathit{int} \mid \mathit{bool}
\end{array}
$$

Type Schemes (aka forall types, polymorphic types)

$$\Sigma ::= \forall \vec{\alpha}.\sigma$$

Free type variables

$$
\begin{array}{rcl}
\mathsf{TV}(\sigma) & = & \text{all type variables in } \sigma \\
\mathsf{TV}(\Sigma) & = & \text{all type variables minus the } \forall\text{-bound ones} \\
\mathsf{TV}(\Gamma) & = & \text{all free type variables of all types in } \Gamma
\end{array}
$$

Polymorphism Recap

$$\begin{array}{l}\textbf{let } i = \lambda x.x \\ \textbf{in } (i\ \textbf{True}, i\ 5)\end{array}$$

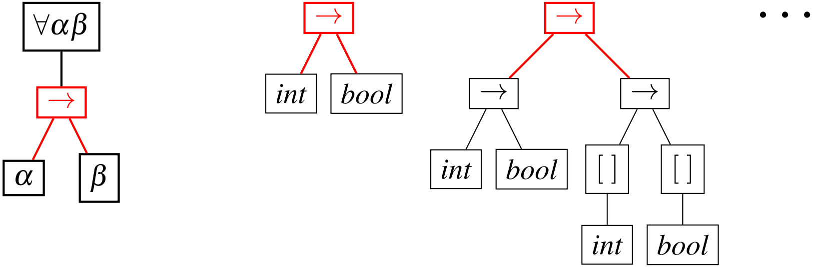
$$(\lambda i.(i\ \textbf{True}, i\ 5))(\lambda x.x)$$

- None of these programs works with simple types
- Both work with fully polymorphic type systems
- Only the first one works with let-polymorphism

Types vs Type Schemes

Type schemes denote *sets of types*:

- All types that fit the scheme
- e.g. $\forall \alpha\beta.\alpha \to \beta$: anything that is a function



Type Checking vs. Type Inference

- Type checking: Given an expression e, a type σ and an environment Γ for the free variables of e, check if $\Gamma \vdash e : \sigma$ holds.
- Type inference: Given an expression e, compute an environment Γ and a type σ such that $\Gamma \vdash e : \sigma$.
- Both type checker and type inferencer generate/solve constraints while (recursively) traversing the expression tree of e.

Motivating example

What do we know about this function?

$$\lambda x.\lambda f.\lambda y.\textbf{if } x \leq 5 \textbf{ then } y \textbf{ else } f\,x$$

- It takes three arguments, so it's type must be $\alpha \to \beta \to \gamma \to \delta$
- x is compared to an integer, so $\alpha = int$
- f is used as a function, so $\beta = \beta_1 \to \beta_2$
- f is applied to x, so $\beta_1 = \alpha$
- y and the result of $f\,x$ are the results of the if-then-else, so $\gamma = \beta_2$
- The result of the function is the result of the if-then-else, so $\delta = \beta_2$
- Putting it all together we get one of:

$$\begin{array}{ccccccccc}
int & \to & (int & \to & \delta) & \to & \delta & \to & \delta \\
int & \to & (int & \to & \gamma) & \to & \gamma & \to & \gamma \\
int & \to & (int & \to & \beta_2) & \to & \beta_2 & \to & \beta_2
\end{array}$$

What Have We Just Done?

Type inference!

$$\lambda x.\lambda f.\lambda y.\textbf{if}\ x \leq 5\ \textbf{then}\ y\ \textbf{else}\ f\ x$$

1. Traverse the AST, look at subexpressions
   - f is used as a function, so ...
2. Generate constraints with *fresh* type variables to record partial information
   - ... $\beta = \beta_1 \to \beta_2$
3. Solve the constraints to find the types of subexpressions

   Some are completely determined $x{:}\mathit{int}$

   Some are partially determined $f{:}\mathit{int} \to \delta$

   Some are undetermined $y{:}\delta$

Today: how to put these ideas into an algorithm

Type Substitutions

A substitution is a function

$$* : TVar \rightarrow \sigma$$

We write

$$[\alpha \mapsto int, \beta \mapsto (int \rightarrow ([int], bool))]$$

Substitutions can be applied to a type:

$$(-)^* : \sigma \rightarrow \sigma$$

Equations:

$$\begin{array}{rcl} \alpha^* & = & *(\alpha), \\ (\sigma_1 \rightarrow \sigma_2)^* & = & \sigma_1^* \rightarrow \sigma_2^* \\ (\sigma_1, \sigma_2)^* & = & (\sigma_1^*, \sigma_2^*) \\ [\sigma]^* & = & [\sigma^*] \\ b^* & = & b \end{array}$$

Note that we are working with *types* here, not with type schemes

Example

$$
\begin{aligned}
*_1 &= [\alpha \mapsto [\beta]] \\
*_2 &= [\beta \mapsto int, \gamma \mapsto bool] \\
*_3 &= [\alpha \mapsto [int], \gamma \mapsto bool] \\
\tau &= (\alpha, \gamma) \\
\tau^{*_1} &= ([\beta], \gamma) \\
(\tau^{*_1})^{*_2} = \tau^{(*_2 \circ *_1)} &= ([int], bool) \\
\tau^{*_3} &= ([int], bool)
\end{aligned}
$$

- Some substitutions are comparable by being more or less *general*
- We can extend $*_1$ by $*_2$ to get $*_3$: $*_2 \circ *_1 = *_3$ $(\neq *_1 \circ *_2)$
- We say $*_a$ is *more general* (or *less specific*) than $*_b$, denoted by $*_a \sqsubseteq *_b$, where

$$*_a \sqsubseteq *_b \iff \exists *.\ *_b = * \circ *_a$$

Unification

In type derivations sometimes types just magically appear
As if we somehow know what we need later in the derivation

$$\frac{\vdash \lambda f.\textbf{if True then } (f\,5) \textbf{ else } (f\,7) : (int \to int) \to int \qquad \vdash \lambda x.\,1 : int \to int}{\vdash (\lambda f.\textbf{if True then } (f\,5) \textbf{ else } (f\,7))(\lambda x.\,1) : int}$$

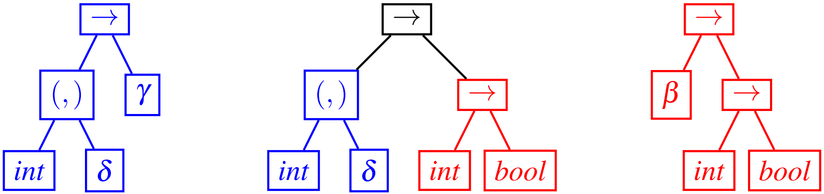
Solution: use place holders (type variables) to delay decisions
And rely on unification to sort out things later

$$\frac{\dfrac{\dfrac{(f\,5)\ \alpha = int \to \beta}{\vdots}}{\vdash \lambda f.\textbf{if True then } (f\,5) \textbf{ else } (f\,7) : \alpha \to int} \qquad \dfrac{\dfrac{\alpha = \gamma \to int}{\vdots}}{\vdash \lambda x.\,1 : \alpha}}{\vdash (\lambda f.\textbf{if True then } (f\,5) \textbf{ else } (f\,7))(\lambda x.\,1) : int}$$

Unification Example

Imagine from one part of a program we know that $\alpha = (int, \delta) \to \gamma$

From another part of a program we learn that $\alpha = \beta \to (int \to bool)$



Unification computes a substitution $*$ such that

$$((int, \delta) \to \gamma)^* = (\beta \to (int \to bool))^*$$

In this case

$$* = [\beta \mapsto (int, \delta), \gamma \mapsto (int \to bool)]$$

Another possibility is

$$\begin{aligned} *' &= [\beta \mapsto (int, int), \gamma \mapsto (int \to bool), \delta \mapsto int] \\ *' &= [\delta \mapsto int] \circ [\beta \mapsto (int, \delta), \gamma \mapsto (int \to bool)] \end{aligned}$$

Unifiers

- A unifier for σ, τ is a substitution $*$ such that

$$\sigma^* = \tau^*$$

- A unifier $*$ is a most general unifier for σ, τ if

$$\forall *' : *' \textit{ is a unifier of } \sigma, \tau \Rightarrow * \sqsubseteq *'$$

Unifier for int, int?	id = empty substitution
Unifier for int, α?	$[\alpha \mapsto int]$
Unifier for $int, bool$?	$\lightning$
Unifier for $[int], [\alpha]$?	$[\alpha \mapsto int]$
Unifier for $[\beta], [\alpha]$?	$[\alpha \mapsto \beta]$ or $[\beta \mapsto \alpha]$

Infinite types?

What is the type of this function? (Try it in Haskell)

```
f x = f x
f :: a -> b
```

What about this one? `g x = g (x, x)`

Let's look at the constraints

$$\begin{array}{rcl} g & : & \alpha \to \beta \\ x & : & \alpha \\ (x,x) & : & (\alpha,\alpha) \\ \alpha \to \beta & = & (\alpha,\alpha) \to \beta \\ \alpha & = & (\alpha,\alpha) \end{array}$$

What is a unifier for this?

But then

$$(\alpha,\alpha) = ((\alpha,\alpha),(\alpha,\alpha))$$

Occurs check

- Whenever we see a constraint with a variable on one side

$$\begin{array}{ccc} \alpha & = & \sigma \quad \text{or} \\ \sigma & = & \alpha \end{array}$$

- Like in the case of

$$\alpha = (\alpha, \alpha)$$

- We need to require that

$$\alpha \notin \mathsf{TV}(\sigma)$$

- Otherwise unification fails with the error "Occurs check: cannot construct the infinite type: $a \sim (a, a)$"

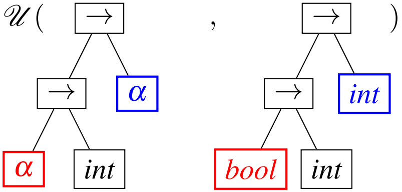
Unification Algorithm

- The recursive function $\mathscr{U}$ computes a most general unifier of σ, τ
- Or fail is such a unifier does not exist.

$$
\begin{array}{rcl}
\mathscr{U}(int, int) & = & id \\
\mathscr{U}(bool, bool) & = & id \\
\mathscr{U}(\alpha, \alpha) & = & id \\
\mathscr{U}(\alpha, \tau) & = & [\alpha \mapsto \tau],\ \text{if } \alpha \notin \mathsf{TV}(\tau) \text{ otherwise } \mathsf{fail} \\
\mathscr{U}(\tau, \alpha) & = & [\alpha \mapsto \tau],\ \text{if } \alpha \notin \mathsf{TV}(\tau) \text{ otherwise } \mathsf{fail} \\
\mathscr{U}([\sigma], [\tau]) & = & \mathscr{U}(\sigma, \tau) \\
\mathscr{U}((\sigma_1, \sigma_2), (\tau_1, \tau_2)) & = & \mathscr{U}(\sigma_2^*, \tau_2^*) \circ *,\ \text{where } * = \mathscr{U}(\sigma_1, \tau_1) \\
\mathscr{U}(\sigma_1 \to \sigma_2, \tau_1 \to \tau_2) & = & \mathscr{U}(\sigma_2^*, \tau_2^*) \circ *,\ \text{where } * = \mathscr{U}(\sigma_1, \tau_1) \\
\mathscr{U}(\cdot, \cdot) & = & \mathsf{fail}
\end{array}
$$

Example

- Why the need to apply the substitution before descending?
- $\mathscr{U}(\sigma_1 \to \sigma_2, \tau_1 \to \tau_2) = \mathscr{U}(\sigma_2^*, \tau_2^*) \circ *,$ where $* = \mathscr{U}(\sigma_1, \tau_1)$



- Wrong answer: $[\alpha \mapsto int, \alpha \mapsto bool]$
- Right answer: $\lightning$ cannot unify *int* and *bool*

Type Inference/Checking

- Unification solves constraints, but where do constraints come from?
- Two orthogonal/complementary approaches:
  1. Define a recursive function $\mathscr{C} : (\Gamma, e, \sigma) \to *_\perp$
  2. Define a recursive function $\mathscr{I} : (\Gamma, e) \to (\sigma, *)_\perp$
- such that either:
  1. - $\mathscr{C}(\Gamma, e, \sigma) = * \Rightarrow \Gamma^* \vdash e : \sigma^*$
     - $\mathscr{C}(\Gamma, e, \sigma) = \mathit{fail} \Rightarrow$ there is no $*$ with $\Gamma^* \vdash e : \sigma^*$
  2. - $\mathscr{I}(\Gamma, e) = (\sigma, *) \Rightarrow \Gamma^* \vdash e : \sigma$
     - $\mathscr{I}(\Gamma, e) = \mathit{fail} \Rightarrow$ there are no $\sigma, *$ with $\Gamma^* \vdash e : \sigma$

$\mathscr{C}$ and $\mathscr{I}$ can be expressed in terms of each other:

$$\begin{aligned} \mathscr{C}(\Gamma, e, \sigma) &= \mathscr{U}(\sigma^*, \tau) \circ * \text{ where } (\tau, *) = \mathscr{I}(\Gamma, e) \\ \mathscr{I}(\Gamma, e) &= (\alpha^*, *) \text{ where } * = \mathscr{C}(\Gamma, e, \alpha), \ \alpha \text{ fresh} \end{aligned}$$

Type Inference

Given an expression e. How do we infer a type for e?

- Let $\mathsf{FV}(e) = \{x_1, \ldots, x_k\}$, and $\alpha_1, \ldots, \alpha_k$ be *fresh* type variables.
- Set $\Gamma_0 = \{x_1{:}\alpha_1, \ldots, x_k{:}\alpha_k\}$
- Compute $\mathscr{I}(\Gamma_0, e)$

Definition of $\mathscr{C}$ (integers)

$$
\begin{array}{rcl}
\mathscr{C}(\Gamma, \mathtt{i}, \sigma) & = & \mathscr{U}(\sigma, int) \\
\mathscr{C}(\Gamma, e_1 + e_2, \sigma) & = & \mathscr{U}(\sigma^*, int) \circ * \\
& & \text{where} \\
& & \qquad \begin{array}{rcl} * & = & \mathscr{C}(\Gamma^{*_1}, e_2, int) \circ *_1 \\ *_1 & = & \mathscr{C}(\Gamma, e_1, int) \end{array} \\
\mathscr{C}(\Gamma, e_1 \leq e_2, \sigma) & = & \mathscr{U}(\sigma^*, bool) \circ * \\
& & \text{where} \\
& & \qquad \begin{array}{rcl} * & = & \mathscr{C}(\Gamma^{*_1}, e_2, int) \circ *_1 \\ *_1 & = & \mathscr{C}(\Gamma, e_1, int) \end{array}
\end{array}
$$

Definition of $\mathscr{C}$ (booleans)

$$
\begin{array}{rcl}
\mathscr{C}(\Gamma, \mathsf{b}, \sigma) & = & \mathscr{U}(\sigma, bool) \\
\mathscr{C}(\Gamma, \textbf{if } e_c \textbf{ then } e_t \textbf{ else } e_e, \sigma) & = & \mathscr{C}(\Gamma^*, e_c, bool) \circ * \\
& & \text{where} \\
& & \qquad \begin{array}{rcl} * & = & \mathscr{C}(\Gamma^{*_1}, e_e, \sigma^{*_1}) \circ *_1 \\ *_1 & = & \mathscr{C}(\Gamma, e_t, \sigma) \end{array} \\
\mathscr{C}(\Gamma, e_1 \,\&\&\, e_2, \sigma) & = & \mathscr{U}(\sigma^*, bool) \circ * \\
& & \text{where} \\
& & \qquad \begin{array}{rcl} * & = & \mathscr{C}(\Gamma^{*_1}, e_2, bool) \circ *_1 \\ *_1 & = & \mathscr{C}(\Gamma, e_1, bool) \end{array}
\end{array}
$$

Definition of $\mathscr{C}$ (tuples)

$$
\begin{array}{rcl}
\mathscr{C}(\Gamma,(e_1,e_2),\sigma) & = & \mathscr{U}(\sigma^*,(\alpha_1,\alpha_2)^*)\circ * \\
& & \text{where} \\
& & \qquad \begin{array}{rcl} * & = & \mathscr{C}(\Gamma^{*_1},e_2,\alpha_2)\circ *_1 \\ *_1 & = & \mathscr{C}(\Gamma,e_1,\alpha_1) \\ & & \alpha_1,\alpha_2 \text{ fresh} \end{array} \\
\mathscr{C}(\Gamma,\mathbf{fst},\sigma) & = & \mathscr{U}(\sigma,(\alpha_1,\alpha_2)\to\alpha_1), \qquad \alpha_1,\alpha_2 \text{ fresh} \\
\mathscr{C}(\Gamma,\mathbf{snd},\sigma) & = & \mathscr{U}(\sigma,(\alpha_1,\alpha_2)\to\alpha_2), \qquad \alpha_1,\alpha_2 \text{ fresh}
\end{array}
$$

Definition of $\mathscr{C}$ (lists)

$$
\begin{array}{rcl}
\mathscr{C}(\Gamma, [], \sigma) & = & \mathscr{U}(\sigma, [\alpha]), \quad \alpha \text{ fresh} \\
\mathscr{C}(\Gamma, e_1 : e_2, \sigma) & = & \mathscr{U}(\sigma^*, [\alpha^*]) \circ * \\
 & & \text{where} \\
 & & \quad \begin{array}{rcl} * & = & \mathscr{C}(\Gamma^{*_1}, e_2, [\alpha^{*_1}]) \circ *_1 \\ *_1 & = & \mathscr{C}(\Gamma, e_1, \alpha) \\ & & \alpha \text{ fresh} \end{array} \\
\mathscr{C}(\Gamma, \mathbf{null}, \sigma) & = & \mathscr{U}(\sigma, [\alpha] \to bool), \quad \alpha \text{ fresh} \\
\mathscr{C}(\Gamma, \mathbf{head}, \sigma) & = & \mathscr{U}(\sigma, [\alpha] \to \alpha), \quad \alpha \text{ fresh} \\
\mathscr{C}(\Gamma, \mathbf{tail}, \sigma) & = & \mathscr{U}(\sigma, [\alpha] \to [\alpha]), \quad \alpha \text{ fresh}
\end{array}
$$

Definition of $\mathscr{C}$ (functions)

$$
\begin{array}{rcl}
\mathscr{C}(\Gamma, e_1 e_2, \sigma) & = & \mathscr{C}(\Gamma^*, e_2, \alpha^*) \circ * \\
 & & \text{where } * = \mathscr{C}(\Gamma, e_1, \alpha \to \sigma) \\
 & & \alpha \text{ fresh} \\
\mathscr{C}(\Gamma, \lambda x.e, \sigma) & = & \mathscr{U}(\sigma^*, \alpha^* \to \beta^*) \circ * \\
 & & \text{where } * = \mathscr{C}((\Gamma, x{:}\alpha), e, \beta) \\
 & & \alpha, \beta \text{ fresh} \\
\mathscr{C}(\Gamma, \textbf{let}\ x = e_1\ \textbf{in}\ e_2, \sigma) & = & \mathscr{C}((\Gamma^*, x{:}\forall\vec{\beta}.\alpha^*), e_2, \sigma^*) \circ * \\
 & & \vec{\beta} = \mathsf{TV}(\alpha^*) - \mathsf{TV}(\Gamma^*) \\
 & & * = \mathscr{C}((\Gamma, x{:}\alpha), e_1, \alpha),\ \alpha \text{ fresh} \\
\mathscr{C}(\Gamma, x, \sigma) & = & \mathscr{U}(\tau[\vec{\alpha} \mapsto \vec{\beta}], \sigma) \\
 & & \text{where } \Gamma(x) = \forall\vec{\alpha}.\tau \\
 & & \vec{\beta} \text{ fresh}
\end{array}
$$

References

- O. Lee, K. Yi. “Proofs about a folklore let-polymorphic type inference algorithm”. TOPLAS, 1998.
- Martin Grabmüller. “Algorithm W Step by Step”. Draft Paper, 2006
- B. Heeren, J. Hage, and D. Swierstra. “Generalizing Hindley-Milner Type Inference Algorithms”. Technical Report UU-CS- 2001-031, Institute of Information and Computing Sciences, Utrecht University, 2002.
- Robin Milner. “A Theory of Type Polymorphism in Programming”. Journal of Computer and System Science, 1978