Big-Step Semantics

CPSC 509: Programming Language Principles

Ronald Garcia*

29 January 2014
(**Time Stamp**: 13:06, Monday 9th November, 2020)

By now, you have learned how to:

1. define sets using *inductive definitions*;
2. prove universal properties of inductively-defined sets using the corresponding principle of derivation induction (and rule induction) (a.k.a. *proof by induction*); and
3. define total functions that map an inductively defined set to other sets using the corresponding principle of recursive definition.

Isn't induction great?!? Given the last of these tools, we can define total functions over recursively defined sets with relative ease. In fact, we can use this principle to define an evaluator for the language of Boolean Expressions (see below).

As our languages get more sophisticated, though, we will find that definition by recursion does not always suffice for defining and analyzing semantics. Some programming language definitions have properties that don't play well with total functions:

1. If your language has programs that *don't terminate*, then you will be hard-pressed to define them recursively;
2. If your language has *nondeterministic behaviour*, meaning that one program or program fragment can produce more than one possible behaviour, then a recursive function definition may awkwardly describe its semantics: a more general relation between programs and possible results may be a more natural specification;
3. Sometimes you want to define a *partial function* somehow related to your language, and an equational definition of this can also be awkward some times.

This set of notes introduces a common approach to defining semantics that addresses some of these issues. We will do so while simultaneously extending our language of Boolean expressions with *arithmetic xpressions*. This is not strictly necessary, but introduces some new language concepts (especially *runtime errors*) along the way. We will use a different technique than recursion to define the evaluator, but the technique we are introducing is actually hidden inside the proof of the principle of recursive definition. Let's tease that out.

1 Boolean Expression Function Revisited

Earlier, we learned the Principle of Definition by Recursion, which enables us to define total functions over inductively defined sets, like the terms of our language of Boolean Expressions. One particular function

over boolean expressions we defined was its evaluator.

$$\textsc{Pgm} = \textsc{Term}, \quad \textsc{Obs} = \textsc{Value}$$
$$eval : \textsc{Pgm} \to \textsc{Obs}$$
$$eval(\mathsf{true}) = \mathsf{true}$$
$$eval(\mathsf{false}) = \mathsf{false}$$
$$eval(\mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3) = eval(t_2) \text{ if } eval(t_1) = \mathsf{true}$$
$$eval(\mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3) = eval(t_3) \text{ if } eval(t_1) = \mathsf{false}$$

Recall that this evaluator definition is justified by the principle of recursive definition from $t \in \textsc{Term}$, which means that we chose:

1. $S = \textsc{Value}$;
2. $s_t = \mathsf{true}$;
3. $s_f = \mathsf{false}$;
4. $H_{if} : \textsc{Value} \times \textsc{Value} \times \textsc{Value} \to \textsc{Value}$
   $H_{if}(\mathsf{true}, v_1, v_2) = v_1$
   $H_{if}(\mathsf{false}, v_1, v_2) = v_2$.

and fed them into the principle to produce a unique function, and we then convince ourselves intuitively that this is in fact our intended evaluator.

If we consider the *proof* of the Principle of Definition by Recursion, however, we see that the first step of the proof is to construct a binary relation $\Downarrow \subseteq \textsc{Term} \times \textsc{Value}$, typically called a *big-step* relation, that we then externally prove is a total function, satisfies the equations given, and is unique.[1] In short, we explicitly constructed our function as a binary relation, and then proved that it's a function. Let's examine that explicit construction.

If we inline our chosen elements into the rules for $\Downarrow$, we get roughly the following:

$$\frac{}{\mathsf{true} \Downarrow \mathsf{true}}\ \text{(etrue)} \qquad \frac{}{\mathsf{false} \Downarrow \mathsf{false}}\ \text{(efalse)}$$

$$\frac{t_1 \Downarrow \mathsf{true} \quad t_2 \Downarrow v_2}{\mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3 \Downarrow v_2}\ \text{(eif-t)} \qquad \frac{t_1 \Downarrow \mathsf{false} \quad t_3 \Downarrow v_3}{\mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3 \Downarrow v_3}\ \text{(eif-f)}$$

Note that the (eif-t) and (eif-f) rules together specialize the following single rule with respect to the definition of H_{if}:

$$\frac{t_1 \Downarrow v_1 \quad t_2 \Downarrow v_2 \quad t_3 \Downarrow v_3}{\mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3 \Downarrow H_{if}(v_1, v_2, v_3)}\ \text{(eif)}$$

In short, it's not hard to prove that

$$\begin{aligned}\Downarrow &= \{\, \langle t, v\rangle \in \textsc{Term} \times \textsc{Value} \mid \exists \mathcal{D} \in \textsc{Deriv}[\text{etrue,efalse,eif}].\mathcal{D} :: \langle t, v\rangle \,\} \\ &= \{\, \langle t, v\rangle \in \textsc{Term} \times \textsc{Value} \mid \exists \mathcal{D} \in \textsc{Deriv}[\text{etrue,efalse,eif-t,eif-f}].\mathcal{D} :: \langle t, v\rangle \,\}\,.\end{aligned}$$

The key observation is that based on the value of v_1, H_{if} discards either v_2 or v_3, so there is no real need for the discarded derivation. This improvement in the rules is just due to some human cleverness: it is not fundamental, but it also matches the last two equations of our recursive definition above. Later, we use this split to motivate a nice implementation of the language.

Based on our proof of the Principle of Definition by Recursion, it is clear that $\Downarrow = eval$, both count as definitions of the same total function. The key difference between the two approaches is a matter of flexibility. Inductive definitions are more general: they need not be total, and they need not be partial functions (i.e., deterministic). We sometimes want that flexibility in the definition of our language semantics. For this reason, among others, inductive definitions like $\Downarrow$ are often the preferred mode of specifying the semantics of programming language. In the next section we use inductive definitions to define a semantics that is deterministic but *partial*.

[1] We use the name $\Downarrow$ for reasons that should be explained shortly.

2 Boolean and Arithmetic Expressions

We extend the syntax of Boolean expressions with *numerals* $nv \in \text{NUM}$, which are some syntactic representation of numbers (another example of numerals is Roman numerals), and several operations on them.

$$\begin{array}{rcl} t \in \text{TERM}, & v \in \text{VALUE}, & nv \in \text{NUM} \\ t & ::= & \mathsf{true} \mid \mathsf{false} \mid \mathsf{if}\ t\ \mathsf{then}\ t\ \mathsf{else}\ t \\ & \mid & \mathsf{z} \mid \mathsf{succ}(t) \mid \mathsf{pred}(t) \mid \mathsf{zero?}(t) \\ v & ::= & \mathsf{true} \mid \mathsf{false} \mid nv \\ nv & ::= & \mathsf{z} \mid \mathsf{succ}(nv) \end{array}$$

Our numerals stand for natural numbers, and are written in a unary notation: For example, the term z stands for the number 0, $\mathsf{succ}(\mathsf{z})$ stands for 1, $\mathsf{succ}(\mathsf{succ}(\mathsf{z}))$ stands for 2, and so on. We can formally relate numerals to natural numbers with a function:

$$\begin{array}{c} nat : \text{NUM} \to \mathbb{N} \\ nat(\mathsf{z}) = 0 \\ nat(\mathsf{succ}(nv)) = 1 + nat(nv). \end{array}$$

In addition to these expressions, $\mathsf{pred}(t)$ computes the predecessor of an expression, while $\mathsf{zero?}(t)$ determines whether a numeric term is z-valued.

2.1 Big-step semantics

To define the meaning of this language, we directly define a *binary relation* $\Downarrow \subseteq \text{TERM} \times \text{VALUE}$ that maps terms in our language to *values*. Conceptually, VALUEs are the expressions that the TERMs map to. In this language $\text{VALUE} \subseteq \text{TERM}$, though that isn't always the case.

This style of relation is called a *big-step* operational semantics, big-step semantics for short, because our relation takes a BIG STEP from programs directly to final values. For historical reasons, it is also sometimes called a *natural* semantics [Kahn, 1987]. In the next section, we'll use this mapping to define our evaluator. Typically the symbol $\Downarrow$ is used as the name of a big-step semantics, though sometimes you will also see a fish hook $\hookrightarrow$ used.

Our big-step relation for Boolean and Arithmetic Expressions is defined by the following rules:

$$\text{(etrue)}\ \frac{}{\mathsf{true} \Downarrow \mathsf{true}} \qquad \text{(efalse)}\ \frac{}{\mathsf{false} \Downarrow \mathsf{false}} \qquad \text{(eif-t)}\ \frac{t_1 \Downarrow \mathsf{true} \quad t_2 \Downarrow v_2}{\mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3 \Downarrow v_2}$$

$$\text{(eif-f)}\ \frac{t_1 \Downarrow \mathsf{false} \quad t_3 \Downarrow v_3}{\mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3 \Downarrow v_3} \qquad \text{(ez)}\ \frac{}{\mathsf{z} \Downarrow \mathsf{z}} \qquad \text{(esucc)}\ \frac{t \Downarrow nv}{\mathsf{succ}(t) \Downarrow \mathsf{succ}(nv)}$$

$$\text{(epred)}\ \frac{t \Downarrow \mathsf{succ}(nv)}{\mathsf{pred}(t) \Downarrow nv} \qquad \text{(ezero?-z)}\ \frac{t \Downarrow \mathsf{z}}{\mathsf{zero?}(t) \Downarrow \mathsf{true}} \qquad \text{(ezero?-s)}\ \frac{t \Downarrow \mathsf{succ}(nv)}{\mathsf{zero?}(t) \Downarrow \mathsf{false}}$$

A few things are worth noting. In each rule, we must take care that each component of $\Downarrow$ is consistent with the overarching specification that $\Downarrow \subseteq \text{TERM} \times \text{VALUE}$. For any valid instance of a rule, the first component must be a TERM and the second component must be a VALUE. For instance, suppose we had written the (esucc) rule as:

$$\text{(esucc)}\ \frac{t \Downarrow v}{\mathsf{succ}(t) \Downarrow \mathsf{succ}(v)}$$

This sample rule would be inconsistent because it could be instantiated with $v = \mathsf{true}$, and $\mathsf{succ}(\mathsf{true}) \notin \text{VALUE}$. We must either consider this rule invalid or expand our notion of VALUEs.

Also, we can tell that the language is partial based on what rules we have available for building derivation trees. For example, $\mathsf{zero?}(\mathsf{true})$ does not evaluate to a value. We can tell by checking which rules could

be instantiated so that they represent evaluation of the above expression. We can instantiate both (ezero?-z) and (ezero?-s) to get:

$$\text{(ezero?-z)}\frac{\mathsf{true} \Downarrow \mathsf{z}}{\mathsf{zero?}(\mathsf{true}) \Downarrow \mathsf{true}} \qquad \text{(ezero?-s)}\frac{\mathsf{true} \Downarrow \mathsf{succ}(nv)}{\mathsf{zero?}(\mathsf{true}) \Downarrow \mathsf{false}}$$

These are valid instances, but their premises are not achievable: true only big-steps to true. Therefore there is no value v such that $\langle \mathsf{zero?}(\mathsf{true}), v\rangle \in \Downarrow$.

Bear in mind that this restriction was a *design decision* for this language. We could have defined the meaning of zero? to give true if the argument is z, give false otherwise. To do so, replace the rule (ezero?-s) with the following:

$$\text{(ezero?-nz)}\frac{t \Downarrow v}{\mathsf{zero?}(t) \Downarrow \mathsf{false}} \text{ where } v \neq \mathsf{z}.$$

With this in place, $\mathsf{zero?}(\mathsf{true}) \Downarrow \mathsf{false}$. The side-condition that $v \neq \mathsf{z}$ is needed to ensure that the new rule cannot be instantiated with $v = \mathsf{z}$ If it could, then there would be two rules that could be instantiated for the same term, i.e.

$$\text{(ezero?-z)}\frac{t \Downarrow \mathsf{z}}{\mathsf{zero?}(t) \Downarrow \mathsf{true}} \qquad \text{(ezero?-nz)}\frac{t \Downarrow \mathsf{z}}{\mathsf{zero?}(t) \Downarrow \mathsf{false}}$$

Together, these rules would make the language both nondeterministic and confusing.

Now this definition of $\Downarrow$ should remind you of the process of proving the Principle of Definition by Recursion. There, you first define a binary relation, and then prove that that binary relation is a total function. Here we have defined a binary relation, and we can then prove that it is a *partial function*

Proposition 1 ($\Downarrow$ is a partial function)**.** *If $t \Downarrow v_1$ and $t \Downarrow v_2$ then $v_1 = v_2$.*

Proof. Left as an exercise for the reader. □

Remember that every total function is a partial function too, but as we informally argued above, $\Downarrow$ is not total.

Proposition 2 ($\Downarrow$ is not total)**.** *For some $t \in$ TERM $t \not\Downarrow v$ for any $v \in$ VALUE.*

Proof. We showed above that $\langle \mathsf{zero?}(\mathsf{true}), v\rangle \notin \Downarrow$ for all v. □

2.2 Defining the evaluator

Now that we have defined our big-step relation, it's time to define our evaluator. It's worth noting that we *could* directly define our evaluator using the $\Downarrow$ relation[2], as in

$$\begin{array}{l}\text{PGM} = \text{TERM}, \quad \text{OBS} = \text{VALUE}\\ \mathit{eval} : \text{PGM} \rightharpoonup \text{OBS}\\ \mathit{eval} = \Downarrow\end{array}$$

But that is not a foregone conclusion. The first day of class we said that the general structure of language semantics involves an evaluator, and that the structure of the evaluator gives you a tool for reasoning. Here we'll see that the same big-step relation can be used to define multiple different evaluators. The side-effect is that each evaluator has the same reasoning tools at its disposal (derived from the structure of the big-step relation), but the answers may be different.

Here is another evaluator based on $\Downarrow$. Define by

$$\begin{array}{lll}\text{PGM} = \text{TERM}, \quad \text{OBS} = \{\,\mathsf{true}, \mathsf{false}\,\} \cup \mathbb{N} & & \\ \mathit{eval}_{BA} : \text{PGM} \to \text{OBS} & & \\ \mathit{eval}_{BA}(t) & = \mathsf{true} & \text{if } t \Downarrow \mathsf{true}\\ \mathit{eval}_{BA}(t) & = \mathsf{false} & \text{if } t \Downarrow \mathsf{false}\\ \mathit{eval}_{BA}(t) & = \mathit{nat}(nv) & \text{if } t \Downarrow nv\end{array}$$

[2]Note that we are still obliged to prove that $\Downarrow$ is a partial function

This evaluator is not much different from the previous one, except that numeric results are presented in terms of the natural numbers (thanks to nat, which we defined earlier), rather than the numerals that we defined for computation purposes.

A significantly different evaluator is defined below. Define $eval_{BAs} : \text{PGM} \to \{\,\mathsf{success}\,\}$ by

$$eval_{BAs}(t) = \mathsf{success} \text{ if } t \Downarrow v$$

This evaluator only produces one observable result: whether the big-step relation yields some value. Thinking in terms of an implementation for a moment, there is another possible outcome: it could crash if there is no way to make progress, as in $\mathsf{zero?}(\mathsf{true})$.

These two evaluators essentially define two *different* languages that happen to have the same syntax. Therefore, questions that we might ask about the two languages would give different answers.

For example, We might consider two programs to be equivalent if evaluating them produces the same value. Under this definition, true and z are not equivalent under $eval_{BA}$, but they *are* equivalent under $eval_{BAs}$.

This example may seem bizarre, but there are more complex languages where a definition like $eval_{BAs}$ is easier to reason about, and it also shares important properties with its less-trivial counterpart.

3 Inversion Lemmas, Revisited

One of the nice things about a recursive function definition is that we can use the equations that define the function to calculate what a recursive function maps its argument to. However, we didn't define the BA language as a set of recursive equations, so we'll need a new strategy for calculating the results of evaluation (either by hand, or by implementing an interpreter).

Before looking at the general strategy, lets consider an instance. How do we determine what (if anything) $\mathsf{zero?}(\mathsf{z})$ evaluates to? Given the structure of our evaluators above, we know that we must determine what it big-steps to. Since the $\Downarrow$ relation is defined inductively, we know that $\mathsf{zero?}(\mathsf{z}) \Downarrow v$ for some v if and only if there is some derivation $\mathcal{D} :: \mathsf{zero?}(\mathsf{z}) \Downarrow v$. Thus, calculating the mapping boils down to *searching* for a derivation that begins with our term:

$$\begin{array}{c} \vdots \\ \mathsf{zero?}(\mathsf{z}) \Downarrow ??? \end{array}$$

We can consider two possible rules at this point, (ezero?-z) and (ezero?-s). If we try (ezero?-s) then we quickly get to a dead-end

$$\frac{\begin{array}{c} X \\ \mathsf{z} \Downarrow \mathsf{succ}(nv) \end{array}}{\mathsf{zero?}(\mathsf{z}) \Downarrow \mathsf{false}},$$

so we can then try to instantiate (ezero?-z):

$$\frac{\begin{array}{c} \vdots \\ \mathsf{z} \Downarrow \mathsf{z} \end{array}}{\mathsf{zero?}(\mathsf{z}) \Downarrow \mathsf{true}},$$

Then, we can instantiate the (ez) rule to close off the top:

$$\frac{\dfrac{}{\mathsf{z} \Downarrow \mathsf{z}}}{\mathsf{zero?}(\mathsf{z}) \Downarrow \mathsf{true}}.$$

This gives us the derivation we wanted, and as a result we have calculated that $\mathsf{zero?}(\mathsf{z}) \Downarrow \mathsf{true}$.

We can make this reasoning process more concrete. Given a term t, we try to find a value that it big-steps to by searching bottom-up for a derivation that has t on its left-hand side, considering the rules that could possibly be instantiated to match our goal. This reasoning is made formal as a set of propositions: *inversion lemmas*, like we have talked about earlier and used in the homework.

Proposition 3 (Inversion, distinguishing the top-level structure of t)**.**

1. *If* $\mathsf{true} \Downarrow v$ *then* $v = \mathsf{true}$.
2. *If* $\mathsf{false} \Downarrow v$ *then* $v = \mathsf{false}$.
3. *If* $\mathsf{if}\ t_1\ \mathsf{then}\ t_2\ \mathsf{else}\ t_3 \Downarrow v$ *then either*
   (a) $t_1 \Downarrow \mathsf{true}$ *and* $t_2 \Downarrow v$ *or*
   (b) $t_1 \Downarrow \mathsf{false}$ *and* $t_3 \Downarrow v$.
4. *If* $\mathsf{z} \Downarrow v$ *then* $v = \mathsf{z}$.
5. *If* $\mathsf{succ}(t) \Downarrow v$ *then* $t \Downarrow v_1$, $v_1 \in \textsc{Num}$, *and* $v = \mathsf{succ}(v_1)$.
6. *If* $\mathsf{pred}(t) \Downarrow v$ *then* $t \Downarrow \mathsf{succ}(v)$ *and* $v \in \textsc{Num}$.
7. *If* $\mathsf{zero?}(t) \Downarrow v$ *then either*
   (a) $t \Downarrow \mathsf{z}$ *and* $v = \mathsf{true}$ *or*
   (b) $t \Downarrow \mathsf{succ}(nv)$ *and* $v = \mathsf{false}$.

To prove these propositions, we first expand them to be formal statements about derivations. For example, item 7 expands to the following:

Proposition 4. $\forall \mathcal{D}.\mathcal{D} :: \mathsf{zero?}(t) \Downarrow v \Rightarrow ((\exists \mathcal{E}.\mathcal{E} :: t \Downarrow \mathsf{z}) \wedge v = \mathsf{true}) \vee ((\exists \mathcal{E}.\mathcal{E} :: t \Downarrow \mathsf{succ}(nv)) \wedge v = \mathsf{false})$

Proof. By cases on the last rule used to form $\mathcal{D}$.
Case (etrue). Vacuous because true does not match $\mathsf{zero?}(t)$.
Case (etrue). Vacuous.
Case (efalse). Vacuous.
Case (eif-t). Vacuous.
Case (eif-f). Vacuous.
Case (ez). Vacuous.
Case (esucc). Vacuous.
Case (epred). Vacuous.
Case (ezero?-z). Then

$$\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}' \\ t \Downarrow \mathsf{z}\end{array}}{\mathsf{zero?}(t) \Downarrow \mathsf{true}}$$

Then we can let $\mathcal{D}'$ be our $\mathcal{E}$, and clearly $v = \mathsf{true}$. Thus the first part of our disjunction is true, making the whole thing true.
Case (ezero?-s). This case is analogous to the (ezero?-z) case:

$$\mathcal{D} = \dfrac{\begin{array}{c}\mathcal{D}' \\ t \Downarrow \mathsf{succ}(nv)\end{array}}{\mathsf{zero?}(t) \Downarrow \mathsf{false}}$$

Then we can let $\mathcal{D}'$ be our $\mathcal{E}$, and clearly $v = \mathsf{false}$. Thus the *second* part of our disjunction is true, making the whole thing true.

□

Each of the statements of the Inversion proposition can be proven in the same manner as this one: by cases on the derivation $\mathcal{D}$. As you can see, the proof is quite straightforward, especially since most of the cases are vacuous because the rule doesn't match. The proof is simple enough that many books or papers do not even bother to state the proof strategy or its cases. However, it is important that you know how one *would* prove such a thing. While it's intuitively obvious to us, it would not be intuitively to a computer. Nonetheless we can make the proof a purely mechanical thing that a mechanized proof assistant on a computer could check for correctness.

As you will see, these propositions can serve as the basis for an implementation of the language.

References

G. Kahn. Natural semantics. In *4th Annual Symposium on Theoretical Aspects of Computer Sciences on STACS 87*, pages 22–39, London, UK, UK, 1987. Springer-Verlag.