

# CPSC 311: Definition of Programming Languages: Error rules; small-step semantics and evaluation contexts ("10-contexts")

Joshua Dunfield  
University of British Columbia

October 10, 2015

## Topics discussed

- a *judgment* is whatever is in the conclusion of a rule, not necessarily  $e \Downarrow v$ , not necessarily about evaluation or dynamic semantics
- notation for judgments and rules
- error rules for Fun... and more... and still more
- *small-step semantics*
  - inside an evaluation derivation, little steps of computation happen
  - we can model these with a different judgment
  - *reduction rules* for the actual steps of computation
  - rule Step-context and *evaluation contexts* for finding the subexpression where the actual step can happen
- BNFs not just for concrete syntax; can also define subsets of abstract syntax
- evaluation contexts: define with a BNF, notation for replacing holes
- **note:** "evaluation semantics"  $\implies$  "big-step semantics"
- only need one small-step rule for free variable errors  
(the lecture on Friday, 2015/10/02 ended around this point)  
(note: I am probably going to skip to something different for Monday's lecture)
- relating small- and big-step semantics
- small-step semantics and recursion

## 1 Collected rules for Fun (again)

Figure 1 collects all the rules, again, showing Eval-app-value rather than Eval-app-expr. I added Eval-rec because it will come in handy.

## §1 Collected rules for Fun (again)

---

In this note I'll keep using the old Fun language rather than the Fun++ language on the assignment, for tedious reasons (I might give away part of the solution) but also to keep the number of rules small.

Some additional features of this figure are explained below.

$e \Downarrow v$  Expression  $e$  evaluates to value  $v$

$$\begin{array}{c} \frac{}{(\text{num } n) \Downarrow (\text{num } n)} \text{Eval-num} \quad \frac{e1 \Downarrow (\text{num } n_1) \quad e2 \Downarrow (\text{num } n_2)}{(\text{add } e1 \ e2) \Downarrow (\text{num } (n_1 + n_2))} \text{Eval-add} \quad \frac{e1 \Downarrow (\text{num } n_1) \quad e2 \Downarrow (\text{num } n_2)}{(\text{sub } e1 \ e2) \Downarrow (\text{num } (n_1 - n_2))} \text{Eval-sub} \\ \\ \frac{e1 \Downarrow v1 \quad \text{subst}(e2, x, v1) \Downarrow v2}{(\text{with } x \ e1 \ e2) \Downarrow v2} \text{Eval-with} \\ \\ \frac{}{(\text{lam } x \ e1) \Downarrow (\text{lam } x \ e1)} \text{Eval-lam} \quad \frac{e1 \Downarrow (\text{lam } x \ eB) \quad e2 \Downarrow v2 \quad \text{subst}(eB, x, v2) \Downarrow v}{(\text{app } e1 \ e2) \Downarrow v} \text{Eval-app-value} \\ \\ \frac{\text{subst}(e, u, (\text{rec } u \ e)) \Downarrow v}{(\text{rec } u \ e) \Downarrow v} \text{Eval-rec} \end{array}$$

$e$  free-variable-error  $e$  raises a free variable error

$$\frac{}{(\text{id } x) \text{ free-variable-error}} \text{FVerr-id}$$

---

Figure 1 Evaluation rules for Fun

---

## 2 Judgments

Now that you have some familiarity with evaluation rules, you may have room in your mind for more terminology.

We started out (in the AE and WAE languages) by saying that “ $e \Downarrow n$ ” meant “ $e$  evaluates to the number  $n$ ”. After we moved on to the Fun language, we needed expressions to evaluate to either numbers or lams, so we stopped using  $e \Downarrow n$  and started using  $e \Downarrow v$  instead.

Rules and derivation trees, however, are not limited to statements about what an expression evaluates to. Gentzen invented the tree notation to represent mathematical proofs, and its invention substantially predates the development of programming language semantics. The statements “ $e \Downarrow n$ ” and “ $e \Downarrow v$ ” are just particular kinds of *judgments*, and “ $e \Downarrow n$ ” and “ $e \Downarrow v$ ” are particular *judgment forms*.

We actually introduced another judgment form without much fuss: “ $e$  free-variable-error”, used as the conclusion of the rule Eval-free-identifier (which I'm going to rename FVerr-id). In Figure 1, we move this rule away from the others, and add headings like this:

## §2 Judgments

---

*judgment* *reading*

Below this heading, we put all the rules whose conclusion has the stated judgment form.

The *reading* is both a “pronunciation key” (how to *read aloud* a judgment of this form), and a suggestion for how to *understand* judgments of this form. The judgment form *e free-variable-error* is rather self-explanatory, but here are some other judgment forms seen in the wild:

$$\begin{array}{l} P \text{ true} \\ \Gamma \vdash P \text{ valid} \\ \tau : \kappa \\ \Gamma \vdash e : \tau \\ \Gamma \vdash e :_{\varphi} A \leftrightarrow M \\ e \longrightarrow e' \\ G_1 \vdash_{\omega}^p e \Downarrow G_2; t \end{array}$$

We’ll learn what some of these mean later in 311. Defining a language via rules usually requires several different judgment forms, both for the dynamic semantics and for the static semantics.

All of the terminology (rule, premise, conclusion, derivation tree, derivable), as well as techniques for working with rules and derivations, such as the “method of hope”, is the same for any judgment form, no matter what the purpose of the judgment is. We’ll see this in a little while when we introduce *small-step semantics* as a different way of specifying the behaviour of an interpreter, and later when we look at static semantics, particularly static typing.

### 3 Error rules for Fun

Ignoring errors, such as free identifiers, trying to add a function to a number, or trying to apply a number to a function, our rules match the interpreter we wrote. However, the specification of errors isn’t very satisfactory.

Let’s consider *only* the free identifier (or free variable) error, and whether our interpreter is sound and complete given that rule.

Last time, we said:

■ **Definition 1.** Completeness of the interpreter: If  $e \Downarrow v$  is derivable then  $(\text{interp } e) = v$ .

■ **Definition 2.** Soundness of the interpreter: If  $(\text{interp } e) = v$  then  $e \Downarrow v$  is derivable.

These notions make sense for other judgments, such as the *e free-variable-error* judgment:

■ **Definition 3.** Completeness of the interpreter’s handling of free variable errors: If *e free-variable-error* is derivable then  $(\text{interp } e)$  raises a free variable error.

### §3 Error rules for Fun

Completeness means that, whenever the rules for deriving  $e$  free-variable-error say that a free variable error has occurred, our interpreter will flag such an error. By “raises” or “flags”, I mean what the Racket (actually PLAI) error function does.

■ **Definition 4.** Soundness of the interpreter’s handling of free variable errors:  
If  $(\text{interp } e)$  raises a free variable error then  $e$  free-variable-error.

Is our interpreter’s handling of free variable errors sound and complete?

If we evaluate  $(\text{interp } (\text{id } 'i))$  our interpreter evaluates  $(\text{error } \text{"free-variable"})$ . This seems to correspond to the one rule that can derive the  $e$  free-variable-error judgment form. Note that  $\text{interp}$ ’s **type-case** branch

```
[id (x)
  (error "free-variable")]
```

doesn’t inspect  $x$ , so it will do the same thing for  $(\text{interp } (\text{id } 'zzzzz))$  or any other symbol.

However, our interpreter is *not* sound, because our interpreter is doing a *much better* job of catching free variables! For example:

```
(interp (add (num 5) (id 'i)))
```

gives a free variable error, which is what we *want*, but isn’t what the rules say! The rules say that the only expression that should cause a free-variable-error is  $\text{id}$ . Not an expression that contains an  $\text{id}$ , but exactly an  $\text{id}$ .

When we say our interpreter is not sound, we must understand that this does not necessarily mean our interpreter is wrong! Soundness is always *with respect to* a definition. Here, it is our *definition* that really needs to change, because our definition doesn’t match our expectation. We expect that a free variable should cause an error even if it’s hiding inside an  $\text{add}$ .

How should we fix our definition? We have to add lots of new rules, like this:

$e$  free-variable-error  $e$  raises a free variable error

$$\frac{}{(\text{id } x) \text{ free-variable-error}} \text{FVerr-id}$$
$$\frac{e1 \text{ free-variable-error}}{(\text{add } e1 \ e2) \text{ free-variable-error}} \text{FVerr-add-left} \quad \frac{e2 \text{ free-variable-error}}{(\text{add } e1 \ e2) \text{ free-variable-error}} \text{FVerr-add-right}$$
$$\frac{e1 \text{ free-variable-error}}{(\text{app } e1 \ e2) \text{ free-variable-error}} \text{FVerr-app-left} \quad \frac{e2 \text{ free-variable-error}}{(\text{app } e1 \ e2) \text{ free-variable-error}} \text{FVerr-app-right}$$

We would need several more rules, which I won’t bore you with, but we have to know when to stop. The following rule would not be helpful, even though it seems to follow the pattern established above.

$$\frac{e \text{ free-variable-error}}{(\text{lam } x \ e) \text{ free-variable-error}} \text{FVerr-lam??}$$

### §3 Error rules for Fun

Why? Well, we want to flag *free* identifiers, but  $x$  will be flagged as free even though the  $\text{lam}$  binds it!

$$\frac{\frac{}{(id\ x)\ \text{free-variable-error}}\ \text{FVerr-id}}{(\text{lam}\ x\ (id\ x))\ \text{free-variable-error}}\ \text{FVerr-lam??}$$

Putting in this rule seems bad. Fortunately, we can leave it out without affecting soundness and completeness. Our interpreter doesn't catch free variable errors until it actually reaches them:

```
> (interp (lam 'a (id 'b)))  
(lam 'a (id 'b))  
> (interp (app (lam 'a (id 'b)) (num 0)))  
⊗ free-variable
```

Whether this is what we really want is another question; for the sake of stability, to keep us from changing both the interpreter and the rules at the same time, let's just assume it is.

But something still doesn't match up:

$$\frac{e2\ \text{free-variable-error}}{(\text{add}\ e1\ e2)\ \text{free-variable-error}}\ \text{FVerr-add-right}$$

This rule is looking too hard: if  $e1$  doesn't evaluate to something—for example, when  $e1$  is  $(\text{rec}\ u\ (id\ u))$ —FVerr-add-right will raise an error, even though our interpreter wouldn't.

$$\frac{e1\ \Downarrow\ v1\quad e2\ \text{free-variable-error}}{(\text{add}\ e1\ e2)\ \text{free-variable-error}}\ \text{FVerr-add-right-better}$$

This should match our interpreter. But we also have to change FVerr-app-right, and other FVerr rules we didn't bother to write down. Now imagine doing this for other errors that our interpreter catches, but that we haven't written rules for. And *then* imagine doing this for an actual language with many more features than Fun!

Our interpreter tries to evaluate the given expression  $e$ ; this involves interpreting the expressions inside  $e$ . But if it notices a free variable, it raises an error. Can we distill what it means to be “about to notice a free variable”, and somehow use that in our rules? Yes, but to do this effectively we need to use a different kind of semantics, small-step semantics, which will turn out to have several advantages over evaluation semantics.

(Aside: My opposition to “interpreter semantics” doesn't mean I'm against allowing a particular interpreter that seems to do something sensible, like our `interp` function in `dynsem-fun.rkt`, to guide our development of the rules. In the end, a definition should be independent of all of its implementations, but to develop that definition, we should welcome insights from anywhere!)

## 4 Error rules for Fun, in painful detail

When is our interpreter about to notice a free variable? That is, when we do  $(\text{interp}\ e)$ , when will our interpreter raise an error?

#### §4 Error rules for Fun, in painful detail

---

- (1) when  $e$  is  $(\text{id } \dots)$
- (2) when  $e$  is  $(\text{add } e_1 e_2)$  and we're about to notice a free variable in  $e_1$
- (3) when  $e$  is  $(\text{add } e_1 e_2)$ , we evaluated  $e_1$  to some value, and we're about to notice a free variable in  $e_2$
- (4) same as (2) and (3), but for  $\text{sub}$
- (5) when  $e$  is  $(\text{app } e_1 e_2)$  and we're about to notice a free variable in  $e_1$
- (6) when  $e$  is  $(\text{app } e_1 e_2)$  and  $e_1$  evaluates to  $(\text{lam } x e_B)$  and we're about to notice a free variable in  $e_2$
- (7) when  $e$  is  $(\text{app } e_1 e_2)$  and  $e_1$  evaluates to  $(\text{lam } x e_B)$  and  $e_2$  evaluates to  $v_2$  and we're about to notice a free variable in  $\text{subst}(e_B, x, v_2)$
- (8) when  $e$  is  $(\text{with } x e_1 e_B)$  and we're about to notice a free variable in  $e_1$
- (9) when  $e$  is  $(\text{with } x e_1 e_B)$ , we evaluated  $e_1$  to some value, and we're about to notice a free variable in  $e_B$

This attempted definition is essentially repeating all the evaluation rules, but with “about to notice a free variable” replacing one premise. We have the evaluation rule

$$\frac{e_1 \Downarrow (\text{lam } x e_B) \quad e_2 \Downarrow v_2 \quad \text{subst}(e_B, x, v_2) \Downarrow v}{(\text{app } e_1 e_2) \Downarrow v} \text{Eval-app-value}$$

and we could write these rules corresponding to (5)–(7):

$$\frac{e_1 \text{ free-variable-error}}{(\text{app } e_1 e_2) \text{ free-variable-error}} \text{FVerr-app-1} \quad \frac{e_1 \Downarrow (\text{lam } x e_B) \quad e_2 \text{ free-variable-error}}{(\text{app } e_1 e_2) \text{ free-variable-error}} \text{FVerr-app-2}$$

$$\frac{e_1 \Downarrow (\text{lam } x e_B) \quad e_2 \Downarrow v_2 \quad \text{subst}(e_B, x, v_2) \text{ free-variable-error}}{(\text{app } e_1 e_2) \text{ free-variable-error}} \text{FVerr-app-3}$$

This is “straightforward” but tedious.  $\text{Eval-app-value}$  covers the situation where “everything works”, that is, when evaluating  $e_1$  gives a  $\text{lam}$ , when evaluating  $e_2$  gives a value, and when evaluating the body under substitution gives a value. The  $\text{FVerr-app-}$  rules have to deal with every possible point of failure:  $e_1$  can't be evaluated ( $\text{FVerr-app-1}$ ),  $e_1$  is fine but  $e_2$  can't be evaluated ( $\text{FVerr-app-2}$ ), or  $e_1$  and  $e_2$  are fine but  $\text{subst}(e_B, x, v_2)$  can't be evaluated.

■ **Exercise 5.** Write an  $\text{FVerr}$  rule for  $\text{rec}$ , and rules for  $\text{with}$ .

At this point (especially if you did the exercise), you might be thinking, “Curse Gentzen and his ‘rules’! I wish I could just raise an error and have it somehow propagate through the rules, like I can use error in  $\text{interp}$ !” Sadly, that isn't feasible. While 311 is skipping the mathematical foundations underlying rules and derivations, those foundations exist and are essential to being able to prove properties of rules, and I don't see how the foundations would survive trying to add exceptions *to the rules mechanism itself*.

Switching to a different semantics will help, though.

## 5 Small-step semantics

Forget about error handling for a moment:

When we use the method of hope to derive an evaluation judgment  $e \Downarrow v$ , we see expressions becoming “more evaluated”. Expressions are replaced with values; bound variables in lam and with get replaced with values. (If we used the expression strategy instead of the value strategy, the bound variables are replaced with expressions that aren’t necessarily values; still, at least they’re known expressions.)

$$\dots \frac{\frac{(\text{add } (\text{num } 1) (\text{num } 2)) \Downarrow \text{---} \quad (\text{num } 10) \Downarrow \text{---}}{(\text{sub } (\text{add } (\text{num } 1) (\text{num } 2)) (\text{num } 10)) \Downarrow \text{---}} \text{Eval-sub}}{(\text{app } (\text{lam } x (\text{sub } (\text{add } (\text{id } x) (\text{num } 2)) (\text{num } 10)) (\text{num } 1))) \Downarrow \text{---}} \text{Eval-app}}$$

Little steps of computation happen along the way, such as when we replace  $n_1$  and  $n_2$  in “ $n_1 + n_2$ ” and add the actual numbers together.

$$\dots \frac{\frac{(\text{add } (\text{num } 1) (\text{num } 2)) \Downarrow (\text{num } 3) \quad (\text{num } 10) \Downarrow \text{---}}{(\text{sub } (\text{add } (\text{num } 1) (\text{num } 2)) (\text{num } 10)) \Downarrow \text{---}} \text{Eval-sub}}{(\text{app } (\text{lam } x (\text{sub } (\text{add } (\text{id } x) (\text{num } 2)) (\text{num } 10)) (\text{num } 1))) \Downarrow \text{---}} \text{Eval-app}}$$

This suggests that instead of saying the meaning of an expression is what it evaluates to, we can think of the meaning of an expression as *the expression we get after doing “one step” of computation*. In this way, the meaning of  $(\text{sub } (\text{add } (\text{num } 1) (\text{num } 2)) (\text{num } 10))$  is

$$(\text{sub } (\text{num } 3) (\text{num } 10))$$

and the meaning of *that* expression is

$$(\text{num } -7)$$

This might seem tedious (and risking Zeno’s paradox?) but we’ll be able to recover the same notion of meaning we had with  $e \Downarrow v$  (and we won’t need a cavalcade of error rules).

The new judgment form will be

$$e1 \longrightarrow e2$$

and is read “ $e1$  steps to  $e2$ ”.

The rules for this judgment form will feel different from the Eval rules we used before. Rather than writing rules that recursively evaluate subexpressions, as in Eval-add which evaluates the subexpressions  $e1$  and  $e2$  before doing its actual work (adding  $n_1$  and  $n_2$ ), most of our rules will only work directly when their subexpressions are already values. Just one rule (backed by a BNF definition) will manage “finding” a subexpression to step.

First, let’s write the “basic” rules, sometimes called *reduction rules*. Think of these rules as where computation really happens. We show these in Figure 2.

The last rule in Figure 2 is not a reduction rule, and is quite concise but depends on another definition, which we’ll explain next: *evaluation contexts*.

## §5 Small-step semantics

---

$e1 \longrightarrow e2$  Expression  $e1$  steps to  $e2$

**Reduction rules:**

$$\frac{}{(\text{add } (\text{num } n_1) (\text{num } n_2)) \longrightarrow (\text{num } n_1 + n_2)} \text{ Step-add}$$

$$\frac{}{(\text{sub } (\text{num } n_1) (\text{num } n_2)) \longrightarrow (\text{num } n_1 - n_2)} \text{ Step-sub}$$

$$\frac{}{(\text{app } (\text{lam } x \text{ eB}) v) \longrightarrow \text{subst}(eB, x, v)} \text{ Step-app-value}$$

$$\frac{}{(\text{with } x \text{ v1 } e2) \longrightarrow \text{subst}(e2, x, v1)} \text{ Step-with} \quad \frac{}{(\text{rec } u \text{ e}) \longrightarrow \text{subst}(e, u, (\text{rec } u \text{ e}))} \text{ Step-rec}$$

**Context rule:**

$$\frac{e \longrightarrow e'}{\mathcal{C}[e] \longrightarrow \mathcal{C}[e']} \text{ Step-context}$$

---

### Figure 2 Small-step semantics

---

The idea is that if we are given an expression that contains a subexpression  $e$ , where  $e$  is “an expression we should step next”, and we can use one of the basic rules (reduction rules) to step  $e$  to  $e'$ , then the whole expression steps to  $\mathcal{C}[e']$ . The expression we get after stepping is the same as the one we started with, except for the part  $e$  that just “took a step”.

To define what  $\mathcal{C}$  means, we’ll use a BNF. We’ve been using BNFs to define the concrete syntax of languages, but BNFs are versatile and can also be used with *abstract* syntax. To (hopefully) clarify that this BNF is describing abstract syntax, not concrete syntax, I’ll follow the convention I’ve been using in the rules, where we write  $e$ ,  $v$ ,  $n$ , etc. rather than using angle brackets  $\langle E \rangle$  (as PLAI and the Algol 60 Report do for concrete syntax).

This is also an opportunity to define values  $v$  using a BNF:

$$\text{Values } v ::= (\text{num } n) \\ | (\text{lam } x \text{ e})$$

■ **Exercise 6.** Update this definition of values  $v$  to reflect Fun++ (the Assignment 2 language).

Now, the definition of evaluation contexts:



## §5 Small-step semantics

Evaluation contexts  $\mathcal{C} ::= []$

- | (add  $\mathcal{C}$   $e$ )
- | (add  $v$   $\mathcal{C}$ )
- | (sub  $\mathcal{C}$   $e$ )
- | (sub  $v$   $\mathcal{C}$ )
- | (app  $\mathcal{C}$   $e$ )
- | (app  $v$   $\mathcal{C}$ )
- | (with  $x$   $\mathcal{C}$   $e$ )

The empty brackets  $[]$  are called a “hole”. Some examples of evaluation contexts:

(add  $[]$  (app  $e3$   $e4$ ))  
(sub (num 5)  $[]$ )  
(app (app  $[]$   $e1$ )  $e2$ ))

In all of these, the hole  $[]$  appears in a position where we should try to step:

- to add, we need two values (numbers), so we should try to step the first subexpression
- to sub, if we have a value (num 5) as the first subexpression we should try to step the second subexpression
- to apply a function to an argument, where the function is itself a function application, we need to step inside that application

We also need some more notation: we want to use these contexts in our rule Step-context, but a context  $\mathcal{C}$  isn’t an expression because it always has a hole  $[]$  in it, and holes aren’t in our abstract syntax! (They’re also not in our concrete syntax, which is just as well.) So we’ll write

$\mathcal{C}[e]$

to mean the context  $\mathcal{C}$  with its hole replaced by  $e$ . This is best understood through examples, so:

if $\mathcal{C}$ is...	and $e$ is...	then $\mathcal{C}[e]$ is...
(add $[]$ (app $e3$ $e4$ ))	(sub $e1$ $e2$ )	(add (sub $e1$ $e2$ ) (app $e3$ $e4$ ))
(add $[]$ (app $e3$ $e4$ ))	(num 1)	(add (num 1) (app $e3$ $e4$ )) *
(sub (num 5) $[]$ )	(app (lam $x$ (id $x$ )) (num 2))	(sub (num 5) (app (lam $x$ (id $x$ )) (num 2)))
(app (app $[]$ $e1$ ) $e2$ ))	(add (lam ...) (lam ...))	(app (app (add (lam ...) (lam ...)) $e1$ ) $e2$ )) **

The second example (marked \*) illustrates that contexts  $\mathcal{C}$  don’t care whether the expression replacing the hole can actually step: (num 1) is a value, so it won’t step to anything. We wouldn’t be able to apply Step-context on this particular  $\mathcal{C}[e]$ , but the definition of  $\mathcal{C}$  doesn’t care. The last example (marked \*\*) also shows this: (add (lam ...) (lam ...)) won’t step, but it can still replace the hole. The definition of  $\mathcal{C}$  is all about finding a *position* within the expression;  $\mathcal{C}$  doesn’t care what’s at that position.

### 5.1 Error handling

Now for the thrilling conclusion: With a small-step semantics, the annoying error rules can be replaced with just one:

$$\overline{\mathcal{C}[(\text{id } x)]} \text{ free-variable-error} \quad \text{FVerr-context}$$

## 5.2 Relating small- and big-step semantics

You’ve seen small-step semantics now, so I can tell you that “evaluation semantics” is often called “big-step semantics”: to evaluate (add  $e_1$   $e_2$ ), evaluation semantics has  $e_1$  take a “big step” and then has  $e_2$  take a “big step”.

Have we accidentally defined a different language? Hopefully not. First we need a useful definition:

■ **Definition 7.** We write  $e \longrightarrow^* e'$  to mean that  $e$  takes 0 or more steps to  $e'$ . (That is, either  $e = e'$ , or  $e \longrightarrow e_2$  and  $e_2 \longrightarrow^* e'$ .)

If we’re really enjoying Gentzen’s notation, we can write this definition using rules:

$$\boxed{e \longrightarrow^* e'} \text{ Expression } e \text{ takes 0 or more steps to } e'$$

$$\frac{}{e \longrightarrow^* e} \text{ Steps-zero} \qquad \frac{e \longrightarrow e_2 \quad e_2 \longrightarrow e'}{e \longrightarrow^* e'} \text{ Steps-step}$$

■ **Exercise 8.** Write rules deriving  $e \longrightarrow^+ e'$  such that  $e \longrightarrow^+ e'$  if and only if  $e$  takes *one* or more steps to  $e'$ . You can use other judgment forms, including  $\longrightarrow^*$ , as premises.

Now the following should hold:

- (1) If  $e \Downarrow v$  then  $e \longrightarrow^* v$ .
- (2) If  $e \longrightarrow^* v$  then  $e \Downarrow v$ .

Proving these is outside the scope of 311, but they should hold unless I’ve made a mistake...

A consequence of (1) and (2) is that, ignoring the whole issue of error handling, we can write an interpreter either by following the Eval rules or by following the Step rules. In the former case, we can look at the interpreter code and (hopefully) see that it directly implements the Eval rules, and then appeal to (1) and (2) to show that our interpreter (indirectly) follows the Step rules. In the latter case, we have to write an interpreter (maybe we should call it a stepper?) that directly implements the Step rules, and again appeal to (1) and (2), to show that we have (indirectly) implemented the Eval rules.

Both small- and big-step semantics are used to define programming languages, but small-step has some important advantages, (partly) explained below. (Big-step is usually considered easier to understand, which is one of the reasons I presented it first.)

5.3 Small-step semantics and recursion

As with big-step semantics ( $e \Downarrow v$ ), attempting to step  $(\text{rec } u \text{ (id } u))$  won't get us anywhere useful. However, small-step "fails" less catastrophically. In big-step, we attempted to construct an infinite derivation tree; in small-step, we can apply rule Step-rec to derive a perfectly good judgment:

$$\frac{}{(\text{rec } u \text{ (id } u)) \longrightarrow (\text{rec } u \text{ (id } u))} \text{ Step-rec}$$

If we keep stepping, we won't get anywhere. If we enjoy Gentzen's notation and regard the Steps-zero and Steps-step rules as the definition of "stepping 0 or more times", we will again attempt to construct an infinite derivation tree.

Being able to talk about taking *one* step is useful. For example, an *information-flow type system* can prove that a given program will never leak secret information (e.g. by printing a cleartext password). For a big-step semantics, this property cannot be stated easily: we can certainly model what evaluation prints, through a judgment

$$e \Downarrow v \text{ prints } s$$

read "e evaluates to value v and prints the string s". But we can't prove the following statement:

For all e such that e passes the information-flow type system,  
 either  $e \Downarrow v$  prints s where s contains no secret information,  
 or e free-variable-error.

We can't prove this because evaluating e might *diverge*, for example, if e is  $(\text{rec } u \text{ (id } u))$ . We would instead have to prove the following (using "e  $\Uparrow$ " to mean e diverges):

For all e such that e passes the information-flow type system,  
 either  $e \Downarrow v$  prints s where s contains no secret information,  
 or e free-variable-error,  
 or  $e \Uparrow$  but e has not yet printed any secret information.

But what does "e has not yet printed any secret information" mean? The judgment  $e \Downarrow v$  prints s is supposed to define what expressions should print which strings, but it only makes sense if e has finished and returned a value.

In small-step semantics, we still have a notion of diverging expressions: e diverges if there exists no value v such that  $e \longrightarrow^* v$ . But we can easily talk about the steps we take as we go:

For all e such that e passes the information-flow type system,  
 either e is a value,  
 or e free-variable-error,  
 or  $e \longrightarrow e'$  prints s where s contains no secret information.

We could then show (by induction on the number of steps) that for any number of steps, no secret information will be printed.

Another argument for small-step semantics is that, while we would like Fun(++) programs to terminate and return a value, not all programs should terminate. An operating system kernel<sup>1</sup> or

<sup>1</sup>I have an ancient Toshiba laptop that stayed up for *over three years*, running OpenBSD. I had to turn it off when I moved back to Canada; the flights here were a little longer than its battery life.

## §5 Small-step semantics

---

web server should, in principle, run forever. We still want to know that the web server will never send sensitive information in the clear (cf. Heartbleed), which is analogous to our information-flow example above.