Scribe: Dan Sheldon, Maksim Orlovich Lecturer: Andrew Myers

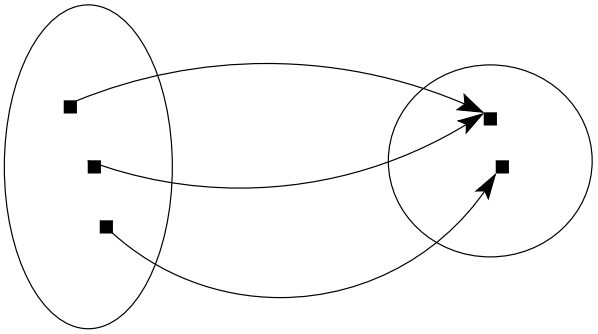
1 Introduction

The goal is to study basic PL features, using the semantic techniques we know:

- Small-step operational semantics
- Big-step operational semantics (also known as "natural" semantics)
- Translation

We will mostly use small-step semantics and translation.

2 Translation



For translation, we map well-formed programs in the original language, into items in the *meaning space*. These items may be

- programs in an another language (*definitional translation*)
- mathematical objects (*denotational semantics*); an example is taking $\lambda x : \text{int}.x$ to $\{0 \to 0, 1 \to 1, \ldots\}$

Because they define the meaning of a program, these translations are also known as meaning functions or semantic functions.

2.1 Translating CBN λ-calculus into CBV λ-calculus

The call-by-name (lazy) version of λ-calculus was defined with the following evaluation context:

$$E[\bullet] ::= [\bullet] \;\mid\; [\bullet]\ e$$

and with the following reduction:

$$(\lambda x.e_1)\ e_2 \longrightarrow e_1\{e_2/x\}$$

The call-by-value (eager) version of λ-calculus was defined with the following evaluation context:

$$E[\bullet] ::= [\bullet] \;\mid\; [\bullet]\ e_1 \;\mid\; v\ [\bullet]$$

and with the following reduction:

$$(\lambda x.e_1)\ v \longrightarrow e_1\{v/x\}$$

Relevant notation: $[\![x]\!]$ uses the *semantic brackets*. These denote a function applied to the syntax of the original language. The bracket may ocassionally be annotated to avoid ambiguity or confusion between multiple functions: either as $[\![e]\!]_{\text{cbn}}$ or $\mathcal{C}[\![e]\!]$

To translate from CBN lambda calculus to CBV lambda calculus we define the semantic function $[\![\bullet]\!]$ as follows by induction on the syntactic structure:

$$\begin{aligned}
[\![x]\!] &= x\ ID \\
[\![\lambda x.e]\!] &= \lambda x.[\![e]\!] \\
[\![e_1\ e_2]\!] &= [\![e_1]\!](\lambda z.[\![e_2]\!]), \text{where } z \notin FV([\![e_2]\!])
\end{aligned}$$

The idea here is to wrap the parameters to functions inside $\lambda-$abstractions to delay their evaluation, and then to finally pass in a dummy parameter to expand them out.

For an example, recall that we defined:

$$\begin{aligned}
TRUE &\triangleq \lambda x.\lambda y.x \\
FALSE &\triangleq \lambda x.\lambda y.y \\
IF &\quad \lambda x.\lambda y.\lambda z.(xyz)
\end{aligned}$$

There is a problem with this construction in CBV $\lambda-$calculus: $IF\ b\ e_1\ e_2$ evaluates both e_1 and e_2. Perhaps the conversion above can be used to fix these to lazily evaluate the arms?

$$\begin{aligned}
[\![TRUE]\!] &= \lambda x.[\![\lambda y.x]\!] \\
&= \lambda x.\lambda y.[\![x]\!] \\
&= \lambda x.\lambda y.(x\ ID)
\end{aligned}$$

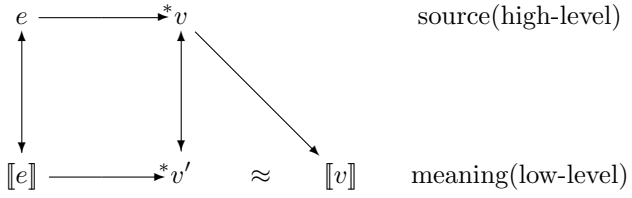
$$[\![FALSE]\!] = \lambda x.\lambda y.(y\ ID)$$

$$\begin{aligned}
[\![IF]\!] &= [\![\lambda x.\lambda y.\lambda z.xyz]\!] \\
&= \lambda x.\lambda y.\lambda z.[\![(xy)z]\!] \\
&= \lambda x.\lambda y.\lambda z.[\![xy]\!](\lambda d.[\![z]\!]) \\
&= \lambda x.\lambda y.\lambda z.[\![xy]\!](\lambda d.z\ ID) \\
&= \lambda x.\lambda y.\lambda z.[\![x]\!](\lambda d.[\![y]\!])(\lambda d.z\ ID) \\
&= \lambda x.\lambda y.\lambda z.(x\ ID)(\lambda d.y\ ID)(\lambda d.z\ ID)
\end{aligned}$$

This isn't quite a solution to implementing lazy evaluation of *IF* arms inside CBV λ-calculus, as the conversion is meant to be used only with a fully converted expression; but this will be adopted later.

2.2 Adequacy

We would like to say that meaning space is *adequate* to represent the source language. To get this, we need the following situation to hold:



That is, if an expression e steps to v in 0 or more steps, then $[\![e]\!]$ must step to v' such that v' is equivalent (e.g. β-equivalent) to $[\![v]\!]$; and further each expression in the low-level language can be expressed in the original language. This is formally stated as two properties: soundness and completeness

2.2.1 Soundness

$$[\![e]\!] \xrightarrow[\text{cbv}]{}{}^* v' \Rightarrow \exists v.e \xrightarrow[\text{cbn}]{}{}^* v \wedge v' \approx [\![v]\!]$$

2.2.2 Completeness

$$[\![e]\!] \xrightarrow[\text{cbn}]{}{}^* v \Rightarrow \exists v'.[\![e]\!] \xrightarrow[\text{cbv}]{}{}^* v' \wedge v' \approx [\![v]\!]$$

2.2.3 Handling non-termination

The above does not list one requirement: the source and meaning forms must also agree on non-terminating execution. We write $e \Uparrow$, read e *diverges*, to denote non-termination. We have $e \Uparrow$ if there exists an infinite trace of expressions $e_1, e_2, \ldots$ such that $e \to e_1 \to e_2 \to \ldots$ With this, there are additional conditions for soundness:

$$[\![e]\!] \Uparrow_{\text{cbv}} \Rightarrow e \Uparrow_{\text{cbn}}$$

and completeness:

$$e \Uparrow_{\text{cbn}} \Rightarrow [\![e]\!] \Uparrow_{\text{cbv}}$$

Note that which direction is considered soundness and which completeness depends on which semantics (the original operational semantics or the translation) is considered the ground truth.

Adequacy is the combination of *soundness* and *completeness*.

2.2.4 Caveats

We have defined adequacy to help show the correctness of translations. But there are a few caveats:

1. *Completeness is not enough*, and may not be useful on its own. For example, consider a trivial meaning space MS where $\forall e, e \xrightarrow[\text{MS}]{} 0$ and $\forall v, v \equiv 0$. Then completeness is satisfied for *any* translation, but conveys no notion of adequacy.
2. Soundness may be hard to show in general. Typically, we show *agreement on divergence and base values*.
3. We'll see that we *need types to show soundness*.

2.3 Example: Augmented Lambda Calculus (uML)

Let's construct an example by augmenting the λ-calculus, and defining its translation to the CBV λ-calculus. We'll call this language uML since it resembles ML, with the "u" standing for "untyped".

2.3.1 Expressions

$$\begin{aligned} e \quad ::= \quad & \lambda x_1 \ldots x_n.e \mid e_0 \ldots e_n \mid x \mid n \mid \textbf{true} \mid \textbf{false} \\ & \mid (e_1, \ldots, e_2) \mid \#ne \mid \textbf{if } e_0 \textbf{ then } e_1 \textbf{ else } e_2 \\ & \mid \textbf{let } x = e_1 \textbf{ in } e_2 \\ & \mid \textbf{letrec } f_1 = \lambda x_1 e_1 \textbf{ and } \ldots \textbf{ and } f_n = \lambda x_n e_n \textbf{ in } e \end{aligned}$$

2.3.2 Values

$$v \quad ::= \quad \lambda x_1 \ldots x_n.e \;\mid\; n \;\mid\; \textbf{true} \;\mid\; \textbf{false} \;\mid\; (v_1, \ldots, v_n)$$

2.3.3 Evaluation context

$$\begin{aligned} E[\bullet] \quad ::= \quad & [\bullet] \;\mid\; v_0 \ldots v_m[\bullet]e_{m+2} \ldots e_n \;\mid\; \#n[\bullet] \\ & \mid\; \textbf{if } [\bullet] \textbf{ then } e_1 \textbf{ else } e_2 \\ & \mid\; \textbf{let } x = [\bullet] \textbf{ in } e_2 \\ & \mid\; (v_1, \ldots, v_m, [\bullet], e_{m+2}, \ldots, e_n) \end{aligned}$$

2.3.4 Reductions

$$\begin{aligned} (\lambda x_1 \ldots x_n.e)v_1 \ldots v_n &\rightarrow e\{v_1/x_1\}\{v_2/x_2\} \ldots \{v_n/x_n\} \\ \#n(v_1, \ldots, v_m) &\rightarrow v_n \qquad (\text{where } 1 \leq n \leq m) \\ \textbf{if true then } e_1 \textbf{ else } e_2 &\rightarrow e_1 \\ \textbf{if false then } e_1 \textbf{ else } e_2 &\rightarrow e_2 \\ \textbf{let } x = v \textbf{ in } e &\rightarrow e\{v/x\} \\ \textbf{letrec } \ldots &\rightarrow \textit{to be continued} \end{aligned}$$

We can already see hints that types will be important for analyzing translations. For example, what happens with the expression "if 3 then 1 else 0"? This evaluation gets *stuck* because 3 is a value and will never reduce to **true** or **false**.

2.3.5 Translating uML to CBV λ-calculus

We begin to define translation rules:

$$\begin{aligned} [\![\lambda x_1 \ldots x_n.e]\!] &= \lambda x_1.\lambda x_2. \ldots \lambda x_n.[\![e]\!] \\ [\![e_0 \ldots e_n]\!] &= ((([\![e_0]\!][\![e_1]\!])[\![e_2]\!]) \ldots)[\![e_n]\!] \\ [\![x]\!] &= x \\ [\![n]\!] &= \lambda f.\lambda x.f^n x \\ [\![\textbf{true}]\!] &= \lambda x.\lambda y.(x\ ID) \\ [\![\textbf{false}]\!] &= \lambda x.\lambda y.(y\ ID) \\ [\![\textbf{if } e_0 \textbf{ then } e_1 \textbf{ else } e2]\!] &= ([\![e_0]\!]\lambda z.[\![e_1]\!])\lambda z.[\![e_2]\!] \end{aligned}$$

Revisiting our earlier example "if 3 then 1 else 0", we see that its translation *will* evaluate in the λ-calculus. because there is no way for a lambda-calculus term to get stuck, even if it no longer soundly represents an evaluation in the source language. This is a disconnect that we will address later on by explicitly adding run-time type checking to the translation.