Notes for Lecture 6 (Fall 2022 Week 3 part 2): Stepping in more detail

Jana Dunfield

September 18, 2022

1 Stepping in more detail

1.1 Substitution notation

The short form of the rule for stepping a lambda is

```
------------------------------
(\x -> eBody) v  =>  eBody[v/x]
                     ^^^^^^^^^^
                     expression eBody
                     with v substituted for x
                     (or "x replaced by v")
```


This is the same rule notation used in CISC 204: if the premises (stuff above the line) are true, the conclusion (stuff above the line) is true. In this rule there are no premises.[1]

The notation `eBody[v/x]` means "the expression `eBody` with `v` substituted for `x`". (Or "`x` replaced by `v`", but to say that you have to read `v` and `x` in a different order from how they're written.)

If you find yourself asking "if I see '... `[v/x]`', am I substituting `v` for `x`, or substituting `x` for `v`?", you can often figure it out from context: substituting `x` for `v` doesn't make sense because `eBody` should be talking about `x`, not `v`.

Another way to remember is that the notation for substitution

```
[f/x]
```

looks like a fraction

$$\texttt{f/x} = \frac{\texttt{f}}{\texttt{x}}$$

If we substitute `f` for `x` in the expression `x`, as would happen when stepping `(\x -> x) f`, we would get `f`. If we think of substitution as multiplication (*this does not work in general, but it works here to help us remember what the notation means*), we have

$$\texttt{x} \cdot \frac{\texttt{f}}{\texttt{x}} = \texttt{f}$$

Remark 1. I'm using this notation because I (mostly) like it and it matches the notation used in CISC 204. In other courses, or in online material, you may encounter different notation for substitution. Some people use braces instead of square brackets. Some write the expression

[1] Because it has no premises, some people would call it an axiom rather than a rule.

being substituted over (here, `eBody`) after the brackets, rather than before. (I do that when I teach CISC 465.) Some people use a backslash instead of a forward slash, or a "maps to" symbol: `eBody[x ↦ v]`.

1.2 Where not to step

With very small expressions like

```
2 + 2
```

there's no ambiguity about where to step: we have to step the whole expression.

With the slightly larger expression

```
5 - (2 + 2)
```

we have two potential sites for stepping:

- `2 + 2`, and
- `5 - (2 + 2)`

The expression `2 + 2` steps to `4` by arithmetic.

The expression `5 - (2 + 2)` steps to `5 - 4` because we can "step inside" arithmetic:

- `2 + 2` steps to `4`;
- therefore, `5 - (2 + 2)` steps to `5 - 4`.

Is there anything we can't "step inside"? Yes: we can't step inside lambdas. We also can't step inside function definitions. Why not?

Suppose we're trying to step the expression

```
(\y -> y + 1) 3
```

We can't step `y + 1` until we substitute something for `y` (in this case, `3`). In general, we will get stuck trying to step inside a lambda until we've substituted an argument for the bound variable.

There are some expressions where we wouldn't get stuck, like

```
(\z -> 2 + 2) 3
```

Here, the function `(\z -> 2 + 2)` doesn't mention its bound variable, so we could in theory step `2 + 2`. But for simplicity, we (and Haskell) *never* step inside a lambda. (Why is that simpler? It gives us no choice about where to step. Choices about where to step should not affect the result—the expression we get after repeatedly stepping—but they can affect the number of steps taken.)

You can try to remember "don't step inside a lambda", but it may be better to remember *why* we don't: we need to eliminate the bound variable (by substituting an argument for it).

1.3 Where to step

Explaining where we can't step is easier than explaining where we can.

A fully detailed explanation of where to step might begin like this.

```
"Step-inside" rules

eFun  =>  eFun'              eArg  =>  eArg'
------------------------     ------------------------
eFun eArg  =>  eFun' eArg    eFun eArg  =>  eFun eArg'
```

The second "step-inside" rule is probably easier to think about:

- if the expression `eArg` steps to `eArg'`, then
- the expression `eFun` applied to `eArg`
  steps to
  the expression `eFun` applied to `eArg'`.

This is the rule that allows us to step

```
   (\x -> x + 1) (2 + 2)
=>  (\x -> x + 1) 4
```

In this example, `eFun` is `(\x -> x + 1)`, `eArg` is `(2 + 2)`, and `eArg'` is `4`.

The first "step-inside" rule may be more surprising, because we may not be used to thinking of a function being replaced by another function.

- if the expression `eFun` steps to `eFun'`, then
- the expression `eFun` applied to `eArg`
  steps to
  the expression `eFun'` applied to `eArg`.

This is the rule that allows us to step

```
   (\x -> (\y -> y * x) 1) 2
=>         (\y -> y * 2) 1
```

In this example, `eFun` is `(\x -> (\y -> y * x) 1)`, `eArg` is `2`, and `eFun'` is `(\y -> y * 2) 1`.

Giving a precise and complete definition of where Haskell steps would require giving rules like the two above, but for pretty much every expression form in Haskell. I won't try to do this:

- it would be exhausting (for you and me);
- it would be complicated (what Haskell really does is more complicated than what we'll get away with in this course).

For CISC 360, we will get away with the following:

- We can't step inside a lambda or a function definition.
- We can't do an arithmetic step unless all the arguments are numbers. (So we can step `(0 - 1) + (calc ...)` because both `0` and `1` are numbers.)

I think I've explained the first of these. The second might sound obvious, but it's not obvious if we think about it enough: why shouldn't we step `x + 0` to `x`? We know that adding zero to something won't change it. As with not stepping inside a lambda, this is a matter of simplicity: "don't do arithmetic unless we know the actual numbers" is a simple rule that won't get us into trouble.[2]

[2] I'm confident that `x + 0` is equal to `x`. I'm not sure if `x + 0.0` is equal to `x`.