CS611 Lecture 8 Evaluation Contexts 14 September, 2005
Scribe: Georgios Piliouras, Raghuram Ramanujan Lecturer: Andrew Myers

In this lecture, we

- revisit the notion of defining functions inductively
- explore the use of *Evaluation Contexts* for creating more compact structural operational semantics
- and observe how non-determinism in a language creates problems for big step semantics

1 Inductive Definitions

Consider the following definition of the Fibonacci sequence:

$$f(x) = \begin{cases} 1 & \text{if } x \leq 2 \\ f(x-1) + f(x-2) & \text{if } x > 2 \\ & \text{where } x \in \mathbf{N} \end{cases}$$

This is a well-founded definition since *f(x)* is defined in terms of *f* applied to successively smaller arguments *i.e.* there is no infinitely descending recursive chain. The function can thus be captured with the following pair of rewrite rules:

$$\frac{}{x \mapsto 1}\ x \leq 2 \qquad \frac{x-1 \mapsto a \quad x-2 \mapsto b}{x \mapsto a+b}\ x > 2$$

In general, in a valid inductive definition, the argument that appears in the rule must be related through some well-founded relation to what appears in the left-hand side of the premise(s). A function with such a definition has a finite derivation tree. Now consider the function:

$$f(x) = f(x) + 1$$

This yields the rewrite rule:

$$\frac{x \mapsto a}{x \mapsto a+1}$$

This is an example of an invalid inductive definition. In fact, this definition derives the empty function.

2 Evaluation Contexts

Recall the context-free grammar that was used to define expressions in the lambda calculus:

$$e \quad ::= \quad x \ \mid \ e_0\, e_1 \ \mid \ \lambda x.e$$

This context-free grammar induces a language. The language corresponding to a context-free grammar is a set comprising all the finitely-derivable strings (or abstract syntax trees). In this example, the following rewrite rules derive all the members of the language:

$$\frac{}{x} \qquad \frac{e}{\lambda x.e} \qquad \frac{e_0 \quad e_1}{e_0\, e_1}$$

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

- reduction rules - these are often more interesting as they offer an insight into the workings of the language
- evaluation order rules - these are typically as not interesting, though they are usually more numerous

Earlier in the semester, we captured the semantics of the call-by-value lambda calculus via the following rewrite rules:

$$\frac{}{(\lambda x.\, e) \;\longrightarrow e\{v/x\}} \qquad \frac{e_1 \longrightarrow e_1'}{e_1\ e_2 \longrightarrow e_1'\ e_2} \qquad \frac{e_2 \longrightarrow e_2'}{v\ e_2 \longrightarrow v\ e_2'}$$

The first rule here (*β reduction*) falls under the category of reduction rules, whereas the latter two are evaluation order rules. Though there are relatively few evaluation order rules in the call-by-value lambda calculus, they are much more numerous in real-world programming languages and thus tedious to write. This motivates the need to find a more compact representation for such rules. *Evaluation Contexts* provide a mechanism to do just that. An evaluation context E (alternately $\mathsf{E}[\bullet]$) is an expression (term) with a 'hole' where a reduction is allowed. For example:

$$(((\lambda x.x)[\bullet])(\lambda z.z\ z)) \text{ where } [\bullet] \text{ represents the hole}$$

If E is a context, then $\mathsf{E}[e]$ represents E with the hole plugged by e. This definition of a context yields a generic *context rule* that can be applied to any defined context E:

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

For the case of the call-by-value lambda calculus that we have been looking at so far, the two evaluation order rules give rise to the following two contexts:

$$\begin{array}{ccc} \dfrac{e_1 \longrightarrow e_1'}{e_1\ e_2 \longrightarrow e_1'\ e_2} & \equiv & [\bullet]\ e \\ \dfrac{e_2 \longrightarrow e_2'}{v\ e_2 \longrightarrow v\ e_2'} & \equiv & v\ [\bullet] \end{array}$$

These define the legal evaluation contexts *i.e.* the ones in which the context rule defined earlier can be applied. Hence, we derive a compact way for representing rules. Namely, for the case of call-by-value lambda calculus we get:

$$\mathsf{E} \quad ::= \quad [\bullet]\ \mathsf{e} \ \mid\ \mathsf{v}\ [\bullet]$$

The benefits of evaluation context will become exceedingly obvious in the future, as we add more features to the language than just applications. For the time being however, we will present an example of its usage for producing the proof tree of a given expression:

$$\dfrac{\dfrac{(\lambda \mathsf{x.x})0 \longrightarrow 0}{(\lambda \mathsf{x.x})((\lambda x.x)0) \longrightarrow (\lambda \mathsf{x.x})0}\ \textit{where}\ \mathsf{E} = (\lambda \mathsf{x.x})[\bullet]}{((\lambda x.x)((\lambda x.x)0)(\lambda \mathsf{z.zz})) \longrightarrow (((\lambda x.x)0)(\lambda \mathsf{z.zz}))}\ \textit{where}\ \mathsf{E} = [\bullet](\lambda \mathsf{z.zz})$$

where in the above tree we indicate the term being reduced in italic typeface.

For the case of call-by-name lambda calculus, given the context rule, we achieve an equally compact representation:

$$\mathsf{E} ::= [\bullet]\ \mathsf{e}$$

$$\frac{}{(\lambda x.e)\ e' \longrightarrow e\{e'/x\}}$$

The contexts defined above look like one of these two abstract syntax trees:

$$(\bullet\ @\ e) \text{ and } (v\ @\ \bullet)$$

where the @ symbol denotes application.

Consider the following tree:

$$((v\ @\ \bullet)\ @\ e)$$

To define a semantic small-step for this piece of syntax, we have to apply the $E[e] \longrightarrow E[e']$ rule twice (once for each @ starting from the topmost @), because by our current definition, E can only have one application (@). The example above illustrates this, with $v = (\lambda x.x)$, $e = (\lambda z.zz)$.

So our definition of contexts does not cover such "deep" contexts in one step. You have to apply the reduction rule multiple times, thus requiring a multi-level proof tree.

But by changing our definition of evaluation contexts to include these "deep" contexts, we can make our proof trees shorter. This fits the bill:

$$\mathsf{E} \quad ::= \quad [\bullet] \ \mid\ \mathsf{E}\ \mathsf{e} \ \mid\ \mathsf{v}\ \mathsf{E}$$

This yields a a modified context rule:

$$\frac{e \longrightarrow_r e'}{\mathsf{E}[e] \longrightarrow \mathsf{E}[e']}$$

Taken together, the new representation flattens out the multiple "boring" steps corresponding to the repeated application of the evaluation order rules to a single level.

This means that the lower arrow in the context rule actually does more stepping than the upper arrow, and so they are not the same arrow. Hence the r suffix on the upper arrow. Technically (for the same reason) $e \longrightarrow_r e'$ should actually be a side condition.

Note that even though the lower arrow does more stepping, it only does the stepping that is defined in the grammar for evaluation contexts: that is, the stepping of evaluation-order rules only, not the reduction rules. So the single level still contains only one reduction.

Revisiting the example above, we get a much shorter proof tree, where we use a single evaluation order rule:

$$\frac{\dfrac{}{(\lambda \mathsf{x}.\mathsf{x})0 \longrightarrow 0}}{((\lambda x.x)((\lambda x.x)0)(\lambda \mathsf{z}.\mathsf{zz})) \longrightarrow (((\lambda x.x)0)(\lambda \mathsf{z}.\mathsf{zz}))} \quad \textit{where } \mathsf{E} = ((\lambda x.x))(\lambda z.z)$$

Lastly, evaluation contexts can be used to easily define the semantics of error exeptions, since we can very easily propagate an error value using the evaluation order rule below:

$$\mathsf{E}[error] \longrightarrow error$$

This alleviates the need to show in painstaking detail how error propagates up through each rewrite rule.

3 Non-determinism

A non-deterministic computation can be informally defined as one, where starting from the same input we might have more than one possible transition, each one leading possibly to different results.

$$\begin{matrix} e \longrightarrow e' \\ e \longrightarrow e'' \end{matrix} \quad \text{where } e' \neq e''$$

Of course, this raises the issue of how can one express non-determinism. As we shall see, a rather natural choice is by the use of small step semantics. Indeed, we can easily define the non-deterministic choice operator $\Box$, where by $c_1 \Box c_2$ we express that the system can do either:

$$\frac{}{\langle c_1 \Box c_2, \sigma \rangle \longrightarrow \langle c_1, \sigma \rangle}$$

$$\frac{}{\langle c_1 \Box c_2, \sigma \rangle \longrightarrow \langle c_2, \sigma \rangle}$$

However, if we try to express non-determinism using big step semantics, then we immediately face severe problems. For example, a natural approach for tackling the issue of non-determinism in big step semantics would be the following:

$$\frac{\langle c_1, \sigma \rangle \Downarrow \sigma'}{\langle c_1 \Box c_2, \sigma \rangle \Downarrow \sigma'} \quad \frac{\langle c_2, \sigma \rangle \Downarrow \sigma'}{\langle c_1 \Box c_2, \sigma \rangle \Downarrow \sigma'}$$

Although at a first glance this approach seems to work, let's examine what happens if exactly one of c_1, c_2 diverges. Indeed, in this case, according to our semantics we will always choose the *good* one, the one that terminates. For obvious reasons this kind of non-determinism is referred to as *angelic nondeterminism.*

A closely related issue to the definition of non-determinism using semantics is capturing the notion of parallel processing. Again, in this case small step semantics seem to provide a rather natural solution to the problem:

$$\frac{\langle c_1, \sigma \rangle \longrightarrow \langle c_1', \sigma' \rangle}{\langle c_1 || c_2, \sigma \rangle \longrightarrow \langle c_1' || c_2, \sigma' \rangle}$$

$$\frac{\langle c_2, \sigma \rangle \longrightarrow \langle c_2', \sigma' \rangle}{\langle c_1 || c_2, \sigma \rangle \longrightarrow \langle c_1 || c_2', \sigma' \rangle}$$

Big step semantics on the other hand does not capture interleaving. For example, we could try to express the semantics for the concurrent statements $c_1 || c_2$ as follows:

$$\frac{\langle c_1, \sigma \rangle \Downarrow \sigma'' \ \ \langle c_2, \sigma'' \rangle \Downarrow \sigma'}{\langle c_1 || c_2, \sigma \rangle \Downarrow \sigma'}$$

However, we have only ended up defining $c_1; c_2$, that is the serial execution of commands c_1 and c_2. Thus, in general, big step semantics are often not faithful when it comes to representing non-determinism.