Our goal is to study programming language features using various semantic techniques. So far we have seen small-step and big-step operational semantics. However, there are other ways to specify semantics, and they can give useful insights that may not be apparent in the operational semantics. A different way to give semantics is by defining a *translation* from the programming language to another language that is better understood (and typically simpler). This is essentially a process of *compilation*, in which a source language is converted to a target language. Later on we will see that the target language can even be mathematical structures, in which case we refer to the semantics as a *denotational semantics*. A third style of semantics is *axiomatic semantics*, which we will also discuss later in the course.

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

2 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:

$$(\lambda x.\, e_1)\, e_2 \ \xrightarrow{1}\ e_1\{e_2/x\} \qquad E \ ::= \ [\cdot] \ \mid \ E\, e.$$

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

$$(\lambda x.\, e)\, v \ \xrightarrow{1}\ e\{v/x\} \qquad E \ ::= \ [\cdot] \ \mid \ E\, e \ \mid \ v\, E.$$

These are fine as operational semantics, but the CBN semantics rules do not adequately capture why CBV is as expressive as CBN. We can see this better by constructing a translation from CBN to CBV. That is, we treat the CBV calculus as the meaning space. 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, 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.

We define the semantic function $[\![\cdot]\!]$ by induction on the structure of the translated expression:

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

For an example, recall that we defined:

$$
\mathsf{true} \triangleq \lambda xy.\, x \qquad\qquad \mathsf{false} \triangleq \lambda xy.\, y \qquad\qquad \mathsf{if} \triangleq \lambda xyz.\, xyz.
$$

The problem with this construction in the CBV λ-calculus is that $\mathsf{if}\, b\, e_1\, e_2$ evaluates both e_1 and e_2, regardless of the truth value of b. The conversion above fixes this problem.

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

Now, translating $\mathsf{if}\,\mathsf{true}\, e_1\, e_2$ and evaluating under the CBV rules,

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

and e_2 was never evaluated.

3 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:

$$
\begin{tikzcd}
e \arrow[rr, "*"] \arrow[d, "{[\![\cdot]\!]}"'] & & v \arrow[d, "{[\![\cdot]\!]}"] \\
{[\![e]\!]} \arrow[r, "*"] & v' \approx & {[\![v]\!]}
\end{tikzcd}
$$

If an expression e converges to a value v in zero or more steps in the source language, then $[\![e]\!]$ must converge to some value v' that is equivalent (e.g. β-equivalent) to $[\![v]\!]$, 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:

3.1 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.

3.2 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 $[\![e]\!]$.

3.3 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.

4 Proving Adequacy *

We would like to show that evaluation commutes with translation in our CBN $\to$ CBV 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 according to the following rules:

$$x \approx x \qquad \frac{t \approx t'}{\lambda x.\, t \approx \lambda x.\, t'} \qquad \frac{t_0 \approx t_0' \qquad t_1 \approx t_1'}{t_0\, t_1 \approx t_0'\, t_1'} \qquad t \approx (\lambda z.\, t)\, \mathsf{id},\text{ where } z \notin FV(t)$$

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. One can show easily that if two terms are equivalent with respect to this relation, then they have the same β-normal form.

To show adequacy, we show that each evaluation step in the source term is mirrored by a sequence of evaluation steps in the corresponding target term, and vice versa. So we define a correspondence $\lesssim$ 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:

$$e \lesssim t \ \wedge \ e \rightarrow e' \quad \Rightarrow \quad \exists t' \ t \xrightarrow{*} t' \ \wedge \ e' \lesssim t' \tag{1}$$

This can be visualized as a commutative 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 $\lesssim$ by the following rules:

$$x \lesssim x \, \mathsf{id} \qquad \frac{e \lesssim t}{\lambda x.\, e \lesssim \lambda x.\, t} \qquad \frac{e_0 \lesssim t_0 \qquad e_1 \lesssim t_1}{e_0 \, e_1 \lesssim t_0 \, (\lambda \, .\, t_1)} \qquad \frac{e \lesssim t}{e \lesssim (\lambda \, .\, t) \, \mathsf{id}} \tag{2}$$

For simplicity, we ignore the fresh variable that would be used in the new lambda abstraction in the last two rules.

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

First, let us 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: $x \lesssim x \, \mathsf{id}$ by definition.
- Case $\lambda x.\, e'$: We have $[\![e]\!] = \lambda x.\, [\![e']\!]$. By the induction hypothesis, $e' \lesssim [\![e']\!]$, so $\lambda x.\, e' \lesssim \lambda x.\, [\![e']\!]$ by the second rule of (2).

- Case $e_0\, e_1$: We have $[\![e]\!] = [\![e_0]\!]\,(\lambda\,.\,[\![e_1]\!])$. By the induction hypothesis, $e_0 \lesssim [\![e_0]\!]$ and $e_1 \lesssim [\![e_1]\!]$. Therefore by the third rule of (2), $e_0\, e_1 \lesssim [\![e_0]\!]\,(\lambda.\,[\![e_1]\!])$.

□

Next, let us show that if e corresponds to t, its translation is equivalent to t.

Lemma 2. $e \lesssim t \;\Rightarrow\; [\![e]\!] \approx t$.

Proof. Induction on the derivation of $e \lesssim t$.

- Case $x \lesssim x\,\mathsf{id}$:
  This case is trivial: $[\![x]\!] = x\,\mathsf{id}$.
- Case $\lambda x.\, e' \lesssim \lambda x.\, t'$ where $e' \lesssim t'$:
  Here, $[\![e]\!] = \lambda x.\,[\![e']\!]$. By the induction hypothesis, $[\![e']\!] \approx t'$, 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 induction hypothesis, $[\![e_0]\!] \approx t_0$ and $[\![e_1]\!] \approx t_1$. From the definition of $\approx$, we have $[\![e_0]\!](\lambda\,.\,[\![e_1]\!]) \approx t_0\,(\lambda\,.\,t_1)$.
- Case $e \lesssim (\lambda\,.\,t)\,\mathsf{id}$ where $e \lesssim t$:
  The induction hypothesis is $[\![e]\!] \approx t$. But $t \approx (\lambda\,.\,t)\,\mathsf{id}$, 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. We proceed by induction on the derivation of $e_2 \lesssim t_2$.

- Case $x \lesssim x\,\mathsf{id}$:
  We have $e_2\,\{e_1/x\} = e_1$ and $t_2\,\{\lambda.\, t_1/x\} = (\lambda.\, t_1)\,\mathsf{id}$. By the fourth rule of (2), we have $e_1 \lesssim (\lambda.\, t_1)\,\mathsf{id}$.
- Case $y \lesssim y\,\mathsf{id}$ where $y \neq x$:
  This case is trivial, as the substitution has no effect.
- Case $\lambda x.\, e \lesssim \lambda x.\, t$ where $e \lesssim t$:
  Again, this case is trivial, as 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 (2), $\lambda y.\, e\,\{e_1/x\} \lesssim \lambda y.\, t\,\{\lambda.\, t_1/x\}$, 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 (2) we have $e\,\{e_1/x\}\, e'\,\{e_1/x\} \lesssim t\,\{\lambda.\, t_1/x\}\,(\lambda.\, t'\,\{t_1/x\})$.

- Case $e_2 \lesssim (\lambda \,.\, t_2')\,\mathsf{id}$, where $e_2 \lesssim t_2'$:
  We need to show that $e_2\,\{e_1/x\} \lesssim ((\lambda \,.\, t_2')\,\mathsf{id})\,\{\lambda \,.\, t_1/x\}$; that is, $e_2\,\{e_1/x\} \lesssim ((\lambda \,.\, t_2'\,\{\lambda \,.\, t_1/x\})\,\mathsf{id})$. From the induction hypothesis, we have $e_2\,\{e_1/x\} \lesssim t_2'\,\{\lambda.\, t_1/x\}$. By (2), this means $e_2\,\{e_1/x\} \lesssim (\lambda \,.\, t_2'\,\{\lambda.\, t_1/x\})\,\mathsf{id}$.

□

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

Lemma 4. $\lambda x.\, e \lesssim t \Rightarrow \exists t' \;\; t \rightarrow \lambda x.\, t' \;\wedge\; e \lesssim t'$.

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

- Case $y \lesssim y\,\mathsf{id}$: 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 $e_0 \lesssim (\lambda.\, t_0)\,\mathsf{id}$, where $e_0 \lesssim t_0$:
  In this case $e_0 = \lambda x.\, e$, and $t = ((\lambda.\, t_0)\,\mathsf{id})$. By the induction hypothesis, there is some t' such that $t_0 \rightarrow \lambda x.\, t'$ and $e \lesssim t'$. Since $t = ((\lambda.\, t_0)\,\mathsf{id}) \xrightarrow{1} t_0$ we have $t \rightarrow \lambda x.\, t'$, as required.

□

We are now ready to prove (1).

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

- Case $x \lesssim x\,\mathsf{id}$: Vacuously true, as there is no evaluation step $e \xrightarrow{1} e'$.
- 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 $e \xrightarrow{1} e'$:
  - Case $e_0\, e_1 \xrightarrow{1} e_0'\, e_1$, where $e_0 \xrightarrow{1} e_0'$:
    By the induction hypothesis, $\exists t_0' \;\, e_0' \lesssim t_0' \wedge t_0 \rightarrow t_0'$. It is easy to see that $t_0\,(\lambda.\, t_1) \rightarrow t_0'\,(\lambda.\, t_1)$. By the third rule of (2), $e_0'\, e_1 \lesssim t_0'\,(\lambda.\, t_1)$, as required.
  - Case $(\lambda x.\, e_2)\, e_1 \xrightarrow{1} e_2\,\{e_1/x\}$:
    Here $\lambda x.\, e_2 \lesssim t_0$ and $e_1 \lesssim t_1$.
    By Lemma 4, there exists a t_2 such that $t_0 \rightarrow \lambda x.\, t_2$ and $e_2 \lesssim t_2$. Therefore, we have $t_0\,(\lambda \,.\, t_1) \rightarrow (\lambda x.\, t_2)\;(\lambda \,.\, t_1) \xrightarrow{1} t_2\,\{\lambda.\, t_1/x\}$. 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 $e_0 \lesssim (\lambda.\, t_0)\;\mathsf{id}$, where $e_0 \lesssim t_0$:
  By the induction hypothesis, $\exists t_0' \;\, e_0 \lesssim t_0' \;\wedge\; t_0 \rightarrow t_0'$. It is easy to see that therefore $((\lambda.\, t_0)\;\mathsf{id}) \xrightarrow{1} t_0 \rightarrow t_0'$, 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 $t \rightarrow v'$, 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 cannot 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 cannot happen.