Second-order and dependent type systems

Jacques Garrigue, 2009/6/11

Simple types are not enough to enjoy the ful relation withl expressiveness of λ-calculus. From the point of view of logic, they restrict us to propositional logic.

By extending the type system, we can recover more expressiveness, and find relations with stronger logics.

1 Second order λ-calculus: $\lambda 2$

If we look at encodings using in untyped λ-terms, a common pattern appears.

$$\begin{array}{l}
\mathsf{t} = \lambda x.\lambda y.x \\
\mathsf{f} = \lambda x.\lambda y.y \\
\mathsf{pair}\ a\ b = \lambda s.(sab) \\
\mathsf{fst} = \lambda p.p\ \mathsf{t} \\
\mathsf{snd} = \lambda p.p\ \mathsf{f}
\end{array}$$

In all these examples, it is esssential that the type of the parameters is not fixed. This works in an untyped context, but not with simple types. The fact that the same function (λ-term) can be applied to values of different types is called *polymorphism*.

In the second orde λ-calculus, we add polymorphic types.

$$\begin{array}{rcll}
t & ::= & \alpha & \text{type variable} \\
 & | & \forall \alpha.t & \text{polymorphic type} \\
 & | & t \to t & \text{function type} \\
M & ::= & x \mid \lambda x{:}t.M \mid (MM) & \\
 & | & \Lambda \tau.M & \text{type abstraction} \\
 & | & M[t] & \text{type application}
\end{array}$$

The following typing rules are needed.

$$\textbf{Tabs}\quad \frac{\Gamma, \alpha : * \vdash M : \tau}{\Gamma \vdash \Lambda\alpha.M : \forall\alpha.\tau}$$

$$\textbf{Tapp}\quad \frac{\Gamma \vdash M : \forall\alpha.\tau}{\Gamma \vdash M[t] : [t/\alpha]\tau}$$

The terms above can be typed in the following way.

$$\begin{array}{ll}
\mathsf{Bool} = \forall\alpha.\alpha \to \alpha \to \alpha & \\
\vdash \mathsf{t} = \Lambda\alpha.\lambda x{:}\alpha.\lambda y{:}\alpha.x & : \mathsf{Bool} \\
\vdash \mathsf{f} = \Lambda\alpha.\lambda x{:}\alpha.\lambda y{:}\alpha.y & : \mathsf{Bool}
\end{array}$$

$$\begin{array}{ll}
\mathsf{Pair}[t_1,t_2] = \forall\gamma.(t_1 \to t_2 \to \gamma) \to \gamma & \\
\vdash \mathsf{pair} = \Lambda\alpha.\Lambda\beta.\lambda a{:}\alpha.\lambda b{:}\alpha.\Lambda\gamma.\lambda s{:}\alpha \to \beta \to \gamma.(s\ a\ b) & : \forall\alpha.\forall\beta.\alpha \to \beta \to \mathsf{Pair}[\alpha,\beta] \\
\vdash \mathsf{fst} = \Lambda\alpha.\Lambda\beta.\lambda p{:}\mathsf{Pair}[\alpha,\beta].p[\alpha]\ (\lambda x{:}\alpha.\lambda y{:}\beta.x) & : \forall\alpha.\forall\beta.\mathsf{Pair}[\alpha,\beta] \to \alpha \\
\vdash \mathsf{snd} = \Lambda\alpha.\Lambda\beta.\lambda p{:}\mathsf{Pair}[\alpha,\beta].p[\beta]\ (\lambda x{:}\alpha.\lambda y{:}\beta.y) & : \forall\alpha.\forall\beta.\mathsf{Pair}[\alpha,\beta] \to \beta
\end{array}$$

Church numerals can be typed too.

$$\begin{array}{ll}
\mathsf{Nat} = \forall\alpha.(\alpha \to \alpha) \to \alpha \to \alpha & \\
\vdash \mathsf{c}_n = \Lambda\alpha.\lambda f : \alpha \to \alpha.\lambda x : \alpha.f^n x & : \mathsf{Nat} \\
\vdash \mathsf{c}_+ = \lambda m : \mathsf{Nat}.\lambda n : \mathsf{Nat}.\Lambda\alpha.\lambda f : \alpha \to \alpha.\lambda x : \alpha.(m[\alpha]\ x\ (n[\alpha]\ f\ x)) & : \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat} \\
\vdash \mathsf{c}_\times = \lambda m : \mathsf{Nat}.\lambda n : \mathsf{Nat}.\Lambda\alpha.\lambda f : \alpha \to \alpha.(m[\alpha]\ (n[\alpha]\ f)) & : \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat} \\
\vdash \mathsf{c}_{\mathsf{exp}} = \lambda m : \mathsf{Nat}.\lambda n : \mathsf{Nat}.\Lambda\alpha.n[\alpha \to \alpha]\ (m[\alpha]) & : \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat}
\end{array}$$

Through this small extension, expressiveness is considerably increased. However, termination can still be proved for second-order λ-calculus, meaning that Y cannot be encode. Conversely, if we add only Y as a δ-rule, second-order λ-calculus becomes universal.

2 Predicate logic and λP

The second-order λ-calculus is more expressive for computations, but from the point of view of logic it is still limited to predicate logic.

In order to encode predicate logic, we need a different kind of extension.

Here are the terms and propositions of predicate logic.

$$\begin{array}{rcll}
t & ::= & x \mid a \mid f(t,\ldots) & \text{terms} \\
A & ::= & \bot \mid A \to A \mid A \wedge A \mid A \vee A & \text{propositions} \\
 & \mid & p(t,\ldots) & \text{predicate} \\
 & \mid & \forall x.A & \text{universal quantifier} \\
 & \mid & \exists x.A & \text{existential quantifier}
\end{array}$$

Propositions look like polymorphic types, however they are quantified on terms rather than types.

The dependently-typed λ-calculus, or λP, extends simple types with quantification over terms.

$$\begin{array}{rcll}
t & ::= & b \mid t \times t & \text{types} \\
 & \mid & \bot & \text{contradiction} \\
 & \mid & p_t\ M & \text{predicate} \\
 & \mid & \Pi x{:}t.t & \text{dependent function} \\
M & ::= & x \mid c_t \mid \lambda x{:}t.M \mid (MM) \mid (M,M) & \text{terms}
\end{array}$$

When x does not occur in t_2, the dependent function type $\Pi x{:}t_1.t_2$ can be written $t_1 \to t_2$.

The typing rules are extended and modified. In particular we need to check the well-formedness of types.

$$\textbf{Type}\quad \frac{\Gamma \vdash M : t}{\Gamma \vdash p_t\ M \text{ ok}} \qquad \frac{\Gamma, x : t \vdash t' \text{ ok}}{\Gamma \vdash \Pi x{:}t.t' \text{ ok}} \qquad \frac{\Gamma, x : t \vdash t' \text{ ok}}{\Gamma \vdash \Sigma x{:}t.t' \text{ ok}}$$

$$\textbf{Abs}\quad \frac{\Gamma, x : t \vdash M : t'}{\Gamma \vdash \lambda x{:}t.M : \Pi x{:}t.t'}$$

$$\textbf{App}\quad \frac{\Gamma \vdash M : \Pi x{:}t.t' \quad \Gamma \vdash N : t}{\Gamma \vdash (M\ N) : [N/x]t'}$$

$$\textbf{Tsubs}\quad \frac{\Gamma \vdash M : [N/x]t \quad N =_{\beta\delta} N'}{\Gamma \vdash M : [N'/x]t}$$

$$\textbf{Neg}\quad \frac{\Gamma \vdash M : \bot \quad \Gamma \vdash t \text{ ok}}{\Gamma \vdash M : t}$$

The above definitions are sufficient to encode predicate logic. Actually, this calculus is more expressive than predicate logic, but there is an injective morphism from predicate logic to λP.

For instance, here is the type encoding the proposition "Humans are mortal, Socrates is a human, so Socrates is mortal".

$$(\mathit{\Pi x{:}Name.Human\ x \rightarrow Mortal\ x}) \rightarrow \mathit{Human\ Socrates} \rightarrow \mathit{Mortal\ Socrates}$$

A proof of this proposition is encoded by the following term.

$$\lambda \mathit{mortal}{:}(\mathit{\Pi x{:}Name.Human\ x \rightarrow Mortal\ x}).\lambda \mathit{human}{:}(\mathit{Human\ Socrates}).\mathit{mortal\ Socrates\ human}$$

3 λP and theorem proving

Not only can we encode propositions and proofs in λP, but the tools of λ-calculus, such as δ rules, are useful in making proofs simpler.

The proposition "$\forall x.x + x = 2 \times x$" can be encoded as follows.

$$\Pi x{:}\mathsf{Nat}.\mathit{eqnat}(\mathit{add}\ x\ x, \mathit{mult}\ 2\ x)$$

Assuming the following δ-rules,

$$\begin{aligned}&\mathit{add}\ 0\ n \rightarrow n\\ &\mathit{add}\ (sm)\ n \rightarrow s\ (\mathit{add}\ m\ n)\\ &\mathit{mult}\ 0\ n \rightarrow 0\\ &\mathit{mult}\ (sm)\ n \rightarrow \mathit{add}\ n\ (\mathit{mult}\ m\ n)\end{aligned}$$

and these exta axioms,

$$\begin{aligned}&\mathit{add\_sym} : \Pi m{:}\mathsf{Nat}.\Pi n{:}\mathsf{Nat}.\mathit{eqnat}(\mathit{add}\ m\ n, \mathit{add}\ n\ m)\\ &\mathit{eq\_sub} : \Pi f{:}(\mathsf{Nat} \rightarrow \mathsf{Nat}).\Pi m{:}\mathsf{Nat}.\Pi n{:}\mathsf{Nat}.\mathit{eqnat}(m, n) \rightarrow \mathit{eqnat}(f\ m, f\ n)\end{aligned}$$

here is its proof term.

$$\lambda x{:}\mathsf{Nat}.\mathit{eq\_sub}\ (\lambda y{:}\mathsf{Nat}.\mathit{add}\ x\ y)\ (\mathit{add\_sym}\ 0\ x)$$

λP can encode predicate logic, but it lacks in polymorphism. By combining λP and λ2, we gain in expressiveness, and this type system provides a foundation for type-theoretic theorem provers.