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 meaning, 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 \( \lambda x : \text{int}. x \) 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 \( [e]_K \) in the meaning space under the semantic function. We may occasionally add other decorations to distinguish between different semantic functions, as for example \( [e]_{cbn} \) or \( C[e] \).

2 Translating CBN \( \lambda \)-Calculus into CBV \( \lambda \)-Calculus

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

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

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

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

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 more clearly 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 \( \lambda \)-calculus to the CBV \( \lambda \)-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 \( \lambda \)-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:

\[
[x] \triangleq x \text{id} \quad (\text{id} = \lambda z. z)
\]

\[
[\lambda x. e] \triangleq \lambda x. [e]
\]

\[
[e_1 e_2] \triangleq [e_1] (\lambda z. [e_2]), \quad \text{where } z \notin \text{FV}([e_2]).
\]

For an example, recall that we defined:

\[
\text{true} \triangleq \lambda xy.x \quad \text{false} \triangleq \lambda xy.y \quad \text{if} \triangleq \lambda xyz.xyz.
\]

The problem with this construction in the CBV $\lambda$-calculus is that 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.

\[
[\text{true}] = \lambda xy. x \quad \text{false} = \lambda xy. y \quad \text{if} = \lambda xyz. xyz.
\]

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

\[
[\text{if true} \ e_1 \ e_2] = ([\text{if} \ (\lambda d. \text{true}) \ (\lambda d. \text{true}) \ (\lambda d. \text{true})] \ (\lambda d. \text{true})) (\lambda d. \text{true}))
\]

\[
= (\lambda xy. x \text{id}) (\lambda d. \text{true}) (\lambda d. \text{true}) (\lambda d. \text{true}) (\lambda d. \text{true})
\]

\[
\xrightarrow{3}\ ((\lambda d. \text{true}) \text{id}) (\lambda d. \text{true}) (\lambda d. \text{true}) (\lambda d. \text{true}) (\lambda d. \text{true})
\]

\[
\xrightarrow{1}\ \lambda xy. x \text{id}
\]

and $e_2$ was never evaluated.

3 Adequacy

Both the CBV and CBN $\lambda$-calculus are deterministic reduction strategies in the sense that there is at most one reduction that is enabled in any term. When an expression $e$ in a language is evaluated in a deterministic system, one of three things can happen:

1. There exists an infinite sequence of expressions $e_1, e_2, \ldots$ such that $e \xrightarrow{1} e_1 \xrightarrow{1} e_2 \xrightarrow{1} \ldots$. In this case, we write $e \uparrow$ and say that $e$ diverges.

2. The expression $e$ produces a value $v$ in zero or more steps. In this case we say that $e$ converges to the value $v$ and write $e \Downarrow v$.

3. The computation converges to a non-value. When this happens, we say the computation is stuck.

A semantic translation is adequate if these three behaviors in the source system are accurately reflected in the target system, and vice versa. This relationship is illustrated in the following diagram:

\footnote{This cannot happen with our CBN-to-CBV translation, but we will see some examples soon enough.}
If $e$ converges to a value $v$ in the source language, then $\llbracket e \rrbracket$ must converge to some value $v'$ that is equivalent (e.g., $\beta$-equivalent) to $\llbracket v \rrbracket$ in the target language, 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:

**Soundness:**

(i) $\llbracket e \rrbracket \downarrow_{cbv} t \Rightarrow \exists v t \approx \llbracket v \rrbracket \land e \downarrow_{cbn} v$

(ii) $\llbracket e \rrbracket \uparrow_{cbv} \Rightarrow e \uparrow_{cbn}$

**Completeness:**

(i) $e \downarrow_{cbn} v \Rightarrow \exists t t \approx \llbracket v \rrbracket \land \llbracket e \rrbracket \downarrow_{cbv} t$

(ii) $e \uparrow_{cbn} \Rightarrow \llbracket e \rrbracket \uparrow_{cbv}$.

where $\approx$ is some notion of target term equivalence that is preserved by evaluation.

Soundness says that the computation in the CBV domain starting from the image $\llbracket e \rrbracket$ of a CBN program $e$ accurately simulates the computation starting from $e$ in the CBN domain. Thus if the CBV process terminates in a value, then so must the CBN process, and the values must be related in the sense described formally above; and if the CBV computation diverges, then so does the CBN computation. Completeness says the opposite: the computation in the CBN domain starting from $e$ is accurately simulated by the computation in the CBV domain starting from $\llbracket e \rrbracket$.

Adequacy is the combination of soundness and completeness.

4 Proving Adequacy

We would like to show that evaluation commutes with our translation $\llbracket \cdot \rrbracket$ from CBN to CBV. 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 $\beta$-redexes that appear during translation. We can define this equivalence by structural induction on CBV target terms according to the following rules:

\[
\begin{align*}
x & \approx x \\
x & \approx x \\
\lambda x.t & \approx \lambda x.t' \\
\lambda z.t & \approx \lambda z.t \\
t_0 & \approx t'_0 \\
t_0 t_1 & \approx t'_0 t'_1 \\
t & \approx (\lambda z.t) \text{id}, \text{ where } z \notin \text{FV}(t)
\end{align*}
\]

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 $\beta$-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 $\preceq$ 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 \(e \leq t\) 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:

\[
e \leq t \land e \rightarrow e' \Rightarrow \exists t' \rightarrow t' \land e' \leq t'
\]  

(1)

This can be visualized as a commutative diagram:

\[
\begin{array}{ccc}
e & \xrightarrow{1} & e' \\
\Downarrow \approx & \searrow \downarrow \swarrow & \Downarrow \leq \\
t & \rightarrow * & t' (\approx [e'])
\end{array}
\]

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 \(\leq\) in such a way that \(e \leq [e]\). 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 \(\leq\) by the following rules:

\[
x \leq x \text{id} \quad e \leq t \quad e_0 \leq t_0 \quad e_1 \leq t_1 \quad e \leq \lambda x. t \\
\lambda x. e \leq \lambda x. t \quad e_0 e_1 \leq t_0 (\lambda . t_1) \quad e \leq \lambda x. [e] \text{id}
\]

(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 \(\beta\)-reductions that may arise during evaluation. Because the right-hand side of the \(\leq\) 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.** \(e \leq [e]\).

**Proof.** An easy structural induction on \(e\).

- Case \(x\): \(x \leq x \text{id}\) by definition.
- Case \(\lambda x. e'\): We have \([e] = \lambda x. [e']\). By the induction hypothesis, \(e' \leq [e']\), so \(\lambda x. e' \leq \lambda x. [e']\) by the second rule of (2).
- Case \(e_0 e_1\): We have \([e] = [e_0] (\lambda . [e_1])\). By the induction hypothesis, \(e_0 \leq [e_0]\) and \(e_1 \leq [e_1]\). Therefore by the third rule of (2), \(e_0 e_1 \leq [e_0] [e_1]\).
Lemma 2. \( e \leq t \Rightarrow \llbracket e \rrbracket \approx t \).

Proof. Induction on the derivation of \( e \leq t \).

- Case \( x \leq x \, \mathrm{id} \):  
  This case is trivial: \( \llbracket x \rrbracket = x \, \mathrm{id} \).

- Case \( \lambda x. e' \leq \lambda x. t' \) where \( e' \leq t' \):  
  Here, \( \llbracket e \rrbracket = \lambda x. \llbracket e' \rrbracket \). By the induction hypothesis, \( \llbracket e' \rrbracket \approx t' \), therefore \( \lambda x. \llbracket e' \rrbracket \approx \lambda x. t' \) as required.

- Case \( e_0 \, e_1 \leq t_0 \, (\lambda \, t_1) \) where \( e_0 \leq t_0 \) and \( e_1 \leq t_1 \):  
  Here, \( \llbracket e_0 \, e_1 \rrbracket = \llbracket e_0 \rrbracket (\lambda \, \llbracket e_1 \rrbracket) \), and by the induction hypothesis, \( \llbracket e_0 \rrbracket \approx t_0 \) and \( \llbracket e_1 \rrbracket \approx t_1 \). From the definition of \( \approx \), we have \( \llbracket e_0 \rrbracket (\lambda \, \llbracket e_1 \rrbracket) \approx t_0 \, \lambda \, t_1 \).

- Case \( e \leq (\lambda \, t) \, \mathrm{id} \) where \( e \leq t \):  
  The induction hypothesis is \( \llbracket e \rrbracket \approx t \). But \( t \approx (\lambda \, t) \, \mathrm{id} \), and \( \approx \) is transitive.

Given these definitions, we can prove (1) by induction on the derivation of \( e \leq t \). 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. \( e_1 \leq t_1 \land e_2 \leq t_2 \Rightarrow e_2 \, \{ e_1 / x \} \leq t_2 \, \{ \lambda \, t_1 / x \} \).

Proof. We proceed by induction on the derivation of \( e_2 \leq t_2 \).

- Case \( x \leq x \, \mathrm{id} \):  
  We have \( e_2 \, \{ e_1 / x \} = e_1 \) and \( t_2 \, \{ \lambda \, t_1 / x \} = (\lambda \, t_1) \, \mathrm{id} \). By the fourth rule of (2), we have \( e_1 \leq (\lambda \, t_1) \, \mathrm{id} \).

- Case \( y \leq y \, \mathrm{id} \) where \( y \neq x \):  
  This case is trivial, as the substitution has no effect.

- Case \( \lambda x. e \leq \lambda x. t \) where \( e \leq t \):  
  Again, this case is trivial, as the substitutions into \( e_2 \) and \( t_2 \) have no effect.

- Case \( \lambda y. e \leq \lambda y. t \) where \( e \leq t \), \( x \neq y \):  
  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 \( e \leq t \), by the induction hypothesis we have \( e \, \{ e_1 / x \} \leq t \, \{ \lambda \, t_1 / x \} \). Therefore by (??), \( \lambda y \, e \, \{ e_1 / x \} \leq \lambda y \, t \, \{ \lambda \, t_1 / x \} \), as required.

- Case \( e \, e' \leq t \, (\lambda \, t') \), where \( e \leq t \) and \( e' \leq t' \):  
  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, \( e \, \{ e_1 / x \} \leq t \, \{ \lambda \, t_1 / x \} \) and \( e' \, \{ e_1 / x \} \leq t' \, \{ \lambda \, t_1 / x \} \). Therefore, by (??) we have \( e \, \{ e_1 / x \} \, e' \, \{ e_1 / x \} \leq t \, \{ \lambda \, t_1 / x \} \, (\lambda \, t' \, \{ t_1 / x \}) \).

- Case \( e_2 \leq (\lambda \, t') \, \mathrm{id} \), where \( e_2 \leq t_2' \):  
  We need to show that \( e_2 \, \{ e_1 / x \} \leq ((\lambda \, t') \, \mathrm{id}) \, \{ \lambda \, t_1 / x \} \); that is, \( e_2 \, \{ e_1 / x \} \leq ((\lambda \, t_2' \, \{ \lambda \, t_1 / x \}) \, \mathrm{id}) \). From the induction hypothesis, we have \( e_2 \, \{ e_1 / x \} \leq t_2' \, \{ \lambda \, t_1 / x \} \). By (??), this means \( e_2 \, \{ e_1 / x \} \leq (\lambda \, t_2' \, \{ \lambda \, t_1 / x \}) \, \mathrm{id} \).

\[ \square \]
The next lemma 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.** \( \lambda x. e \leq t \Rightarrow \exists t' \ t \rightarrow \lambda x. t' \land e \leq t' \).

**Proof.** By induction on the derivation of \( \lambda x. e \leq t \).

- Case \( y \leq y \text{id} \): Impossible, as \( y \neq \lambda x. e \).
- Case \( \lambda x. e \leq \lambda x. t' \) where \( e \leq t' \):
  Here, \( t = \lambda x. t' \), and the result is immediate.
- Case \( e_0 e_1 \leq t_0 (\lambda . t_1) \): Impossible, as \( e_0 e_1 \neq \lambda x. e \).
- Case \( e_0 \leq (\lambda . t_0) \text{id} \), where \( e_0 \leq t_0 \):
  In this case \( e_0 = \lambda x. e \), and \( t = ((\lambda . t_0) \text{id}) \). By the induction hypothesis, there is some \( t' \) such that \( t_0 \rightarrow \lambda x. t' \) and \( e \leq t' \). Since \( t = ((\lambda . t_0) \text{id}) \frac{1}{t_0} \) we have \( t \rightarrow \lambda x. t' \), as required.

We are now ready to prove (1).

**Proof.** By induction on the derivation of \( e \leq t \):

- Case \( x \leq x \text{id} \): Vacuously true, as there is no evaluation step \( e \frac{1}{t} e' \).
- Case \( \lambda x. e \leq \lambda x. t \): A value: also vacuously true.
- Case \( e_0 e_1 \leq t_0 (\lambda . t_1) \), where \( e_0 \leq t_0 \) and \( e_1 \leq t_1 \):
  We show this by cases on the derivation of \( e \frac{1}{t} e' \):
  - Case \( e_0 e_1 \frac{1}{t} e'_0 e_1 \), where \( e_0 \frac{1}{t} e'_0 \):
    By the induction hypothesis, \( \exists t_0' e'_0 \leq t_0' \land t_0 \rightarrow t_0' \). It is easy to see that \( t_0 (\lambda . t_1) \rightarrow t_0' (\lambda . t_1) \).
    By the third rule of (2), \( e'_0 e_1 \leq t'_0 (\lambda . t_1) \), as required.
  - Case \( (\lambda x. e_2) e_1 \frac{1}{t} e_2 \{e_1/x\} \):
    Here \( \lambda x. e_2 \leq t_0 \) and \( e_1 \leq t_1 \).
    By Lemma 4, there exists a \( t_2 \) such that \( t_0 \rightarrow \lambda x. t_2 \) and \( e_2 \leq t_2 \). Therefore, we have \( t_0 (\lambda . t_1) \rightarrow (\lambda x. t_2) (\lambda . t_1) \frac{1}{t_2} \{\lambda . t_1/x\} \). But from the substitution lemma above (Lemma 3), we know that \( e_2 \{e_1/x\} \leq t_2 \{\lambda . t_1/x\} \), as required.
- Case \( e_0 \leq (\lambda . t_0) \text{id} \), where \( e_0 \leq t_0 \):
  By the induction hypothesis, \( \exists t_0' e_0 \leq t_0' \land t_0 \rightarrow t_0' \). It is easy to see that therefore \( ((\lambda . t_0) \text{id}) \frac{1}{t_0} \rightarrow t_0 \rightarrow t_0' \), 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 \( e \leq \| e \| \). From (1), we know that each step of evaluation of \( e \) is mirrored by execution on the target side that preserves \( e \leq t \). 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 $v \preceq t$. 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.