CPSC 311: Definition of Programming Languages: Evaluation semantics: Functions (“06-functions”)

Joshua Dunfield
University of British Columbia

September 23, 2015

1 Topics discussed

- Recipe for extending a language
- The “Fun language”: concrete syntax, abstract syntax
- The Fun language: evaluation rules
  - why evaluation in Fun produces a *value*, not just a number
  - evaluation rule for `lam`
  - updated evaluation rules for features already in WAE
  - rule for function application?

  (**the lecture on Monday, 2015/09/21 ended around this point**)
- rule for function application: Eval-app-expr
- the “expression strategy”
- example with Eval-app-expr
- the “method of hope”; complete derivations
- another example with Eval-app-expr
- the “value strategy” and Eval-app-value
- advantages and disadvantages of each strategy

(**the lecture on Wednesday, 2015/09/23 ended here)**

2 Functions

We started our journey through evaluation semantics with the AE language. We got a slightly larger language, WAE, by adding the with construct, which lets us give names to values and then use those names. To model with in the evaluation semantics, we had to define substitution.

Adding just one more feature will turn WAE into a surprisingly powerful language.

To add functions, we’ll try to follow the same recipe as for with:

1. **Extend the concrete syntax** (EBNF grammar).
2. **Extend the abstract syntax** (**define-type**).
3. **Add evaluation rules.**

After finishing these steps, we will have extended our *language*: a language is defined by its syntax and semantics. To implement the language, we'll need to extend our parsing function (to reflect the new syntax) and our interpreter (to reflect the new evaluation rules).

Attention! From this point on, and until further notice, **any similarity to the PLAI book will be coincidental**. Hold on tight!

Remark. In real life, or at least in real programming languages research, there would probably be a fourth step: **Prove that the new evaluation rules have good properties.** Exactly what properties are "good" depends on the language. For the AE and WAE languages, one good property (the only one I can think of right now) would be *determinism*, which says that evaluating the same expression should give consistent results:

$$\text{"If } e \Downarrow n_1 \text{ and } e \Downarrow n_2 \text{, then } n_1 = n_2 \text{."}$$

Later in 311, we'll discuss these sorts of properties (without actually proving them).

3 The Fun language: syntax

It's time to add functions to the WAE language. Nearly a century of tradition would have us use the syntax "lambda" or λ, but to help distinguish the lambda in Fun from the lambda in Racket (which can also be written λ), we'll use "lam".

Instances of the identifier bound by `lam` can be represented by `id`, just as we did for `with`.

$$\begin{aligned} \langle \mathrm{E} \rangle \ ::= \ & \langle \mathrm{num} \rangle \\ & \mid \{+ \ \langle \mathrm{E} \rangle \ \langle \mathrm{E} \rangle\} \\ & \mid \{- \ \langle \mathrm{E} \rangle \ \langle \mathrm{E} \rangle\} \\ & \mid \{\mathrm{with} \ \{\langle \mathrm{id} \rangle \ \langle \mathrm{E} \rangle\} \ \langle \mathrm{E} \rangle\} \\ & \mid \langle \mathrm{id} \rangle \\ & \mid \{\mathrm{lam} \ \langle \mathrm{id} \rangle \ \langle \mathrm{E} \rangle\} \end{aligned}$$

And here's the abstract syntax:

```
(define-type E
  [num (n number?)]
  [add (lhs E?) (rhs E?)]
  [sub (lhs E?) (rhs E?)]
  [with (name symbol?) (named-expr E?) (body E?)]
  [id (name symbol?)]

  [lam (name symbol?) (body E?)]
)
```

We're not done with the syntax, though. In a previous lecture, I mentioned that a language can be organized by what kinds of data it has, and how it "introduces" and "eliminates" each kind of data. The AE and WAE languages only had one kind of data, numbers; functions are a new kind of data. We can introduce functions with "lam", but we need a way to use them. We'll call this "app". The first expression will represent the function being applied, and the second expression will represent the argument—what the function is being applied to.

$$\begin{aligned} \langle \mathrm{E} \rangle \ ::= \ & \langle \mathrm{num} \rangle \\ & \mid \{+ \ \langle \mathrm{E} \rangle \ \langle \mathrm{E} \rangle\} \\ & \mid \{- \ \langle \mathrm{E} \rangle \ \langle \mathrm{E} \rangle\} \\ & \mid \{\mathrm{with} \ \{\langle \mathrm{id} \rangle \ \langle \mathrm{E} \rangle\} \ \langle \mathrm{E} \rangle\} \\ & \mid \langle \mathrm{id} \rangle \\ & \mid \{\mathrm{lam} \ \langle \mathrm{id} \rangle \ \langle \mathrm{E} \rangle\} \\ & \mid \{\mathrm{app} \ \langle \mathrm{E} \rangle \ \langle \mathrm{E} \rangle\} \end{aligned}$$

Here's the abstract syntax with both functions `lam` and function application `app`:

```
(define-type E
  [num (n number?)]
  [add (lhs E?) (rhs E?)]
  [sub (lhs E?) (rhs E?)]
  [with (name symbol?) (named-expr E?) (body E?)]
  [id (name symbol?)]

  [lam (name symbol?) (body E?)]
  [app (function E?) (argument E?)]
)
```

4 The Fun language: Evaluation rules

We'll try to reuse all the rules from the WAE language. Because we don't know yet whether that will work, we'll write a ? before each rule name to emphasize its provisional status.

When we write a meta-variable e in a rule, it will stand for a Fun expression rather than a WAE expression.

$$\frac{}{(\texttt{num}\ n) \Downarrow n}\text{?Eval-num} \qquad \frac{e1 \Downarrow n_1 \qquad e2 \Downarrow n_2}{(\texttt{add}\ e1\ e2) \Downarrow n_1 + n_2}\text{?Eval-add} \qquad \frac{e1 \Downarrow n_1 \qquad e2 \Downarrow n_2}{(\texttt{sub}\ e1\ e2) \Downarrow n_1 - n_2}\text{?Eval-sub}$$

$$\frac{e1 \Downarrow v1 \qquad \mathit{subst}(e2, x, (\texttt{num}\ v1)) \Downarrow v2}{(\texttt{with}\ x\ e1\ e2) \Downarrow v2}\text{?Eval-with} \qquad \frac{}{(\texttt{id}\ x)\ \text{free-variable-error}}\text{?Eval-free-identifier}$$

We added two productions to the EBNF and two variants to the abstract syntax, so we expect to need two new evaluation rules. We might also expect (following the pattern of Eval-add, Eval-sub, and Eval-with) that, since a `lam` has one expression inside it and an `app` has two expressions inside it, the rule for Eval-lam will have one premise and the rule for Eval-app will have two premises:

$$\frac{??}{(\texttt{lam}\ x\ e1) \Downarrow ??}\text{??Eval-lam} \qquad \frac{?? \qquad ??}{(\texttt{app}\ e1\ e2) \Downarrow ??}\text{??Eval-app}$$

Let's think about ??Eval-lam. What should its premise be? The only expression we have is $e1$, so maybe we should evaluate $e1$ in the premise.

$$\frac{e1 \Downarrow v1}{(\texttt{lam}\ x\ e1) \Downarrow v1??}\text{??Eval-lam}$$

This seems to follow the pattern of our other rules, but (as with programming) we should think about examples (test cases). What is the *simplest possible* function? I claim it is the identity function, $(\texttt{lam}\ x\ (\texttt{id}\ x))$. So let's try that function with our proposed rule:

$$\frac{(\texttt{id}\ x) \Downarrow \_\_\_}{(\texttt{lam}\ x\ (\texttt{id}\ x)) \Downarrow \_\_\_}$$

Now we have a problem: the expression $(\texttt{id}\ x)$ doesn't evaluate to anything—instead, we get a "free-variable-error". So we can't derive $(\texttt{id}\ x) \Downarrow \_\_\_$.

The problem is that we're trying to evaluate what's *inside* the function before we've applied it *to* anything! We don't know what x is until we pass the function an argument. A function is a machine that turns arguments into results; evaluating a function without an argument is like running a dishwasher when it's empty.

We already have numbers that evaluate to themselves (Eval-num), so we'll make functions evaluate to themselves as well.

$$\frac{}{(\texttt{lam}\ x\ e1) \Downarrow (\texttt{lam}\ x\ e1)}\ \text{Eval-lam}$$

Irritatingly, this doesn't quite match what we've done so far. Instead of always evaluating to numbers—deriving

$$\ldots \Downarrow n$$

where n is a number, we can now use Eval-lam to derive

$$\ldots \Downarrow (\texttt{lam}\ x\ e1)$$

We could say that evaluation produces either a number n or an expression of the form $(\texttt{lam}\ x\ e1)$. It will be a little easier to say that evaluation produces a *value*, where a value is a particular kind of expression: one that is either $(\texttt{num}\ n)$, or $(\texttt{lam}\ x\ e1)$.[1]

Making this change requires several changes to the rules from the WAE language:

$$\frac{}{(\texttt{num}\ n) \Downarrow (\texttt{num}\ n)}\ \text{Eval-num} \qquad \frac{e1 \Downarrow (\texttt{num}\ n_1) \qquad e2 \Downarrow (\texttt{num}\ n_2)}{(\texttt{add}\ e1\ e2) \Downarrow (\texttt{num}\ (n_1 + n_2))}\ \text{Eval-add} \qquad \frac{e1 \Downarrow (\texttt{num}\ n_1) \qquad e2 \Downarrow (\texttt{num}\ n_2)}{(\texttt{sub}\ e1\ e2) \Downarrow (\texttt{num}\ (n_1 - n_2))}\ \text{Eval-sub}$$

$$\frac{e1 \Downarrow v1 \qquad \mathit{subst}(e2, x, (\cancel{\texttt{num}}\ v1)) \Downarrow v2}{(\texttt{with}\ x\ e1\ e2) \Downarrow v2}\ \text{Eval-with} \qquad \frac{}{(\texttt{id}\ x)\ \textsf{free-variable-error}}\ \text{Eval-free-identifier}$$

$$\frac{}{(\texttt{lam}\ x\ e1) \Downarrow (\texttt{lam}\ x\ e1)}\ \text{Eval-lam}$$

Interestingly, one of the rules—Eval-with—became simpler, and more general: it should now work with any kind of value $v1$, not just numbers.

Figure 1 summarizes all our rules so far—we're still missing a rule for app.

$$\frac{}{(\texttt{num}\ n) \Downarrow (\texttt{num}\ n)}\ \text{Eval-num} \qquad \frac{e1 \Downarrow (\texttt{num}\ n_1) \qquad e2 \Downarrow (\texttt{num}\ n_2)}{(\texttt{add}\ e1\ e2) \Downarrow (\texttt{num}\ (n_1 + n_2))}\ \text{Eval-add} \qquad \frac{e1 \Downarrow (\texttt{num}\ n_1) \qquad e2 \Downarrow (\texttt{num}\ n_2)}{(\texttt{sub}\ e1\ e2) \Downarrow (\texttt{num}\ (n_1 - n_2))}\ \text{Eval-sub}$$

$$\frac{e1 \Downarrow v1 \qquad \mathit{subst}(e2, x, v1) \Downarrow v2}{(\texttt{with}\ x\ e1\ e2) \Downarrow v2}\ \text{Eval-with} \qquad \frac{}{(\texttt{id}\ x)\ \textsf{free-variable-error}}\ \text{Eval-free-identifier}$$

$$\frac{}{(\texttt{lam}\ x\ e1) \Downarrow (\texttt{lam}\ x\ e1)}\ \text{Eval-lam}$$

Figure 1 Evaluation rules for Fun (still missing a rule for app)

4.1 Evaluating application

Here's our guess at the shape of the Eval-app rule, with a lot of ??'s.

$$\frac{?? \qquad ??}{(\texttt{app}\ e1\ e2) \Downarrow\ ??}\ ??\text{Eval-app}$$

[1]By themselves, values are inert. They don't do anything. A function does nothing until it's called. Unfortunately, many "real" programming languages make this hard to remember. For example, DrRacket claims that the result of evaluating `(lambda (x) x)` is "`#<procedure>`". But you should think of that as just a strange notation for the function you entered. It's still `(lambda (x) x)`.

Ideas?

At this point, I was expecting to (eventually) reach a particular rule, but it's not the one that you suggested during lecture. (I should have expected this, since I suggested that we have a rule with two premises, and the one I was thinking of has three premises!) However, the rule you suggested is quite reasonable.

The first suggestion was to evaluate `e1` to a `lam`. Why does it have to be a `lam`? Well, evaluation produces a value. We have two kinds of values so far: numbers and `lam`s. Applying a number as a function doesn't make much sense, so it's got to be a `lam`. Also, requiring `e1` to evaluate to a `lam` corresponds to Eval-add, where the expressions being added have to evaluate to `num`s.

$$\frac{e1 \Downarrow (\texttt{lam}\ x\ eB) \qquad ??}{(\text{app}\ e1\ e2) \Downarrow ??}\ \text{?Eval-app}$$

I can't think of any other way of writing this first premise. This is good progress, so I dropped one of the ?'s from the name of the rule.

Now we come to a "fork in the road". The choice we make now will decide what *evaluation strategy* this language has. This is usually considered an important and fundamental language design choice, especially for functional languages (like Racket and this Fun language).

4.1.1 The "expression strategy"

The next suggestion during lecture was to substitute $e2$ for x, like this:[2]

$$\frac{e1 \Downarrow (\texttt{lam}\ x\ eB) \qquad \mathit{subst}(eB, x, e2) \Downarrow v}{(\text{app}\ e1\ e2) \Downarrow v}\ \text{Eval-app-expr}$$

We'll call this the *expression strategy*[3], because we're taking the expression $e2$—the argument being passed to $(\texttt{lam}\ x\ eB)$—and substituting it for x, *without* evaluating $e2$.

Let's look at some examples.

- **Evaluating an application of the identity function.**

  Suppose we apply the identity function $(\texttt{lam}\ x\ (\texttt{id}\ x))$ to some simple expression, like $(\texttt{add}\ (\texttt{num}\ 2)\ (\texttt{num}\ 3))$.

  (It would be even simpler to apply the identity function to a `num`, but that wouldn't illustrate what I'm trying to illustrate.)

  So we need to derive

  $$\big(\text{app}\ (\texttt{lam}\ x\ (\texttt{id}\ x))\ (\texttt{add}\ (\texttt{num}\ 2)\ (\texttt{num}\ 3))\big) \Downarrow \cdots$$

  First we "match" expressions to meta-variables in Eval-app-expr:

  $$\big(\text{app}\ \underbrace{(\texttt{lam}\ x\ (\texttt{id}\ x))}_{e1}\ \underbrace{(\texttt{add}\ (\texttt{num}\ 2)\ (\texttt{num}\ 3))}_{e2}\big) \Downarrow \cdots$$

  Then we plug in those expressions:

  $$\frac{(\texttt{lam}\ x\ (\texttt{id}\ x)) \Downarrow (\texttt{lam}\ x\ eB) \qquad \mathit{subst}(eB, x, (\texttt{add}\ (\texttt{num}\ 2)\ (\texttt{num}\ 3))) \Downarrow v}{\big(\text{app}\ (\texttt{lam}\ x\ (\texttt{id}\ x))\ (\texttt{add}\ (\texttt{num}\ 2)\ (\texttt{num}\ 3))\big) \Downarrow v}\ \text{Eval-app-expr}$$

  We need to be careful about Gentzen's notation. We *want* to apply the rule Eval-app-expr, but we haven't really applied it yet, because we haven't derived its two premises. A rule is an "if... then

[2]During lecture, I called the result of evaluation $v2$ rather than v. It will be less confusing to call it v.
[3]There is a more traditional name, which we'll talk about in due course.

...” statement; you don’t get to conclude the “then” part without showing that the “if” holds. What I wrote above is based on the power of hope: I *want* to derive $(\mathtt{app}\ \cdots\ \cdots) \Downarrow v$, so I’m going to *try* to apply Eval-app-expr. To apply Eval-app-expr, I need to derive each premise, using the same method of hope.

At each step in this process, any leaf in my derivation tree that doesn’t have a horizontal line over it is a *goal* that remains to be proved.[4]

If I can get a derivation tree whose leaves all have horizontal lines over them, I will know that I have derived $(\mathtt{app}\ \cdots\ \cdots) \Downarrow v$, but not before.

We now want to derive

$$(\mathtt{lam}\ x\ (\mathtt{id}\ x)) \Downarrow (\mathtt{lam}\ x\ eB) \qquad \text{(first goal)}$$

and

$$\mathit{subst}(eB, x, (\mathtt{add}\ (\mathtt{num}\ 2)\ (\mathtt{num}\ 3))) \Downarrow v \qquad \text{(second goal)}$$

For the first evaluation, we have a rule that evaluates `lam`s: Eval-lam. Plugging in for the meta-variable $e1$ in that rule, we get

$$\frac{}{(\mathtt{lam}\ x\ (\mathtt{id}\ x)) \Downarrow (\mathtt{lam}\ x\ \underbrace{(\mathtt{id}\ x)}_{eB})}$$

Note that this tells us what eB is, so we can revise our second goal by plugging in $(\mathtt{id}\ x)$ for eB:

$$\mathit{subst}((\mathtt{id}\ x), x, (\mathtt{add}\ (\mathtt{num}\ 2)\ (\mathtt{num}\ 3))) \Downarrow v \qquad \text{(second goal, revised)}$$

Since *subst* is now applied to “real” arguments (without unknown meta-variables), we can use the definition of *subst*. We really should revise the definition of *subst* to our extended language, but the old version works for this example: replacing all instances of the identifier x in $(\mathtt{id}\ x)$ with $(\mathtt{add}\ (\mathtt{num}\ 2)\ (\mathtt{num}\ 3))$ is just $(\mathtt{add}\ (\mathtt{num}\ 2)\ (\mathtt{num}\ 3))$.

$$(\mathtt{add}\ (\mathtt{num}\ 2)\ (\mathtt{num}\ 3)) \Downarrow v \qquad \text{(second goal, revised again)}$$

■ **Exercise 1.** Derive this second goal, following the “method of hope” as above. (This is *not* exactly like a similar example for the AE language, because we changed our notion of evaluation to produce expressions—more specifically, values—rather than numbers.) I left some space above for you to write the derivation tree.

I’ll assume you’ve derived the second goal, and to keep track, I’ll put a checkmark ✓ above it. I’m also assuming you got $(\mathtt{num}\ 5)$, so I’m plugging that in for the meta-variable v.

$$\frac{\dfrac{}{(\mathtt{lam}\ x\ (\mathtt{id}\ x)) \Downarrow (\mathtt{lam}\ x\ (\mathtt{id}\ x))}\text{Eval-lam} \qquad \overset{\checkmark}{(\mathtt{add}\ (\mathtt{num}\ 2)\ (\mathtt{num}\ 3)) \Downarrow (\mathtt{num}\ 5)}}{(\mathtt{app}\ (\mathtt{lam}\ x\ (\mathtt{id}\ x))\ (\mathtt{add}\ (\mathtt{num}\ 2)\ (\mathtt{num}\ 3))) \Downarrow (\mathtt{num}\ 5)}\text{Eval-app-expr}$$

Everything in this tree has either a horizontal line or a checkmark, so we have a complete evaluation derivation.

[4]If you have used a logic programming language, such as Prolog (taught in CPSC 312), yes, there is a connection; we may not have time to explore it in 311, though.

Remember: Following the method of hope, a derivation tree is **complete** if each leaf of the tree is either (1) an application of a rule with no premises (for example, Eval-num), or (2) a conclusion of some other complete derivation tree.

You can tell (1) because the leaf has a horizontal line with nothing above it.

To keep track of (2), write a checkmark above the leaf.

If a leaf doesn't have a horizontal line or a checkmark, that leaf is a goal that needs to be be derived, and the derivation is **incomplete**.

- **Evaluating an application of the doubling function.**

  In Fun, we can write a function that doubles its argument:

$$(\texttt{lam}\ x\ (\texttt{add}\ (\texttt{id}\ x)\ (\texttt{id}\ x)))$$

  What happens when we apply this function to $(\texttt{add}\ (\texttt{num}\ 2)\ (\texttt{num}\ 3))$? Hopefully, we will get 10, since $2 \cdot (2+3) = 10$. But remember that we are evaluating to expressions now, so we actually want to evaluate to $(\texttt{num}\ 10)$.

$$\dfrac{\dfrac{}{\ldots \Downarrow (\texttt{lam}\ x\ (\texttt{add}\ (\texttt{id}\ x)\ (\texttt{id}\ x)))}\text{Eval-lam} \qquad \mathit{subst}((\texttt{add}\ (\texttt{id}\ x)\ (\texttt{id}\ x)), x, (\texttt{add}\ (\texttt{num}\ 2)\ (\texttt{num}\ 3))) \Downarrow v}{(\texttt{app}\ (\texttt{lam}\ x\ (\texttt{add}\ (\texttt{id}\ x)\ (\texttt{id}\ x)))\ (\texttt{add}\ (\texttt{num}\ 2)\ (\texttt{num}\ 3))) \Downarrow v}\text{Eval-app-expr}$$

  To save space, I wrote "…", but since Eval-lam has the same thing on both sides of the "$\Downarrow$", it has to be $(\texttt{lam}\ x\ (\texttt{add}\ (\texttt{id}\ x)\ (\texttt{id}\ x)))$.

  The second premise of Eval-app-expr has neither a horizontal line above nor a checkmark, so we have an incomplete derivation. To figure out which rule to try, we need to know what expression we have, so we look up the definition of *subst*. That gives:

$$\dfrac{\dfrac{}{\ldots \Downarrow (\texttt{lam}\ x\ (\texttt{add}\ (\texttt{id}\ x)\ (\texttt{id}\ x)))}\text{Eval-lam} \qquad (\texttt{add}\ (\texttt{add}\ (\texttt{num}\ 2)\ (\texttt{num}\ 3))\ (\texttt{add}\ (\texttt{num}\ 2)\ (\texttt{num}\ 3))) \Downarrow v}{(\texttt{app}\ (\texttt{lam}\ x\ (\texttt{add}\ (\texttt{id}\ x)\ (\texttt{id}\ x)))\ (\texttt{add}\ (\texttt{num}\ 2)\ (\texttt{num}\ 3))) \Downarrow v}\text{Eval-app-expr}$$

  We now know exactly what expression we have, so we can try to apply a rule. We have an add, so we'll try Eval-add.

$$\dfrac{\dfrac{}{\ldots \Downarrow (\texttt{lam}\ x\ (\texttt{add}\ (\texttt{id}\ x)\ (\texttt{id}\ x)))}\text{Eval-lam} \qquad \dfrac{(\texttt{add}\ (\texttt{num}\ 2)\ (\texttt{num}\ 3)) \Downarrow (\texttt{num}\ \underline{\quad}) \qquad (\texttt{add}\ (\texttt{num}\ 2)\ (\texttt{num}\ 3)) \Downarrow (\texttt{num}\ \underline{\quad})}{(\texttt{add}\ (\texttt{add}\ (\texttt{num}\ 2)\ (\texttt{num}\ 3))\ (\texttt{add}\ (\texttt{num}\ 2)\ (\texttt{num}\ 3))) \Downarrow (\texttt{num}\ \underline{\quad}+\underline{\quad})}\text{Eval-add}}{(\texttt{app}\ (\texttt{lam}\ x\ (\texttt{add}\ (\texttt{id}\ x)\ (\texttt{id}\ x)))\ (\texttt{add}\ (\texttt{num}\ 2)\ (\texttt{num}\ 3))) \Downarrow (\texttt{num}\ \underline{\quad}+\underline{\quad})}\text{Eval-app-expr}$$

  Conveniently, you already did a complete derivation for $(\texttt{add}\ (\texttt{num}\ 2)\ (\texttt{num}\ 3))$ and found that this expression evaluated to $(\texttt{num}\ 5)$, so we can fill in all of the blanks with 5.

$$\dfrac{\dfrac{}{\ldots \Downarrow (\texttt{lam}\ x\ (\texttt{add}\ (\texttt{id}\ x)\ (\texttt{id}\ x)))}\text{Eval-lam} \qquad \dfrac{\overset{\checkmark}{(\texttt{add}\ (\texttt{num}\ 2)\ (\texttt{num}\ 3)) \Downarrow (\texttt{num}\ 5)} \qquad \overset{\checkmark}{(\texttt{add}\ (\texttt{num}\ 2)\ (\texttt{num}\ 3)) \Downarrow (\texttt{num}\ 5)}}{(\texttt{add}\ (\texttt{add}\ (\texttt{num}\ 2)\ (\texttt{num}\ 3))\ (\texttt{add}\ (\texttt{num}\ 2)\ (\texttt{num}\ 3))) \Downarrow (\texttt{num}\ 5+5)}\text{Eval-add}}{(\texttt{app}\ (\texttt{lam}\ x\ (\texttt{add}\ (\texttt{id}\ x)\ (\texttt{id}\ x)))\ (\texttt{add}\ (\texttt{num}\ 2)\ (\texttt{num}\ 3))) \Downarrow (\texttt{num}\ 5+5)}\text{Eval-app-expr}$$

  Finally, we replace $5+5$ with 10 in the derivation tree.

$$\dfrac{\dfrac{}{\ldots \Downarrow (\texttt{lam}\ x\ (\texttt{add}\ (\texttt{id}\ x)\ (\texttt{id}\ x)))}\text{Eval-lam} \qquad \dfrac{\overset{\checkmark}{(\texttt{add}\ (\texttt{num}\ 2)\ (\texttt{num}\ 3)) \Downarrow (\texttt{num}\ 5)} \qquad \overset{\checkmark}{(\texttt{add}\ (\texttt{num}\ 2)\ (\texttt{num}\ 3)) \Downarrow (\texttt{num}\ 5)}}{(\texttt{add}\ (\texttt{add}\ (\texttt{num}\ 2)\ (\texttt{num}\ 3))\ (\texttt{add}\ (\texttt{num}\ 2)\ (\texttt{num}\ 3))) \Downarrow (\texttt{num}\ 10)}\text{Eval-add}}{(\texttt{app}\ (\texttt{lam}\ x\ (\texttt{add}\ (\texttt{id}\ x)\ (\texttt{id}\ x)))\ (\texttt{add}\ (\texttt{num}\ 2)\ (\texttt{num}\ 3))) \Downarrow (\texttt{num}\ 10)}\text{Eval-app-expr}$$

Notice that in last example, we ended up with *two* copies of the same evaluation derivation (the one you did as an exercise). This is because we had two instances of x in the body of the function being applied, and our rule Eval-app-expr substitutes the entire expression $\big(\texttt{add}\ (\texttt{num}\ 2)\ (\texttt{num}\ 3)\big)$.

Does that really matter, except for making the derivation tree bigger? Well, maybe. If the evaluation semantics seems to be doing work twice, an interpreter based on the semantics will *also* do that work twice! That doesn't matter much for this small example, but imagine if, instead of the argument being $\big(\texttt{add}\ (\texttt{num}\ 2)\ (\texttt{num}\ 3)\big)$, it were some expression that computed the sum of the first million prime numbers. Our interpreter would evaluate that huge expression twice, even though it only appears once in the input expression $\big(\texttt{app}\ \cdots\ \cdots\big)$.

4.2 The "value strategy"

An alternative strategy, which we'll call the *value strategy*, is to have this rule instead of Eval-app-expr:

$$\dfrac{e1 \Downarrow (\texttt{lam}\ x\ eB) \qquad e2 \Downarrow v2 \qquad \mathit{subst}(eB, x, v2) \Downarrow v}{(\texttt{app}\ e1\ e2) \Downarrow v}\ \text{Eval-app-value}$$

The first premise is the same as Eval-app-expr, but a new second premise evaluates $e2$ *immediately*, and in a third premise, we substitute the *result* of that evaluation for x.

We can see the difference from Eval-app-expr by evaluating the same Fun expression we evaluated just above, the doubling function applied to $(\texttt{add}\ (\texttt{num}\ 2)\ (\texttt{num}\ 3))$. The expression still evaluates to $(\texttt{num}\ 10)$, but the derivation looks rather different:

$$\dfrac{\dfrac{}{\ldots \Downarrow \big(\texttt{lam}\,x\,(\texttt{add}\,(\texttt{id}\,x)\,(\texttt{id}\,x))\big)}\text{Eval-lam} \qquad \overset{\checkmark}{(\texttt{add}\ (\texttt{num}\ 2)\ (\texttt{num}\ 3)) \Downarrow \overbrace{(\texttt{num}\ 5)}^{v2}} \qquad \big(\texttt{add}\ (\texttt{num}\ 5)\ (\texttt{num}\ 5)\big) \Downarrow (\texttt{num}\ 10)}{\big(\texttt{app}\ \big(\texttt{lam}\,x\,(\texttt{add}\,(\texttt{id}\,x)\,(\texttt{id}\,x))\big)\ \big(\texttt{add}\ (\texttt{num}\ 2)\ (\texttt{num}\ 3)\big)\big) \Downarrow (\texttt{num}\ 10)}\ \text{Eval-app-value}$$

4.3 Advantages and disadvantages

Think about the "expression strategy" and the "value strategy". The value strategy gave us a smaller derivation for one of our examples—and would also give us a faster interpreter for that example.

- Both strategies seem to be giving us the same answers—evaluation is giving us the same values. Is that true in general?
  - There are different ways to read "in general". For our language[5], we will indeed get the same value regardless of which strategy we use, for any expression. (Caveat: I haven't proved this.) The size of the derivations, and the amount of time the interpreter will take, may be very different, but we'll get the same value (either a `num` or a `lam`).
  - If we mean languages generally, there are languages where this doesn't hold. In a language with *effects*, the argument $e2$ might do something that allows us to distinguish the two evaluation strategies. For example, $e2$ might print a string, and a different number of strings would be printed depending on the evaluation strategy. The values returned wouldn't change, however: either you print once, and return a value, or you print twice and return the same value.

    We *could* get different values, however, if our language had *mutable state*, such as an incrementable counter. If $e2$ increments this counter, the contents of the counter might affect the value returned.

    Languages with mutable state have more complicated semantics, which we'll look at later on.

[5]Or rather, our languages: a language is syntax and semantics together, so we should really talk about two languages: the "Fun with Eval-app-expr" language, and the "Fun with Eval-app-value" language.

- Are there any Fun expressions where the *expression* strategy (Eval-app-expr) would be faster?

  - Yes—expressions that apply a function that doesn't use its argument:

$$\dfrac{\dfrac{}{(\texttt{lam}\ x\ (\texttt{num}\ 4)) \Downarrow (\texttt{lam}\ x\ (\texttt{num}\ 4))}\,\text{Eval-lam} \qquad \mathit{subst}\big((\texttt{num}\ 4), x, (\texttt{add}\ (\texttt{num}\ 2)\ (\texttt{num}\ 3))\big) \Downarrow v}{(\texttt{app}\ (\texttt{lam}\ x\ (\texttt{num}\ 4))\ (\texttt{add}\ (\texttt{num}\ 2)\ (\texttt{num}\ 3))) \Downarrow v}\,\text{Eval-app-expr}$$

    Since x doesn't occur in $(\texttt{num}\ 4)$, the result of *subst* on $(\texttt{num}\ 4)$ is just $(\texttt{num}\ 4)$:

$$\dfrac{\dfrac{}{(\texttt{lam}\ x\ (\texttt{num}\ 4)) \Downarrow (\texttt{lam}\ x\ (\texttt{num}\ 4))}\,\text{Eval-lam} \qquad (\texttt{num}\ 4) \Downarrow v}{(\texttt{app}\ (\texttt{lam}\ x\ (\texttt{num}\ 4))\ \underbrace{(\texttt{add}\ (\texttt{num}\ 2)\ (\texttt{num}\ 3))}_{e2}) \Downarrow v}\,\text{Eval-app-expr}$$

    Here, we never evaluate the argument $e2$ at all! The value strategy would evaluate $e2$ even though it's not needed.

  - There's another evaluation strategy, *lazy evaluation*, which doesn't evaluate the argument $e2$ until it's used inside the `lam`. At that time, it evaluates $e2$ and remembers the value it gets. Other instances of x will reuse that value instead of evaluating $e2$ again. This strategy is a little more complicated to define, but we'll come back to it later in 311.

- Does evaluation in Racket work like the expression strategy, or like the value strategy? How about Java? Haskell? Algol-60?

  - What Racket does isn't exactly the same as either of our evaluation rules, but it's very close to the value strategy.
  - Java: same answer. (In Java, and similar languages, you often pass *pointers* or *references* around, but those are really just values, albeit of a different kind than the values in Fun.)
  - Haskell uses lazy evaluation (see above), so it's kind of like the expression strategy.
  - Algol-60 supports *both* the value strategy and the expression strategy. The expression strategy is the default, but programmers can designate specific function arguments as following the value strategy. (The Algol-60 committee had *just invented the expression strategy*. Years later, several committee members were still angry that the report's editor, Peter Naur—also the 'N' in 'BNF'—decided, on his own, to make the expression strategy be the default.)