Week 1 part 2: Stepping; Haskell intro

Jana Dunfield

September 6, 2022

References to “Thompson” are to *Haskell: The Craft of Functional Programming*, 3rd ed.

1 Functional programming in industry

- http://cufp.org/2015/fighting-spam-with-haskell-at-facebook.html (Haskell at Facebook)
- http://cufp.org/2016/yaron-minsky-keynote.html (OCaml at Jane Street, a financial services firm)
- http://cufp.org/2017/using-functional-programming-to-accelerate-translational-research-at-pfizer.html (functional programming for pharmaceuticals)

An (arguably) functional programming language, Rust, is used in substantial parts of Firefox—for both reliability and performance reasons (functional programs are often easier to parallelize).

2 Stepping

The behaviour of Haskell *expressions*, such as $2 + 2$ and `(\x -> x + 1) 3`, can be predicted by a process of stepping. Thompson calls stepping "calculation"; another name is "tracing".

For very simple examples, stepping is hard to distinguish from equality in algebra:

	`2 + 2`		`2 + 2`	
$=$	`4`	$\Rightarrow$	`4`	by arithmetic

(Thompson uses a squiggly arrow $\rightsquigarrow$ instead of $\Rightarrow$.) If you are taking notes on a laptop, I suggest typing ==> to avoid confusion with =>, which is part of Haskell's type syntax.

For slightly larger examples, however, stepping distinguishes itself from equality—we do only *one* operation per step:

	`(2 + 2) + (3 + 4)`		`(2 + 2) + (3 + 4)`	
$=$	`11`	$\Rightarrow$	`4 + (3 + 4)`	by arithmetic
		$\Rightarrow$	`4 + 7`	by arithmetic
		$\Rightarrow$	`11`	by arithmetic

And whereas mathematical equality may have no particular goal—we can write $5 = 1 + (6 - 2) = 3 + 2 = 2 + 1 + 2$—stepping moves towards a result. The Haskell expression `11` will not step "backwards" to `4 + 7`.

Adding integers may seem simple, but it is not always clear how stepping should work. How should `1 / 3` step? In a previous iteration of 360, I received four suggestions:

1. `(1 / 3)`
   $\Rightarrow$ `0` by arithmetic

2. `(1 / 3)`
   $\Rightarrow$ `0.3333333333333333` by arithmetic

3. `(1 / 3)`
   $\Rightarrow$ `1` by arithmetic

4. `(1 / 3)`
   $\Rightarrow$ $\frac{1}{3}$ by arithmetic

All of these are reasonable choices, and at least three of them are found in actual programming languages; the operator / exists in most languages, but they don't agree on its meaning (its *semantics*).

- Suggestion 1 interprets / as integer division, rounding down. C, Java and Python interpret / this way.

  (In algorithms textbooks, this would be written $\lfloor 1 / 3 \rfloor$: mathematicians usually assume / means division on real numbers, so they have to "floor" or "truncate" the result $\frac{1}{3}$ by applying the operation $\lfloor \ldots \rfloor$.)

- Suggestion 2 interprets / as real (floating-point) division. Haskell interprets / this way.

- Suggestion 3 interprets / as integer division, rounding *up* (like $\lceil 1 / 3 \rceil$). I don't know of any language that does this, but there probably are some.

- Suggestion 4 interprets / as exact division or rational division (1/3 being the ratio of two integers). The Racket language interprets / this way (though in Racket, you write `(/ 1 3)` rather than `1 / 3`). A student said that Matlab does this, and I expect that Maple and Mathematica do as well.

Aside:

```
/* example showing the semantics of the / operator in Java */
public class hmm {
    public static void main (String[] args) {
        System.out.println (1 / 3);           /* prints 0 */

        System.out.println (3 * (1.0 / 3));   /* prints 1.0 */
    }
}
```


3 Using GHCi's read-eval-print loop

We interact with GHCi through its "read-eval-print loop" (which some people call a REPL, pronounced "repple"). Read-eval-print loops are available for many languages, including Python, but not others (including Java, C and C++).

The read-eval-print loop is:

1. **read** input
2. **evaluate** input
3. **print** result
4. loop back to 1

When you enter an expression such as `2 + 2` into GHCi, the loop **reads** the expression, **evaluates** it to a result (namely `4`), **prints** the result, and then waits to read more input.

(If you are familiar with command-line shells such as `bash`, they are generally not "read-eval-print", but "read-run": by default, they don't print the exit status. If you are not familiar with command-line shells, don't worry about this.)

Typing large expressions is tedious and unreliable, however, so most of the time we will enter Haskell code in files (with the extension `.hs`), load them into GHCi (as described below), and then evaluate some expressions. For example, if we define a function `triple` in a `.hs` file, we can type

```
triple 10
```

and (hopefully) get the result `30`.

3.1 Loading files

To load `filename.hs` into GHCi, enter a *load command*:

```
:load filename.hs
```
or
```
:load filename
```
or
```
:l filename
```

Once the file is loaded, the prompt "`Prelude>`" becomes "`*Main>`". (If the file contains a module, then the module's name will be printed instead of "Main".)

A load command is *not* a Haskell expression: When you enter a line beginning with `:` into GHCi, it interprets the line as a command. Commands ask GHCi to take some action that is not "evaluate a Haskell expression". You can see a list of GHCi commands by entering `:?` or `:help`, though I find this more useful for reminding myself of commands' spelling than for learning how they work.

The `:reload` command, which can be abbreviated `:r`, reloads the previously loaded file. So if you change `filename.hs` and want to use the updated version, enter `:r`.

3.2 Seeing the type of an expression

Like Java, but unlike Python, Haskell is a "statically typed" language. You can ask GHCi for the type of an expression by entering the command

`:type` *expression*

Haskell's type system prioritizes power (being good at catching subtle bugs) over ease of learning, so this can give surprising results:


```
*Main> :type 3
3 :: Num a => a
```

I won't explain this in full detail now, but roughly, this is Haskell "hedging its bets": the expression `3` can have a variety of numeric types (including integers and floating-point numbers, which behave differently), so Haskell refuses to choose just one. Instead, it says `3` has type 'a', where all we know is that `a` is some numeric type ('`Num a`')".

If we tell Haskell that we want a specific numeric type, say `Int`, it accepts our request and says the expression `(3 :: Int)` has type `Int`.

```
*Main> :type (3 :: Int)
(3 :: Int) :: Int
```

If `3` is an `Int`, then Haskell decides (reasonably) that adding `4` to it gives an `Int`, even though we wrote `4` and not `(4 :: Int)`:

```
*Main> :type (3 :: Int) + 4
(3 :: Int) + 4 :: Int
```

This is all somewhat confusing, but Haskell is more decisive when you ask it about definitions in loaded files: if you put

```
x = 3
```

in a file `a.hs`, load `a.hs`, and then ask for the type of `x`, you get a clearer answer. Try it!

4 Functions

An anonymous function, called a *lambda* (λ), is written in Haskell by writing a backslash (which sort of resembles λ), followed by the name of the argument (called the *bound variable*), followed by `->`, followed by the function body:

```
    \x -> x + 1
     ^    ^^^^^
  bound    body
 variable
```

We can call this function by writing it, then giving an argument:

```
(\x -> x + 1) 6
```

This is a small expression: try it in the GHCi read-eval-print loop.
How does this step? Since GHCi prints 7 at the end, there must be a sequence of steps

$$\begin{array}{ll} & \texttt{(\textbackslash x -> x + 1) 6} \\ \Rightarrow & \\ & \vdots \\ \Rightarrow & \texttt{7} \end{array}$$

It seems plausible that the last step is to add 6 and 1:

$$\begin{array}{lll} & \texttt{(\textbackslash x -> x + 1) 6} & \\ \Rightarrow & & \\ & \vdots & \\ \Rightarrow & \texttt{6 + 1} & \\ \Rightarrow & \texttt{7} & \text{by arithmetic} \end{array}$$

In fact, there are just two steps: the second is arithmetic, the first is a function application (another name for a function call).

$$\begin{array}{lll} & \texttt{(\textbackslash x -> x + 1) 6} & \\ \Rightarrow & \texttt{6 + 1} & \text{by function application} \\ \Rightarrow & \texttt{7} & \text{by arithmetic} \end{array}$$

If we can step from `(\x -> x + 1) 6` to `6 + 1` there must be a stepping rule that explains how to do this.

4.1 Stepping rule for function application

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

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

Read the last line as:

f applied to argument arg steps to body with arg substituted for x

Exercise 1. Using the stepping rule for function application, then the stepping rule for arithmetic, fill in the following:

```
  (\z -> 8 - z) 0
⇒                     by function application
⇒                     by arithmetic
```

Exercise 2. Fill in the following:


```
  (\f -> f - f) (2 + 2)
⇒                     by arithmetic
⇒
⇒
```


(There is only one correct *result*, zero, but several reasonable sequences of steps. Only one sequence of steps corresponds to what Haskell actually does. But I haven't yet explained what Haskell actually does!)

5 Optional readings for this lecture

- Thompson, chapter 1
  - Section 1.10, "Calculation and evaluation", is another presentation of *stepping*.

6 Other remarks

Remark 1. In Thompson's discussion of fixed-size integers, the bound 2147483647—which is $2^{31} - 1$—is almost certainly out of date. To see the value on your system, enter (in GHCi)

```
maxBound :: Int
```

If you enter `maxBound` without the "`:: Int`", you won't get anything useful. (GHCi won't even give you an error message, though Hugs will.) GHC has different maximum values for different types; when you enter `maxBound` by itself, it doesn't know what type you want.

(It's usually better to use `Integer` instead of `Int` anyway, because `Integer` avoids the possibility of overflow.)