1 Evaluation Contexts

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

- *reduction rules*, which describe the actual computation steps; and
- *evaluation order rules*, which constrain the choice of reductions that can be performed next.

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

$$\frac{}{(\lambda x.\, e)\, v \to e\{v/x\}} \qquad \frac{e_1 \to e_1'}{e_1\, e_2 \to e_1'\, e_2} \qquad \frac{e_2 \to e_2'}{v\, e_2 \to v\, e_2'} \tag{1}$$

The leftmost rule is a reduction rule (β-reduction), whereas the other two rules are evaluation order rules. The evaluation order rules 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 evaluation order rules in the CBV λ-calculus, there are typically many more in real-world programming languages. This motivates the desire to find a more compact representation for such rules.

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

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

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

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

For the case of the CBV λ-calculus, the rightmost two rules of (1) can be represented more compactly by the two evaluation context schemes $[\cdot]\, e$ and $v\, [\cdot]$. Thus we could specify the CBV λ-calculus simply by writing

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

The CBN λ-calculus has an equally compact specification:

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

2 Nested Contexts

Note that in CBV, the evaluation contexts $[\cdot]\, e$ and $v\, [\cdot]$ do not specify *all* contexts in which β-reduction may be applied. There are also compound contexts obtained from nested applications of the rules (1). For example, the context

$$(v\, [\cdot])\, e \tag{2}$$

is also a valid evaluation context for CBV, since it can be derived from two applications of the rules (1):

$$\frac{\dfrac{e_1 \to e_2}{v\,e_1 \to v\,e_2}}{(v\,e_1)\,e \to (v\,e_2)\,e} \tag{3}$$

Here we have applied the rightmost rule of (1) in the first step and the middle rule of (1) in the second. The evaluation context (2) represents the abbreviated rule

$$\frac{e_1 \to e_2}{(v\,e_1)\,e \to (v\,e_2)\,e}$$

obtained by collapsing the two steps of (3).

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

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

3 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

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

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:

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

4 Error Propagation

Evaluation contexts can be used to define the semantics of error exceptions. If we have a special error value error, we can very easily propagate it using the evaluation order rule

$$E[\mathsf{error}] \to \mathsf{error}.$$

This obviates the need to show in painstaking detail how error propagates up through a series of applications of rewrite rules. We will revisit this idea later on when we talk about exception handling mechanisms.

The benefits of evaluation contexts will become exceedingly clear in the future as we add more features to the language.

5 Semantics via Translation

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.

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 other decorations to distinguish between different semantic functions, as for example $[\![e]\!]_{\mathrm{cbn}}$ or $\mathcal{C}[\![e]\!]$.

6 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 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 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:

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

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,

$$\begin{aligned}
[\![\mathsf{if}\, \mathsf{true}\, e_1\, e_2]\!] &= [\![\mathsf{if}]\!]\, (\lambda d.\, [\![\mathsf{true}]\!])\, (\lambda d.\, [\![e_1]\!])\, (\lambda d.\, [\![e_2]\!]) \\
&= (\lambda xyz.\, (x\ \mathsf{id})\, (\lambda d.\, y\ \mathsf{id})\, (\lambda d.\, z\ \mathsf{id}))\, (\lambda d.\, [\![\mathsf{true}]\!])\, (\lambda d.\, [\![e_1]\!])\, (\lambda d.\, [\![e_2]\!]) \\
&\xrightarrow{3} ((\lambda d.\, [\![\mathsf{true}]\!])\ \mathsf{id})\, (\lambda d.\, (\lambda d.\, [\![e_1]\!])\ \mathsf{id})\, (\lambda d.\, (\lambda d.\, [\![e_2]\!])\ \mathsf{id}) \\
&\xrightarrow{1} [\![\mathsf{true}]\!]\, (\lambda d.\, (\lambda d.\, [\![e_1]\!])\ \mathsf{id})\, (\lambda d.\, (\lambda d.\, [\![e_2]\!])\ \mathsf{id}) \\
&= (\lambda xy.\, x\ \mathsf{id})\, (\lambda d.\, (\lambda d.\, [\![e_1]\!])\ \mathsf{id})\, (\lambda d.\, (\lambda d.\, [\![e_2]\!])\ \mathsf{id}) \\
&\xrightarrow{2} ((\lambda d.\, (\lambda d.\, [\![e_1]\!])\ \mathsf{id})\ \mathsf{id}) \\
&\xrightarrow{2} [\![e_1]\!],
\end{aligned}$$

and e_2 was never evaluated.