# Notes for Lecture 3 (Fall 2022 week 2, part 1): More stepping, more Haskell

Jana Dunfield

September 11, 2022

Code for this lecture is in `lec3.hs`.

If you haven't yet, please install GHC. If you have problems, ask on Piazza. It may be easier to install Hugs, which is similar (but the error messages will be different).

## 1 Warm-up

Download the file `lec3.hs` and try:

1. Start GHCi.
2. Load the file (use the `:load` command).
3. Enter

   ```
   triple 2
   ```

   and make sure you get 6.

4. Edit the file to add the following function:

   ```
   myfun = \w -> not w
   ```

5. Typing

   ```
   myfun False
   ```

   will give an error, because we haven't reloaded the file.
6. Reload the file. You can do this by giving the same load command again, but it's easier to type

   ```
   :r
   ```

7. Try

   ```
   myfun False
   ```

   again.
8. Now try editing the file to add

   ```
   what = \f -> f (f False)
   ```

9. Reload and experiment with `what`. It may help to ask for the type of `what`, by entering

   ```
   :type what
   ```

## 2 Prefix and infix in Haskell

This section looks like it's about syntax, but it will also expose you to something that will become important later. Don't expect to understand everything here yet. Try things out and experiment, but don't get stuck—I will explain this more in the next lecture, and in later lectures.

In mathematics (and programming languages), operators that can be written before their argument(s) are prefix operators. Negation is one such operator: the negation operator $-$ is written before its argument $(x * 2)$.

$$-(x * 2)$$

Operators that are written between their arguments are infix operators. The multiplication operator $*$ is an infix operator.

Haskell's opinion about which operators are prefix and infix is mostly in accord with mathematical notation. Haskell allows something that isn't common in math, however: If you want to use an infix operator as a prefix operator, you can surround it in parentheses. The following will both print 7; in the first line, + is used in the standard way as an infix operator, and in the second line it is (temporarily) used as a prefix operator "(+)":

```
3 + 4

(+) 3 4
```

The second line may seem less readable than the first; we're used to writing $+$ between its arguments, not before. The second line is not why Haskell provides this feature of turning infix operators into prefix ones. Haskell provides this so we can refer to the addition operator *by itself*.

If we try to ask for the type of addition like this, we get an error:

```
Prelude> :type +

<interactive>:1:1: parse error on input '+'
```

If we add parentheses, we don't get an error:

```
Prelude> :type (+)
(+) :: Num a => a -> a -> a
Prelude>
```

What we do get needs some explanation, though. This type says: "if a is a numeric type, then if you give me one thing of type a, and another thing of type a, I will return something of type a." The "Num a" part means "a is a numeric type" (addition works on several different numeric types, not only integers), and the => separates the Num a part from the rest of the type. You can think of a -> a -> a as the "actual" type, and the Num a part as some extra information.

I wrote "if you give me one thing of type a, and another thing of type a…" rather than saying *two* things of type a. Does Haskell let us give addition only one argument? Another way to phrase the question: Does Haskell let us apply addition to only argument?

We need to use parentheses (writing "2 +" would be a syntax error):

```
Prelude> (+) 2
```

```
<interactive>:7:1:
    No instance for (Show (a0 -> a0))
      (maybe you haven't applied enough arguments to a function?)
      arising from a use of ‘print’
    In the first argument of ‘print’, namely ‘it’
    In a stmt of an interactive GHCi command: print it
```

This is also an error, but the message does *not* say that GHC can't apply `(+)` to one argument; it's saying GHC can't *print* the result of doing that! (So the "(maybe you haven't applied enough arguments to a function?)" part is somewhat misleading.)

When we see an error message that says "No instance for `(Show ...)`", GHC is saying that it did what we asked—it computed the result of `(+) 2`—but it can't print the result. GHC is a read-evaluate-print loop: if it complains about not being able to print, it must have been able to evaluate.

To try to understand what's going on, we can ask GHC for the *type* of `(+) 2`. GHC will not evaluate expressions that don't have a type; we know that GHC evaluated, so there must be such sa type.

```
Prelude> :type (+) 2
(+) 2 :: Num a => a -> a
```

If we remove the `Num a` part, we get `a -> a`, which is a function that expects an `a` and returns an `a`.

```
Prelude> (+) 2 3
5
```

(No, I don't enjoy GHC's error messages either.)

## 3 Stepping: Rule for function application

(From earlier lecture)

If $f$ is a function,
and the bound variable of $f$ is $x$,
and the body of $f$ is $body$, then

$$f\ arg \quad \Rightarrow \quad body \text{ with } arg \text{ substituted for } x$$

### 3.1 Stepping with function definitions

The rule can also be used with function definitions.

```
triple z = 3 * z
```

$$\begin{array}{lll} & \texttt{triple 2} & \\ \Rightarrow & \texttt{3 * 2} & \text{by function application} \\ \Rightarrow & \texttt{6} & \text{by arithmetic} \end{array}$$

Going into more detail for the first step:

We are *instantiating* the above rule for function application as follows:

1. the function $f$ is `triple`
2. the bound variable $x$ is `z`
3. the body of $f$ (that is, the body of `triple`) is `3 * z`
4. the argument $arg$ is 2

Writing out the rule again (generally not something I'd ask you to do; done here to illustrate):
If `triple` is a function,
and the bound variable of `triple` is $z$,
and the body of `triple` is `3 * z`, then

$$\begin{aligned} & \texttt{triple 2} \\ \Rightarrow\ & \texttt{3 * z} \text{ with 2 substituted for } \texttt{z} \\ =\ & \texttt{3 * 2} \end{aligned}$$

Two different things just happened. First, we instantiated the function assignment rule and wrote out "$\Rightarrow$ `3 * z` with 2 substituted for `z`". Second, we wrote the line "= `3 * 2`". Only the *first* of these is a "step"; in the second, we observed that

`3 * z` with 2 substituted for `z`

is *equal* to

`3 * 2`

which is not a step; it is an observation about what "substituted" means.

In fact, there is no rule that says

`3 * z` with 2 substituted for `z`

*steps* to

`3 * 2`

Since these two things are *equal*, such a rule would mean that

`3 * 2`

steps to

`3 * 2`

which it does not. (It is very rare for an expression to step to itself, and `3 * 2` certainly does not step to itself. It steps to `6`.)

## 3.2 Recursive functions

The rule for function application also works for recursive functions, like `diag` (in `lec3.hs`):

```
diag n = if n == 0 then 0 else n + diag (n - 1)
```

This is a recursive function, since the "else" branch calls `diag` itself.

I like this example because it has a nice geometric property (switching to mathematical notation for a moment):

$$\begin{aligned}
\mathsf{diag}(3) &= 3 + \mathsf{diag}(2) \\
&= 3 + 2 + \mathsf{diag}(1) \\
&= 3 + 2 + 1 + \mathsf{diag}(0) \\
&= 3 + 2 + 1 + 0 \\
&= 6
\end{aligned}$$

This can be visualized as

```
3 + 2 + 1 + 0

X
X   X
X   X   X
^   ^   ^
|   |   contributed by diag(1)
|   |
|   contributed by diag(2)
|
contributed by diag(3)
```

which is a triangle, suggesting that the result of $\mathsf{diag}(n)$ is roughly $\frac{n^2}{2}$. (It is actually $\frac{n \cdot (n+1)}{2}$, which is consistent with $\mathsf{diag}(3) = 6 = \frac{3 \cdot 4}{2} = \frac{12}{2}$.)

Getting back on topic, the *bound variable* of `diag` is `n`, and the *body* is everything after the `=`:

```
  diag n = if n == 0 then 0 else n + diag (n - 1)
       ^   ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

bound var.                  body
```

Let's start with a *non*-recursive case:

| | | |
|---|---|---|
| | `diag 0` | |
| $\Rightarrow$ | `if 0 == 0 then 0 else 0 + diag (0 - 1)` | by function application |
| $\Rightarrow$ | `if True then 0 else 0 + diag (0 - 1)` | by equality rule (Section 5) |
| $\Rightarrow$ | `0` | by if-then-else rule |

■ **Exercise 1.** In the first step above, identify $f$, $body$, and $arg$, and write it out in full detail (writing "... with ... substituted ... = ...") as we did for `triple`.

Now we can do a *non*-recursive case:

```
   diag 1
⇒  if 1 == 0 then 0 else 1 + diag (1 - 1)   by function application
⇒  if False then 0 else 1 + diag (1 - 1)    by equality rule (Section 5)
⇒  1 + diag (1 - 1)                          by if-then-else rule
⇒  1 + diag (0)                              by arithmetic
⇒  1 + 0                                     above, we found diag 0 ⇒ ... ⇒ 0
⇒  1                                         by arithmetic
```

Not too bad? I have some bad news and some good news.

The bad news is that *this is not what Haskell actually does*.

The good news is that (for the next few weeks, at least) *it doesn't matter*.

But, if you're curious, this is what Haskell actually does:

```
   diag 1
⇒  if 1 == 0 then 0 else 1 + diag (1 - 1)                      by function application
⇒  if False then 0 else 1 + diag (1 - 1)                       by equality rule (Section 5)
⇒  1 + diag (1 - 1)                                             by if-then-else rule
⇒  1 + (if (1 - 1) == 0 then 0 else 1 + diag ((1 - 1) - 1))    by function application
⇒  1 + (if 0 == 0 then 0 else 1 + diag (0 - 1))                by arithmetic
⇒  1 + (if True then 0 else 1 + diag (0 - 1))                  by equality rule
⇒  1 + 0                                                        by if-then-else rule
⇒  1                                                            by arithmetic
```

Haskell defers the subtraction `(1 - 1)` until *after* the function application—but once the subtraction is done, the `(1 - 1)` in the else branch is *also* done! This is *lazy evaluation*; until further notice, we will ignore this. I would accept either sequence of steps as a correct answer.

## 4 Stepping: Rule for if-then-else

This is really two rules, but we won't worry about that.

`if True then` then-part `else` else-part
⇒ then-part

`if False then` then-part `else` else-part
⇒ else-part

**Question:** Do we have to memorize this rule (or the other rules)? Do they have to be called "then-part" and "else-part"?

In general, you don't have to memorize any of the rules (but you should understand what they do); you certainly don't need to memorize names like "then-part" (which are arbitrary; it would be traditional to call it "e1").

If I ask a quiz question, and it requires that you justify each step of an evaluation (by writing "by arithmetic", "by fun. app.", etc.), then I will expect you to identify the rule you are using—*but* I will give you all the necessary rules. What I don't want to see is a justification that could be a "guess", like "by Haskell" (which could mean anything) or "by conditionals" (unless I decide to call this rule "conditionals"). Being able to make reasonable guesses (while often an effective test-taking skill) doesn't show that you understand the material.

## 5 Stepping: Equality rule (for the == operator)

If e is a number, then

$$e \;\texttt{==}\; e \;\Rightarrow\; \texttt{True}$$

If e1 and e2 are numbers,
and e1 is not equal to e2 then

$$e1 \;\texttt{==}\; e2 \;\Rightarrow\; \texttt{False}$$

This isn't as simple as it seemed during the lecture! Just saying

$$x \;\texttt{==}\; x \;\Rightarrow\; \texttt{True}$$

will work for calling diag, but promises too much—Haskell only allows you to compare certain kinds of things with ==. You can't compare functions, for example:

```
(\z -> z + 1) == (\z -> z + 1)
```

doesn't work (it will give you a charming type error). Even the "obvious"

```
triple == triple
```

is not allowed. So in completing these notes, I added a condition to the rule: e has to be a number. (Actually, == works for *some* non-numbers, but the explanation wouldn't make sense to you yet, and right now we just need a stepping rule that lets us compare numbers.)