Practical Foundations for Programming Languages

Jason Hu, Zhe Zhou, Ramana Nagasamudram

June 27, 2019

1 Introduction

The semantics of variables plays a key role when comparing by-name and by-value languages. Recall that in a by-name setting, variables range over computations and in a by-value setting, they range over values. We've seen in the previous lecture that by-value languages are inherently more expressive than by-name languages – we can emulate by-name methods in a by-value setting using a *computational modality*. The converse isn't possible precisely because we aren't afforded control over the order of evaluation in the by-name setting. The ability to control the sequencing of events (in the by-value setting) also provides us with a natural way to account for exceptions.

Generalize $\tau\ \mathtt{comp}$ to $\tau_1 \& \ldots \& \tau_n$, or $\tau\ \mathtt{seq}$. We can also have dynamically many unevaluated computations. This naturally gives rise to parallelism. Doing this in a by-value setting we obtain "work-efficiency".

The central issue is the meaning/semantics of variables. We have to consider what variables range over.

Recall: $\mathtt{fix}[\tau](x.e)$ is done in an ad-hoc way. Additionally, $\mathtt{comp}(x.m)$ is also done in an ad-hoc way.

2 FPC – recursive types

Origin: Scott's model of the λ-calculus. Relationship between computability and continuity. *Computational trinitarianism.*

We start out working in the by-name setting, and will then move on to the by-value setting.

$$\begin{aligned} \tau ::= 0 \mid t \mid 1 \mid \tau_1 + \tau_2 \mid \tau_1 \times \tau_2 \\ \mid \tau_1 \to \tau_2 \mid \mathtt{rec}(t.\tau) \end{aligned}$$

Where type "1" is $\mathtt{unit}$. And type "0" is empty type, which means there is not any instance for this type. (In many settings, we want $\tau_1 + \tau_2 + \cdots + \tau_n$).

Example: $2 \triangleq 1 + 1$

$$\frac{e_1 : \tau_1}{\mathtt{in}[1][\tau_1][\tau_2](e_1) : \tau_1 + \tau_2} \qquad \frac{e_2 : \tau_2}{2 \cdot e_2 : \tau_1 + \tau_2} \qquad \frac{e : \tau_1 + \tau_2 \qquad x_1 : \tau_1 \vdash e_1 : \tau \qquad x_2 : \tau_2 \vdash e_2 : \tau}{\mathtt{case}\ (e; x_1.e_1; x_2.e_2) : \tau}$$

$$\frac{e : 0}{\mathtt{cant}(e) : \tau} \qquad \frac{e : [\mathtt{rec}(t.\tau)/t]\tau}{\mathtt{fold}[t.\tau](e) : \mathtt{rec}(t.\tau)} \qquad \frac{e : \mathtt{rec}(t.\tau)}{\mathtt{unfold}(e) : [\mathtt{rec}(t.\tau)/t]\tau}$$

Where we have "$\mathtt{in}[1][\tau_1][\tau_2](e_1)$" to represent building a sum type and assuming e_1 has the left one type "τ_1". For short, we can just use notation "$1 \cdot e_1$" for instead. So does "$2 \cdot e_2$".

The `cant` rule expresses vacuity.

Example:

$$\begin{aligned}
\mathtt{tt} &\triangleq 1.\langle\rangle \\
\mathtt{ff} &\triangleq 2.\langle\rangle \\
\mathtt{if}(e; e; e) &\triangleq \mathtt{case}(e; _.e_1; _.e_2) \\
n &\triangleq 1 + 1 + \cdots + 1 \\
\tau\ \mathtt{opt} &\triangleq \tau + 1 \\
\omega &\triangleq \mathtt{rec}(t.1 + t)(\approxeq 1 + \omega)
\end{aligned}$$

$$\frac{}{\mathtt{fold}(e)\ \mathtt{val}} \qquad \frac{e \mapsto e'}{\mathtt{unfold}(e) \mapsto \mathtt{unfold}(e')} \qquad \frac{}{\mathtt{unfold}(\mathtt{fold}(e)) \mapsto e}$$

$$\begin{aligned}
\mathtt{zero} &\triangleq \mathtt{fold}[t.1 + t](1.\langle\rangle) \\
\mathtt{succ}(e) &\triangleq \mathtt{fold}[t.1 + t](2.e) \\
\mathtt{ifz}(e; e_1; e_2) &\triangleq \mathtt{case}(\mathtt{unfold}(e); _.e_1; x.e_2)
\end{aligned}$$

Key idea of self-reference is self-application (Kleene/Church). Let's consider the following type ("self-referential computations of type τ"),

$$\tau\ \mathtt{self}$$

Example: $\mathtt{fact} : (\omega \to \omega)\ \mathtt{self}$.

$$\tau\ \mathtt{self} \triangleq \mathtt{rec}(t.t \to \tau)$$

This type has a "negative" occurence of t.

$$\frac{x : \tau\ \texttt{self} \vdash e : \tau}{\texttt{self}(x.e) : \tau\ \texttt{self}}\ \textsc{Intro}$$

$$\begin{aligned}
\texttt{self}(x.e) &\triangleq \texttt{fold}[t.t \to \tau](\lambda[\tau\ \texttt{self}](x.e)) \\
\texttt{unroll}(e) &\triangleq \texttt{unfold}(e)(e) \\
Y &\triangleq \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))
\end{aligned}$$

Suppose we are given $F : (\omega \to \omega)\ \texttt{self}$, then how do we call F. We use `unroll` it to get a function and just apply the function we get. That is:

$$\begin{aligned}
\texttt{selfap}(F, e) &\triangleq \texttt{ap}(\texttt{unroll}(F)(e)) \\
&\triangleq \texttt{ap}((\texttt{unfold}(F)(F))(e))
\end{aligned}$$

3 FPC by-value

Main ideas are PCF by-value. If you want to have laziness you use $\texttt{comp}(\cdot)$ (the programming language analog of TBD – to be determined).

$$\begin{aligned}
v ::= &\langle\rangle \mid \langle v_1, v_2\rangle \mid v_1 \otimes v_2 \mid \texttt{comp}(m) \mid 1.v \mid 2.v \mid \dots \\
&\mid \lambda[\tau_1](x.m) \mid \texttt{fold}(v) \mid x
\end{aligned}$$

$$m ::= \texttt{ret}(v) \mid \texttt{bnd}(v; x.m) \mid \texttt{split}(v; x_1, x_2.m) \mid \texttt{case} \cdots \mid \texttt{cant}(v) \mid \texttt{ap}(v_1, v_2) \mid \texttt{unfold}(v)$$

How do we handle self-referentiality?

$$\tau\ \texttt{self} \triangleq \texttt{rec}(t.t \to \tau\ \texttt{comp})$$

We want to have:

$$\frac{x : \tau\ \texttt{self} \vdash m \dot{\sim} \tau}{\texttt{self}(x.m) : \tau\ \texttt{self}}$$

Or,

$$\frac{x : \tau\ \texttt{self} \vdash m \dot{\sim} \tau}{\texttt{fold}[t.t \to \tau](\lambda[\tau\ \texttt{self}].\texttt{ret}(\texttt{comp}(m))) : \tau\ \texttt{self}}$$

$$\begin{aligned}
\texttt{unroll}(v : \tau\ \texttt{self}) &\triangleq \texttt{unfold}(v)(v)\ [\textbf{compute}\ \texttt{unfold}(v)\ \textbf{first}] \\
&\triangleq \texttt{bnd}(\texttt{comp}(\texttt{unfold}(v)); x.\texttt{ap}(x, v))\ [\textbf{compute}\ \texttt{ap}(x, v)\ \textbf{now}] \\
&\triangleq \texttt{bnd}(\texttt{comp}(\texttt{unfold}(v)); x.\texttt{bnd}(\texttt{ap}(x, v); y.\texttt{ret}(y)))
\end{aligned}$$

Additional Notes: the following part is from the textbook PFPL.

How to use self referentiality to simulate an abritrary recursive function($\texttt{fix}$)?

$$\begin{aligned}
\texttt{fix}[\tau](x.e) &\triangleq \texttt{unroll}(\texttt{self}[\tau](y.([\texttt{unroll}(y)/x]e))) \\
&\triangleq [\texttt{self}[\tau](y.([\texttt{unroll}(y)/x]e))/y]([\texttt{unroll}(y)/x]e) \\
&\triangleq [\texttt{unroll}(\texttt{self}[\tau](y.([\texttt{unroll}(y)/x]e)))/x]e \\
&\triangleq [\texttt{fix}[\tau](x.e)/x]e
\end{aligned}$$

Exercise: design a self-referential value of $\tau\ \texttt{comp}$

Example: a stream of nat's – $\texttt{rec}(t.(\omega \times t)\ \texttt{comp})$

4 Universal Domain

The idea is to have one big pot or one "universal" type. Unfortunately, this also means that anything that parses does something.

$$u ::= x \mid \texttt{num}[n] \mid \texttt{ifz}(u; u_1; x.u_2) \mid \texttt{nil} \mid \texttt{cons}(u_1; u_2) \mid \ldots$$

Statics amounts to saying $x_1\ \texttt{ok}, \ldots, x_n\ \texttt{ok} \vdash u\ \texttt{ok}$.

Dynamics $u \mapsto u'$, $u\ \texttt{val}$, $u\ \texttt{err}$

We also need $u\ \texttt{err}$. For example, consider the term $\texttt{ifz}(\texttt{nil}; u_1; x.u_2)\ \texttt{err}$.

$$\frac{}{x : \mathcal{U} \vdash u : \mathcal{U}}$$

The one universal type is

$$\mathcal{U} \triangleq \texttt{rec}(t.[\texttt{nil} \hookrightarrow 1, \texttt{cons} \hookrightarrow t \times t, \texttt{fun} \hookrightarrow t \to t, \texttt{num} \hookrightarrow \omega, \ldots])$$

For the untyped lambda calculus, you just consider the $\texttt{fun}$ clause.

4.1 Dynamics

$$
\begin{aligned}
\texttt{nil} &\triangleq \texttt{fold}[\texttt{nil} \hookrightarrow 1] \\
\texttt{num}[n] &\triangleq \texttt{fold}[\texttt{num} \hookrightarrow n] \\
\texttt{cons}(u_1; u_2) &\triangleq \texttt{fold}[\texttt{cons} \hookrightarrow \langle u_1; u_2 \rangle]
\end{aligned}
$$

$$
\texttt{IFZ}(u; u_1; x.u_2) \triangleq \texttt{case}\Big(\texttt{unfold}(u); \begin{array}{l} \texttt{num} \hookrightarrow x.\texttt{ifz}(x; u_1; x'[\texttt{fold}[\texttt{num} \hookrightarrow x']/x]u_2 \\ \texttt{nil} \hookrightarrow \texttt{err} \\ \vdots \end{array}\Big)
$$

This is quite inefficient.