Polymorphism

Tikhon Jelvis (tikhon@jelv.is)

February 13, 2014

Untyped -Calculus

- model computation with functions
- simple structure:

$$
\begin{aligned}
e ::= \; & x && \text{variable} \\
\mid \; & \lambda x.e && \text{abstraction} \\
\mid \; & e_1 e_2 && \text{application}
\end{aligned}
$$

-Calculus Evaluation

- key idea: application by substitution

$$(\lambda x.e)s \Rightarrow [s/x]e$$

- $[s/x]e$ = "replace x with s in e"
- handy mnemonic (thanks Sergei): multiplying by $\frac{s}{x}$ and canceling
- remember to worry about "capturing"

Simple Types

- extend -calculus with **types**
- base types
  - **unit**, **int**... etc
- function types
  - **int** $\rightarrow$ **int**
  - (**unit** $\rightarrow$ **unit**) $\rightarrow$ **int** $\rightarrow$ **int**

Syntax: Terms and Types

$\tau ::= \mathbf{unit}$	unit type
$\mid \quad \tau_1 \to \tau_2$	function types
$e ::= ()$	unit value
$\mid \quad x$	variable
$\mid \quad \lambda x : \tau . e$	abstraction
$\mid \quad e_1 \, e_2$	application

Typing Rules

- functions:

$$\frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash (\lambda x : \tau . e) : \tau \rightarrow \tau'}$$

- application:

$$\frac{\Gamma \vdash e_1 : \tau \rightarrow \tau' \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 e_2 : \tau'}$$

Problem: Repetition

- every type has **an** identity function:

$$\lambda x : \tau . x$$

- **different** for every possible τ

$$id_{\textbf{unit}}, id_{\textbf{int}}, id_{\textbf{int} \rightarrow \textbf{int}} \dots$$

- a single term can have **multiple** incompatible types

Solution: System F

- add polymorphism to our types
  - types parameterized by other types
- what is a "parameterized term"x?
  - abstraction (function)
- so: function **for** types
- *id* would take a type τ and give you id_τ

New Syntax

$$
\begin{aligned}
\tau ::= \ & \textbf{unit} && \text{unit type} \\
\mid \ & \alpha && \text{type variable} \\
\mid \ & \tau_1 \to \tau_2 && \text{function types} \\
\mid \ & \forall \alpha . \tau && \text{type quantification} \\
e ::= \ & () && \text{unit value} \\
\mid \ & x && \text{variable} \\
\mid \ & \lambda x : \tau . e && \text{abstraction} \\
\mid \ & e_1 e_2 && \text{application} \\
\mid \ & \Lambda \alpha . e && \text{type abstraction} \\
\mid \ & e_1 [\tau] && \text{type application}
\end{aligned}
$$

Type Variables

- behave mostly like value-level variables
- type variables can be **free** or **bound**
  - free variables are not defined inside expression
- **substitute** types for type variables:
  - $[\sigma/\alpha]\tau$ means "replace α with σ in type τ"

Evaluation

- simply typed -calculus—just like untyped:

$$(\lambda x : \tau . e)s \Rightarrow [s/x]e$$

- one more rule, for type abstractions:

$$(\Lambda \alpha . e)[\tau] \Rightarrow [\tau/\alpha]e$$

- **type-level** version of the first rule
- reduction is still **very simple**

Typing Rules

- Γ now covers both type and term variables
- basic rules just like STLC
- new rules:

$$\frac{\Gamma, \alpha \vdash x : \tau}{\Gamma \vdash \Lambda\alpha.x : \forall\alpha.\tau}$$

$$\frac{\Gamma \vdash x : \forall\alpha.\tau}{\Gamma \vdash x[\sigma] : ([\sigma/\alpha]\tau)}$$

- compare to normal abstraction and application

Running Example: id

- function:

$$id : \forall\alpha.\alpha \to \alpha$$
$$id = \Lambda\alpha.\lambda(x : \alpha).x$$

- reduction:

$$\begin{aligned} & (\Lambda\alpha.\lambda(x : \alpha).x)[\textbf{unit}]() \\ \Rightarrow & (\lambda(x : \textbf{unit}).x)() \\ \Rightarrow & () \end{aligned}$$

Another Example: app

- Untyped term, impossible in STLC:

$$\lambda f.\lambda x.fx$$

- we can type function application:

$$app : \forall\alpha.\forall\beta.(\alpha \to \beta) \to \alpha \to \beta$$
$$app = \Lambda\alpha.\Lambda\beta.\lambda(f : \alpha \to \beta).\lambda(x : \alpha).fx$$

  - Haskell $, OCaml <|: really just *id* with restricted type

Interesting Example: self application

- We cannot even **express** self-application in STLC

$$\lambda f.ff$$

- but we **can** with polymorphism:

$$\begin{aligned} \mathit{self} &: (\forall\alpha.\alpha \to \alpha) \to (\forall\beta.\beta \to \beta) \\ \mathit{self} &= \lambda(f : \forall\alpha.\alpha \to \alpha).f[\forall\beta.\beta \to \beta]f \end{aligned}$$

- however, still no infinite loops

Data Structures

- consider untyped booeans:

$$\begin{aligned} true &= \lambda x.\lambda y.x \\ false &= \lambda x.\lambda y.y \end{aligned}$$

- typed version:

$$\begin{aligned} true, false &: \forall\alpha.\alpha \to \alpha \to \alpha \\ true &= \Lambda\alpha.\lambda(x:\alpha).\lambda(y:\alpha).x \\ false &= \Lambda\alpha.\lambda(x:\alpha).\lambda(y:\alpha).y \end{aligned}$$

- types prevent malformed "booleans"

Products

- easy in untyped ; added to STLC explicitly:

$$\begin{aligned} \sigma \times \tau &: \forall\alpha.(\sigma \to \tau \to \alpha) \to \alpha \\ \langle s, t \rangle &= \Lambda\alpha.\lambda(f : \sigma \to \tau \to \alpha).fst \\ fst &: \sigma \times \tau \to \sigma \\ fst &= \lambda(p : \sigma \times \tau).p[\sigma](\lambda s : \sigma.\lambda t : \tau.s) \end{aligned}$$

- we can do sum types similarly

Type Inference

- this is a handy system
- unfortunately, **type inference is undecideable**
- we can make type inferrable with a simple restriction:
  - **prenex form**: all quantifiers at the front
  - types where all foralls are left of parentheses
- Haskell, ML... etc do this

Hindley-Milner

- important insight: **most general type**
- every untyped term has a **unique** most general type

$$\lambda x.x : \forall \alpha . \alpha \to \alpha$$

- we can easily model this with logic programming
  - faster algorithms exist as well

Curry-Howard

- System F maps to 2nd-order logic
  - quantifiers **only** over predicates
- predicate logic with $\forall$ but no “domains”
  - no external sets to quantify over
- consider: Λ defines a function from types to values
  - but not vice-versa

Experimenting

- Standard Haskell, ML… etc: prenex form
- Haskell with `RankNTypes`: everything we've covered
  - along with recursion and recursive types
- OCaml can also do the equivalent of `RankNTypes` but awkwardly