Reduction (or Contextual) Semantics

CPSC 509: Programming Language Principles

Ronald Garcia*

4 March 2014

We learned about structural operational small-step semantics, which let us reason about the process of computation when evaluating a value. These semantics capture program execution as small rewrites on bits of a program, captured by a derivation tree that finds a single piece of work and does it. We can make this idea of computation-as-local-rewrites more explicit which can make for concise specifications, and later will be useful for defining the semantics of programming language features that implement complex control operations like exceptions and continuations.

We introduce *reduction semantics* a.k.a. *contextual semantics*, which is yet another way to define the $\longrightarrow$ single-step relation.

1 Evaluation Contexts

We already learned about the idea of frames for abstracting the structural rules of a structural operational semantics:

$$f \in \text{FRAME}$$
$$f ::= \text{if } \square \text{ then } t_1 \text{ else } t_2 \mid \square \text{ and } t \mid \square \text{ or } t \mid \square \text{ xor } t \mid v \text{ xor } \square \mid \text{not } \square$$

Each frame captures a single expression with a hole in it. Then our (sf) frame rule can repeatedly appeal to frames to decompose a program and find an interesting computation.

Reduction semantics refactor the combination of frames and rules by pushing the combination of frames into the syntax of *evaluation* contexts, leaving the need for only a single level of rule application to define the single-step relation.

To capture the entire context in which a computation may happen, we define the set ECTXT of *evaluation contexts*:

$$E \in \text{ECTXT}$$
$$E ::= \square \mid E[\text{if } \square \text{ then } t \text{ else } t] \mid E[\square \text{ and } t] \mid E[\square \text{ or } t] \mid E[\square \text{ xor } t] \mid E[v \text{ xor } \square] \mid E[\text{not } \square]$$

The idea here is that we can build up a context of computation by following the small-step structural rules and replacing them with context frames. So for example, $\square$ is the empty context. If we place

a $\text{if } \square \text{ then true else false}$ frame inside its hole, we get a new context; if we place a $\square \text{ and true}$ frame inside of *that* context's hole, then we get yet another context. Putting this all together, the evaluation context $\square[\text{if } \square \text{ then true else false}][\square \text{ and true}]$ represents a program with a hole in it that looks like the expression $\text{if } (\square \text{ and true}) \text{ then true else false}$.

Now bear in mind that the notation we use for evaluation contexts is meant to give you an intuition for what things mean. The hole notation $\square$ gives you a sense that you could stick something in there to either make a bigger context or a whole program. That's why we use $E[..]$ as notation for sticking .. in E's hole. However, the use of brackets in this notation is *not* the same mathematical expression as the plug function that we defined for frames. To fully appreciate this, it may be helpful to consider a desugared tree-like

*© Ronald Garcia. Not to be copied, used, or revised without explicit written permission from the copyright owner.

representation of evaluation contexts, much like our early presentation of the Boolean language, where we had atoms like if and we drew trees in parenthesized style. Here is a TREE-like presentation of evaluation contexts:

$$E ::= \mathsf{hole} \mid \mathsf{in\text{-}if}(E, t_1, t_2) \mid \mathsf{in\text{-}and}(E, t) \mid \mathsf{in\text{-}or}(E, t) \mid \mathsf{in\text{-}xor\text{-}l}(E, t) \mid \mathsf{in\text{-}xor\text{-}r}(E, v) \mid \mathsf{in\text{-}not}(E)$$

If we strip away the syntactic sugar and look at this structure, a few things become apparent. Notice that there is only one explicit reference to a hole, in the top-level hole structure, but the tree structures like in-if just carry an evaluation context. This is so because it is an implicit property of in-if that there is a hole in its argument position. To make this clearer, suppose we also wanted to support evaluation in the consequent position of an if expression. In structural operational semantics, that would correspond to adding a rule:

$$\text{(s-if-conseq)}\frac{t_2 \longrightarrow t_2'}{\mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3 \longrightarrow \mathsf{if}\ t_1\ \mathsf{then}\ t_2'\ \mathsf{else}\ t_3}$$

In our sugared evaluation contexts, this corresponds to adding a new case:

$$E ::= \ldots \mid E[\mathsf{if}\ t_1\ \mathsf{then}\ \square\ \mathsf{else}\ t_3]$$

but in our tree-based representation it does as well!

$$E ::= \ldots \mid \mathsf{in\text{-}if\text{-}conseq}(E, t_1, t_3)$$

So we tell where the hole is in an if expression based on which context frame we are introducing: in-if or in-if-conseq.

Finally, notice that the construction $\mathsf{in\text{-}and}(E, t)$ takes as one of its argument its *surrounding context*. That's how this corresponds to $E[\square\ \mathsf{and}\ t]$. This is why this form of evaluation context is called *inside-out* evaluation contexts because when you have a context E, you can immediately see the inner-most context frame, and as you decompose it, you work your way out to the top of the program[1].

The program-with-a-hole

$$(\mathsf{true\ xor}\ \square)\ \mathsf{xor\ false}$$

is written using our tree-based representation,

$$\square[\mathsf{true\ xor}\ \square][\square\ \mathsf{xor\ false}],$$

which desugars to

$$\mathsf{in\text{-}xor\text{-}l}(\mathsf{in\text{-}xor\text{-}r}(\mathsf{hole}, \mathsf{true}), \mathsf{false}).$$

2 Plugging

Now that we have a representation of a program with a hole in it, we need a way to talk about an entire program. Intuitively, if I have a context E, and I plug its hole with a term t, then the result should be an entire program. This is exactly the case, and we capture this by defining a plug function that works just like the one for frames $\cdot[\cdot] : \text{ECTXT} \times \text{TERM} \to \text{TERM}$ as follows:

$$\begin{aligned}
\square[t] &= t \\
(E[\mathsf{if}\ \square\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2])[t] &= E[\mathsf{if}\ t\ \mathsf{then}\ t_1\ \mathsf{else}\ t_2] \\
(E[\square\ \mathsf{and}\ t_1])[t] &= E[t\ \mathsf{and}\ t_1] \\
(E[\square\ \mathsf{or}\ t_1])[t] &= E[t\ \mathsf{or}\ t_1] \\
(E[\square\ \mathsf{xor}\ t_1])[t] &= E[t\ \mathsf{xor}\ t_1] \\
(E[v_1\ \mathsf{xor}\ \square])[t] &= E[v_1\ \mathsf{xor}\ t] \\
(E[\mathsf{not}\ \square])[t] &= E[\mathsf{not}\ t]
\end{aligned}$$

[1] There is also a notion of *outside-in* context, but we defer its description for now

Notice that plug is defined by recursion on the structure of evaluation contexts. As a helpful exercise, you should state the principle of recursion for evaluation contexts, and see if you can redefine plug in longhand form.

Note that the plug *function* $E[t]$ is technically a different thing from the context *constructors* like $E[\mathsf{succ}(\square)]$. It's a bit unfortunate that the same notation is used for technically different things, but on the other hand the two are intuitively similar, so this overloading is a bit nice but could be formally misleading. To tell the constructor notation apart from the plug function, look at what is in the hole: if it looks like a frame, then this is a constructor, but if it looks like a term, it's a reference to the plug function.

3 Reduction Semantics

We now have all of the machinery needed to re-present the semantics of the Boolean and Arithmetic Language as a reduction semantics. Keep in mind that both reduction semantics and structural operational semantics are examples of small step semantics. Not only this, both proceed by defining a small-step relation $\longrightarrow$, and then define the evaluator in terms of $\longrightarrow^*$, the reflexive transitive closure of $\longrightarrow$. In fact, the resulting relation $\longrightarrow$ is *exactly* the same in both cases. So what changed? As usual, what changed was the *structure* of the definition, which gives you a different way of looking at the same language, and also gives you different principles for reasoning about programs and the language as a whole.

Here are the main pieces of the reduction semantics. We start with the syntax of the language:

$$
\begin{array}{lll}
t \in \textsc{Term}, & v \in \textsc{Value}, & r \in \textsc{Redex} \\
t & ::= & \mathsf{true} \mid \mathsf{false} \mid \mathsf{if}\ t\ \mathsf{then}\ t\ \mathsf{else}\ t \mid t\ \mathsf{and}\ t \mid t\ \mathsf{or}\ t \mid t\ \mathsf{xor}\ t \\
v & ::= & \mathsf{true} \mid \mathsf{false} \\
r & ::= & \mathsf{if}\ v\ \mathsf{then}\ t\ \mathsf{else}\ t \mid v\ \mathsf{and}\ t \mid v\ \mathsf{or}\ t \mid v\ \mathsf{xor}\ v \mid \mathsf{not}\ v
\end{array}
$$

Then we introduce the syntax of evaluation contexts, which replaces all of the structural rules from the structural operational semantics:

$$
\begin{array}{l}
E \in \textsc{ECtxt} \\
E ::= \square \mid E[\mathsf{if}\ \square\ \mathsf{then}\ t\ \mathsf{else}\ t] \mid E[\square\ \mathsf{and}\ t] \mid E[\square\ \mathsf{or}\ t] \mid E[\square\ \mathsf{xor}\ t] \mid E[v\ \mathsf{xor}\ \square] \mid E[\mathsf{not}\ \square]
\end{array}
$$

Then we introduce the interesting computational rules, which are called *notions of reduction* (for no great reason as far as I can tell). The notions of reduction form a binary relation $\rightsquigarrow \subseteq \textsc{Redex} \times \textsc{Term}$:

$$
\begin{array}{rl}
\mathsf{if\ true\ then}\ t_2\ \mathsf{else}\ t_3 & \rightsquigarrow t_2 \\
\mathsf{if\ false\ then}\ t_2\ \mathsf{else}\ t_3 & \rightsquigarrow t_3 \\
\mathsf{true\ and}\ t_2 & \rightsquigarrow t_2 \\
\mathsf{false\ and}\ t_2 & \rightsquigarrow \mathsf{false} \\
\mathsf{true\ or}\ t_2 & \rightsquigarrow \mathsf{true} \\
\mathsf{false\ or}\ t_2 & \rightsquigarrow t_2 \\
v\ \mathsf{xor}\ v & \rightsquigarrow \mathsf{false} \\
\mathsf{false\ xor\ true} & \rightsquigarrow \mathsf{true} \\
\mathsf{true\ xor\ false} & \rightsquigarrow \mathsf{true} \\
\mathsf{not\ true} & \rightsquigarrow \mathsf{false} \\
\mathsf{not\ false} & \rightsquigarrow \mathsf{true}
\end{array}
$$

The notions of reduction *only* characterize the interesting steps of computation. We need to combine them with the evaluation contexts if we are to describe computation over arbitrary programs. We now define our single-step relation $\longrightarrow$ in terms of our notions of reduction, our *plug* function, and our evaluation contexts:

$$\frac{t_1 \rightsquigarrow t_2}{E[t_1] \longrightarrow E[t_2]}$$

As with frame-style S.O.S., this definition uses the plug function as a sugared side-condition. We can rewrite the above inference rule to better indicate what components of it are side-conditions and what components are part of the rule:

$$\frac{}{t_a \longrightarrow t_b} \text{ where } t_a = E[t_1], t_b = E[t_2], \text{ and } t_1 \rightsquigarrow t_2.$$

In fact, since there are no premises, this is a rather flat inductive definition (to see why I say so, write down the principle of induction for this definition). We could also write it down as in set comprehension notation and add quantifiers to make it a little more precise:

$$\longrightarrow = \{ \langle t_a, t_b \rangle \subseteq \textsc{Term} \times \textsc{Term} \mid \exists E \in \textsc{ECtxt}.\, \exists t_1 \in \textsc{Term}.\, \exists t_2 \in \textsc{Term}.\, t_a = E[t_1] \wedge t_b = E[t_2] \wedge t_1 \rightsquigarrow t_2 \}.$$

This definition is equivalent to the rule-based definition above.

To make these concepts clearer, consider the program:

$$\text{if zero?(pred(succ(z))) then false else true.}$$

Under reduction semantics, we can simply underline the term that is subject to the notion of reduction at each step.

$$\begin{aligned} & \text{if } (\text{if } \underline{(\text{true or } (\text{true xor true}))} \text{ then false else true}) \text{ then false else true} \\ \longrightarrow\ & \text{if } \underline{(\text{if true then false else true})} \text{ then false else true} \\ \longrightarrow\ & \underline{\text{if false then false else true}} \\ \longrightarrow\ & \text{true} \end{aligned}$$

Under structural operational semantics, each of these steps requires a proof tree. Under reduction semantics, on the other hand, each step requires an evaluation context E, and a notion of reduction $t_1 \rightsquigarrow t_2$.

4 Discussion

One nice feature of reduction semantics is that they are very compact: provide a syntax for evaluation contexts, a set of notions of reduction, and you can always assume the definition of $\longrightarrow$. It's not much more compact than frame-style S.O.S., but the former is more common in the literature.

However, we'll find that reduction semantics make some language features easier to define, or clearer to understand. It's also the case that this style of definition is more closely related to how you might implement such a language than structural operational semantics.

The downside is that we lose our ability to easily prove properties of a reduction step by induction, since the S.O.S. rules were our source of induction principles (induction over the structure of the context is often not helpful). However, if a reduction semantics is well structured, then we can reverse-engineer a structural operational semantics from it if we need one to make proving properties of the langauge easier.

References

A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Inf. Comput.*, 115(1):38–94, Nov. 1994.