1 Rule Induction

Let us use our newfound wisdom on well-founded induction to prove some properties of the semantics we have seen so far.

1.1 Example 1: evaluation preserves closedness

Theorem If $e \to e'$ under the CBV reduction rules, then $FV(e') \subseteq FV(e)$. In other words, CBV reductions cannot introduce any new free variables.

Proof. By induction on the CBV derivation of $e \to e'$. There is one case for each CBV rule, corresponding to each way $e \to e'$ could be derived.

Case 1: $\dfrac{e_1 \to e_1'}{e_1\ e_2 \to e_1'\ e_2}$.

We assume that the desired property is true of the premise—this is the induction hypothesis—and we wish to prove under this assumption that it is true for the conclusion. Thus we are assuming that $FV(e_1') \subseteq FV(e_1)$ and wish to prove that $FV(e_1'\ e_2) \subseteq FV(e_1\ e_2)$.

$$\begin{array}{rcll} FV(e_1'\ e_2) & = & FV(e_1') \cup FV(e_2) & \text{by the definition of } FV \\ & \subseteq & FV(e_1) \cup FV(e_2) & \text{by the induction hypothesis} \\ & = & FV(e_1\ e_2) & \text{again by the definition of } FV. \end{array}$$

Case 2: $\dfrac{e_2 \to e_2'}{v\ e_2 \to v\ e_2'}$.

This case is similar to Case 1, where now $e_2 \to e_2'$ is used in the induction hypothesis.

Case 3: $\dfrac{}{(\lambda x.\, e)v \to e\{v/x\}}$.

There is no induction hypothesis for this case, since there is no premise in the rule; thus this case constitutes the basis of our induction. We wish to show, independently of any inductive assumption, that $FV(e\{v/x\}) \subseteq FV((\lambda x.\, e)v)$.

This case requires a lemma, stated below, to show that $FV(e\{v/x\}) \subseteq (FV(e) - \{x\}) \cup FV(v)$. Once that is shown, we have

$$\begin{array}{rcll} FV(e\{v/x\}) & \subseteq & (FV(e) - \{x\}) \cup FV(v) & \text{by the lemma to be proved} \\ & = & FV(\lambda x.\, e) \cup FV(v) & \text{by the definition of } FV \\ & = & FV((\lambda x.\, e)v) & \text{again by the definition of } FV. \end{array}$$

We have now considered all three rules of derivation for the CBV λ-calculus, so the theorem is proved.

□

Lemma $FV(e\{v/x\}) \subseteq (FV(e) - \{x\}) \cup FV(v)$ (this lemma is used by case 3 in the above theorem).

Proof. By structural induction on e. There is one case for each clause in the definition of the substitution operator. We have assumed previously that values are closed terms, so $FV(v) = \emptyset$ for any value v; but actually we do not need this for the proof, and we do not assume it.

Case 1: $e = x$.

$$\begin{aligned} FV(e\{v/x\}) &= FV(x\{v/x\}) \\ &= FV(v) \quad \text{by the definition of the substitution operator} \\ &= (\{x\} - \{x\}) \cup FV(v) \\ &= (FV(x) - \{x\}) \cup FV(v) \quad \text{by the definition of } FV \\ &= (FV(e) - \{x\}) \cup FV(v). \end{aligned}$$

Case 2: $e = y$, $y \neq x$.

$$\begin{aligned} FV(e\{v/x\}) &= FV(y\{v/x\}) \\ &= FV(y) \quad \text{by the definition of the substitution operator} \\ &= \{y\} \quad \text{by the definition of } FV \\ &\subseteq (\{y\} - \{x\}) \cup FV(v) \\ &= (FV(y) - \{x\}) \cup FV(v) \quad \text{again by the definition of } FV \\ &= (FV(e) - \{x\}) \cup FV(v). \end{aligned}$$

Case 3: $e = e_1\ e_2$.

$$\begin{aligned} FV(e\{v/x\}) &= FV((e_1\ e_2)\{v/x\}) \\ &= FV(e_1\{v/x\}\ e_2\{v/x\}) \quad \text{by the definition of the substitution operator} \\ &\subseteq (FV(e_1) - \{x\}) \cup FV(v) \cup (FV(e_2) - \{x\}) \cup FV(v) \quad \text{by the induction hypothesis} \\ &= ((FV(e_1) \cup FV(e_2)) - \{x\}) \cup FV(v) \\ &= (FV(e_1\ e_2) - \{x\}) \cup FV(v) \quad \text{again by the definition of } FV \\ &= (FV(e) - \{x\}) \cup FV(v). \end{aligned}$$

Case 4: $e = \lambda x.\, e'$.

$$\begin{aligned} FV(e\{v/x\}) &= FV((\lambda x.\, e')\{v/x\}) \\ &= FV(\lambda x.\, e') \quad \text{by the definition of the substitution operator} \\ &= FV(\lambda x.\, e') - \{x\} \quad \text{because } x \notin FV(\lambda x.\, e') \\ &\subseteq (FV(e) - \{x\}) \cup FV(v). \end{aligned}$$

Case 5: $e = \lambda y.\, e'$, $y \neq x$. This is the most interesting case, because it involves a change of bound variable. Using the fact $FV(v) = \emptyset$ for values v would give a slightly simpler proof. Let v be a value and z a variable such that $z \neq x$, $z \notin FV(e')$, and $z \notin FV(v)$.

$$\begin{aligned} FV(e\{v/x\}) &= FV((\lambda y.\, e')\{v/x\}) \\ &= FV(\lambda z.\, e'\{z/y\}\{v/x\}) \quad \text{by the definition of the substitution operator} \\ &= FV(e'\{z/y\}\{v/x\}) - \{z\} \quad \text{by the definition of } FV \\ &= ((((FV(e') - \{y\}) \cup FV(z)) - \{x\}) \cup FV(v)) - \{z\} \quad \text{by the induction hypothesis twice} \\ &= (((FV(\lambda y.\, e') \cup \{z\}) - \{x\}) \cup FV(v)) - \{z\} \quad \text{by the definition of } FV \\ &= ((FV(\lambda y.\, e') - \{x\}) \cup FV(v) \cup \{z\}) - \{z\} \\ &= (FV(e) - \{x\}) \cup FV(v). \end{aligned}$$

□

There is a subtle point that arises in case 5. We said at the beginning of the proof that we would be doing structural induction on e; that is, induction on the well-founded subterm relation $<$. This was a lie. Because of the change of bound variable necessary in case 5, we are actually doing induction on the relation of subterm modulo α-equivalence:

$$e <_\alpha e' \quad \triangleq \quad \exists e''\ e'' < e' \wedge e =_\alpha e''.$$

But a moment's thought reveals that this relation is still well-founded, since α-reduction does not change the size or shape of the term, so we are ok.

1.2 Example 2: agreement of big-step and small-step semantics

As we saw earlier, we can express the idea that the two semantics should agree on terminating executions by connecting the $\longrightarrow^*$ and $\Downarrow$ relations:

$$\langle c, \sigma\rangle \longrightarrow^* \langle \textbf{skip}, \sigma'\rangle \iff \langle c, \sigma\rangle \Downarrow \sigma'$$

This can be proved using induction. To prove the $\Rightarrow$ direction, we can use structural induction on c. The $\Leftarrow$ direction requires induction on the derivation of the big-step evaluation. We are given $\langle c, \sigma\rangle \Downarrow \sigma'$, so we know that there is a derivation. The form of the derivation depends on the form of c. Here we show just a few of the cases for c.

- Case **skip**. In this case we know $\sigma = \sigma'$, and trivially, $\langle \textbf{skip}, \sigma\rangle \longrightarrow^* \langle \textbf{skip}, \sigma\rangle$.
- Case $x := a$. In this case we know from the premises that $\langle a, \sigma\rangle \Downarrow n$ for some n, and that $\sigma' = \sigma[x \mapsto n]$.

  We will need a lemma that $\langle a, \sigma\rangle \Downarrow n \Rightarrow \langle a, \sigma\rangle \longrightarrow^* n$. This can be proved using the same technique being used on commands. We will also need a lemma showing that $\langle a, \sigma\rangle \longrightarrow^* n$ implies $\langle x := a, \sigma\rangle \longrightarrow^* \langle x := n, \sigma\rangle$. This obvious result can be proved easily using an induction on the number of steps taken.

  Given these lemmas, we have $\langle x := a, \sigma\rangle \longrightarrow^* \langle x := n, \sigma\rangle$ and $\langle x := n, \sigma\rangle \longrightarrow \langle \textbf{skip}, \sigma[x \mapsto n]\rangle$, so $\langle x := a, \sigma\rangle \longrightarrow^* \langle \textbf{skip}, \sigma[x \mapsto n]\rangle$.
- Case **while** b **do** c, where b evaluates to false. In this case we have $\langle b, \sigma\rangle \Downarrow \mathit{false}$ and $\sigma = \sigma'$. Consider what happens in the small-steps semantics. Given two more lemmas, that $\langle b, \sigma\rangle \Downarrow t \Rightarrow \langle b, \sigma\rangle \longrightarrow^* t$, and that $\langle b, \sigma\rangle \longrightarrow^* t \Rightarrow \langle \textbf{while } b \textbf{ do } c, \sigma\rangle \longrightarrow^* \langle \textbf{while } t \textbf{ do } c, \sigma\rangle$, we have $\langle \textbf{while } b \textbf{ do } c, \sigma\rangle \longrightarrow^* \langle \textbf{skip}, \sigma\rangle$, as desired.
- Case **while** b **do** c, where b evaluates to true. This is the most interesting case in the whole proof. We need one more obvious lemma for stitching together small-step executions:

$$(\langle c_1, \sigma\rangle \longrightarrow^* \langle \textbf{skip}, \sigma'\rangle \wedge \langle c_2, \sigma'\rangle \longrightarrow^* \langle \textbf{skip}, \sigma''\rangle) \implies \langle c_1; c_2, \sigma\rangle \longrightarrow^* \langle \textbf{skip}, \sigma''\rangle \tag{1}$$

  This can be proved by induction on the number of steps.

  Now, because $\langle \textbf{while } b \textbf{ do } c, \sigma\rangle \Downarrow \langle \textbf{skip}, \sigma'\rangle$, we know that $\langle c, \sigma\rangle \Downarrow \sigma''$ and $\langle \textbf{while } b \textbf{ do } c, \sigma''\rangle \Downarrow \sigma'$. Further, because the derivations of these two assertions are subderivations of that for $\langle \textbf{while } b \textbf{ do } c, \sigma\rangle$, the induction hypothesis gives us that $\langle c, \sigma\rangle \longrightarrow^* \langle \textbf{skip}, \sigma''\rangle$ and that $\langle \textbf{while } b \textbf{ do } c, \sigma''\rangle \longrightarrow^* \sigma'$. Using Lemma 1, we have $\langle c; \textbf{while } b \textbf{ do } c, \sigma\rangle \longrightarrow^* \langle \textbf{skip}, \sigma'\rangle$.

  Now consider the small-step side. We have an initial step

$$\langle \textbf{while } b \textbf{ do } c, \sigma\rangle \longrightarrow \langle \textbf{if } b \textbf{ then } (c; \textbf{while } b \textbf{ do } c) \textbf{ else skip}, \sigma\rangle$$

  From prior lemmas, we know this will step to $\langle c; \textbf{while } b \textbf{ do } c, \sigma\rangle$, which we just showed will step to $\langle \textbf{skip}, \sigma'\rangle$ as desired.

  Notice that we could not have used structural induction for this proof, because the induction step involved relating an evaluation of the command **while** b **do** c to a different evaluation of the same command rather than to an evaluation of a subexpression.

2 Remark

The value of the reasoning framework we have set up is that formal reasoning about the semantics of programming languages, including such seemingly complicated notions as reductions and substitutions, can be reduced to the mindless application of a few simple rules. There is no hand-waving or magic involved. There is nothing hidden, it is all right there in front of you. To the extent that we can do this for real programming languages, we will be better able to understand what is going on.

3 Evaluation Contexts

[1] The rules for structural operational semantics can be classified into two types:

- *reduction rules*, which describe the actual computation steps; and
- *structural congruence rules*, which constrain the choice of reductions that can be performed next, thus defining both the order of evaluation and whether subexpressions are evaluated lazily.

For example, the CBV reduction strategy for the λ-calculus was captured in the following rules:

$$\frac{}{(\lambda x.\, e)\ v \longrightarrow e\{v/x\}} \tag{2}$$

$$\frac{e_1 \longrightarrow e_1'}{e_1\ e_2 \longrightarrow e_1'\ e_2} \qquad \frac{e_2 \longrightarrow e_2'}{v\ e_2 \longrightarrow v\ e_2'} \tag{3}$$

Rule (2), β-reduction, is a reduction rule, whereas rules (3) are structural congruence rules. The rules (3) say essentially that a reduction may be applied to a redex on the left-hand side of an application anytime, and may be applied to a redex on the right-hand side of an application provided the left-hand side is already fully reduced.

Although there are only two structural congruence rules in the CBV λ-calculus, there are typically many more in real-world programming languages. It would be nice to have a more compact way to express them.

Evaluation contexts provide a mechanism to do just that. An evaluation context E, sometimes written $E[\,\bullet\,]$, is a λ-term or a metaexpression representing a family of λ-terms with a special variable $[\,\cdot\,]$ called the *hole*. If $E[\,\bullet\,]$ is an evaluation context, then $E[\,e\,]$ represents E with the term e substituted for the hole.

Every evaluation context $E[\,\bullet\,]$ represents a *context rule*

$$\frac{e \longrightarrow e'}{E[\,e\,] \longrightarrow E[\,e'\,]},$$

which says that we may apply the reduction $e \longrightarrow e'$ in the context $E[\,e\,]$.

For the case of the CBV λ-calculus, the two structural congruence rules (3) are specified by the two evaluation context schemes $([\,\cdot\,]\ e)$ and $(v\ [\,\cdot\,])$. These are just a compact way of representing the rules (3). Thus we could specify the CBV λ-calculus simply:

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

The CBN λ-calculus has an equally compact specification:

$$(\lambda x.\, e)\ e' \longrightarrow e\{e'/x\} \qquad [\,\cdot\,]\ e$$

4 Nested Contexts

Note that in CBV, the evaluation contexts $[\,\cdot\,]\ e$ and $v\ [\,\cdot\,]$ do not specify *all* contexts in which the reduction rule (2) may be applied. There are also compound contexts obtained from nested applications of the rules (3). For example, the context

$$(v\ [\,\cdot\,])\ e \tag{4}$$

[1] Sections marked with an asterisk may not have been covered in lecture, and are optional in this case.

is also a valid evaluation context for CBV, since it can be derived from two applications of the rules (3):

$$\frac{\dfrac{e_1 \longrightarrow e_2}{v\ e_1 \longrightarrow v\ e_2}}{(v\ e_1)\ e \longrightarrow (v\ e_2)\ e}. \tag{5}$$

Here we have applied the right-hand rule of (3) in the first step and the left-hand rule of (3) in the second. The evaluation context (4) represents the abbreviated rule

$$\frac{e_1 \longrightarrow e_2}{(v\ e_1)\ e \longrightarrow (v\ e_2)\ e}$$

obtained by collapsing the two steps of (5).

The set of *all* valid evaluation contexts for the CBV λ-calculus is represented by the grammar

$$E \quad ::= \quad [\,\cdot\,] \ \mid\ E\ e \ \mid\ v\ E.$$

5 Annotated Proof Trees

We can also use evaluation contexts to indicate exactly where a reduction is applied in each step of a proof tree. For example, consider the annotated proof tree

$$\frac{\dfrac{(\lambda x.\, x)\ 0 \longrightarrow 0}{(\lambda x.\, x)\ ((\lambda x.\, x)\ 0) \longrightarrow (\lambda x.\, x)\ 0}\ ((\lambda x.\, x)\ [\,\cdot\,])}{(\lambda x.\, x)\ ((\lambda x.\, x)\ 0)\ \lambda z.\, z\ z \longrightarrow (\lambda x.\, x)\ 0\ \lambda z.\, z\ z}\ ([\,\cdot\,]\ \lambda z.\, z\ z)$$

We have labeled each step to indicate the context in which the β-reduction was applied.

As above, we can simplify the tree by collapsing the two steps and annotating the resulting abbreviated tree with the corresponding nested context:

$$\frac{(\lambda x.\, x)\ 0 \longrightarrow 0}{(\lambda x.\, x)\ ((\lambda x.\, x)\ 0)\ \lambda z.\, z\ z \longrightarrow (\lambda x.\, x)\ 0\ \lambda z.\, z\ z}\ ((\lambda x.\, x)\ [\,\cdot\,]\ \lambda z.\, z\ z)$$