Polymorphic Types

CS242

Lecture 6

Let Expressions

Extend the lambda calculus with one new expression

$e \rightarrow x \mid \lambda x.e \mid e\ e \mid \text{let } f = \lambda x.e \text{ in } e \mid i$

$t \rightarrow \alpha \mid t \rightarrow t \mid \text{int}$

Let Expressions

Nothing new here, really:

$$\text{let } f = \lambda x.e \text{ in } e' \qquad \text{is equivalent to} \qquad (\lambda f.e')\ \lambda x.e$$

And note we are getting closer to standard syntax:

$$\text{let } f\ x = e \text{ in } e' \qquad \text{is syntactic sugar for} \qquad \text{let } f = \lambda x.e \text{ in } e'$$

Type Rules

$$\frac{}{A, x: t \vdash x : t}\ [\text{Var}]$$

$$\frac{A, x: t \vdash e : t'}{A \vdash \lambda x{:}t.e : t \rightarrow t'}\ [\text{Abs}]$$

$$\frac{}{A \vdash i : int}\ [\text{Int}]$$

$$\frac{A \vdash \lambda x.e : t \qquad A, f: t \vdash e' : t'}{A \vdash \text{let } f = \lambda x.e \text{ in } e' : t'}\ [\text{Let}]$$

$$\frac{A \vdash e_1 : t \rightarrow t' \qquad A \vdash e_2 : t}{A \vdash e_1\, e_2 : t'}\ [\text{App}]$$

Recall ...

The program

let f = λx.x in f f

is untypable, but

(λx.x) (λy.y)

is typable (in simply typed lambda calculus)

Polymorphic Types

$e \to x \;|\; \lambda x.e \;|\; e\ e \;|\; \text{let } f = \lambda x.e \text{ in } e \;|\; i$

$t \to \alpha \;|\; t \to t \;|\; \text{int}$

$o \to \forall \alpha.o \;|\; t$

Polymorhpic Let Type Rule

$$\frac{A \vdash \lambda x.e : t \qquad A, f: \forall\alpha.t \vdash e' : t' \quad \text{if } \alpha \notin FV(A)}{A \vdash \text{let } f = \lambda x.e \text{ in } e' : t'} \text{[Let]}$$

$FV(A, x{:}t) = FV(A) \cup FV(t)$

$FV(\emptyset) = \emptyset$

$FV(int) = \emptyset$

$F(t \rightarrow t') = FV(t) \cup FV(t')$

$FV(\forall\alpha.t) = FV(t) - \{\alpha\}$

$FV(\alpha) = \{\alpha\}$

The Idea

If we prove $e : t$ and the proof does not use any facts about α, then we have also proven $e: \forall\alpha.t.$

Instantiation Rule

$$\frac{}{A, f: \forall\alpha.t \vdash f: t[\alpha := t']} \quad [\text{Inst}]$$

Example

$$\dfrac{\dfrac{x : \beta \vdash x : \beta}{\vdash \lambda x.x : \beta \to \beta} \qquad \dfrac{I : \forall\alpha.\ \alpha \to \alpha \vdash I : (\rho \to \rho) \to (\rho \to \rho) \qquad I : \forall\alpha.\ \alpha \to \alpha \vdash I : \rho \to \rho}{I : \forall\alpha.\ \alpha \to \alpha \vdash I\ I : \rho \to \rho}}{\vdash \text{let } I = \lambda x.x \text{ in } I\ I : \rho \to \rho}$$

Multiple Type Variables

$$\frac{A \vdash \lambda x.e : t \qquad A, f: \forall\alpha_1,\ldots,\alpha_n.t \vdash e' : t' \quad \text{if } \alpha_1,\ldots,\alpha_n \notin FV(A)}{A \vdash \text{let } f = \lambda x.e \text{ in } e' : t'} \text{[Let]}$$

$FV(A, x{:}t) = FV(A) \cup FV(t)$

$FV(\emptyset) = \emptyset$

$FV(int) = \emptyset$

$F(t \rightarrow t') = FV(t) \cup FV(t')$

$FV(\forall\alpha_1,\ldots,\alpha_n.t) = FV(t) - \{\alpha_1,\ldots,\alpha_n\}$

$FV(\alpha) = \{\alpha\}$

Type Inference for Polymorphic Let

- To do type inference with polymorphic let, we need to know the type derivation for $\lambda x.e$ to do the generalization step
  - Because we need to compute the set of free variables in the environment
  - And we need to know the variables in the type of the function to generalize
- Thus, we need to solve the constraints and produce a valid typing of $\lambda x.e$ to proceed
  - So we solve the constraints and substitute the solution back into the proof at each let.
  - Compute $FV(A)$
  - Generalize

$$\frac{A \vdash \lambda x.e : t \qquad A, f: \forall \alpha_1,\dots,\alpha_n.t \vdash e' : t' \quad \text{if } \alpha_1,\dots,\alpha_n \notin FV(A)}{A \vdash \text{let } f = \lambda x.e \text{ in } e' : t'} \text{[Let]}$$

Example – Full Derivation

$$\dfrac{\dfrac{\dfrac{x:\beta\to\beta \vdash x:\beta\to\beta}{\vdash \lambda x.x : (\beta\to\beta)\to(\beta\to\beta)} \qquad \dfrac{y:\beta\vdash y:\beta}{\vdash \lambda y.y:\beta\to\beta}}{\vdash (\lambda x.x)\ (\lambda y.y):\beta\to\beta} \qquad \beta\notin FV(\emptyset) \qquad \dfrac{I:\forall\alpha.\ \alpha\to\alpha \vdash I:(\rho\to\rho)\to(\rho\to\rho) \qquad I:\forall\alpha.\ \alpha\to\alpha\vdash I:\rho\to\rho}{I:\forall\alpha.\ \alpha\to\alpha \vdash I\ I:\rho\to\rho}}{\vdash \text{let } I = (\lambda x.x)\ (\lambda y.y) \text{ in } I\ I : \rho\to\rho}$$

Outside the allowed syntax, but this example still works.

Example – Type Derivation Skeleton

$$\dfrac{\dfrac{\dfrac{x: \quad \vdash x:}{\vdash \lambda x.x :} \qquad \dfrac{y: \quad \vdash y:}{\vdash \lambda y.y:}}{\vdash (\lambda x.x)\ (\lambda y.y) : \qquad ?? \notin FV(\emptyset)} \qquad \dfrac{I: \qquad \vdash I : \qquad I: \qquad \vdash I:}{I: \quad \vdash I\ I :}}{\vdash \text{let } I = (\lambda x.x)\ (\lambda y.y) \text{ in } I\ I :}$$

Example – Type Inference

x: ⊢ x:

⊢ λx.x :

y: ⊢ y:

⊢ λy.y:

⊢ (λx.x) (λy.y) :   ?? ∉ FV(∅)

I: ⊢ I :   I: ⊢ I:

I: ⊢ I I :

⊢ let I = (λx.x) (λy.y) in I I :

First we run type inference (from last lecture) on the innermost let binding.

Example – Type Inference

$x{:}\ \alpha_x \ \vdash x{:}$

$\vdash \lambda x.x :$

$y{:}\ \alpha_y \ \vdash y{:}$

$\vdash \lambda y.y{:}$

$\vdash (\lambda x.x)\ (\lambda y.y) : \quad ?? \notin FV(\emptyset)$

$I{:} \quad \vdash I : \quad\quad I{:} \quad \vdash I{:}$

$I{:} \quad \vdash I\ I :$

$\vdash \text{let } I = (\lambda x.x)\ (\lambda y.y) \text{ in } I\ I :$

Example – Type Inference

$$\dfrac{x: \alpha_x \quad \vdash x: \alpha_x}{\vdash \lambda x.x : \alpha_x \to \alpha_x} \qquad \dfrac{y: \alpha_y \quad \vdash y: \alpha_y}{\vdash \lambda y.y: \alpha_y \to \alpha_y}$$

$$\dfrac{\dfrac{\vdash \lambda x.x : \alpha_x \to \alpha_x \qquad \vdash \lambda y.y: \alpha_y \to \alpha_y}{\vdash (\lambda x.x)\ (\lambda y.y) : \beta} \qquad ?? \notin FV(\emptyset) \qquad \dfrac{I: \quad \vdash I : \qquad I: \quad \vdash I:}{I: \quad \vdash I\ I :}}{\vdash \text{let } I = (\lambda x.x)\ (\lambda y.y) \text{ in } I\ I :}$$

$$\alpha_x \to \alpha_x = (\alpha_y \to \alpha_y) \to \beta$$

Solving the Equations

$\alpha_x \to \alpha_x = (\alpha_y \to \alpha_y) \to \beta$

$\alpha_x = \alpha_y \to \alpha_y$ [Structure]

$\alpha_x = \beta$

$\beta = \alpha_x$ [Reflexivity]

$\beta = \alpha_y \to \alpha_y$ [Transitivity]

Substitution:

$\alpha_x = \alpha_y \to \alpha_y$

$\beta = \alpha_y \to \alpha_y$

Example – Type Inference

$x: \alpha_y \to \alpha_y \quad \vdash x: \alpha_y \to \alpha_y$

$\vdash \lambda x.x : (\alpha_y \to \alpha_y) \to (\alpha_y \to \alpha_y)$

$y: \alpha_y \quad \vdash y: \alpha_y$

$\vdash \lambda y.y: \alpha_y \to \alpha_y$

$\vdash (\lambda x.x)\ (\lambda y.y) : \alpha_y \to \alpha_y$

$?? \notin FV(\emptyset)$

$I: \quad \vdash I : \quad I: \quad \vdash I:$

$I: \quad \vdash I\ I :$

$\vdash \text{let } I = (\lambda x.x)\ (\lambda y.y) \text{ in } I\ I :$

Example – Generalization

$$\dfrac{x: \alpha_y \to \alpha_y \quad \vdash x: \alpha_y \to \alpha_y}{\vdash \lambda x.x : (\alpha_y \to \alpha_y) \to (\alpha_y \to \alpha_y)} \qquad \dfrac{y: \alpha_y \quad \vdash y: \alpha_y}{\vdash \lambda y.y: \alpha_y \to \alpha_y}$$

$$\dfrac{\dfrac{\cdots}{\vdash (\lambda x.x)\ (\lambda y.y) : \alpha_y \to \alpha_y} \quad \alpha_y \notin FV(\emptyset) \qquad \dfrac{I: \forall\alpha.\ \alpha \to \alpha \ \vdash I : \qquad I: \forall\alpha.\ \alpha \to \alpha \vdash I:}{I: \forall\alpha.\ \alpha \to \alpha \ \vdash I\ I :}}{\vdash \text{let } I = (\lambda x.x)\ (\lambda y.y) \text{ in } I\ I :}$$

Example – Type Inference

Next we run type inference on the body of the let.

$$
\dfrac{
\dfrac{
\dfrac{x: \alpha_y \to \alpha_y \quad \vdash x: \alpha_y \to \alpha_y}{\vdash \lambda x.x : (\alpha_y \to \alpha_y) \to (\alpha_y \to \alpha_y)}
\qquad
\dfrac{y: \alpha_y \quad \vdash y: \alpha_y}{\vdash \lambda y.y: \alpha_y \to \alpha_y}
}{\vdash (\lambda x.x)\ (\lambda y.y) : \alpha_y \to \alpha_y}
\qquad \alpha_y \notin FV(\emptyset)
\qquad
\dfrac{I: \forall\alpha.\ \alpha \to \alpha \ \vdash I : \qquad I: \forall\alpha.\ \alpha \to \alpha \vdash I:}{I: \forall\alpha.\ \alpha \to \alpha \ \vdash I\ I :}
}{\vdash \text{let } I = (\lambda x.x)\ (\lambda y.y) \text{ in } I\ I :}
$$

Example – Type Inference

$$\dfrac{\dfrac{\dfrac{x: \alpha_y \to \alpha_y \;\vdash x: \alpha_y \to \alpha_y}{\vdash \lambda x.x : (\alpha_y \to \alpha_y) \to (\alpha_y \to \alpha_y)} \qquad \dfrac{y: \alpha_y \;\vdash y: \alpha_y}{\vdash \lambda y.y: \alpha_y \to \alpha_y}}{\vdash (\lambda x.x)\ (\lambda y.y) : \alpha_y \to \alpha_y \qquad \alpha_y \notin FV(\emptyset)} \qquad \dfrac{I: \forall\alpha.\ \alpha \to \alpha \;\vdash I : \gamma \to \gamma \qquad I: \forall\alpha.\ \alpha \to \alpha \vdash I: \rho \to \rho}{\begin{array}{c} I: \forall\alpha.\ \alpha \to \alpha \;\vdash I\ I : \mu \\ \gamma \to \gamma = (\rho \to \rho) \to \mu \end{array}}}{\vdash \text{let } I = (\lambda x.x)\ (\lambda y.y) \text{ in } I\ I : \mu}$$

Solving the Equations

$\gamma \rightarrow \gamma = (\rho \rightarrow \rho) \rightarrow \mu$

$\gamma = \rho \rightarrow \rho$ [Structure]

$\gamma = \mu$

$\mu = \gamma$ [Reflexivity]

$\mu = \rho \rightarrow \rho$ [Transitivity]

Substitution:

$\gamma = \rho \rightarrow \rho$

$\mu = \rho \rightarrow \rho$

Example – Full Derivation

$$\dfrac{\dfrac{\dfrac{x: \alpha_y \rightarrow \alpha_y \ \vdash x: \alpha_y \rightarrow \alpha_y}{\vdash \lambda x.x : (\alpha_y \rightarrow \alpha_y) \rightarrow (\alpha_y \rightarrow \alpha_y)} \qquad \dfrac{y: \alpha_y \ \vdash y: \alpha_y}{\vdash \lambda y.y: \alpha_y \rightarrow \alpha_y}}{\vdash (\lambda x.x)\ (\lambda y.y) : \alpha_y \rightarrow \alpha_y} \qquad \alpha_y \notin FV(\emptyset) \qquad \dfrac{I: \forall \alpha.\ \alpha \rightarrow \alpha \ \vdash I :(\rho \rightarrow \rho) \rightarrow (\rho \rightarrow \rho) \qquad I: \forall \alpha.\ \alpha \rightarrow \alpha \vdash I: \rho \rightarrow \rho}{I: \forall \alpha.\ \alpha \rightarrow \alpha \ \vdash I\ I : \rho \rightarrow \rho}}{\vdash \text{let } I = (\lambda x.x)\ (\lambda y.y) \text{ in } I\ I : \rho \rightarrow \rho}$$

Summary

Polymorphism allows one to write and use generic functions.

Data types:

Cons: $\forall \alpha.\ \alpha \rightarrow List(\alpha) \rightarrow List(\alpha)$

Nil: $\forall \alpha.\ List(\alpha)$

Higher order functions:

Map: $\forall \alpha, \beta.\ (\alpha \rightarrow \beta) \rightarrow List(\alpha) \rightarrow List(\beta)$

Function composition: $\forall \alpha, \beta, \rho.\ (\alpha \rightarrow \rho) \rightarrow (\rho \rightarrow \beta) \rightarrow (\alpha \rightarrow \beta)$

Discussion

- *Parametric polymorphism* allows functions to be defined once and used at many different types
  - Does not eliminate all cases where code must be duplicated to satisfy the type checker, but it goes a very long way.
- The type inference algorithm produces the most general possible type
  - No better type is possible within the type system
- Considered a major breakthrough when it was discovered in the late 1970's
  - Robin Milner received the Turing Award for this work

Impact

- All typed functional languages use parametric polymorphism
  - ML, Haskell
  - The functional languages also use type inference
- Also the basis of templates/generics in C++ and Java

History

Consider a function type: $A \rightarrow B$

This looks a lot like the syntax for logical implication ...

There is a connection! A type can be read as saying that a computation of type $A \rightarrow B$ is a proof that given something of type A, we can construct something of type B.

These are *constructive logics:* Don't just prove that the thing of type B exists, but actually produce the element of B (using the computation)

Typed vs. Untyped

- Typed languages always rule out some desirable programs
  - Response: Various kinds of polymorphism

- Typed languages require a lot more work (writing types)
  - Response: Type inference

- Typed languages provide a powerful form of program verification, guaranteeing certain behavior for all inputs
  - Response: Maybe we only care about a subset of the inputs, not all inputs

- Bottom line: Modern typed languages cover 95%+ of what you want to write and require only a small amount of extra work
  - But, programmers still need to understand the type system to use it!
  - This is the real cost

Utility

- Polymorphic type inference can make you a better programmer

- Especially when you program in untyped languages!

- If you learn this type discipline, you will find yourself mentally applying it to your own code
  - And making many fewer type errors, even without a type checker
  - Covers > 95% of code people write (excluding objects ...)