Principals of Programming Languages: Revision Lecture

Sam Barrett

January 4, 2021

1 The Simply Typed λ-Calculus

1.1 Syntax

The syntax of the Simply Typed λ-Calculus can be defined as:

$$T ::= \mathbb{B} | T \to T$$

$$M ::= x | \lambda x : T.M | MM | \texttt{true} | \texttt{false} | \texttt{if}\ M\ \texttt{then}\ M\ \texttt{else}\ M$$

You can see here that we use the Church style for typing whereby, variables in λ abstractions are annotated with types.

Values are atomic, i.e. they cannot be evaluated further and are of the form:

$$V ::= \lambda x : T.M | \texttt{true} | \texttt{false}$$

When we compute a term we are typically trying to reduce it to a value.

1.2 Evaluation Contexts

When we want to define the call-by-value small-step operational semantics of a language we use evaluation contexts. The Call-by-value evaluation contexts for the small-step operational semantics of λ-Calculus is defined as:

$$C ::= \bullet | CM | VC | \texttt{if}\ C\ \texttt{then}\ M\ \texttt{else}\ M$$

A context is a term with a *hole* ($\bullet$) in it.

You can tell that this is the call-by-value evaluation context as you can see that we always evaluate the arguments of a application before the application itself.

These contexts yield the following rules:

$$\frac{}{(\lambda x : T.M)V \to_v M[x \backslash V])}\beta$$

$$\frac{}{\texttt{if true then}\ M\ \texttt{else}\ N \to_v M}\texttt{IteT}$$

$$\frac{M \to_v N}{C[M] \to_v C[N]}\texttt{CTX}_C$$

$$\frac{}{\texttt{if false then}\ M\ \texttt{else}\ N \to_v N}\texttt{IteF}$$

1.3 Typing Rules

And to facilitate the typing of these expressions we use the following typing rules:

$$\frac{}{\Gamma, x : T \vdash x : T}\texttt{VAR}$$

$$\frac{\Gamma, x : T \vdash M : U}{\Gamma \vdash \lambda x : T.M : T \to U}\texttt{ABS}$$

$$\frac{\Gamma \vdash M : T \to U \Gamma \vdash N : T}{\Gamma \vdash MN : U}\texttt{APP}$$

$$\frac{}{\Gamma \vdash \texttt{true} : \mathbb{B}}\texttt{T}$$

$$\frac{}{\Gamma \vdash \texttt{false} : \mathbb{B}}\texttt{F}$$

$$\frac{\Gamma \vdash M : \mathbb{B} \Gamma \vdash N : T \Gamma \vdash P : T}{\Gamma \vdash \texttt{if } M \texttt{ then } N \texttt{ else } P : T}\texttt{ITE}$$

1.4 Church-Numerals

We can define Church Numerals in the Simply Typed λ-Calculus as having the type $\texttt{Nat} = (\mathbb{B} \to \mathbb{B}) \to \mathbb{B} \to \mathbb{B}$.

$$\begin{aligned}
\underline{0} &= \lambda f : \mathbb{B} \to \mathbb{B}.\lambda x : \mathbb{B}.x \\
\underline{1} &= \lambda f : \mathbb{B} \to \mathbb{B}.\lambda x : \mathbb{B}.fx \\
\underline{2} &= \lambda f : \mathbb{B} \to \mathbb{B}.\lambda x : \mathbb{B}.f(fx) \\
\ldots
\end{aligned}$$

Essentially, the number is a counter of how many applications of f appear.

We can now define a successor function, `succ` of type $\texttt{Nat} \to \texttt{Nat}$ and an `add` function of type $\texttt{Nat} \to \texttt{Nat} \to \texttt{Nat}$:

$$\begin{aligned}
\texttt{succ} &= \lambda a : \texttt{Nat}.\lambda f : \mathbb{B} \to \mathbb{B}.\lambda x : \mathbb{B}.f(afx) \\
\texttt{add} &= \lambda a : \texttt{Nat}.\lambda b : \texttt{Nat}.\lambda f : \mathbb{B} \to \mathbb{B}.\lambda x : \mathbb{B}.af(bfx)
\end{aligned}$$

Our `add` function essentially concatenates the fs in a with the fs in b and our `succ` function appends an f to the value a.

This encoding can be used to iterate over a function of type $\mathbb{B} \to \mathbb{B}$ by applying a function $f : \mathbb{B} \to \mathbb{B}$ to a base case $x : \mathbb{B}$ $n : \texttt{Nat}$ times.

$$nfx$$

However, if one wants to iterate over a function of another type, say, $T \to T$ one will need to define a new set of Church numerals over the type T, i.e. a type of the form $(T \to T) \to T \to T)$ (This is later solved through the use of System-F's paramerisation of types, similar to Monads)

1.4.1 Exercise 1

Can we iterate over a function of type $\texttt{Nat} \to \texttt{Nat}$ using this method? I.e. can we define $\texttt{add} = \lambda a : \texttt{Nat}.\lambda b : \texttt{Nat}.a \texttt{ succ } b$?

For this to be possible, a would need to have the type $(\texttt{Nat} \to \texttt{Nat}) \to (\texttt{Nat} \to \texttt{Nat})$. However, our value a is of the type $(\mathbb{B} \to \mathbb{B}) \to (\mathbb{B} \to \mathbb{B})$. These types are not compatible, if we were to redefine $\texttt{Nat}$ to fit this function we would end up with a recursive type definition which is not allowed within the Simply Typed λ-Calculus.

Note: This is possible using System-F

1.5 System-F

We define the Church-style syntax of System-F as:

$$\begin{aligned} T &::= \alpha|\mathbb{B}|\mathbb{N}|T \to T|\forall\alpha.T \\ M &::= x|\lambda x : T.M|MM|\texttt{true}|\texttt{false}|\texttt{if } M \texttt{ then } M \texttt{ else } M| \\ &\texttt{let } x = M \texttt{ in } M|\texttt{zero}|\texttt{succ}M|\texttt{pred}M|\texttt{iszero}M|\lambda\alpha.M|M\{T\} \end{aligned}$$

Here we have both Boolean $\mathbb{B}$ and Nat $\mathbb{N}$ ground types. This leads to simpler examples.

System-F utilises a system of parameterised types. The general form of these types is $\forall\alpha.T$ This defines a family of types whereby for **any** type α. For example, given an expression $M : \forall\alpha.T$, we can construct a any type of the form $M : T[\alpha\backslash T']$ such as $M : T[\alpha\backslash\mathbb{B}]$ or $M : T[\alpha\backslash\mathbb{N}]$

We also have what are known as *type abstractions* in the form of $\lambda\alpha.M$ and *type applications*T of the form $M\{T\}$

Our Values take the form:

$$V ::= \lambda x : T.M|\texttt{true}|\texttt{false}|\texttt{zero}|\texttt{succ}M|\lambda\alpha.M$$

We define our Call-by-value evaluation contexts as:

$$C ::= \bullet|CM|VC|\texttt{if } C \texttt{ then } M \texttt{ else } M|\texttt{let } x = C \texttt{ in } M|\texttt{pred } C|\texttt{iszero } C|C\{T\}$$

Which is the same as before with the extension of let, pred, iszero and type application.

We also have an extended set of Call-by-value small-step operational semantics rules:

$$\frac{}{(\lambda x : T.M)V \to_v M[x\backslash V]}\ \beta$$

$$\frac{M \to_v N}{C[M] \to_v C[N]}\ \texttt{CTX}_C$$

$$\frac{}{\texttt{if true then } M \texttt{ else } N \to_v M}\ \texttt{IteT}$$

$$\frac{}{\texttt{if false then } M \texttt{ else } N \to_v N}\ \texttt{IteF}$$

$$\frac{}{\texttt{let}x = V\texttt{in}M \to_v M[x\backslash V]}\ \texttt{LetV}$$

$$\frac{}{\texttt{pred zero} \to_v \texttt{zero}}\ \texttt{PredZ}$$

$$\frac{}{\texttt{pred}(\texttt{succ}\ \ M) \rightarrow_v M}\ \texttt{PredS}$$

$$\frac{}{\texttt{iszero}\ \ \texttt{zero} \rightarrow_v \texttt{true}}\ \texttt{IsZZ}$$

$$\frac{}{\texttt{iszero}(\texttt{succ}M) \rightarrow_v \texttt{false}}\ \texttt{IsZS}$$

$$\frac{}{(\lambda a.M)\{T\} \rightarrow_v M[\alpha \backslash T]}\ \texttt{T}\beta$$

And our typing rules are as follows:

$$\frac{}{\Gamma, x : T \vdash x : T}\ \texttt{VAR}$$

$$\frac{\Gamma, x : T \vdash M : U}{\Gamma \vdash \lambda x : T.M : T \rightarrow U}\ \texttt{ABS}$$

$$\frac{\Gamma \vdash M : T \rightarrow U\ \Gamma \vdash N : T}{\Gamma \vdash MN : U}\ \texttt{APP}$$

$$\frac{}{\Gamma \vdash \texttt{true} : \mathbb{B}}\ \texttt{T}$$

$$\frac{}{\Gamma \vdash \texttt{false} : \mathbb{B}}\ \texttt{F}$$

$$\frac{\Gamma \vdash M : \mathbb{B}\ \Gamma \vdash N : T\ \Gamma \vdash P : T}{\Gamma \vdash \texttt{if}\ \ M\ \ \texttt{then}\ \ N\ \ \texttt{else}\ \ P : T}\ \texttt{ITE}$$

$$\frac{}{\Gamma \vdash \texttt{zero} : \mathbb{N}}\ \texttt{Z}$$

$$\frac{\Gamma \vdash M : \mathbb{N}}{\Gamma \vdash \texttt{succ}M : \mathbb{N}}\ \texttt{S}$$

$$\frac{\Gamma \vdash M : \mathbb{N}}{\Gamma \vdash \texttt{pred}M : \mathbb{N}}\ \texttt{P}$$

$$\frac{\Gamma \vdash M : \mathbb{N}}{\Gamma \vdash \texttt{iszero}M : \mathbb{B}}\ \texttt{I}$$

$$\frac{\Gamma \vdash M : T\ \Gamma, x : T \vdash N : U}{\Gamma \vdash \texttt{let}x = M\ \ \texttt{in}\ \ N : U}\ \texttt{LET}$$

$$\frac{\Gamma \vdash M : T}{\Gamma \vdash \lambda\alpha.M : \forall\alpha.T}\ \texttt{TABS}$$

Note TABS also requires that $\alpha \notin \texttt{FV}(\Gamma)$ is satisfied Where $\texttt{FV}(\Gamma)$ is the set of free variables in Γ i.e. α should be a *new* variable with respect to Γ in the hypothesis.

$$\frac{\Gamma \vdash M : \forall\alpha.T}{\Gamma \vdash M\{U\} : T[\alpha \backslash U]}\ \texttt{TAPP}$$

1.5.1 Examples

We previously saw that in the Simply typed λ-Calculus we could not construct a function, `add` of type `Nat` $\rightarrow$ `Nat`. Using System-F this is possible through an abstracted definition of Church Numerals:

$$
\begin{aligned}
\underline{0} &= \lambda\alpha.\lambda f : \alpha \rightarrow \alpha.\lambda x : \alpha.x \\
\underline{1} &= \lambda\alpha.\lambda f : \alpha \rightarrow \alpha.\lambda x : \alpha.fx \\
\underline{2} &= \lambda\alpha.\lambda f : \alpha \rightarrow \alpha.\lambda x : \alpha.f(fx) \\
\ldots
\end{aligned}
$$

Again `succ` has type `Nat` $\rightarrow$ `Nat` $\rightarrow$ `Nat` and `add` type `Nat` $\rightarrow$ `Nat`:

$$\texttt{succ} = \lambda a : \texttt{Nat}.\lambda\alpha.\lambda f : \alpha \rightarrow \alpha.\lambda x : \alpha.f(a\{\alpha\}fx) \tag{1}$$

$$\texttt{add} = \lambda a : \texttt{Nat}.\lambda b : \texttt{Nat}.\lambda\alpha.\lambda f : \alpha \rightarrow \alpha.\lambda x : \alpha.a\{\alpha\}f(b\{\alpha\}fx) \tag{2}$$

Now, given any numeral n : `Nat` we can iterate over a function F of type $T \rightarrow T$ as follows: $n\{T\}F$ and we can define `add` more simply using `succ` as:

$$\texttt{add} = \lambda a : \texttt{Nat}.\lambda b : \texttt{Nat}.a\{\texttt{Nat}\}\texttt{succ}b$$