Dependent Types

Tikhon Jelvis (tikhon@jelv.is)

February 13, 2014

Untyped lambda-calculus

- terms: functions (λ), variables, application

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

- evaluation via substitution:

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

Simply typed lambda-calculus

- extend untyped lambda-calculus with “simple” types
- simple: atomic “base” types + functions
  - base: **unit**, **int**, . . .
  - function: **unit** $\rightarrow$ **unit**, . . .
- not Turing-complete any more
  - safety at the expense of expressiveness

Simply typed lambda-calculus

- new type-level syntax

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

- evaluation is unchanged

System F

- parametric polymorphism
- allow "families" of simply typed terms
  - polymorphic *id* is a family of $id_{\mathbf{unit}}, id_{\mathbf{int}}, id_{\mathbf{int} \to \mathbf{int}}, \ldots$
- a "family" is a function from types to values

System F

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

Type abstractions

- type functions: $\Lambda\alpha.\tau$
- qualified types: $\forall\alpha.\tau$
- type application: $e_1[\tau]$
- example:

$$
\begin{aligned}
id &: \forall\alpha.\alpha \to \alpha \\
id &= \Lambda\alpha.\lambda(x : \alpha).x
\end{aligned}
$$

Type abstractions

- hey, these are like normal λ s, but **simpler**
- functions: $\lambda(x : \tau).e$
- function types: $\tau_1 \to \tau_2$
- function application: $e_1\ e_2$

Unifying types and terms

- let's combine these two **similar** constructs
- **everything is a term**
  - values can appear in types
  - no more clean type/value separation

Why?

- System F: repetition over different **types**
  - $id_{\mathbf{unit}}, id_{\mathbf{int}}, id_{\mathbf{int} \to \mathbf{int}}, \ldots$
- consider other repetitive patterns:
  - $\mathbf{int}, \mathbf{int} \times \mathbf{int}, \mathbf{int} \times \mathbf{int} \times \mathbf{int}, \ldots$
  - $\mathbf{vector}\ 1, \mathbf{vector}\ 2, \mathbf{vector}\ 3, \ldots$
- we would love to write

$$dot : \mathbf{vector}\ n \times \mathbf{vector}\ n \to int$$

Dependent types

- no type syntax: e and τ are **expressions**

$$
\begin{array}{rcll}
e, \tau & ::= & () & \text{unit value} \\
& | & \textbf{unit} & \text{unit type} \\
& | & \star & \text{type of types} \\
& | & x & \text{variable} \\
& | & \forall (x : \tau).\tau' & \text{dependent function} \\
& | & e_1 \; e_2 & \text{application} \\
& | & \lambda (x : \tau).e & \text{abstraction}
\end{array}
$$

- $\forall (x : \tau).\tau'$ replaces function arrows $(\to)$

Typing rules: types of types

- What type does $\star$ have?
  - simple (but dubious) answer: $\star : \star$
  - more interesting: infinite hierarchy $\star_1 : \star_2 : \star_3 : \ldots$
- let's go with simplicity:

$$\frac{}{\Gamma \vdash \star : \star}$$

Typing rules: dependent abstractions

- dependent abstractions are terms too:

$$\frac{\Gamma \vdash \tau : \star \quad \Gamma, x : \tau \vdash \tau' : \star}{\Gamma \vdash (\forall (x : \tau).\tau') : \star}$$

- key idea: τ' can **depend on** x
- $\forall (x : \tau).\tau'$ is like a function type $\tau \to \tau'$ except also parametrized by a value x
  - alternative syntax: $(x : \tau) \to \tau'$

Typing rules: application

- remember: the new function type is a $\forall$:

$$\frac{\Gamma \vdash e_1 : \forall(x : \tau).\tau' \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1\ e_2 : [e_2/x]\tau'}$$

- substitution happening at the **type level**
  - basically like enabling the type system with evaluation

Typing rules: type equivalence

- since types are terms, we need to evaluate them to check for equivalence
- read $e_1 \Downarrow e_2$ as "e_1 evaluates to e_2"

$$\frac{\Gamma \vdash e : \tau_1 \quad \tau_1 \Downarrow \tau \quad \tau_2 \Downarrow \tau}{\Gamma \vdash e : \tau_2}$$

- consider: **vector** 5 vs **vector** $(2+3)$ vs **vector** $(3+2)$
- again: evaluation in type checking

Typing rules: abstractions

- pretty straightforward: $\rightarrow$ just becomes $\forall$:

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

- note: x can occur in body **and** type

Example: id

- from System F:

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

- with dependent types:

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

- the System F Λ has become a normal λ !

Nicer notation

- often, we just want normal functions
  - **ignore** argument x
- special syntax: $\rightarrow$
- consider *id*: $\forall(\alpha : \star).\alpha \rightarrow \alpha$
- no extra name (x) introduced

Example: vectors

- assume built-in numbers: $\mathbb{N} : \star$ and $n : \mathbb{N}$
- vectors indexed by length:

$$\forall(\alpha : \star).\forall(n : \mathbb{N}).\mathbf{vec}\ \alpha\ n$$

- constructors:

$$\begin{aligned}
&\mathit{Nil} : \forall(\alpha : \star).\mathbf{vec}\ \alpha\ 0\\
&\mathit{Cons} : \forall(\alpha : \star).\forall(n : \mathbb{N}).\alpha \rightarrow\\
&\mathbf{vec}\ \alpha\ n \rightarrow \mathbf{vec}\ \alpha\ (n+1)
\end{aligned}$$

Example: zero vector

- we can have sized vectors in Haskell
  - numbers at the type level—redundant
- however, Haskell can't do this:

$$\mathit{zero} : \forall(n : \mathbb{N}).\mathbf{vec\ int}\ n$$

- creates a vector of length n, full of 0 s
- argument n is part of result type!