## 1. Definitions

First, we give a brief overview of second order propositional logic. This extends the usual language of propositional logic to also include quantifiers over propositions. Under the BHK interpretation, a construction of $\forall p \varphi(p)$ is a function which, given a proposition $p$, returns a proof of $\varphi(p)$.

In System F, we expand our set of types to include all second order propositional logic formulas which can be built from the connectives $\to$ and $\forall$.

**Definition:** We say $M$ is a **term of type $\boldsymbol{\tau}$** in $\Gamma$, when $M$ can be derived using the following rules.

$$\text{(Var)} \qquad \Gamma, x : \tau \vdash x : \tau$$

$$\text{(Abs)} \qquad \frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash (\lambda x : \sigma\ M) : \sigma \to \tau}$$

$$\text{(App)} \qquad \frac{\Gamma \vdash M : \sigma \to \tau \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash (M\,N) : \tau}$$

$$\text{(U. Abs)} \qquad \frac{\Gamma \vdash M : \sigma}{\Gamma \vdash \Lambda\alpha.M : \forall \alpha\, \sigma} \qquad (\alpha \notin \mathrm{FV}(\tau) \text{ for each } x^\tau \in \mathrm{FV}(M))$$

$$\text{(U. App)} \qquad \frac{\Gamma \vdash M : \forall \alpha\, \sigma}{\Gamma \vdash M\tau : \sigma[\alpha := \tau]} \qquad (\sigma \text{ any type})$$

The first three rules are the same as for simply typed $\lambda$-calculus; the last two (universal abstraction and application) deal with quantification over types. Here, $\alpha$ is any atomic type, called a *type variable.*

The purpose of the restriction in universal abstraction is to ensure that types of terms remain well defined. For instance, consider the term $\Lambda\alpha.x^\alpha$. Recall that the environment $\Gamma$ must assign types to all free variables in expressions on the right of the $\vdash$; but in the above expression the free variable $x$ does not have a well defined type, as $\alpha$ is a type variable which ranges over all types! Note that it is acceptable to form terms such as $\Lambda\alpha\lambda x^\alpha.x^\alpha$, as $x$ is not free in this expression.

We should define precisely what we mean by a free variable:

**Definition:** The set of **free variables** $\mathrm{FV}(M)$ of a term $M$ is the set of all object and type variables free in $M$.

Substitution into object variables is exactly the same as in the simply typed $\lambda$-calculus, with the additional rules

$$(M\tau)[x := P] = M[x := P]\tau$$

$$(\Lambda\alpha.M)[x := P] = \Lambda\alpha.M[x := P] \qquad (\alpha \notin \mathrm{FV}(P))$$

We now also have substitution of *type* variables, which use the following rules:

$$x^\rho[\alpha := \sigma] = x^{\rho[\alpha:=\sigma]}$$

$$(MN)[\alpha := \sigma] = M[\alpha := \sigma]N[\alpha := \sigma]$$

$$(\lambda x^\rho.M)[\alpha := \sigma] = \lambda x^{\rho[\alpha:=\sigma]}.M[\alpha := \sigma]$$

$$(M\tau)[\alpha := \sigma] = M[\alpha := \sigma]\tau[\alpha := \sigma]$$

$$(\Lambda\beta.M)[\alpha := \sigma] = \Lambda\beta.M[\alpha := \sigma] \qquad (\beta \notin \mathrm{FV}(\sigma) \cup \{p\})$$

Likewise, we have two notions of $\beta$-reduction:

$$(\lambda x^\tau.M)N \to_\beta M[x := N] \qquad\qquad (\Lambda\alpha.M)\tau \to_\beta M[\alpha := \tau]$$

As before, the subject reduction theorem holds, and hence well-typed expressions remain well-typed after $\beta$-reduction:

**Theorem:** If $\Gamma \vdash M^\tau$ and $M \twoheadrightarrow_\beta N$ then $\Gamma \vdash N^\tau$.

The proof is essentially the same as the simply typed case, just with more possibilities to check in the induction.

## 2. Basic Theorems

As in simply typed $\lambda$-calculus, we have the strong normalisation and Church-Rosser properties:

**Theorem:** If $M_1 \in \Lambda_{\text{wt}}$ then there exists $M_2 \in \text{NF}$ with $M_1 \twoheadrightarrow_\beta M_2$.

**Theorem:** If $M_1 \twoheadrightarrow_\beta M_2$ and $M_1 \twoheadrightarrow_\beta M_3$ then there exists $M_4 \in \Lambda_{\text{wt}}$ with $M_2 \twoheadrightarrow_\beta M_4$ and $M_3 \twoheadrightarrow_\beta M_4$.

This tells us that normal forms uniquely exist, and hence the equivalence of any two terms in System F is a decidable problem. However, the obvious decision procedure (reduce both terms to their normal forms and compare) has *non-elementary* time complexity; it is not bounded by any tower of exponentials.

The proof for the strong normalisation theorem is based on the corresponding proof for simply typed $\lambda$-calculus, but the naive translation is problematic. Given a term $M^{\forall\alpha\,\alpha}$, we would like to define $M$ to be reducible if and only if $M\tau$ is reducible for all $\tau$. But this definition is circular, as one possible $\tau$ is indeed $\forall\alpha\,\alpha$! The idea is instead to use a method known as *reducibility candidates*, but this is beyond the scope of this talk.

Since System F is strongly normalising, it cannot be Turing complete; indeed it can only express programs which halt. Luckily, this turns out to not be such a problem, as:

**Theorem:** The class of integer functions expressible in System F are exactly the functions provably total in second-order arithmetic.

Here, by *provably total* we mean that second order arithmetic proves the formula which expresses "for all $n$, the program $e$ with input $n$ terminates and returns an integer", where $e$ is an algorithm that represents the function $f$.

The remainder of the talk will be devoted to showing exactly how System F can express functions, integers and other data types.

## 3. Expressibility of System F

Part of the power of System F comes from the fact that we can represent inductive data types in the language. Suppose we wish to model some data type $\rho$ with constructors $f_1, ..., f_n$; that is, $n$ functions which take in some number of inputs, and return an element of the data type. In other words, each $f_i$ is a function of type $\sigma_i = \tau_i^1 \to ... \to \tau_i^{k_i} \to \rho$. Define $\rho$ to be the data type:

$$\rho = \forall\alpha.\sigma_1[\rho := \alpha] \to ... \to \sigma_n[\rho := \alpha] \to \alpha$$

We now give some examples:

**Example 1:** (Church numerals)

The two basic constructors for natural numbers are a constant $Z$ (zero), and a function $S$ from $\omega \to \omega$ (successor). Natural numbers therefore correspond to the type

$$\omega = \forall\alpha.\alpha \to (\alpha \to \alpha) \to \alpha$$

The two constructors may be represented by the $\lambda$-terms:

$$Z = \Lambda\alpha\lambda x^\alpha\lambda y^{\alpha\to\alpha}.x \qquad S = \lambda t^\omega\Lambda\alpha\lambda x^\alpha\lambda y^{\alpha\to\alpha}.y(t\alpha xy)$$

This is entirely analogous to Church numerals for simply typed or untyped $\lambda$-calculus. The integer $n$ is represented by $\overline{n} = \Lambda\alpha\lambda x^\alpha\lambda y^{\alpha\to\alpha}.y(y(...(yx)...))$. Additionally, we have the following:

**Proposition:** The only closed normal terms of type $\omega$ are the Church numerals.

Recall a term is closed if it contains no free variables.

*Proof:* Any closed normal term of type $\omega$ must be in head normal form; that is, of the form $X = \Lambda\alpha\lambda x^\alpha\lambda y^{\alpha\to\alpha}.M$, where $M$ is normal and of type $\alpha$. Since $\alpha$ is a type variable, $M$ cannot be an abstraction. We will prove by induction that $M = y(...(yx)...)$ for some number of $y$.

Suppose for a contradiction that $M = RS$ or $M = R\tau$ where $R \neq y$. Then since $M$ is normal, $R$ cannot be an abstraction; nor can it be a variable since $M$ is well typed. Hence $R$ must be of the form $R'S'$ or $R'\alpha'$. Since $X$ was closed, and the type of $R'$ is more complex than both that of $x$ and $y$, it follows that $X$ must be an abstraction. But this is a contradiction, as then $R$ would be a redex.

We conclude that either $M$ is $x$, or $M = yM'$ for some $M'$; the result now follows by induction (applying the same argument to $M'$ of type $\alpha$).

**Example 2:** (Lists)

Given a type $\tau$, we wish to form the type $L_\tau$, whose objects are finite lists of elements from $\tau$. The two constructors are $N$ of type $L_\tau$ (the constant list) and $C$ of type $\tau \to L_\tau \to L_\tau$ (the function which appends a single element to a given list).

Analogously to the above, we have:

$$\begin{aligned} L_\tau &= \forall\alpha.\alpha \to (\tau \to \alpha \to \alpha) \to \alpha \\ N &= \Lambda\alpha\lambda x^\alpha \lambda y^{\tau\to\alpha\to\alpha}.x \\ C &= \lambda s^\tau \lambda t^{L_\tau} \Lambda\alpha\lambda x^\alpha \lambda y^{\tau\to\alpha\to\alpha}.ys(t\alpha xy) \end{aligned}$$

The list $(s_1, ..., s_n)$ is represented by

$$Cs_1(Cs_2(...(Cs_nN)...)) = \Lambda\alpha\lambda x^\alpha \lambda y^{\tau\to\alpha\to\alpha}.ys_1(ys_2(...(ys_nx)...))$$

It is possible to encode various familiar functions for lists using this definition. For instance, the length function is given by:

$$\text{len} = \lambda l^{L_\tau}\Lambda\alpha\lambda x^\alpha \lambda y^{\alpha\to\alpha}.l\alpha x(\lambda t^\tau.y)$$

We can concatenate two lists by just composing the corresponding $\lambda$-terms in the usual way:

$$\text{concat} = \lambda l^{L_\tau}\lambda m^{L_\tau}\Lambda\alpha\lambda x^\alpha \lambda y^{\tau\to\alpha\to\alpha}.(l\alpha(m\alpha xy)y)$$

It is also possible to define functions such as deletion of the first element, or reversal of a list. Such constructions are not simple; this is not a shortcoming of System F so much as it is a shortcoming of $\lambda$-calculus in general. The difficulties are not unlike those seen when constructing a predecessor function for Church numerals in the untyped $\lambda$-calculus.

**Example 3:** (Binary trees)

Trees can be built inductively from two smaller trees. The constructors are therefore $N : T$ and $C : T \to T \to T$, given by:

$$\begin{aligned} T_\tau &= \forall\alpha.\alpha \to (\alpha \to \alpha \to \alpha) \to \alpha \\ N &= \Lambda\alpha\lambda x^\alpha \lambda y^{\alpha\to\alpha\to\alpha}.x \\ C &= \lambda s^T \lambda t^T \Lambda\alpha\lambda x^\alpha \lambda y^{\alpha\to\alpha\to\alpha}.y(s\alpha xy)(t\alpha xy) \end{aligned}$$

These trees do not actually store any data; the only real information content is just the shape of the tree. However it is fairly straightforward to modify this definition to be able to store data of type $\tau$ at each node however; simply change the constructor $C : T \to T \to T$ to be $C : \tau \to T \to T \to T$.

## 4. References


1. Sørensen, Urzycz; *Lectures on the Curry-Howard Isomorphism*, Ch.11
2. Girard; *Proofs and Types*, Ch.11, 15