System F

Jacques Garrigue, 2018/06/05

The simply-typed λ-calculus, invented by Church in 1941, was the first type discipline for λ-calculus. However, it is rather weak, both as a λ-calculus (it cannot even encode Church numerals!), and as a logic (it is limited to propositional logic).

It was followed in 1958 by Gödel's *System T*, which was actually rather based on combinatory logic, and mostly concerned by providing a computational interpretation of intuitionistic predicate logic.

System F was introduced in 1971 by Girard, as a purer and more expressive extension of λ-calculus. It follows some ideas of Martin-Löf's Type Theory (another follower of Gödel), solving some of its technical problems. Interestingly, the same system was discovered independently by Reynolds, who called it *Second-order lambda-calculus*.

1 Types and terms

Simple types are extended with variables and universal quantification.

$$\begin{array}{rcll} T & ::= & T \to T & \text{function} \\ & | & \alpha & \text{type variable} \\ & | & \forall\alpha.T & \text{polymorphic type} \end{array}$$

Note that, thank to extra expressiveness, many types, including natural numbers and the Cartesian product, can be encoded just using functions and quantification.

Terms are extended with type abstraction and application.

$$\begin{array}{rcll} M & ::= & x \mid \lambda x : T.M \mid M\ M & \\ & | & \Lambda\alpha.M & \text{type abstraction} \\ & | & M[T] & \text{type application} \end{array}$$

β-reduction is defined both for term and type application:

$$\begin{array}{rcl} (\lambda x : T.M)\ N & \to & [N/x]M \\ (\Lambda\alpha.M)[T] & \to & [T/\alpha]M \end{array}$$

where we need to define type substitution:

$$\begin{array}{rcll} [T/\alpha]\alpha & = & T & \\ [T/\alpha]\beta & = & \beta & \alpha \neq \beta \\ [T/\alpha](T_1 \to T_2) & = & [T/\alpha]T_1 \to [T/\alpha]T_2 & \\ [T/\alpha]\forall\beta.T_1 & = & \forall\beta.[T/\alpha]T_1 & \beta \notin \mathsf{ftv}(T) \cup \{\alpha\} \\ [T/\alpha](\lambda x : T_1.M) & = & \lambda x : [T/\alpha]T_1.[T/\alpha]M & \\ [T/\alpha](\Lambda\beta.M) & = & \Lambda\beta.[T/\alpha]M & \beta \notin \mathsf{ftv}(T) \cup \{\alpha\} \\ [T/\alpha](M[T_1]) & = & ([T/\alpha]M)[[T/\alpha]T_1] & \\ & \cdots & & \end{array}$$

Renaming of bound variables (α-conversion) also applies to $\forall\alpha$ and $\Lambda\alpha$.

2 Type derivation

The shape of typing judgments do not change.

$$\Gamma \vdash M : T$$

We just need new rules for the new constructs.

$$\textbf{Var} \qquad \Gamma \vdash x : T \quad (x : T \in \Gamma)$$

$$\textbf{Abs} \qquad \frac{\Gamma, x : T_1 \vdash M : T_2}{\Gamma \vdash \lambda x{:}T.M : T_1 \to T_2}$$

$$\textbf{App} \qquad \frac{\Gamma \vdash M : T_2 \to T_1 \quad \Gamma \vdash N : T_2}{\Gamma \vdash M\ N : T_1}$$

$$\textbf{TAbs} \qquad \frac{\Gamma \vdash M : T}{\Gamma \vdash \Lambda\alpha.M : \forall\alpha.T}\ (\alpha \notin \mathsf{ftv}(\Gamma))$$

$$\textbf{TApp} \qquad \frac{\Gamma \vdash M : \forall\alpha.T}{\Gamma \vdash M[T_1] : [T_1/\alpha]T}$$

Here $\mathsf{ftv}(T)$ is defined on types in a way similar as $\mathsf{fv}(M)$ was defined on terms:

$$\begin{aligned} \mathsf{ftv}(T_1 \to T_2) &= \mathsf{ftv}(T_1) \cup \mathsf{ftv}(T_2) \\ \mathsf{ftv}(\alpha) &= \{\alpha\} \\ \mathsf{ftv}(\forall\alpha.T) &= \mathsf{ftv}(T) \setminus \{\alpha\} \\ \mathsf{ftv}(x_1 : T_1, \ldots, x_n : T_n) &= \textstyle\bigcup_{i=1}^{n} \mathsf{ftv}(T_i) \end{aligned}$$

The condition $\alpha \notin \mathsf{ftv}(\Gamma)$ in **TAbs** ensures that in $\Gamma \vdash M : T$, free type variables are shared between Γ, M, and T.

Again, the following theorems can be proved.

Theorem 1 (Confluence) *If $M \to \ldots \to N$ and $M \to \ldots \to P$ then there exists R such that $N \to \ldots \to R$ and $P \to \ldots \to R$.*

Theorem 2 (Subject reduction) *If $\Gamma \vdash M : \tau$ can be derived and $M \to N$, then $\Gamma \vdash N : \tau$ is derivable.*

Theorem 3 (Strong normalization) *If $\Gamma \vdash M : \tau$ is derivable for some Γ and τ, then there is no infinite reduction starting from M.*

3 Computational power

Booleans, Church numerals, and pairs can be directly encoded in System F.

$$\begin{aligned} &\vdash \mathsf{true} = \Lambda\alpha.\lambda x : \alpha.\lambda y : \alpha.x : \mathsf{Bool} \\ &\vdash \mathsf{false} = \Lambda\alpha.\lambda x : \alpha.\lambda y : \alpha.y : \mathsf{Bool} \\ &\vdash \mathsf{not} = \lambda b : \mathsf{Bool}.b[\mathsf{Bool}]\ \mathsf{false}\ \mathsf{true} : \mathsf{Bool} \to \mathsf{Bool} \end{aligned}$$

where $\mathsf{Bool} = \forall\alpha.\alpha \to \alpha \to \alpha$.

$$\vdash \mathsf{c}_n = \Lambda\alpha.\lambda f : \alpha \to \alpha.\lambda x : \alpha.f^n x : \mathsf{Nat}$$
$$\vdash \mathsf{c}_+ = \lambda m : \mathsf{Nat}.\lambda n : \mathsf{Nat}.\Lambda\alpha.\lambda f : \alpha \to \alpha.\lambda x : \alpha.(m[\alpha]\ x\ (n[\alpha]\ f\ x)) : \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat}$$
$$\vdash \mathsf{c}_\times = \lambda m : \mathsf{Nat}.\lambda n : \mathsf{Nat}.\Lambda\alpha.\lambda f : \alpha \to \alpha.(m[\alpha]\ (n[\alpha]\ f)) : \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat}$$
$$\vdash \mathsf{c}_{\mathsf{pow}} = \lambda m : \mathsf{Nat}.\lambda n : \mathsf{Nat}.\Lambda\alpha.n[\alpha \to \alpha]\ (m[\alpha]) : \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat}$$

where $\mathsf{Nat} = \forall\alpha.(\alpha \to \alpha) \to \alpha \to \alpha$.

$$\vdash \mathsf{pair} = \Lambda\alpha.\Lambda\beta.\lambda x : \alpha.\lambda y : \beta.\Lambda\gamma.\lambda f : \alpha \to \beta \to \gamma.f\ x\ y : \forall\alpha.\forall\beta.\alpha \to \beta \to \mathsf{Pair}(\alpha,\beta)$$
$$\vdash \mathsf{fst} = \Lambda\alpha.\Lambda\beta.\lambda p : \mathsf{Pair}(\alpha,\beta).p[\alpha]\ (\lambda x : \alpha.\lambda y : \beta.x) : \forall\alpha.\forall\beta.\mathsf{Pair}(\alpha,\beta) \to \alpha$$
$$\vdash \mathsf{snd} = \Lambda\alpha.\Lambda\beta.\lambda p : \mathsf{Pair}(\alpha,\beta).p[\beta]\ (\lambda x : \alpha.\lambda y : \beta.y) : \forall\alpha.\forall\beta.\mathsf{Pair}(\alpha,\beta) \to \beta$$

where $\mathsf{Pair}(\alpha,\beta) = \forall\gamma.(\alpha \to \beta \to \gamma) \to \gamma$.

These encodings alone show that System F can compute all primitive recursive functions. Actually, thanks to the ability of passing functions as arguments to other functions, it can do even more than that, computing the Ackermann function for instance. However, strong normalization means that Y cannot be encoded, and one cannot define arbitrary recursive functions, or compute arbitrary loops.

Exercise 1 *Write a typed version of* $\mathsf{if0} : \mathsf{Nat} \to \mathsf{Bool}$.

4 Logical view

In the Curry-Howard correspondence, System F is isomorphic to second-order intuitionistic (propositional) logic. That is, a logic in which one can quantify over propositions[1]. It is *impredicative*, meaning that a polymorphic function can be applied to its own polymorphic type.

Second-order logic allows to express many logical connectives with just implication and quantification. In System F, one can directly encode True, False, conjunction and disjunction.

$$\begin{array}{rcl}
\mathsf{True} & = & \forall\alpha.\alpha \to \alpha \\
\mathsf{False} & = & \forall\alpha.\alpha \\
S \wedge T & = & \forall\alpha.(S \to T \to \alpha) \to \alpha \\
S \vee T & = & \forall\alpha.(S \to \alpha) \to (T \to \alpha) \to \alpha \\
 & \vdash & \Lambda\alpha.\lambda x : \alpha.x : \mathsf{True} \\
s : S, t : T & \vdash & \Lambda\alpha.\lambda f : S \to T \to \alpha.f\ s\ t : S \wedge T \\
s : S & \vdash & \Lambda\alpha.\lambda f : S \to \alpha.\lambda g : T \to \alpha.f\ s : S \vee T \\
t : T & \vdash & \Lambda\alpha.\lambda f : S \to \alpha.\lambda g : T \to \alpha.g\ t : S \vee T
\end{array}$$

One cannot build a term of type False, but it is possible to encode **Ex-falso**:

$$\frac{\Delta \vdash \bot}{\Delta \vdash A}$$

by the following derivation

$$\frac{\Gamma \vdash M : \mathsf{False}}{\Gamma \vdash M[A] : A}$$

[1]Not to confuse with first-order predicate logic, which quantifies over terms.