CS611 Lecture 9 — Semantics via Translation — 18 September 2006

Lecturer: Dexter Kozen

1 Overview

Our goal is to study basic programming language features using the semantic techniques we know:

- small-step operational semantics;
- big-step operational 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 a *meaning space*. These items may be

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

Because they define the meaning of a program, these translations are also known as *meaning functions* or *semantic functions.* We usually denote the semantic function under consideration by $[\![\cdot]\!]$. An object e in the original language is mapped to an object $[\![e]\!]$ in the meaning space under the semantic function. We may occasionally add an annotation to distinguish between different semantic functions, as for example $[\![e]\!]_{\mathrm{cbn}}$ or $\mathcal{C}[\![e]\!]$.

3 Translating CBN λ-Calculus into CBV λ-Calculus

The call-by-name (lazy) λ-calculus was defined with the following reduction rule and evaluation contexts:

$$(\lambda x.\, e_1)\ e_2 \ \rightarrow\ e_1\{e_2/x\} \qquad E\ ::=\ [\,\bullet\,]\ \mid\ E\ e.$$

The call-by-value (eager) λ-calculus was similarly defined with

$$(\lambda x.\, e)\ v \ \rightarrow\ e\{v/x\} \qquad E\ ::=\ [\,\bullet\,]\ \mid\ E\ e\ \mid\ v\ E.$$

To translate from the CBN λ-calculus to the CBV λ-calculus, we define the semantic function $[\![\cdot]\!]$ by induction on the syntactic structure:

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

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

For an example, recall that we defined:

$$\begin{aligned}
\mathbf{true} &\triangleq \lambda xy.\, x \\
\mathbf{false} &\triangleq \lambda xy.\, y \\
\mathbf{if} &\triangleq \lambda xyz.\, xyz.
\end{aligned}$$

The problem with this construction in the CBV λ-calculus is that **if** $b\ e_1\ e_2$ evaluates both e_1 and e_2, regardless of the truth value of b. Perhaps the conversion above can be used to fix these to evaluate them lazily.

$$\begin{aligned}
\llbracket \mathbf{true} \rrbracket &= \llbracket \lambda xy.\, x \rrbracket \\
&= \lambda xy.\, \llbracket x \rrbracket \\
&= \lambda xy.\, x \text{ ID} \\
\llbracket \mathbf{false} \rrbracket &= \lambda xy.\, y \text{ ID} \\
\llbracket \mathbf{if} \rrbracket &= \llbracket \lambda xyz.\, xyz \rrbracket \\
&= \lambda xyz.\, \llbracket (xy)z \rrbracket \\
&= \lambda xyz.\, \llbracket xy \rrbracket\ (\lambda d.\, \llbracket z \rrbracket) \\
&= \lambda xyz.\, \llbracket x \rrbracket\ (\lambda d.\, \llbracket y \rrbracket)\ (\lambda d.\, \llbracket z \rrbracket) \\
&= \lambda xyz.\, (x \text{ ID})\ (\lambda d.\, y \text{ ID})\ (\lambda d.\, z \text{ ID}).
\end{aligned}$$

This is not a complete solution, as the conversion does not work for all expressions, but only fully converted ones. But if used as intended, it has the desired effect. For example, evaluating under the CBV rules,

$$\begin{aligned}
\llbracket \mathbf{if}\ \mathbf{true}\ e_1\ e_2 \rrbracket &= \llbracket \mathbf{if} \rrbracket\ (\lambda d.\, \llbracket \mathbf{true} \rrbracket)\ (\lambda d.\, \llbracket e_1 \rrbracket)\ (\lambda d.\, \llbracket e_2 \rrbracket) \\
&= (\lambda xyz.\, (x \text{ ID})\ (\lambda d.\, y \text{ ID})\ (\lambda d.\, z \text{ ID}))\ (\lambda d.\, \llbracket \mathbf{true} \rrbracket)\ (\lambda d.\, \llbracket e_1 \rrbracket)\ (\lambda d.\, \llbracket e_2 \rrbracket) \\
&\to ((\lambda d.\, \llbracket \mathbf{true} \rrbracket) \text{ ID})\ (\lambda d.\, (\lambda d.\, \llbracket e_1 \rrbracket) \text{ ID})\ (\lambda d.\, (\lambda d.\, \llbracket e_2 \rrbracket) \text{ ID}) \\
&\to \llbracket \mathbf{true} \rrbracket\ (\lambda d.\, \llbracket e_1 \rrbracket)\ (\lambda d.\, \llbracket e_2 \rrbracket) \\
&= (\lambda xy.\, x \text{ ID})\ (\lambda d.\, \llbracket e_1 \rrbracket)\ (\lambda d.\, \llbracket e_2 \rrbracket) \\
&\to (\lambda d.\, \llbracket e_1 \rrbracket) \text{ ID} \\
&\to \llbracket e_1 \rrbracket,
\end{aligned}$$

and e_2 was never evaluated.

4 Adequacy

Both the CBV and CBN λ-calculus are *deterministic* systems in the sense that there is at most one reduction that can be performed on any term. When an expression e in a language is evaluated in a deterministic system, one of three things can happen:

1. The computation can converge to a value: $e \Downarrow v$.
2. The computation can converge to a non-value. When this happens, we say the computation is *stuck*.
3. The computation can diverge: $e \Uparrow$.

A semantic translation is *adequate* if these three behaviors in the source system are accurately reflected in the target system, and vice versa. One aspect of this relationship is captured in the following diagram:

$$\begin{tikzcd}
e \arrow[r, "*"] \arrow[d, leftrightarrow] & v \arrow[d, leftrightarrow] \arrow[rrd] & & \\
\llbracket e \rrbracket \arrow[r, "*"] & v' \arrow[rr, phantom, "\approx"] & & \llbracket v \rrbracket
\end{tikzcd}$$

If an expression e converges to a value v in zero or more steps in the source language, then $\llbracket e \rrbracket$ must converge to some value v' that is equivalent (e.g. β-equivalent) to $\llbracket v \rrbracket$, and vice-versa. This is formally stated as two properties, *soundness* and *completeness*. For our CBN-to-CBV translation, these properties take the following form:

4.1 Soundness

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

In other words, any computation in the CBV domain starting from the image $[\![e]\!]$ of a CBN program e must accurately reflect the computation in the CBN domain.

4.2 Completeness

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

In other words, any computation in the CBN domain starting from e must be accurately reflected by the computation in the CBV domain starting from the image $[\![e]\!]$.

4.3 Nontermination

It must also be the case that the source and target agree on nonterminating executions. We write $e \Uparrow$ and say that e *diverges* if there exists an infinite sequence of expressions $e_1, e_2, \ldots$ such that $e \to e_1 \to e_2 \to \ldots$. The additional condition for adequacy is

$$e \Uparrow_{\text{cbn}} \quad \Leftrightarrow \quad [\![e]\!] \Uparrow_{\text{cbv}} .$$

The direction $\Leftarrow$ of this implication can be considered part of the requirement for soundness, and the direction $\Rightarrow$ can be considered part of the requirement for completeness. *Adequacy* is the combination of soundness and completeness.

5 Untyped ML (uML)

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

5.1 Expressions

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

5.2 Values

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

5.3 Evaluation Contexts

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

5.4 Reductions

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

We can already see that types will be important for establishing soundness. For example, what happens with the expression **if** 3 **then** 1 **else** 0? The evaluation is stuck, because there is no reduction rule that applies to this term.

5.5 Translating uML to the CBV λ-Calculus

We define some of the translation rules:

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

Revisiting our earlier example **if** 3 **then** 1 **else** 0, we see that the translation to CBV is not sound, because its image $[\![$**if** 3 **then** 1 **else** 0$]\!]$ reduces to a value under the CBV rules—there is no way for a closed term to get stuck in the CBV or CBN λ-calculus, as we proved last time. However, this value does not correspond to the stuck non-value **if** 3 **then** 1 **else** 0 in the uML language.

One possible solution to this difficulty is to introduce rules that reduce stuck expressions to a special error value. This is essentially the same as runtime type checking. Another approach is to rule out offending programs by constraining the syntax using typing rules.