Adding Types

Tikhon Jelvis (tikhon@jelv.is)

December 13, 2013

Untyped Lambda Calculus

- simple model of functions
- few parts:

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

Untyped Lambda Calculus

- simple evaluation
- just function application!

$$\frac{(\lambda x.e)e'}{[e'/x]e}$$

- replace x with the argument in the body

The Song that Never Ends

- Lambda calculus is Turing-complete (Church-Turing thesis)
- infinite loops:

$$(\lambda x.xx)(\lambda y.yy) \Rightarrow (\lambda y.yy)(\lambda y.yy)$$

- good for programming, bad for logic

Preventing Self-Application

- problem: self-application
  - xx leads to infinite loops
- we need a rule to **prevent** self-application (and infinite loops in general)
  - simple
  - syntactic
  - static
- **conservative** by necessity

Why?

- helps lambda calculus as a logic
- provides **simple** model of real type systems
- helps design new types and type systems
- usual advantages of static typing

Base Types

- start with some "base" types (like axioms)
  - ints, booleans... whatever
- even just the *unit* type is fine
- base types have values:
  - () is of type *unit*
  - 1 is of type *int*
- ultimately, the exact base types don't matter

Function Types

- one type constructor: $\rightarrow$ (like axiom schema)
- represents function types
- $unit \rightarrow unit$
- $int \rightarrow unit \rightarrow int$
- values are functions

Assigning Types

- we need some way to give a type to an expression
- **only** depends on the static syntax
- **typing judgement**: $x : \tau$

Context

- **depends** on what's in scope (typing context):

$$\Gamma \vdash x : \tau$$

- things in scope: "context", Γ
- set of typing judgements for **free variables**:

$$\Gamma = \{x : \tau, y : \tau \to \tau, ...\}$$

New Syntax

$$
\begin{aligned}
\tau ::= \; & \mathit{unit} && \text{unit type} \\
\mid \; & \mathit{int} && \text{int type} \\
\mid \; & \tau_1 \to \tau_2 && \text{function types}
\end{aligned}
$$

- $$
\begin{aligned}
e ::= \; & () && \text{unit value} \\
\mid \; & n && \text{integer} \\
\mid \; & e_1 + e_2 && \text{arithmetic} \\
\mid \; & x && \text{variable} \\
\mid \; & \lambda x : \tau . e && \text{abstraction} \\
\mid \; & e_1 \, e_2 && \text{application}
\end{aligned}
$$

Typing Rules

- we can assign types following a few "typing rules"
- idea: if we see expression "x", we know "y"
- just like implication in logic

$$\frac{\text{condition}}{\text{result}}$$

- remember the context matters: Γ

Base rules

- note: **no** prerequisites!

$$\frac{}{\Gamma \vdash n : \mathit{int}}$$

$$\frac{}{\Gamma \vdash () : \mathit{unit}}$$

- **base cases** for recursion

Main Rules

- contexts:

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}$$

- function bodies:

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

Main Rules

- application:

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

- **recursive cases** in the type system
- think of a function over syntactic terms
  - similar to evaluation!

Domain-Specific Rules

- we add rules for our "primitive" operations

$$\frac{\Gamma \vdash e_1 : int \quad \Gamma \vdash e_2 : int}{\Gamma \vdash e_1 + e_2 : int}$$

- imagine other base types like booleans

$$\frac{\Gamma \vdash c : bool \quad \Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash c e_1 e_2 : \tau}$$

- easy to extend

No Polymorphsim

- we do not have any notion of polymorphism
- function arguments **have** to be annotated
- untyped: $\lambda x.x$
- typed:
  - $\lambda x : unit.x$
  - $\lambda x : int.x$
  - $\lambda x : int \to unit \to int.x$

Numbers

- remember numbers as repeated application
- untyped:
  - 0: $\lambda f.\lambda x.x$
  - 1: $\lambda f.\lambda x.fx$
  - 2: $\lambda f.\lambda x.f(fx)$
  - 3: $\lambda f.\lambda x.f(f(fx))$

Typed Numbers

- we can add types:
  - 0: $\lambda f : unit \rightarrow unit.\lambda x : unit.x$
  - 1: $\lambda f : unit \rightarrow unit.\lambda x : unit.fx$
  - 2: $\lambda f : unit \rightarrow unit.\lambda x : unit.f(fx)$
  - 3: $\lambda f : unit \rightarrow unit.\lambda x : unit.f(f(fx))$
- numbers: $(unit \rightarrow unit) \rightarrow unit \rightarrow unit$

Pairs

- remember pair encoding:
  - cons: $\lambda x.\lambda y.\lambda f.fxy$
  - first: $\lambda x.\lambda y.x$
  - second: $\lambda x.\lambda y.y$
- lets us build up data types, like lisp

Typed Pairs

- cons:
$$\lambda x : \tau.\lambda y : \tau.\lambda(f : \tau \to \tau \to \tau).fxy$$
- but we want pairs of **different** types!
- we should add pairs (“product types”) to our system

Product Types

- new type syntax: $\tau_1 \times \tau_2$
- like Haskell's `(a, b)` or OCaml's `a * b`
- constructor:

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2}$$

Product Types

- accessors (first and second):

$$\frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \mathsf{first}\ e : \tau_1}$$

$$\frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \mathsf{second}\ e : \tau_2}$$

Sum Types

- sum types: disjoint/tagged unions, variants
- like Haskell's `Either`
- new type syntax: $\tau_1 + \tau_2$
- construction:

$$\frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \text{left } e : \tau_1 + \tau_2}$$

$$\frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \text{right } e : \tau_1 + \tau_2}$$

Sum Types

- matching

$$\frac{\Gamma \;\vdash\; e : \tau_1 + \tau_2 \quad \Gamma,\; x : \tau_1 \;\vdash\; e_1 : \tau' \quad \Gamma,\; y : \tau_2 \;\vdash\; e_2 : \tau'}{\Gamma \vdash (e x e_1 y e_2) : \tau'}$$

Algebraic Data Types

- this basically gives us algebraic data types
- now we just need **recursive types** and **polymorphism**