Recall from Lecture 6 that we can define the semantics of a program *denotationally* by showing how to translate programs into a different mathematical space that we understand better. In that case, we translated programs into mathematical objects (functions from stores to either numbers, booleans, or stores). Another useful form of denotational semantics is *semantics by translation*, where we translate programs in one language into another language that we understand better—essentially the process of compilation.

To see how these relate, we will look at translating λ-calculus to a different version of itself. Specifically, we see how to translate call-by-name (CBN) λ-calculus into call-by-value (CBV) λ-calculus. This exploration has a few different purposes.

- By showing that we can translate CBN λ-calculus into CBV λ-calculus, we show that CBV λ-calculus is at least as *expressive* as CBN. That is, there are no behaviors in CBN λ-calculus that we cannot (locally) simulate in CBV.
- We will see how to precisely state and prove that a translation is correct.
- We will expose some thorny issues around implementing lazy programming languages.

1 Translating CBN λ-calculus to CBV λ-calculus

In Lecture 10 we defined the operational semantics for call-by-name (lazy) λ-calculus as follows.

$$\frac{}{(\lambda x.\, e_1)\ e_2 \longrightarrow e_1[x \mapsto e_2]} \qquad \frac{e_1 \longrightarrow e_1'}{e_1\ e_2 \longrightarrow e_1'\ e_2}$$

We defined the operational semantics for call-by-value (strict) λ-calculus as follows.

$$\frac{}{(\lambda x.\, e)\ v \longrightarrow e[x \mapsto v]} \qquad \frac{e_1 \longrightarrow e_1'}{e_1\ e_2 \longrightarrow e_1'\ e_2} \qquad \frac{e \longrightarrow e'}{v\ e \longrightarrow v\ e'}$$

These are perfectly good operational semantics, but they tell us nothing about why CBV is as expressive as CBN. We can see this more clearly by *translating* CBN into CBV. That is, we create a denotational semantics for CBN that treats CBV as the meaning space.

To translate from the CBN λ-calculus to the CBV λ-calculus, the key issue is how to make it lazy. That is, how do we stop CBV from evaluating its arguments before applying a function? Normally CBV evaluation eagerly evaluates its arguments, so we need to protect those arguments from evaluation. To do this, notice that CBV does have a way to encapsulate a pending computation and save if for later: wrap it in a λ-abstraction. Later, when the value of the argument is needed, applying the abstraction to a dummy argument will extract the body. Such a wrapped computation is referred to as a *thunk*.

For our translation, we will use the notation $\lambda\_.\, e$, which is shorthand for $\lambda x.\, e$ where $x \notin \mathrm{FV}\, e$. We also need a dummy argument to apply our thunks and extract the computation, so we will use $\mathsf{id} = \lambda x.\, x$ for that.

Formally, we define the translation function $[\![\cdot]\!]$ recursively on the structure of expressions as follows.

$$[\![x]\!] \triangleq x\ \mathsf{id} \qquad [\![\lambda x.\, e]\!] \triangleq \lambda x.\, [\![e]\!] \qquad [\![e_1\ e_2]\!] \triangleq [\![e_1]\!]\ (\lambda\_.\, [\![e_2]\!])$$

To see how this works, we will looks at the example of Church booleans. Recall the following definition from Lecture 8.

$$\mathsf{true} \triangleq \lambda xy.\, x \qquad \mathsf{false} \triangleq \lambda xy.\, y \qquad \mathsf{if} \triangleq \lambda btf.\, b\ t\ f$$

There is a slight problem with this construction in CBV λ-calculus: if $b\ e_1\ e_2$ will always evaluate both e_1 and e_2 regardless of the value of b. In CBN, however, this problem goes away. That means that by applying our translation to the above terms, we should get a CBV version of this encoding that behaves the way we expect.

$$\begin{aligned}
[\![\mathsf{true}]\!] &= [\![\lambda xy.\, x]\!] = \lambda xy.\, [\![x]\!] = \lambda xy.\, x\ \mathsf{id} \\
[\![\mathsf{false}]\!] &= [\![\lambda xy.\, y]\!] = \lambda xy.\, [\![y]\!] = \lambda xy.\, y\ \mathsf{id} \\
[\![\mathsf{if}]\!] &= [\![\lambda btf.\, b\ t\ f]\!] = \lambda btf.\, [\![(b\ t)\ f]\!] = \lambda btf.\, [\![b\ t]\!]\ (\lambda_.\, [\![f]\!]) \\
&= \lambda btf.\, [\![b]\!]\ (\lambda_.\, [\![t]\!])\ (\lambda_.\, [\![f]\!]) \\
&= \lambda btf.\, (b\ \mathsf{id})\ (\lambda_.\, t\ \mathsf{id})\ (\lambda_.\, f\ \mathsf{id})
\end{aligned}$$

Now we can see what happens when we first translate $\mathsf{if}\ \mathsf{true}\ e_1\ e_2$ and then evaluate it with CBV rules.

$$\begin{aligned}
[\![\mathsf{if}\ \mathsf{true}\ e_1\ e_2]\!] &= [\![\mathsf{if}]\!]\ (\lambda_.\, [\![\mathsf{true}]\!])\ (\lambda_.\, [\![e_1]\!])\ (\lambda_.\, [\![e_2]\!]) \\
&= \big(\lambda btf.\, (b\ \mathsf{id})\ (\lambda_.\, t\ \mathsf{id})\ (\lambda_.\, f\ \mathsf{id})\big)\ (\lambda_.\, [\![\mathsf{true}]\!])\ (\lambda_.\, [\![e_1]\!])\ (\lambda_.\, [\![e_2]\!]) \\
&\longrightarrow^3 \big((\lambda_.\, [\![\mathsf{true}]\!])\ \mathsf{id}\big)\ \big(\lambda_.\, (\lambda_.\, [\![e_1]\!])\ \mathsf{id}\big)\ \big(\lambda_.\, (\lambda_.\, [\![e_2]\!])\ \mathsf{id}\big) \\
&\longrightarrow [\![\mathsf{true}]\!]\ \big(\lambda_.\, (\lambda_.\, [\![e_1]\!])\ \mathsf{id}\big)\ \big(\lambda_.\, (\lambda_.\, [\![e_2]\!])\ \mathsf{id}\big) \\
&= (\lambda xy.\, x\ \mathsf{id})\ \big(\lambda_.\, (\lambda_.\, [\![e_1]\!])\ \mathsf{id}\big)\ \big(\lambda_.\, (\lambda_.\, [\![e_2]\!])\ \mathsf{id}\big) \\
&\longrightarrow^2 \big(\lambda_.\, (\lambda_.\, [\![e_1]\!])\ \mathsf{id}\big)\ \mathsf{id} \\
&\longrightarrow^2 [\![e_1]\!]
\end{aligned}$$

Notice that e_2 was never evaluated. This is exactly what we were hoping for!

2 Adequacy

We now have two semantics for CBN λ-calculus: the small-step operational semantics, and the translation to CBN λ-calculus. That raises the important question: are they the same? We say a second semantics is *adequate* with respect to the first if it fundamentally defines the same meaning.

To formalize that notion, note that both the CBV and CBN λ-calculus are *deterministic* reduction strategies in the sense at most one reduction is possible for any term (up to α-equivalence). When an expression e in a language is evaluated in a deterministic system, one of three things can happen:

1. There is an infinite sequence of expressions $e_1, e_2, \ldots$ such that $e \longrightarrow e_1 \longrightarrow e_2 \longrightarrow \cdots$. In this case we write $e\Uparrow$ and say e *diverges*.
2. The expression produces a value v in zero or more steps. That is, $e \longrightarrow^* v$. In this case we say e *converges* to v, or write $e \Downarrow v$.
3. The computation converges to a non-value. That is, $e \longrightarrow^* e'$ where e' is not a value but e' cannot step. In this case we say the computation is *stuck*.[1]

A semantic translation is *adequate* if these three behaviors in the source system are adequately represented in the target system, and vice versa. This relation is illustrated with the following commutative diagram.

$$\begin{array}{ccc}
e & \longrightarrow^* & v \\
{\scriptstyle [\![\cdot]\!]}\downarrow & & \downarrow{\scriptstyle [\![\cdot]\!]} \\
[\![e]\!] & \longrightarrow^* & t \approx [\![v]\!]
\end{array}$$

[1] This cannot happen in CBV or CBN λ-calculus for closed terms, but we will see examples soon.

Here $\approx$ is some notion of target term equivalence that is preserved by evaluation, such as β-equivalence, and we use t to represent a target terms to distinguish them from source terms e.

That is, if e converges to some value v in CBN semantics, then $[\![e]\!]$ must converge to some value t in CBV semantics that is equivalent to v, and vice versa. Formally, we state this as the combination between *soundness* and *completeness* of the translation.

Definition 1 (Soundness)**.** The translation from CBN to CBV is *sound* if any behavior any possible behavior of the translated program is possible in the source program. That is,

- If $[\![e]\!] \Downarrow t$, then there is some v such that $e \Downarrow v$ and $t \approx [\![v]\!]$.
- If $[\![e]\!]\Uparrow$ then $e\Uparrow$.

Definition 2 (Completeness)**.** The translation from CBN to CBV is *complete* if any behavior any possible behavior of the source program is possible for the translated program. That is,

- If $e \Downarrow v$, then there is some t such that $[\![e]\!] \Downarrow t$ and $t \approx [\![v]\!]$.
- If $e\Uparrow$ then $[\![e]\!]\Uparrow$.

3 Proving Adequacy

We would like to show that evaluation commutes with our translation $[\![\cdot]\!]$ from CBN to CBV. To do this, we first need a notion of target term equivalence ($\approx$) that is preserved by evaluation. This is challenging, because in the evaluation sequence in the target language, intermediate terms may appear that are not the translation $[\![e]\!]$ of any source term e. For some translations (but not this one), the reverse may also happen. The equivalence must allow for these extra β-redexes that appear during translation.

For our CBN-to-CBV translation, an appropriate notion is to say two terms are equivalent if the differ only by thunks waiting to be applied. That is, $t \approx (\lambda\_.\, t)\ \mathsf{id}$, and $\approx$ is a structural congruence—meaning $t \approx t$ and if $t_1 \approx t_2$ then $\lambda x.\, t_1 \approx \lambda x.\, t_2$, etc. We can think of this equivalence as saying that if $t_1 \approx t_2$, then t_1 and t_2 are the same up to optimizing away applied thunks everywhere (including under λ-abstractions).

Adequacy will follow from a series of lemmas, all of which are proved by induction in some form. Most of the work is contained in Lemma 5. We write $t_1 \xrightarrow[\text{cbv}]{}^{k} t_2$ if t_1 reduces to t_2 in n steps with CBV semantics. If we do not care about the exact number of steps, we use $t_1 \xrightarrow[\text{cbv}]{}^{*} t_2$ for zero or more and $t_1 \xrightarrow[\text{cbv}]{}^{+} t_2$ for one or more.

To show adequacy, we show that each CBN evaluation step starting from e is mirrored exactly by a sequence of CBV evaluation steps starting from $[\![e]\!]$. To keep track of corresponding stages in the two evaluation sequences, we define a *simulation relation* $\lesssim$ between source and target terms that is more general than the translation $[\![\cdot]\!]$ and is preserved during evaluation of both source and target. Intuitively, $e \lesssim t$ means that CBN term e is simulated by the CBV term t.

Formally, we define $\lesssim$ as follows.

$$\frac{}{x \lesssim x\ \mathsf{id}} \qquad \frac{e \lesssim t}{\lambda x.\, e \lesssim \lambda x.\, t} \qquad \frac{e_1 \lesssim t_1 \qquad e_2 \lesssim t_2}{e_1\ e_2 \lesssim t_1\ (\lambda\_.\, t_2)} \qquad \frac{e \lesssim t}{e \lesssim (\lambda\_.\, t)\ \mathsf{id}}$$

The first three rules ensure that a source term corresponds to its translation. The last rule takes care of the extra β-reduction (from applied thunks) that may arise during evaluation—$\approx$-equivalent terms may appear.

To use this relation in our proof of adequacy, we being with a simple but important lemma.

Lemma 1 (Projection Simulates)**.** *For all expressions e,* $e \lesssim [\![e]\!]$.

Proof. The proof follows by structural induction on e.

Case $e = x$**:** $x \lesssim x\ \mathsf{id} = [\![x]\!]$ by definition.

Case $e = \lambda x.\, e'$**:** By induction, $e' \lesssim [\![e']\!]$, so by the second rule, $e = \lambda x.\, e' \lesssim \lambda x.\, [\![e']\!] = [\![e]\!]$.

Case $e = e_1\ e_2$**:** By induction, $e_1 \lesssim [\![e_1]\!]$ and $e_2 \lesssim [\![e_2]\!]$. Therefore, by the third rule,

$$e = e_1\ e_2 \lesssim [\![e_1]\!]\ (\lambda\_.\, [\![e_2]\!]) = [\![e_1\ e_2]\!] = [\![e]\!]\,. \qquad \square$$

Next we show that if e is simulated by t, then its translation is $\approx$-equivalent to t.

Lemma 2 (Related Terms Project to Equivalent Terms)**.** *If* $e \lesssim t$ *then* $[\![e]\!] \approx t$.

Proof. This is a proof by induction on the derivation of $e \lesssim t$.

Case $x \lesssim x\ \mathsf{id}$**:** Here $e = x$ and $t = x\ \mathsf{id}$, so $[\![e]\!] = t$, and we are done.

Case $\lambda x.\, e' \lesssim \lambda x.\, t'$ **where** $e' \lesssim t'$**:** By induction, $[\![e']\!] \approx t'$, so therefore $[\![e]\!] = \lambda x.\, [\![e']\!] \approx \lambda x.\, t' = t$.

Case $e_1\ e_2 \lesssim t_1\ (\lambda\_.\, t_2)$ **where** $e_1 \lesssim t_1$ **and** $e_2 \lesssim t_2$**:** By induction, $[\![e_1]\!] \approx t_1$ and $[\![e_2]\!] \approx t_2$. Therefore,

$$[\![e]\!] = [\![e_1\ e_2]\!] = [\![e_1]\!]\ (\lambda\_.\, [\![e_2]\!]) \approx t_1\ (\lambda\_.\, t_2) = t.$$

Case $e \lesssim (\lambda\_.\, t)\ \mathsf{id}$ **where** $e \lesssim t$**:** By induction, $[\![e]\!] \approx t$. So by the definition of $\approx$, $[\![e]\!] \approx (\lambda\_.\, t)\ \mathsf{id}$. $\square$

The next lemma says that if a value $\lambda x.\, e$ relates to a term t, then t always reduces to some value $\lambda x.\, t'$ while preserving the correspondence.

Lemma 3 (Reduction to Related Values)**.** *If* $\lambda x.\, e \lesssim t$*, then there is some* t' *such that* $t \xrightarrow[\text{cbv}]{}{}^{*} \lambda x.\, t'$ *and* $e \lesssim t'$.

Proof. This is a proof by induction on the derivation of $\lambda x.\, e \lesssim t$. Note that the first and third rules are impossible, as $x \neq \lambda x.\, e$ and $e_1\ e_2 \neq \lambda x.\, e$.

Case $\lambda x.\, e \lesssim \lambda x.\, t'$ **where** $e \lesssim t'$**:** This is immediate in zero steps.

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 $t_0 \xrightarrow[\text{cbv}]{}{}^{*} \lambda x.\, t'$ and $e \lesssim t'$. Therefore $t = (\lambda\_.\, t_0)\ \mathsf{id} \xrightarrow[\text{cbv}]{} t_0 \xrightarrow[\text{cbv}]{}{}^{*} \lambda x.\, t'$. $\square$

The next lemma deals with substitution.

Lemma 4 (Substitution)**.** *If* $e_1 \lesssim t_1$ *and* $e_2 \lesssim t_2$*, then* $e_1[x \mapsto e_2] \lesssim t_1[x \mapsto \lambda\_.\, t_2]$.

Proof. This is a proof by induction on the derivation of $e_1 \lesssim t_1$.

Case $y \lesssim y\ \mathsf{id}$**:** There are two sub-case, depending on whether or not $y = x$.

- $y = x$*:* Here $e_1[x \mapsto e_2] = e_2$ and $t_1[x \mapsto \lambda\_.\, t_2] = (\lambda\_.\, t_2)\ \mathsf{id}$, so the fourth rule applies.
- $y \neq x$*:* Here $e_1[x \mapsto e_2] = y$ and $t_1[x \mapsto \lambda\_.\, t_2] = t_1$, and the case follows from $y \lesssim y\ \mathsf{id}$.

Case $\lambda y.\, e_1' \lesssim \lambda y.\, t_1'$**:** We again have two sub-cases, depending on whether or not $y = x$.

- $y = x$*:* Here $(\lambda x.\, e_1')[x \mapsto e_2] = \lambda x.\, e_1'$ and $(\lambda x.\, t_1')[x \mapsto \lambda\_.\, t_2] = \lambda x.\, t_1'$, so the result is immediate.

- $y \neq x$: Without loss of generality, we can α-convert so $y \notin \mathrm{FV}(e_1) \cup \mathrm{FV}(t_2)$, meaning

$$(\lambda y.\, e_1')[x \mapsto e_2] = \lambda y.\, e_1'[x \mapsto e_2] \quad \text{and} \quad (\lambda y.\, t_1')[x \mapsto \lambda\_.\, t_2] = \lambda y.\, t_1'[x \mapsto \lambda\_.\, t_2].$$

  By induction, $e_1'[x \mapsto e_2] \lesssim t_1'[x \mapsto \lambda\_.\, t_2]$, so the second $\lesssim$ rules completes the case.

Case $e\ e' \lesssim t\ (\lambda\_.\, t')$ **where** $e \lesssim t$ **and** $e' \lesssim t'$**:** By definition, substitution distributes over application, so

$$e_1[x \mapsto e_2] = (e\ e')[x \mapsto e_2] = (e[x \mapsto e_2])\ (e'[x \mapsto e_2])$$
and
$$t_1[x \mapsto \lambda\_.\, t_2] = (t\ t')[x \mapsto \lambda\_.\, t_2] = (t[x \mapsto \lambda\_.\, t_2])\ (t'[x \mapsto \lambda\_.\, t_2]).$$

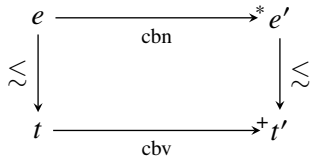
  By the induction hypotheses, we have $e[x \mapsto e_2] \lesssim t[x \mapsto \lambda\_.\, t_2]$ and $e'[x \mapsto e_2] \lesssim t'[x \mapsto \lambda\_.\, t_2]$. The third (application) rule of the definition of $\lesssim$ completes the case.

Case $e_1 \lesssim (\lambda\_.\, t_1')\ \mathsf{id}$ **where** $e_1 \lesssim t_1'$**:** By induction, $e_1[x \mapsto e_2] \lesssim t_1'[x \mapsto \lambda\_.\, t_2]$. Therefore, using the fourth inference rule,

$$e_1[x \mapsto e_2] \lesssim (\lambda\_.\, t_1'[x \mapsto \lambda\_.\, t_2])\ \mathsf{id} = (\lambda\_.\, t_1')\ \mathsf{id}[x \mapsto \lambda\_.\, t_2] = t_1[x \mapsto \lambda\_.\, t_2]. \qquad \square$$

We now have enough machinery to show that the $\lesssim$ relation is preserved by stepping.

Lemma 5 (Simulation Preservation)**.** *If $e \lesssim t$ and $e \xrightarrow[\text{cbn}]{} e'$, then there is a t' such that $t \xrightarrow[\text{cbv}]{}^{+} t'$ and $e' \lesssim t'$.*



Proof. We again proceed by induction on the derivation of $e \lesssim t$.

Case $x \lesssim x\ \mathsf{id}$**:** In this case e cannot step, so this case is impossible.

Case $\lambda x.\, e' \lesssim \lambda x.\, t'$ **where** $e' \lesssim t'$**:** Again e cannot step, so this case is impossible.

Case $e_1\ e_2 \lesssim t_1\ (\lambda\_.\, t_2)$ **where** $e_1 \lesssim t_1$ **and** $e_2 \lesssim t_2$**:** There are two subcases depending on the form of the derivation of $e \xrightarrow[\text{cbn}]{} e'$.

- *Where $e_1\ e_2 \xrightarrow[\text{cbn}]{} e_1'\ e_2$ with $e_1 \xrightarrow[\text{cbn}]{} e_1'$:* By the induction hypothesis, there is some t_1' such that $t_1 \xrightarrow[\text{cbv}]{}^{+} t_1'$ and $e_1' \lesssim t_1'$. Therefore $t_1\ (\lambda\_.\, t_2) \xrightarrow[\text{cbv}]{}^{+} t_1'\ (\lambda\_.\, t_2)$, and by the third rule defining our simulation relation, $e_1'\ e_2 \lesssim t_1'\ (\lambda\_.\, t_2)$.
- *Where $e_1 = \lambda x.\, e_1'$ and $(\lambda x.\, e_1')\ e_2 \xrightarrow[\text{cbn}]{} e_1'[x \mapsto e_2]$:* By assumption, $\lambda x.\, e_1' \lesssim t_1$, so by Lemma 3, $t_1 \xrightarrow[\text{cbv}]{}^{*} \lambda x.\, t_1'$ where $e_1' \lesssim t_1'$. We therefore have that

$$t = t_1\ (\lambda\_.\, t_2) \xrightarrow[\text{cbv}]{}^{*} (\lambda x.\, t_1')\ (\lambda\_.\, t_2) \xrightarrow[\text{cbv}]{} t_1'[x \mapsto \lambda\_.\, t_2].$$

  Lemma 4 now shows that $e_1'[x \mapsto e_2] \lesssim t_1'[x \mapsto \lambda\_.\, t_2]$, completing the case.

Case $e \lesssim (\lambda\_.\, t_0)\ \mathsf{id}$ **where** $e \lesssim t_0$**:** By induction, there exists some t' such that $t_0 \xrightarrow[\text{cbv}]{}^{+} t'$ such that $e' \lesssim t'$. Therefore $t = (\lambda\_.\, t_0)\ \mathsf{id} \xrightarrow[\text{cbv}]{} t_0 \xrightarrow[\text{cbv}]{}^{+} t'$, completing the case. $\square$

We are now ready to complete our proof of adequacy.

Theorem 1 (Adequacy of $[\![\cdot]\!]$)**.** *The CBN-to-CBV translation* $[\![\cdot]\!]$ *is sound and complete.*

Proof. We begin with completeness. Given a source term e and its translation $[\![e]\!]$, Lemma 1 gives us that $e \lesssim [\![e]\!]$. Lemma 5 then shows that each step of the CBN evaluation of e is mirrored precisely by a CBV execution that preserves $e \lesssim t$. Thus, if the evaluation of e diverges, so too will the evaluation of $[\![e]\!]$. Similarly, if the evaluation of e converges to a value v, then by Lemma 3, the evaluation of $[\![e]\!]$ will converge to a value t such that $v \lesssim t$. Lemma 2 shows $[\![v]\!] \approx t$, completing the proof of completeness.

For soundness, 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 $[\![e]\!] \xrightarrow[\text{cbv}]{}{}^{*} t$ for a value t. There are three possibilities for the evaluation of e: (*i*) the evaluation could get stuck. This cannot happen for the source language because all terms are either values or have a valid evaluation step. (*ii*) the evaluation of e could terminate with a value v. In this case $v \lesssim t$ by Lemma 5 because the target language evaluation is deterministic. (*iii*) the evaluation of e could diverge. In this case, Lemma 5 proves there must be some divergent target-language evaluation. The determinism of the target language proves this cannot happen. □