CPSC 311: Conditionals and syntactic sugar **(DRAFT)** ("lec-conditionals")

Joshua Dunfield
University of British Columbia

September 25, 2016

Our language so far can do arithmetic and apply functions (even to other functions), which is not completely useless but can't express functions like factorial $x!$ or even absolute value $|x|$. Factorial needs some way of repeating a multiplication (either recursion or iteration), and needs a conditional to test whether x is zero. Absolute value must test whether its argument is less than zero, to tell it whether to return $-x$ (which would be written (Sub (Num 0) (Id x))) in Fun abstract syntax, or {- 0 x} in Fun concrete syntax) or just x (in abstract syntax, (Id x); in concrete syntax, x).

Actually, our language can *already* express recursion *and* conditionals, but the programs you write will look very strange. More on that later.

1 Ifzero

We need a way for a Fun expression to *test* a value, and evaluate one of two expressions depending on what the value is. So we'll add "Ifzero".

$$\begin{array}{rcl} \langle \text{E} \rangle & ::= & \vdots \\ & & \mid \{\text{Ifzero } \langle \text{E} \rangle \ \langle \text{E} \rangle \ \langle \text{E} \rangle\} \end{array}$$

```
(define-type E
  [Num (n number?)]
  [Add (lhs E?) (rhs E?)]
  [Sub (lhs E?) (rhs E?)]
  [Let (name symbol?) (named-expr E?) (body E?)]
  [Id (name symbol?)]
  [Lam (name symbol?) (body E?)]
  [App (function E?) (argument E?)]
  [Ifzero (scrutinee E?) (zero-branch E?) (nonzero-branch E?)]
)
```

With function application App, we saw that we could use either the value strategy, or the expression strategy, and each had advantages and disadvantages. For Ifzero, the first step is to evaluate the "scrutinee" (because Ifzero is inspecting or "scrutinizing" this expression, to see if it evaluates to zero). But should we evaluate both branches, or just one?

Once we add recursion, the answer will need to be "just one"; otherwise, we would always recurse forever. Even without recursion, the answer should be "just one". Suppose we have an Ifzero expression shaped like this:

$$\Big(\text{Ifzero } \big(\text{App } (\text{Lam } z \ e\text{B}) \ (\text{Num } 77)\big) \\ \big(\text{Num } 0\big) \\ \big(\text{App } (\text{Lam } x \ e\text{THINKING}) \ (\text{Num } 100)\big)\Big)$$

I'm assuming that $e\mathsf{B}$ is some interesting expression (so we can't immediately see whether applying $(\mathsf{Lam}\ z\ e\mathsf{B})$ to $(\mathsf{Num}\ 77)$ is going to return $(\mathsf{Num}\ 0)$) and $e\mathsf{THINKING}$ is a very complicated expression that will take a long time to evaluate. If the first expression evaluates to zero, the whole Ifzero is going to return its "zero branch", which is $(\mathsf{Num}\ 0)$. We shouldn't waste time on $e\mathsf{THINKING}$—we would throw away its result anyway.

The following rules evaluate only one branch:

$$\frac{e \Downarrow (\mathsf{Num}\ 0) \qquad e\mathsf{Z} \Downarrow v}{(\mathsf{Ifzero}\ e\ e\mathsf{Z}\ e\mathsf{NZ}) \Downarrow v}\ \text{??Eval-ifzero-zero} \qquad \frac{e \Downarrow (\mathsf{Num}\ n) \qquad e\mathsf{NZ} \Downarrow v}{(\mathsf{Ifzero}\ e\ e\mathsf{Z}\ e\mathsf{NZ}) \Downarrow v}\ \text{??Eval-ifzero-nonzero}$$

Is this right? Something is missing.

What's missing is a premise in rule Eval-ifzero-nonzero saying that $n \neq 0$. Without this premise, when $e \Downarrow (\mathsf{Num}\ 0)$, we could apply either rule. That's really bad if we're trying to use Ifzero to prevent unbounded recursion. It also violates determinism, that is:

"For all expressions e, if $e \Downarrow v_1$ and $e \Downarrow v_2$, then $v_1 = v_2$."

There are good reasons to violate determinism (can you think of any?), but forgetting a premise isn't one of them.

So we really want these new rules:

$$\frac{e \Downarrow (\mathsf{Num}\ 0) \qquad eZ \Downarrow v}{(\mathsf{Ifzero}\ e\ eZ\ eNZ) \Downarrow v}\ \text{Eval-ifzero-zero} \qquad \frac{e \Downarrow (\mathsf{Num}\ n) \qquad n \neq 0 \qquad eNZ \Downarrow v}{(\mathsf{Ifzero}\ e\ eZ\ eNZ) \Downarrow v}\ \text{Eval-ifzero-nonzero}$$

Question: These rules sort of suggest that my interpreter should only evaluate one of eZ and eNZ—either I apply Eval-ifzero-zero, or Eval-ifzero-nonzero, and each of those two rules evaluates only one of eZ and eNZ. What if I evaluate both of them? Can my interpreter still return the right value? And if it does, isn't that "following the rules"?

For the particular language we have now, you can evaluate both eZ and eNZ in your interpreter (meaning, call `interp` recursively on both eZ and eNZ), and you will always get the correct result—assuming you don't do the strange tricks I alluded to earlier to obtain recursion!

Once we add recursion, you will not be able to evaluate both branches: if one of the branches recurses forever, but it's not the branch you need to take, your interpreter will recurse forever even though a derivation exists.

2 Syntactic sugar

This Ifzero expression doesn't seem too versatile; what if we want to test if a number is *less* than zero? Should we add another kind of expression, Iflessthanzero? We could, but a better, more general design is to add booleans and a general "if-then-else" (like Racket's `if`) to the language, which will be part of the next assignment.

Exercise 1. Write a Fun expression that behaves like Iflessthanzero, using only Ifzero and the other features of Fun (including recursion). It only needs to work for integers; don't worry about other numbers. (I think I have a solution, but I haven't written it down... it has a peculiar Turing-machine flavour.)

Less perversely, we can code up Ifequal: instead of $(\mathsf{Ifequal}\ e1\ e2\ eEq\ eNotEq)$, write $(\mathsf{Ifzero}\ (\mathsf{Sub}\ e1\ e2)\ eEq\ eNotEq)$. It would be annoying to actually write that instead of Ifequal. On the other hand, it would be annoying to add Ifequal to the language: we would have to

- extend the grammar,
- add a variant to the abstract syntax,
- update our parser,
- figure out new evaluation rules,

- extend the definition of substitution, and
- add code to our interpreter.

We could certainly do these tasks, but it's not just work for us—we're also making everything in the specification of the language bigger. If we were proving things in 311, we would also want to extend our proofs of whatever language properties we care about, such as determinism. And if our language specification is given using rules, the new evaluation rules will become part of the language manual, making it bigger.

There's another option that avoids doing most of the above tasks. It's a common practice in language design: add a new feature as *syntactic sugar*. We still have to extend the grammar and update our parser, but *nothing else has to change*, because we will translate ("desugar") Ifequal within the parser:

{Ifequal *e*1 *e*2 *e*Eq *e*NotEq} is parsed as (Ifzero (Sub *e*1 *e*2) *e*Eq *e*NotEq)

This seems to save a lot of work. Is there any reason not to do this?

Unfortunately, yes: parsing and unparsing are no longer inverse operations. That is, transforming concrete syntax to abstract syntax (parsing) and then transforming the abstract syntax back to concrete syntax (unparsing) won't necessarily give the original concrete syntax back.

That might not sound too bad... except that unparsing is also how we would want to print error messages. So the error messages will be confusing, because they refer to code the user didn't write! For example, our interpreter should print an error message if you try to use Ifequal with Lams—and indeed it will, assuming that subtracting Lams prints an error message. But the error message will say that Sub was given invalid arguments, not that Ifequal was!

Here's an example from a real programming language, SML (my favourite language). To make any sense of this, you probably need to know that case is SML's version of **type-case** and that SOME is a variant (constructor) declared by (SML's version of) **define-type**; I'll try to explain the rest as we go.

```
Standard ML of New Jersey v110.72 [built: Tue Jan 11 13:30:58 2011]
- fun f x = case x of SOME y => y
                    | SOME z => z;
stdIn:1.14-2.36 Error: match redundant and nonexhaustive
          SOME y => ...
    -->   SOME z => ...
```

SML is complaining that I've written the same constructor twice in two branches ("redundant") and also that I didn't write another constructor at all ("nonexhaustive"), errors you've already seen (with different terminology) with **type-case**.

In SML (and in PLAI), it's common to write a function that immediately does a case (**type-case**), so SML allows you to write functions in "clausal form", like this:

```
fun fib 0 = 0
  | fib 1 = 1
  | fib n = fib (n-2) + fib (n-1);
```

This closely resembles mathematical notation for defining functions by cases, but it behaves exactly like

```
fun fib x = case x of 0 => 0
                    | 1 => 1
                    | n => fib (n-2) + fib (n-1);
```

The Definition of Standard ML defines clausal form to be a "derived form", which is a fancy name for syntactic sugar: the meaning of a clausal function is given by a translation to a function whose body is a case. That is, the clausal form syntax is derived from the "real" syntax (case).

Since the error message shows the unparsing of the abstract syntax, it shows code that doesn't match what I wrote:

```
- fun f (SOME y) = y
    | f (SOME z) = z;
= stdIn:1.9-3.23 Error: match redundant and nonexhaustive
          SOME y => ...
    -->   SOME z => ...
```

Showing the "wrong" code teaches SML programmers which language features are derived forms, which is somewhat useful but probably doesn't make up for the frustration.