Polymorphic Types

CS242

Lecture 5

But First, Back to Data Types

Integers: *N* applies its first argument *N* times to its second argument

$n\ f\ x = f^n(x)$

$0 = \lambda f.\lambda x.x$

$\text{Succ} = \lambda n.\lambda f.\lambda x.\ f\ (n\ f\ x)$

How Does This Work?

$0 = \lambda f.\lambda x.x$

$\text{Succ} = \lambda n.\lambda f.\lambda x.\ f\ (n\ f\ x)$

Note there are two constructors of the data type:

0 (of arity 0)

Succ (of arity 1, taking a number as the argument)

How Does This Work?

$0 = \lambda f.\lambda x.x$

$\text{Succc} = \lambda n.\lambda f.\lambda x.\ f\ (n\ f\ x)$

$$4 = \text{Succ}(\text{Succ}(\text{Succ}(\text{Succ}(0))))$$

How Does This Work?

$0 = \lambda f.\lambda x.x$

$\text{Succ} = \lambda n.\lambda f.\lambda x.\ f\ (n\ f\ x)$

$$4 = \text{Succ}(\text{Succ}(\text{Succ}(\text{Succ}(0))))$$

$$4\ \text{add2}\ \text{one} = \text{add2}(\text{add2}(\text{add2}(\text{add2}(\text{one}))))$$

Takes two arguments of the same arities as succ and 0, and replaces them 1-1 in the data type, producing an expression to be evaluated.

In General

- For an n-constructor data type, each value takes n function arguments, each of the arity of the corresponding constructor.

- Each constructor is replaced by the corresponding function in the structure.

- The resulting expression is evaluated.

Lists

cons = $\lambda h.\lambda t.\lambda f.\lambda x.\ f\ h\ (t\ f\ x)$

- cons is of arity two, taking a head and tail of a list and building a new list
- The resulting list value takes two arguments, one for cons and one for nil

nil = $\lambda f.\lambda x.x$

- Nil is of arity zero – it is a list value
- Like other list values, it takes two argumets, one for cons and one for nil

- Note how case analysis is built into the constructors – they “know” whether they are cons or nil and act accordingly

Lists

cons = $\lambda h.\lambda t.\lambda f.\lambda x.\ f\ h\ (t\ f\ x)$

nil = $\lambda f.\lambda x.x$

Z = cons one (cons one (cons one nil))

Z add 0 = add one (add one (add one zero))

Z ($\lambda h.\lambda t.$cons (succ h) t) nil = cons two (cons two (cons two nil)))

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:

let f = λx.e in e' is equivalent to (λf.e') λx.e

And note we are getting closer to standard syntax:

let f x = e in e' is syntactic sugar for let f = λx.e in e'

Type Rules

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

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

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

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

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

Recall …

The program

let f = λx.x in x x

is untypable, but

(λx.x) (λy.y)

is typable (in simply typed lambda calculus)

Polymorphic Types

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

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

$o \rightarrow \forall \alpha.o \mid t$

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

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

Example

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

Multiple Type Variables

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

$FV(A, x{:}t) = FV(A) \cup FV(t)$
$FV(\emptyset) = \emptyset$
$FV(int) = \emptyset$
$F(t \rightarrow t') = FV(t) \cup FV(t')$
$FV(\forall \alpha_1,\ldots,\alpha_n.t) = FV(t) - \{\alpha_1,\ldots,\alpha_n\}$
$FV(\alpha) = \{\alpha\}$

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

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

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

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

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

Example – Type Inference

$x: \quad \vdash x:$ $\qquad$ $y: \quad \vdash y:$

$\vdash \lambda x.x :$ $\qquad$ $\vdash \lambda y.y:$

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

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

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

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

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

Example – Type Inference

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

Example – Type Inference

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

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

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

Solving the Equations

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

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

$\alpha_x = \beta$

$\beta = \alpha_x$ [Reflexivity]

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

Substitution:

$\alpha_x = \alpha_y \rightarrow \alpha_y$

$\beta = \alpha_y \rightarrow \alpha_y$

Example – Type Inference

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

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

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

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

$I: \quad \vdash I :$

$I: \quad \vdash I:$

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

$?? \notin FV(\emptyset)$

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

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

Example – Generalization

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

Example – Type Inference

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

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

Example – Type Inference

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

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

$$\dfrac{I : \forall\alpha.\ \alpha \to \alpha \;\vdash\; I : \gamma \to \gamma \qquad I : \forall\alpha.\ \alpha \to \alpha \;\vdash\; I : \rho \to \rho}{I : \forall\alpha.\ \alpha \to \alpha \;\vdash\; I\ I : \mu}$$

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

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

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

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

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: : $A \to B$

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 $A \to B$ 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 certain 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 them!
  - 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 ...)