Lecturer: Andrew Myers

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 $\{(0, 0), (1, 1), \ldots\}$.

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 $[\![e]\!]_{\mathcal{C}}$.

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:

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

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

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

These are fine as operational semantics, but the CBN semantics rules don't do a good job of capturing why CBN is more expensive than CBV. We can see this better by constructing a translation from CBN to CBV. That is, we treat the CBV calculus as the meaning space. Because CBV is closer to how the underlying machine works, this translation exposes some issues that need to be addressed when implementing a lazy language.

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

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

The key issue is how to make function application lazy in the arguments. CBV evaluation will eagerly evaluate all the argument expressions, so they need to be protected from evaluation. This is accomplished by wrapping the expressions passed as function arguments inside λ-abstractions to delay their evaluation. When the value of a variable is really needed, the abstraction can be passed a dummy parameter to evaluate its body.

For an example, recall that we defined:

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

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

$$\begin{aligned}
[\![\textbf{true}]\!] &= [\![\lambda xy.\, x]\!] \\
&= \lambda xy.\, [\![x]\!] \\
&= \lambda xy.\, x\ I \\
[\![\textbf{false}]\!] &= \lambda xy.\, y\ I \\
[\![\textbf{if}]\!] &= [\![\lambda xyz.\, xyz]\!] \\
&= \lambda xyz.\, [\![(xy)z]\!] \\
&= \lambda xyz.\, [\![xy]\!]\ (\lambda d.\, [\![z]\!]) \\
&= \lambda xyz.\, [\![x]\!]\ (\lambda d.\, [\![y]\!])\ (\lambda d.\, [\![z]\!]) \\
&= \lambda xyz.\, (x\ I)\ (\lambda d.\, y\ I)\ (\lambda d.\, z\ I).
\end{aligned}$$

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,

$$\begin{aligned}
[\![\textbf{if true}\ e_1\ e_2]\!] &= [\![\textbf{if}]\!]\ (\lambda d.\, [\![\textbf{true}]\!])\ (\lambda d.\, [\![e_1]\!])\ (\lambda d.\, [\![e_2]\!]) \\
&= (\lambda xyz.\, (x\ I)\ (\lambda d.\, y\ I)\ (\lambda d.\, z\ I))\ (\lambda d.\, [\![\textbf{true}]\!])\ (\lambda d.\, [\![e_1]\!])\ (\lambda d.\, [\![e_2]\!]) \\
&\longrightarrow ((\lambda d.\, [\![\textbf{true}]\!])\ I)\ (\lambda d.\, (\lambda d.\, [\![e_1]\!])\ I)\ (\lambda d.\, (\lambda d.\, [\![e_2]\!])\ I) \\
&\longrightarrow [\![\textbf{true}]\!]\ (\lambda d.\, [\![e_1]\!])\ (\lambda d.\, [\![e_2]\!]) \\
&= (\lambda xy.\, x\ I)\ (\lambda d.\, [\![e_1]\!])\ (\lambda d.\, [\![e_2]\!]) \\
&\longrightarrow (\lambda d.\, [\![e_1]\!])\ I \\
&\longrightarrow [\![e_1]\!],
\end{aligned}$$

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 reach a non-value from which there is no further transition. 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:

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

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 in some way (e.g., β-equivalence) 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

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

In other words, any computation in the CBV domain starting from the image $\llbracket e \rrbracket$ of a CBN program e must accurately reflect some computation in the CBN domain. Intuitively, no bad things happen in the target language.

4.2 Completeness

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

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 $\llbracket e \rrbracket$. Intuitively, all good things can happen in the target language.

4.3 Nontermination

It must also be the case that the source and target agree on nonterminating executions. Assuming that the source language never gets stuck, this follows from the soundness and completeness properties. But in general, the source language may get stuck. We write $e \Uparrow$ and say that e *diverges* if there exists an infinite sequence of expressions $e_1, e_2, \ldots$ such that $e \longrightarrow e_1 \longrightarrow e_2 \longrightarrow \ldots$. The additional condition for adequacy is

$$e \Uparrow_{\text{cbn}} \quad \Longleftrightarrow \quad \llbracket e \rrbracket \Uparrow_{\text{cbv}} .$$

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

5 Proving adequacy *

We would like to show that evaluation commutes with translation in our CBV$\rightarrow$CBN translation. To do this we first need a notion of target term equivalence ($\approx$) that is preserved by evaluation. This is made more challenging because as evaluation takes place in the target language, intermediate terms are generated that are not the translation of any source term. For some translations (but not this one), the reverse may also happen. Therefore, equivalence needs to allow for some extra β redexes that appear during translation. We can define this equivalence by structural induction on CBV target terms.

$$
\begin{array}{rcll}
x & \approx & x & \\
\lambda x.\, t & \approx & \lambda x.\, t' & (\text{if } t \approx t') \\
t_0\ t_1 & \approx & t_0'\ t_1' & (\text{if } t_0 \approx t_0' \text{ and } t_1 \approx t_1') \\
t & \approx & (\lambda z.\, t)\ I & (\text{if } z \notin FV(t))
\end{array}
$$

Here, t represents target terms, to keep them distinct from source terms e. We also include rules so that the relation $\approx$ is reflexive, symmetric, and transitive. Clearly, if two terms are considered equivalent with respect to this relation, they will have the same β-normal form.

The approach to showing adequacy is to show that each step in the source term is mirrored by evaluation steps in the corresponding target term, and vice versa. So we define a correspondence between source and target terms that is more general than the translation $[\![\cdot]\!]$, and is preserved during evaluation of both source and target.

We write $e \lesssim t$ to mean that CBN term e corresponds to CBV term t. The following proposition captures the idea that CBV evaluation simulates CBN evaluation at the level of individual steps:

$$
e \lesssim t \wedge e \rightarrow e' \Rightarrow \exists t'.t \rightarrow^{*} t' \wedge e' \lesssim t' \tag{1}
$$

This can be visualized as a commutation diagram:

$$
\begin{tikzcd}
e \arrow[r] \arrow[d, leftrightarrow, "\lesssim"] & e' \arrow[d, leftrightarrow, "\lesssim"] \\
t \arrow[r, "*"] & t' \ (\approx\ [\![e']\!])
\end{tikzcd}
$$

In fact, since in this case the source language cannot get stuck during evaluation, and both languages have deterministic evaluation, (1) ensures that evaluation in each language corresponds to the other.

We define the relation $\lesssim$ in such a way that $e \lesssim [\![e]\!]$. Then, using (1), we can show that any trace in the source language produces a corresponding trace in the target, by induction on the number of source-language steps.

We define the relation $\lesssim$ as follows:

$$
\begin{array}{rcll}
x & \lesssim & x\ I & (2) \\
\lambda x.\, e & \lesssim & \lambda x.\, t \quad (\text{if } e \lesssim t) & (3) \\
e_0\ e_1 & \lesssim & t_0\ (\lambda\,.\, t_1) \quad (\text{if } e_0 \lesssim t_0,\ e_1 \lesssim t_1) & (4) \\
e & \lesssim & (\lambda\,.\, t)\ I \quad (\text{if } e \lesssim t) & (5)
\end{array}
$$

For simplicity, we ignore the fresh variable that would be used in the new lambda abstraction in line (4).

Lines (2–4) straightforwardly ensure that a source term corresponds to its translation. Line (5) is different; it takes care of the extra β reductions that crop up during evaluation. Because the t side of the $\lesssim$ relation becomes structurally smaller in this rule's premise, the definition of the relation is still well-founded. Lines (2–4) are well-founded based on the structure of e; Line (5) is well-founded based on the structure of t. If we were proving a more complex translation correct, we would need more rules like (5) for other meaning-preserving target-language reductions.

First, let's warm up by showing that a term corresponds to its translation.

Lemma 1

$$
e \lesssim [\![e]\!]
$$

Proof: an easy structural induction on e.

- Case x: $x \lesssim x\ I$ by definition.
- Case $\lambda x.\, e'$: We have $[\![e]\!] = \lambda x.\, [\![e']\!]$. By the induction hypothesis (IH), $e' \lesssim [\![e']\!]$, so $\lambda x.\, e' \lesssim \lambda x.\, [\![e']\!]$ by (3).
- Case $e_0\ e_1$: We have $[\![e]\!] = [\![e_0]\!]\ (\lambda\,.\,[\![e_1]\!])$. By the IH, $e_0 \lesssim [\![e_0]\!]$ and $e_1 \lesssim [\![e_1]\!]$. Therefore by (4), $e_0\ e_1 \lesssim [\![e_0]\!]\ [\![e_1]\!]$.

Next, let's show that if e corresponds to t, its translation is equivalent to t:

Lemma 2

$$e \lesssim t \Rightarrow [\![e]\!] \approx t$$

Proof: an induction on the derivation of $e \lesssim t$.

- Case $x \lesssim x\ I$:
  Trivial: $[\![x]\!] = x\ I$.
- Case $\lambda x.\, e' \lesssim \lambda x.\, t'$ where $e' \lesssim t'$:
  Here, $[\![e]\!] = \lambda x.\, [\![e']\!]$. IH: $[\![e']\!] \approx t'$. Therefore $\lambda x.\, [\![e']\!] \approx \lambda x.\, t'$ as required.
- Case $e_0\ e_1 \lesssim t_0\ (\lambda\,.\,t_1)$ where $e_0 \lesssim t_0$ and $e_1 \lesssim t_1$:
  Here, $[\![e_0\ e_1]\!] = [\![e_0]\!](\lambda\,.\,[\![e_1]\!])$, and by the IH, $[\![e_0]\!] \approx t_0$ and $[\![e_1]\!] \approx t_1$. So from the definition of $\approx$, we have $[\![e_0]\!](\lambda\,.\,[\![e_1]\!]) \approx t_0\ (\lambda\,.\,t_1)$.
- Case $e \lesssim (\lambda\,.\,t)\ I$ where $e \lesssim t$:
  IH: $[\![e]\!] \approx t$. But $t \approx (\lambda\,.\,t)\ I$, and $\approx$ is transitive.

Given these definitions, we can prove (1) by induction on the derivation of $e \lesssim t$. We will need two useful lemmas. The first is a substitution lemma that says substituting corresponding terms into corresponding terms produces corresponding terms:

Lemma 3

$$e_1 \lesssim t_1 \wedge e_2 \lesssim t_2 \Rightarrow e_2\{e_1/x\} \lesssim t_2\{\lambda.\, t_1/x\}$$

Proof. By induction on the derivation of $e_2 \lesssim t_2$.

- Case $x \lesssim x\ I$:
  We have $e_2\{e_1/x\} = e_1$ and $t_2\{\lambda.\, t_1/x\} = (\lambda.\, t_1)\ I$. By rule (5), we have $e_1 \lesssim (\lambda.\, t_1)\ I$.
- Case $y \lesssim y\ I$ where $y \neq x$:
  Trivial: substitution has no effect.
- Case $\lambda x.\, e \lesssim \lambda x.\, t$ where $e \lesssim t$:
  Trivial: The substitutions into e_2 and t_2 have no effect.
- Case $\lambda y.\, e \lesssim \lambda y.\, t$ where $e \lesssim t$, $x \neq y$:
  Here $e_2\{e_1/x\} = \lambda y.\, e\{e_1/x\}$ and $t_2\{\lambda.\, t_1/x\} = \lambda y.\, t\{\lambda.\, t_1/x\}$. Since $e \lesssim t$, by the induction hypothesis we have $e\{e_1/x\} \lesssim t\{\lambda.\, t_1/x\}$. Therefore by (3), $\lambda y.\, e\{e_1/x\} \lesssim \lambda y.\, t\{\lambda.\, t_1/x\}$, as required.
- Case $e\ e' \lesssim t\ (\lambda.\, t')$, where $e \lesssim t$ and $e' \lesssim t'$:
  We have $e_2\{e_1/x\} = e\{e_1/x\}\ e'\{e_1/x\}$, and $t_2\{\lambda.\, t_1/x\} = t\{\lambda.\, t_1/x\}\ (\lambda.\, t'\{t_1/x\})$. From the induction hypothesis, $e\{e_1/x\} \lesssim t\{\lambda.\, t_1/x\}$ and $e'\{e_1/x\} \lesssim t'\{\lambda.\, t_1/x\}$. Therefore, by (4) we have $e\{e_1/x\}\ e'\{e_1/x\} \lesssim t\{\lambda.\, t_1/x\}\ (\lambda.\, t'\{t_1/x\})$.

- Case $e_2 \lesssim (\lambda\,.\,t_2')\ I$, where $e_2 \lesssim t_2'$:
  We need to show that $e_2\{e_1/x\} \lesssim ((\lambda\,.\,t_2')\ I)\{\lambda\,.\,t_1/x\}$; that is, $e_2\{e_1/x\} \lesssim ((\lambda\,.\,t_2'\{\lambda\,.\,t_1/x\})\ I)$. From the induction hypothesis, we have $e_2\{e_1/x\} \lesssim t_2'\{\lambda.\,t_1/x\}$. By (5), this means $e_2\{e_1/x\} \lesssim (\lambda\,.\,t_2'\{\lambda.\,t_1/x\})\ I$.

The next lemma we need says that if a value $\lambda x.\,e$ corresponds to a term t, then t reduces to a corresponding lambda term $\lambda\,.\,t'$.

Lemma 4

$$\lambda x.\,e \lesssim t \Rightarrow \exists t'.\ t \longrightarrow^* \lambda x.\,t' \ \wedge\ e \lesssim t'$$

Proof. By induction on the derivation of $\lambda x.\,e \lesssim t$.

- Case $y \lesssim y\ I$: Impossible, as $y \neq \lambda x.\,e$.
- Case $\lambda x.\,e \lesssim \lambda x.\,t'$ where $e \lesssim t'$:
  Here, $t = \lambda x.\,t'$, and the result is immediate.
- Case $e_0\ e_1 \lesssim t_0\ (\lambda.\,t_1)$: Impossible, as $e_0\ e_1 \neq \lambda x.\,e$.
- Case $e_0 \lesssim (\lambda.\,t_0)\ I$, where $e_0 \lesssim t_0$:
  In this case $e_0 = \lambda x.\,e$, and $t = ((\lambda.\,t_0)\ I)$. By the inductive hypothesis, there is some t' such that $t_0 \longrightarrow^* \lambda x.\,t'$ and $e \lesssim t'$. Since $t = ((\lambda.\,t_0)\ I) \longrightarrow t_0$ we have $t \longrightarrow^* \lambda x.\,t'$, as required.

We are now ready to prove (1).

Proof. By induction on the derivation of $e \lesssim t$:

- Case $x \lesssim x\ I$: Vacuously true, as there is no evaluation step $e \longrightarrow e'$.
- Case $\lambda x.\,e \lesssim \lambda x.\,t$: A value: also vacuously true.
- Case $e_0\ e_1 \lesssim t_0\ (\lambda\,.\,t_1)$, where $e_0 \lesssim t_0$ and $e_1 \lesssim t_1$:
  We show this by cases on the derivation of $e \longrightarrow e'$:
  - Case $e_0\ e_1 \longrightarrow e_0'\ e_1$, where $e_0 \longrightarrow e_0'$:
    By the induction hypothesis, $\exists t_0'.e_0' \lesssim t_0' \wedge t_0 \longrightarrow^* t_0'$. It is easy to see that therefore $t_0\ (\lambda.\,t_1) \longrightarrow^* t_0'\ (\lambda.\,t_1)$. So by (4), $e_0'\ e_1 \lesssim t_0'\ (\lambda.\,t_1)$, as required.
  - Case $(\lambda x.\,e_2)\ e_1 \longrightarrow e_2\{e_1/x\}$:
    Here $\lambda x.\,e_2 \lesssim t_0$ and $e_1 \lesssim t_1$.
    By Lemma 4, there exists a t_2 such that $t_0 \longrightarrow^* \lambda x.\,t_2$ and $e_2 \lesssim t_2$. Therefore, we have $t_0\ (\lambda\,.\,t_1) \longrightarrow^* (\lambda x.\,t_2)\ (\lambda\,.\,t_1) \longrightarrow t_2\{\lambda.\,t_1/x\}$. But from the substitution lemma above (Lemma 3), we know that $e_2\{e_1/x\} \lesssim t_2\{\lambda.\,t_1/x\}$, as required.
- Case $e_0 \lesssim (\lambda.\,t_0)\ I$, where $e_0 \lesssim t_0$:
  By the induction hypothesis, $\exists t_0'.e_0 \lesssim t_0' \wedge t_0 \longrightarrow^* t_0'$. It is easy to see that therefore $((\lambda.\,t_0)\ I) \longrightarrow t_0 \ \longrightarrow^* t_0'$, as required.

Having proved (1), we can show completeness of the translation. If we start with a source term e and its translation $[\![e]\!]$, we know from Lemma 1 that $e \lesssim [\![e]\!]$. From (1), we know that each step of evaluation of e is mirrored by execution on the target side that preserves $e \lesssim t$. If the evaluation of e diverges, so will the evaluation of $[\![e]\!]$. If the evaluation of e converges on a value v, then the evaluation of $[\![e]\!]$ will reach a convergent (by Lemma 4) term t such that $v \lesssim t$. And by Lemma 2, $[\![v]\!] \approx t$. This demonstrates completeness.

To show soundness of the translation, we need to show that every evaluation in the target language corresponds to some evaluation in the source language. Suppose we have a target-language evaluation $t \longrightarrow^* v'$, where $t = [\![e]\!]$, but there is no corresponding source-language evaluation of e. There are three possibilities. First, the evaluation of e could get stuck. This can't happen for this source language because all terms are either values or have a legal evaluation. Second, the evaluation of e could evaluate to a value v. But then v must correspond to v', because the target-language evaluation is deterministic. Third, the evaluation of e might diverge. But then (1) says there is a divergent target-language evaluation. The determinism of the target language ensures that can't happen.