Contextual Semantics

Andrew Tolmach

April 11, 2024

Small-step semantics rules, such as those for the language of booleans and numbers presented in TAPL Chapter 3, can generally be divided into two different categories. Rules such as

$$\texttt{if true then } \texttt{t}_2 \texttt{ else } \texttt{t}_3 \to \texttt{t}_2 \qquad \text{(E-IfTrue)}$$

and

$$\texttt{iszero true} \to \texttt{true} \qquad \text{(E-IsZeroZero)}$$

capture the essence of the language's computational behavior, and hence are sometimes called "computation rules" (or "basic rules"). (Actually, these are rule schemas, since they can contain metavariables.) Rules such as

$$\frac{\texttt{t}_1 \to \texttt{t}_1'}{\texttt{if } \texttt{t}_1 \texttt{ then } \texttt{t}_2 \texttt{ else } \texttt{t}_3 \to \texttt{if } \texttt{t}_1' \texttt{ then } \texttt{t}_2 \texttt{ else } \texttt{t}_3} \qquad \text{(E-If)}$$

and

$$\frac{\texttt{t}_1 \to \texttt{t}_1'}{\texttt{iszero } \texttt{t}_1 \to \texttt{iszero } \texttt{t}_1'} \qquad \text{(E-IsZero)}$$

serve to indicate where the computation rules can be applied inside larger terms; they essentially enforce an *evaluation strategy* (order of evaluation) for the language. These rules are sometimes called "congruence rules" or "structural rules." Arguably, presenting both kinds of rules together in a single uniform system tends to obscure the computation rules, which are in some sense more important. Moreover, if we examine the congruence rules, we can see that they all share a common pattern: the premise of each rule is an arbitrary evaluation step of the form $\texttt{t} \to \texttt{t}'$, and the conclusion is an evaluation step in which the left and right sides of the rule are the same term except that one instance of $\texttt{t}$ has been replaced by $\texttt{t}'$. This suggests that the congruence rules could be represented more compactly. *Contexts* provide a means to do this.

A *(term) context* is simply a term (or a term schema, if it contains metavariables) that has a hole, typically written [ ] or □, in place of a subterm. For example, $\texttt{if [ ] then } \texttt{t}_2 \texttt{ else } \texttt{t}_3$ and $\texttt{iszero pred [ ]}$ are two possible contexts for the term language of Chapter 3. The idea is that the hole can be "filled in" later to form a complete term. If $\texttt{c}$ is a context, we write $\texttt{c}[\texttt{t}]$ to represent the result of substituting $\texttt{t}$ for the hole in $\texttt{c}$.[1] For example,

[1]Note that this is simple textual substitution, unlike the capture-avoiding substitution studied in Chapter 5.

$$(\texttt{if [ ] then 0 else succ 0})[\texttt{true}] = \texttt{if true then 0 else succ 0}$$

(where the parentheses on the left-hand side are just to clarify that the substitution applies to the entire term). Dually, we can also view a context as a *pattern* that can be matched against a term, where a hole is like a "wild-card" that can match anything; if the match is successful, we can read out the subterm that corresponds to the hole.

Using contexts, we can rewrite the E-IF rule as

$$\frac{\texttt{t}_1 \to \texttt{t}_1'}{(\texttt{if [ ] then t}_2\texttt{ else t}_3)[\texttt{t}_1] \to (\texttt{if [ ] then t}_2\texttt{ else t}_3)[\texttt{t}_1']} \qquad \text{(E-IF}'\text{)}$$

Note that the hole in each conclusion context specifies the position of the subterm to be rewritten according to the premise transition. Clearly we could do the same thing with the other congruence rules.

Now we would like to capitalize on the regularity in congruence rules noted above by somehow condensing all the congruence rules into a single rule. In fact, we can condense *all combinations* of the congruence rules into a single rule. To do so, we define a set of contexts, using a (perfectly ordinary) grammar. For the language of Chapter 3, the appropriate grammar is

$$\texttt{C} ::= [\ ] \mid \texttt{succ C} \mid \texttt{pred C} \mid \texttt{iszero C} \mid \texttt{if C then t}_2\texttt{ else t}_3$$

This defines an (infinite) set of contexts

$$C = \{[\ ], \texttt{succ [ ]}, \texttt{pred [ ]}, \texttt{iszero [ ]}, \texttt{if [ ] then t}_2\texttt{ else t}_3, \texttt{succ succ [ ]}, \texttt{succ pred [ ]}, \ldots\}$$

The intuition is that if a context $\texttt{c} \in \texttt{C}$ matches a top-level term $\texttt{t}$, then the hole in $\texttt{c}$ corresponds to a subterm in $\texttt{t}$ where a computation rule can potentially be used (if a suitable one exists). Thus matching against a context works much like repeated use of congruence rules to guide us to a spot where a computation rule can be used.

More precisely, we can define a new variety of rule-based semantics, called *contextual semantics*, as follows:

(a) Define a basic stepping relation $\texttt{t} \to_{cmp} \texttt{t}'$ consisting of just the computation rules from the small-step semantics (with the *cmp* subscript added to the arrow in each rule).

(b) Fix a grammar $\texttt{C}$ of contexts, and define an evaluation relation $\texttt{t} \to_{ctx} \texttt{t}'$ on top-level terms (only) as the single rule

$$\frac{\texttt{t} \to_{cmp} \texttt{t}'}{\texttt{C}[\texttt{t}] \to_{ctx} \texttt{C}[\texttt{t}']} \qquad \text{(E-STEP)}$$

where this rule should be read as a schema in which $\texttt{C}$ may be replaced by any context $\texttt{c} \in \texttt{C}$ (and $\texttt{t}$ and $\texttt{t}'$ may be replaced by any term, as usual). We take $\texttt{t} \to_{ctx} \texttt{t}'$ as the fundamental definition of the semantic behavior of terms.

EXAMPLE: Consider the term `if iszero pred 0 then 0 else pred 0`.

As a reminder, under the small-step semantics, the one-step evaluation behavior of this term is given by the following derivation:

$$\dfrac{\dfrac{\dfrac{}{\texttt{pred 0} \to \texttt{0}}\ \textsc{E-PredZero}}{\texttt{iszero pred 0} \to \texttt{iszero 0}}\ \textsc{E-Iszero}}{\texttt{if iszero pred 0 then 0 else pred 0} \to \texttt{if iszero 0 then 0 else pred 0}}\ \textsc{E-If}$$

Under contextual semantics, the one-step evaluation behavior of any term is given by applying E-STEP. To do that, we must first find a context $\texttt{c} \in \texttt{C}$ that matches the term. In fact, there are several:

Context	Subterm matching hole
$\texttt{c}_1 =$ `[ ]`	`if iszero pred 0 then 0 else pred 0`
$\texttt{c}_2 =$ `if [ ] then t else t′`	`iszero pred 0`
$\texttt{c}_3 =$ `if iszero [ ] then t else t′`	`pred 0`
$\texttt{c}_4 =$ `if iszero pred [ ] then t else t′`	`0`

Of these, only the hole of context $\texttt{c}_3$ matches a computation rule (E-PREDZERO). Hence, E-STEP can only be applied by instantiating `C` with $\texttt{c}_3$, giving the following overall deriviation:

$$\dfrac{\dfrac{}{\texttt{pred 0} \to_{cmp} \texttt{0}}\ \textsc{E-PredZero}}{\texttt{if iszero pred 0 then 0 else pred 0} \to_{ctx} \texttt{if iszero 0 then 0 else pred 0}}\ \textsc{E-Step}$$

Compared with the small-step derivation, this one condenses all the uses of congruence rules (E-IF,E-ISZERO) into a single use of E-STEP. Note that the details of which context $\texttt{c} \in \texttt{C}$ is used and how this context matches the term are elided in this derivation. The fact that only one choice of `c` works is a natural corollary of the language's being deterministic.

As we would hope, the evaluation behavior is the same under both semantics. Indeed, for the small-step semantics of Chapter 3 and the definition of `C` above, we have

THEOREM 1: $\texttt{t} \to \texttt{t}' \iff \texttt{t} \to_{ctx} \texttt{t}'$.

Proof: ($\Leftarrow$). Proof is via a lemma: Suppose $\texttt{u} \to_{cmp} \texttt{u}'$; then, for any $\texttt{c} \in \texttt{C}$, $\texttt{c}[\texttt{u}] \to \texttt{c}[\texttt{u}']$. Assuming this lemma, we proceed as follows: If $\texttt{t} \to_{ctx} \texttt{t}'$ then, by inversion of E-STEP, there must exist a context $\texttt{c} \in \texttt{C}$ and a term `u` such that $\texttt{t} = \texttt{c}[\texttt{u}]$, $\texttt{u} \to_{cmp} \texttt{u}'$, and $\texttt{t}' = \texttt{c}[\texttt{u}']$. Applying the lemma to this `u`, $\texttt{u}'$ and `c` immediately gives us $\texttt{t} \to \texttt{t}'$ as required.

We prove the lemma by structural induction on `c`. If $\texttt{c} = [\ ]$, then $\texttt{c}[\texttt{u}] = \texttt{u}$ and $\texttt{c}[\texttt{u}'] = \texttt{u}'$; since $\texttt{u} \to_{cmp} \texttt{u}'$, certainly $\texttt{u} \to \texttt{u}'$, so the conclusion is immediate. If $\texttt{c} = \texttt{if } \texttt{c}_1 \texttt{ then } \texttt{t}_2 \texttt{ else } \texttt{t}_3$, then $\texttt{c}[\texttt{u}] = \texttt{if } \texttt{c}_1[\texttt{u}] \texttt{ then } \texttt{t}_2 \texttt{ else } \texttt{t}_3$ and $\texttt{c}[\texttt{u}'] = \texttt{if } \texttt{c}_1[\texttt{u}'] \texttt{ then } \texttt{t}_2 \texttt{ else } \texttt{t}_3$. By induction, $\texttt{c}_1[\texttt{u}] \to \texttt{c}_1[\texttt{u}']$. So by E-IF, $\texttt{c}[\texttt{u}] \to \texttt{c}[\texttt{u}']$ as required. The other cases in the context grammar are similar.

($\Rightarrow$) Left as an exercise. $\Box$

This approach for developing contextual semantics is quite general; it does not rely on any peculiar features of the Chapter 3 language. In particular, it extends to non-deterministic languages, where E-STEP might allow a computation rule to be applied at more than one subterm.

Exercise 1: Proof the $\Rightarrow$ direction of Theorem 1.

Exercise 2: Suppose we proceed as in Exercise 3.5.13.(2) and add the congruence rule E-Funny2 to our small-step semantics. What corresponding change do we need to make in the grammar of C to produce an equivalent contextual semantics?

Exercise 3: Change the definition of the `eval1` function in the `arith` implementation to use contextual style.

Notes

Contextual semantics were introduced in Felleisen and Hieb [1992]. Harper [2016, sect. 5.3] has a formal treatment, including a proof of Theorem 1 (for a somewhat different language). The suggested solution for coding `eval1` in inspired by the Haskell code given in Hutton [2023].

Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. *Theor. Comput. Sci.*, 103(2):235–271, 1992. URL `https://doi.org/10.1016/0304-3975(92)90014-7`.

Robert Harper. *Practical Foundations for Programming Languages (2nd. Ed.)*. Cambridge University Press, 2016. URL `http://www.cs.cmu.edu/~rwh/pfpl/abbrev.pdf`.

Graham Hutton. Programming language semantics: It's easy as 1,2,3. *Journal of Functional Programming*, October 2023. URL `https://www.cambridge.org/core/journals/journal-of-functional-programming/article/programming-language-semantics-its-easy-as-123/EC2C046CF94382B3B408036B84475DC7`.

Solutions to Exercises

Exercise 1 Proof of $\Rightarrow$ case of Theorem 1: We assume $\mathtt{t} \to \mathtt{t}'$ and must show $\mathtt{t} \to_{ctx} \mathtt{t}'$ using E-Step. So we must show that there exist $\mathtt{c} \in \mathtt{C}$ and a term $\mathtt{u}$ such that $\mathtt{t} = \mathtt{c}[\mathtt{u}]$, $\mathtt{u} \to_{cmp} \mathtt{u}'$, and $\mathtt{t}' = \mathtt{c}[\mathtt{u}']$. We proceed by induction on the structure of the derivation of $\mathtt{t} \to \mathtt{t}'$ and case analysis on the rule used at the root of the derivation. If the root is a computational rule, we take $\mathtt{c} = [\ ]$; then $\mathtt{t} = \mathtt{u}$ and $\mathtt{t}' = \mathtt{u}'$, so the result is immediate. If the root is rule T-If, then the conclusion of the rule has the form $\mathtt{if}\ \mathtt{t}_1\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3 \to \mathtt{if}\ \mathtt{t}_1'\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3$ and the premise is $\mathtt{t}_1 \to \mathtt{t}_1'$. By induction on that premise, there exist a context $\mathtt{c}_1 \in \mathtt{C}$ and a term $\mathtt{u}$ such that $\mathtt{t}_1 = \mathtt{c}_1[\mathtt{u}]$, $\mathtt{u} \to_{cmp} \mathtt{u}'$, and $\mathtt{t}_1' = \mathtt{c}_1[\mathtt{u}']$. Take $\mathtt{c} = \mathtt{if}\ \mathtt{c}_1\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3$. Then $\mathtt{c} \in \mathtt{C}$ and $\mathtt{t} = \mathtt{c}[\mathtt{u}]$ and $\mathtt{t}' = \mathtt{c}[\mathtt{u}']$, as required. The cases for the other congruence rules are similar.

Exercise 2

$$C ::= [\ ] \mid \mathtt{succ}\ C \mid \mathtt{pred}\ C \mid \mathtt{iszero}\ C \mid \mathtt{if}\ C\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3 \mid \mathtt{if}\ \mathtt{t}_1\ \mathtt{then}\ C\ \mathtt{else}\ \mathtt{t}_3$$

Exercise 3 In `arith/core.ml` replace the definition of `eval1` with the following code:

```
type context =
    CtxHole
  | CtxIf of context * term * term
  | CtxSucc of context
  | CtxPred of context
  | CtxIsZero of context

let rec subst (c:context) (t:term) : term = match c with
  | CtxHole -> t
  | CtxIf(c1,t2,t3) -> TmIf(dummyinfo,subst c1 t,t2,t3)
  | CtxSucc c1 -> TmSucc(dummyinfo,subst c1 t)
  | CtxPred c1 -> TmPred(dummyinfo,subst c1 t)
  | CtxIsZero c1 -> TmIsZero(dummyinfo,subst c1 t)

let rec split (t:term) : (context * term) list  =
  (CtxHole,t)::
  match t with
    TmTrue(_) -> []
  | TmFalse(_) -> []
  | TmZero(_) -> []
  | TmSucc(_,t1) -> List.map (fun (c',t') -> (CtxSucc c',t')) (split t1)
  | TmPred(_,t1) -> List.map (fun (c',t') -> (CtxPred c',t')) (split t1)
  | TmIsZero(_,t1) -> List.map (fun (c',t') -> (CtxIsZero c',t')) (split t1)
  | TmIf(_,t1,t2,t3) -> List.map (fun (c',t') -> (CtxIf(c',t2,t3),t')) (split t1)

let eval_cmp (t:term) : term  = match t with
    TmIf(_,TmTrue(_),t2,t3) ->
      t2
  | TmIf(_,TmFalse(_),t2,t3) ->
      t3
  | TmPred(_,TmZero(_)) ->
      TmZero(dummyinfo)
  | TmPred(_,TmSucc(_,nv1)) when (isnumericval nv1) ->
      nv1
  | TmIsZero(_,TmZero(_)) ->
      TmTrue(dummyinfo)
  | TmIsZero(_,TmSucc(_,nv1)) when (isnumericval nv1) ->
      TmFalse(dummyinfo)
  | _ ->
      raise NoRuleApplies

let eval1 (t:term) : term  =
  match List.concat
          (List.map
             (fun (c,t) ->
                 try [subst c (eval_cmp t)]
                 with NoRuleApplies -> [])
             (split t)) with
    h::_ -> h
  | _ -> raise NoRuleApplies
```