CPSC 311: Recursion **(DRAFT)**
(“lec-recursion”)

Joshua Dunfield
University of British Columbia

September 27, 2016

1 Recursion

A reasonable objection to our Fun language so far is that we can’t write recursive functions, so let’s address that.

The approach we’ll take is not to add recursive *functions* as such, but a recursion *expression* whose body can be a function. For example, we can write $(\mathsf{Rec}\ u\ (\mathsf{Lam}\ x\ e))$, where e is some expression that can refer to u and x.

(It would be slightly more standard to write “fix” instead of Rec, for “fixed point”, but we won’t concern ourselves with whatever a fixed point might be. I mention this only to encourage you to yell at me if I write fix by accident.)

$$
\begin{array}{rcl}
\langle \mathrm{E} \rangle & ::= & \vdots \\
& & \mid \{\mathsf{Rec}\ \langle \mathrm{symbol} \rangle\ \langle \mathrm{E} \rangle\}
\end{array}
$$

What does this thing mean? To answer (?) that, we need an evaluation rule.

$$
\frac{\big[(\mathsf{Rec}\ u\ e)/u\big]e \Downarrow v}{(\mathsf{Rec}\ u\ e) \Downarrow v}\ \text{Eval-rec}
$$

The identifier u in $(\mathsf{Rec}\ u\ e)$ is a way for the expression e to refer to *itself*. So, to evaluate $(\mathsf{Rec}\ u\ e)$, we replace u with... $(\mathsf{Rec}\ u\ e)$! Unfortunately, this can lead to trouble...

A very simple example of a Rec is the expression

$$
\big(\mathsf{Rec}\ u\ (\mathsf{Lam}\ x\ (\mathsf{Id}\ x))\big)
$$

Evaluating this is no trouble, but the Rec doesn’t really serve any purpose here: u doesn’t appear in $(\mathsf{Lam}\ x\ (\mathsf{Id}\ x))$, so substituting for u has no effect:

$$
\frac{\big[\big(\mathsf{Rec}\ u\ (\mathsf{Lam}\ x\ (\mathsf{Id}\ x))\big)/u\big](\mathsf{Lam}\ x\ (\mathsf{Id}\ x)) \Downarrow v}{\big(\mathsf{Rec}\ u\ (\mathsf{Lam}\ x\ (\mathsf{Id}\ x))\big) \Downarrow v}\ \text{Eval-rec}
$$

Using the definition of substitution, the premise is really

$$
\frac{(\mathsf{Lam}\ x\ (\mathsf{Id}\ x)) \Downarrow v}{\big(\mathsf{Rec}\ u\ (\mathsf{Lam}\ x\ (\mathsf{Id}\ x))\big) \Downarrow v}\ \text{Eval-rec}
$$

Using Eval-lam, we get

$$
\frac{\dfrac{}{(\mathsf{Lam}\ x\ (\mathsf{Id}\ x)) \Downarrow (\mathsf{Lam}\ x\ (\mathsf{Id}\ x))}\ \text{Eval-lam}}{\big(\mathsf{Rec}\ u\ (\mathsf{Lam}\ x\ (\mathsf{Id}\ x))\big) \Downarrow (\mathsf{Lam}\ x\ (\mathsf{Id}\ x))}\ \text{Eval-rec}
$$

This is a perfectly good derivation, but we could have obtained the same value by omitting the Rec and just writing $(\mathsf{Lam}\ x\ (\mathsf{Id}\ x))$.

Let's try to evaluate the simplest possible Rec expression that *does* use u.

$$\frac{\big[(\mathsf{Rec}\ u\ (\mathsf{Id}\ u))/u\big](\mathsf{Id}\ u) \Downarrow v}{(\mathsf{Rec}\ u\ (\mathsf{Id}\ u)) \Downarrow v}\ \text{Eval-rec}$$

Rewriting our goal (the premise of Eval-rec) using the definition of *subst*, we get

$$\frac{(\mathsf{Rec}\ u\ (\mathsf{Id}\ u)) \Downarrow v}{(\mathsf{Rec}\ u\ (\mathsf{Id}\ u)) \Downarrow v}\ \text{Eval-rec}$$

So now we need to derive $(\mathsf{Rec}\ u\ (\mathsf{Id}\ u)) \Downarrow v$. The only rule that could possibly work is Eval-rec:

$$\frac{\dfrac{\big[(\mathsf{Rec}\ u\ (\mathsf{Id}\ u))/u\big](\mathsf{Id}\ u) \Downarrow v}{(\mathsf{Rec}\ u\ (\mathsf{Id}\ u)) \Downarrow v}\ \text{Eval-rec}}{(\mathsf{Rec}\ u\ (\mathsf{Id}\ u)) \Downarrow v}\ \text{Eval-rec}$$

This new goal uses *subst*, so we follow that definition again...

$$\frac{\dfrac{(\mathsf{Rec}\ u\ (\mathsf{Id}\ u)) \Downarrow v}{(\mathsf{Rec}\ u\ (\mathsf{Id}\ u)) \Downarrow v}\ \text{Eval-rec}}{(\mathsf{Rec}\ u\ (\mathsf{Id}\ u)) \Downarrow v}\ \text{Eval-rec}$$

This isn't going anywhere!

We should clarify our idea of what a derivation is: a derivation *must be finite*. Endlessly applying the same rule to get an infinite tree isn't allowed.

1.1 Base and recursive cases

Our first attempt to use Rec didn't really do anything; we wrote Rec but didn't use it, kind of like the base case of a recursive function. Our second attempt had us endlessly trying to derive the same thing (and an interpreter following the Eval-rec rule would, in fact, run forever).

A third idea:

$$(\mathsf{Rec}\ u\ (\mathsf{Lam}\ x\ (\mathsf{Add}\ (\mathsf{Id}\ x)\ (\mathsf{App}\ (\mathsf{Id}\ u)\ (\mathsf{Id}\ x)))))$$

This does use u. Evaluating this expression is fine—it evaluates to a Lam. However, when we apply that Lam to something, we'll try to evaluate forever again (though with an ever-changing goal).

Earlier, we added ifzero, so we now have a way for a Fun expression to test an argument and do different things based on it.

2 Soundness and completeness

What does "following the rules" really mean?

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

If completeness does not hold, we say the interpreter is *incomplete*. For example, you might forget to implement an evaluation rule.

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

If soundness does not hold, we say the interpreter is *unsound*.

The words "is derivable" are not quite necessary, but I included them to emphasize that the definition of $e \Downarrow v$ is given by rules.

2.1 Bonus rant

(Skipped during lecture; feel free to skip it here too.) In "interpreter semantics", the *interpreter itself* defines what $e \Downarrow v$ means. So the definitions of soundness and completeness collapse: "If $e \Downarrow v$ then $e \Downarrow v$."

Suppose two of you are (separately) implementing interpreters. One of you implements function application in a way that corresponds to Eval-app-value, and the other implements function application in a way that corresponds to Eval-app-expr. Under interpreter semantics, you have *both* implemented a "correct" interpreter, because *the act of writing an interpreter* (according to interpreter semantics) defines what the language is.

Interpreter semantics has another drawback: you cannot construct a language definition in which behaviour is undefined, because whatever your interpreter happens to do *is* the definition of the language. Now, *which* behaviours should be left undefined can be debated, but real programming languages have multiple implementations (even if we're only counting patches and bug fixes to a single "canonical" implementation!) and run in different environments and processor architectures; you usually can't define everything.

2.2 Undefined behaviour

To be sound, your interpreter must not evaluate an expression successfully (that is, return a value) unless the rules say it does. So your interpreter must not return a value for the expression

$$(\mathsf{Add}\ (\mathsf{Lam}\ \cdots)\ (\mathsf{Lam}\ \cdots))$$

unless $(\mathsf{Add}\ (\mathsf{Lam}\ \cdots)\ (\mathsf{Lam}\ \cdots)) \Downarrow v$ is derivable according to the rules (which it's not for the languages Fun and Fun++ that we've discussed).

For free identifiers, we wrote a rule Eval-free-identifier that says that evaluating $(\mathsf{Id}\ x)$ is a "free-variable-error". But we haven't written rules for other "errors", like adding two Lams. Thus, your interpreter can't return a value, but it's free to treat adding two Lams any way you like. You could:

- generate an error (similar to free-variable-error);
- loop forever (which sounds kind of silly, but we already loop for $(\mathsf{Rec}\ u\ (\mathsf{Id}\ u))\ldots$);
- something else entirely.

(Viktor Vafeiadis, who studies the C++ memory model, likes to give this example of undefined behaviour: "You could launch the missiles.")

Generating an error in such cases sounds like the most organized (precise) option. Writing the rules for this, however, would get rather tedious. More later.

2.2.1 Examples of undefined, unspecified, and implementation-dependent behaviour

- **OCaml**: Feels like one compiler (you download "ocaml", not two different compilers) but has two "back ends": one that generates machine code, and one that generates OCaml virtual machine bytecode. One back end evaluates function application left to right; the other evaluates function application right to left. If you care about order of evaluation, you need to use OCaml's let.
- **C**: Arithmetic overflow is undefined for signed integers. For unsigned integers, it must "wrap around". This seems to be because C predates the consensus that computer architectures should use "two's complement" to represent integers.[1]
- **C** (and many other languages): The size of an `int` was completely unspecified in 1970s/1980s C, and is now partly specified. C99 says that an `int` must be at least 16 bits—well, not quite. Rather, it must be able to represent values between -32767 and 32767. In two's complement representation, 16 bits also gives you $-3276\underline{8}$.
- **C++**: On parallel architectures, which is most of them now that most CPUs have multiple cores, the C++ "memory model"—that is, the guarantees C++ offers about when code running on one core can actually see the effects of code on other cores—is. . . interesting.

If my memory serves (and if this hasn't changed in the last, oh, 20 years), Java has an unusual and refreshing shortage of undefined behaviour, which was probably motivated by the goal of "mobile code": a Java program should run anywhere with the same behaviour.

■ **Exercise 3.** Do some digging (a few Google searches may be enough) and read about undefined behaviour in your favourite (or least favourite) language. If you find something interesting, surprising, or horrifying, and you probably will, post a note on Piazza.

■ **Exercise 4.** (Not an exercise you can expect to actually *do*; just something to think about.) Suppose your programming language allows you to spawn threads that communicate with each other. How would you write an evaluation semantics for such a language?

[1] stackoverflow.com/questions/18195715/why-is-unsigned-integer-overflow-defined-behavior-but-signed-integer-overflow-is