FizzBuzz in Haskell by Embedding a Domain-Specific Language

by Maciej Piróg ⟨maciej.adam.pirog@gmail.com⟩

April 23, 2014

The FizzBuzz problem is simple but not trivial, which makes it a popular puzzle during job interviews for software developers. The conundrum lies in a peculiar but not unusual control-flow scenario: the default action is executed only if some previous actions were not executed. In this tutorial, we ask if we can accomplish this without having to check the conditions for the previous actions twice; in other words, if we can make the control flow follow the information flow without loosing modularity. The goal is to have the most beautiful code!

We deliver a rather non-standard, and a bit tongue-in-cheek solution. First, we design a drastically simple domain-specific language (DSL), which we call, after the three commands of the language, Skip-Halt-Print. For each natural number n, we devise a Skip-Halt-Print program that solves FizzBuzz for n. Then, we implement this in Haskell, and, through a couple of simple transformations, we obtain the final program. The corollary is a reminder of the importance of higher-order functions in every functional programmer's toolbox.

The FizzBuzz problem

FizzBuzz is a simple game for children, and therefore a really hard nut to crack for programmers and computer scientists. To quote the rules [1]:

> Players generally sit in a circle. The player designated to go first says the number '1', and each player thenceforth counts one number in turn. However, any number divisible by three is replaced by the word *fizz* and any divisible by five by the word *buzz*. Numbers divisible by both become *fizzbuzz*.

In this tutorial, we focus on a single step of the game, that is to convert a natural number n into *fizz*, *buzz*, *fizzbuzz*, or its string representation.

There are a lot of solutions floating around the Internet, but most of them are, from our point of view, unsatisfactory. Exhibit A:

```
fizzbuzz :: Int → String
fizzbuzz n =
  if n `mod` 3 ≡ 0 ∧ n `mod` 5 ≡ 0 then
    "fizzbuzz"
  else if n `mod` 3 ≡ 0 then
    "fizz"
  else if n `mod` 5 ≡ 0 then
    "buzz"
  else
    show n
```

Exhibit B:

```
fizzbuzz :: Int → String
fizzbuzz n =
  if n `mod` 3 ≡ 0
    then "fizz" ++ if n `mod` 5 ≡ 0
                     then "buzz"
                     else ""
    else if n `mod` 5 ≡ 0
           then "buzz"
           else show n
```

Though both programs are correct with respect to the specification, Exhibit A, in some cases, performs the `*mod*`3 and `*mod*`5 tests more than once, while Exhibit B disperses the *buzzing* code into more than one place in the program. Meanwhile, we want there to be at most one place that outputs *fizz*, *buzz*, or the string representation, and each test to be performed only once. Outputting *fizzbuzz* should be done by executing the (one and only) piece of code that outputs *fizz*, followed by the (one and only) piece that outputs *buzz*. That is because *fizzing* and *buzzing* are two separate activities – consider the FizzBuzzHissHowl problem, where *hiss* and *howl* are printed for multiples of 7 and 11 respectively. The program design of Exhibit A and B would lead to an explosion of code complexity.

We can examine the output itself, which gives us an opportunity for yet another unsatisfactory attempt, Exhibit C:

```
(◁) :: String → String → String
"" ◁ s = s
```

$$
\begin{array}{l}
a \;\; \lhd\, s = a \\
\mathit{fizzbuzz} :: \mathit{Int} \to \mathit{String} \\
\mathit{fizzbuzz}\ n = ((\ \mathbf{if}\ n\ \text{`}\mathit{mod}\text{`}\ 3 \equiv 0\ \mathbf{then}\ \texttt{"fizz"}\ \mathbf{else}\ \texttt{""}) \\
\qquad\qquad\quad + \!\! +\ \mathbf{if}\ n\ \text{`}\mathit{mod}\text{`}\ 5 \equiv 0\ \mathbf{then}\ \texttt{"buzz"}\ \mathbf{else}\ \texttt{""}) \\
\qquad\qquad\quad \lhd\ \mathit{show}\ n
\end{array}
$$

The problem with this solution is far more subtle: many might say this solution is simple and elegant. Be that as it may, we do not like the fact that the $\lhd$ operator has to check if its first argument is empty. After all, we have already checked the conditions `*mod*`3 and `*mod*`5, so the third test ($\lhd$'s pattern matching) seems redundant from the information-flow point of view (compare Exhibit B, which always performs only two tests).

So, is out there a program that reflects the information-flow structure as Exhibit B, but, at the same time, is as modular as Exhibit C? Let's find out!

Skip-Halt-Print and contexts

If one feels overwhelmed by the number of (better or worse) possible ways to solve such a simple problem as FizzBuzz in Haskell, they can start with a simpler language. The one we propose is called Skip-Halt-Print, and it is very imperative.

A program in Skip-Halt-Print is a (possibly empty) list of commands, which are executed sequentially. There are only three different commands:

- SKIP is an idle instruction; it does nothing at all;
- HALT stops the computation; the rest of the program is not executed;
- PRINT `s` prints out the string `s`.

More formally, the syntax is given by the following grammar, where c denotes commands, p denotes programs, and ϵ is the empty program:

$$
\begin{aligned}
c &::= \textsc{Skip} \mid \textsc{Halt} \mid \textsc{Print}\ \texttt{s} \\
p &::= c;\ p \mid \epsilon
\end{aligned}
$$

For brevity, we will give `s` also as an integer literal with implicit conversion. For example, the command PRINT 42 is meant to print out the string of characters `42`.

The formal semantics can be given by a denotation function $[\![-]\!] : p \to \mathit{String}$, where *String* is the set of all strings of characters. In the following, $+\!\!+$ denotes concatenation and `""` denotes the empty string.

$$
\begin{aligned}
[\![\textsc{Skip};\ p]\!] &= [\![p]\!] \\
[\![\textsc{Halt};\ p]\!] &= [\![\epsilon]\!] = \texttt{""} \\
[\![\textsc{Print}\ \texttt{s};\ p]\!] &= \texttt{s} +\!\!+ [\![p]\!]
\end{aligned}
$$

For example:

$$\begin{aligned}
[\![\textsc{Print}\ \texttt{"studio"};\ \textsc{Skip};\ \textsc{Print}\ 54]\!] &= \texttt{studio54}\\
[\![\textsc{Print}\ \texttt{"nuts"};\ \textsc{Halt};\ \textsc{Print}\ \texttt{"and bolts"}]\!] &= \texttt{nuts}
\end{aligned}$$

For every natural number n, we construct a Skip-Halt-Print program that solves FizzBuzz for n. The building blocks for this construction are called *contexts* – they are programs with holes (one hole per context). We denote contexts by putting angle brackets $\langle - \rangle$ around the programs, while holes are denoted by the $\bullet$ symbol. For example:

$$\langle \textsc{Print}\ \texttt{"keep"};\ \bullet;\ \textsc{Print}\ \texttt{"calm"} \rangle$$

A hole is a place in which we can stick another program and get a new program as a result. We denote this operation by juxtaposition:

$$\begin{aligned}
&\langle \textsc{Print}\ \texttt{"keep"};\ \bullet;\ \textsc{Print}\ \texttt{"calm"} \rangle\ (\textsc{Print}\ \texttt{"nervous and never"})\\
&= \textsc{Print}\ \texttt{"keep"};\ \textsc{Print}\ \texttt{"nervous and never"};\ \textsc{Print}\ \texttt{"calm"}
\end{aligned}$$

Two contexts can be composed, and for that we use the $\circ$ symbol:

$$\begin{aligned}
&\langle \textsc{Skip};\ \bullet;\ \textsc{Print}\ 0 \rangle \circ \langle \textsc{Halt};\ \bullet \rangle\\
&= \langle \textsc{Skip};\ \textsc{Halt};\ \bullet;\ \textsc{Print}\ 0 \rangle
\end{aligned}$$

What does the FizzBuzz program do? Essentially, it prints out n, unless something else (like *fizzing* or *buzzing*) happens. This behaviour is captured by the following context:

$$\mathit{base}(n) = \langle \bullet;\ \textsc{Print}\ n \rangle$$

What about *fizzing*? If only n is divisible by 3, it prints out `fizz`, but it also needs to prevent the default action from happening by HALT-ing the computation. In between PRINT-ing and HALT-ing anything (like *buzzing*) can happen:

$$\mathit{fizz}(n) = \begin{cases} \langle \textsc{Print}\ \texttt{"fizz"};\ \bullet;\ \textsc{Halt} \rangle & \text{if } n \bmod 3 = 0\\ \langle \bullet \rangle & \text{otherwise} \end{cases}$$

The context for *buzzing* is analogous:

$$\mathit{buzz}(n) = \begin{cases} \langle \textsc{Print}\ \texttt{"buzz"};\ \bullet;\ \textsc{Halt} \rangle & \text{if } n \bmod 5 = 0\\ \langle \bullet \rangle & \text{otherwise} \end{cases}$$

The program that solves FizzBuzz for n, which we call $fb(n)$, is a composition of all these contexts, which we cork with $\textsc{Skip}$:

$$fb(n) = (base(n) \circ fizz(n) \circ buzz(n))\ \textsc{Skip}$$

Examples:

$$\begin{aligned}
fb(1) &= \textsc{Skip};\ \textsc{Print}\ 1\\
fb(3) &= \textsc{Print}\ \texttt{"fizz"};\ \textsc{Skip};\ \textsc{Halt};\ \textsc{Print}\ 3\\
fb(5) &= \textsc{Print}\ \texttt{"buzz"};\ \textsc{Skip};\ \textsc{Halt};\ \textsc{Print}\ 5\\
fb(15) &= \textsc{Print}\ \texttt{"fizz"};\ \textsc{Print}\ \texttt{"buzz"};\ \textsc{Skip};\ \textsc{Halt};\ \textsc{Halt};\ \textsc{Print}\ 15
\end{aligned}$$

Exercise 1. For any natural number n, give a Skip-Halt-Print program that solves the FizzBuzzHissHowl problem for n.

Exercise 2. What is the formal definition of the operations on contexts described above? Show that contexts with composition form a monoid; that is, $\circ$ is associative, $(f \circ g) \circ h = f \circ (g \circ h)$, and $\langle \bullet \rangle$ is its left and right unit, $\langle \bullet \rangle \circ f = f$ and $f \circ \langle \bullet \rangle = f$ respectively.

Haskell implementation

Now, to solve FizzBuzz in Haskell, we implement Skip-Halt-Print, both syntax and semantics, together with the language of contexts. For each n, we construct the right composition of contexts as described above and then execute the resulting program.

Then, we apply a series of algebraic transformations that simplify the code into our proposed solution. By "algebraic", we mean transformations that depend only on local properties of the components, without the actual understanding of the implemented algorithm. In other words, something that can be deduced solely from the shape of the code, like the *fold* pattern, and applied by simple equational calculation.

Direct definition

The commands of Skip-Halt-Print are implemented as a three-constructor data type *Cmd*, and the program is, of course, a list of commands. We call the $[\![-]\!]$ function *interp*.

```
data Cmd     = Skip | Halt | Print String
type Program = [Cmd]
```

```
interp :: Program → String
interp (Skip    : xs) = interp xs
interp (Halt    : xs) = ""
interp (Print s : xs) = s ++ interp xs
interp []             = ""
```

Contexts are more tricky. Instead of specifying their syntax and operations, we encode them as functions from programs to programs (this technique is sometimes called *higher-order abstract syntax*). In this case, sticking the program in a context becomes Haskell's function application, and the composition of contexts becomes simply Haskell's ∘. However, note that not every Haskell function of the type *Program* → *Program* is a valid context in the sense specified in the previous section.

```
type Cont = Program → Program

fizz, buzz, base :: Int → Cont
fizz n | n `mod` 3 ≡ 0 = λx → [Print "fizz"] ++ x ++ [Halt]
       | otherwise     = id
buzz n | n `mod` 5 ≡ 0 = λx → [Print "buzz"] ++ x ++ [Halt]
       | otherwise     = id
base n                 = λx → x ++ [Print (show n)]

fb :: Int → Program
fb n = (base n ∘ fizz n ∘ buzz n) [Skip]

fizzbuzz :: Int → String
fizzbuzz n = interp (fb n)
```

Interpretation is a fold

To solve FizzBuzz for n, we first build a program (a datastructure) and then interpret it (by traversing the datastructure). This calls for some deforestation – the removal of the intermediate structures! First, we notice a known pattern here: *interp* is a fold. We can rewrite it as follows:

```
step :: Cmd → String → String
step Skip      t = t
step Halt      t = ""
step (Print s) t = s ++ t

interp = foldr step ""
```

Additionally, *foldr* has the following property (see Exercise 3):

$$\mathit{foldr}\ \mathit{step}\ \texttt{""}\ p = \mathit{foldr}\ (\circ)\ \mathit{id}\ (\mathit{fmap}\ \mathit{step}\ p)\ \texttt{""}$$

So, instead of writing programs like

$$[\mathit{Skip}, \mathit{Halt}, \mathit{Print}\ \texttt{"c"}]$$

and interpreting them by folding with *step*, we can write programs like

$$[\mathit{step}\ \mathit{Skip}, \mathit{step}\ \mathit{Halt}, \mathit{step}\ (\mathit{Print}\ \texttt{"c"})]$$

and interpret them by folding $\circ$. Also, we can inline the definition of *step*:

$$\begin{aligned} & [\mathit{step}\ \mathit{Skip}, \mathit{step}\ \mathit{Halt}, \mathit{step}\ (\mathit{Print}\ \texttt{"c"})] \\ = {} & [\lambda t \to t, \lambda t \to \texttt{""}, \lambda t \to \texttt{"c"} +\!\!+\ t] \\ = {} & [\mathit{id}, \mathit{const}\ \texttt{""}, (\texttt{"c"} +\!\!+)] \end{aligned}$$

Why build and then interpret? We can manually deforest the situation by fusing the two: instead of

$$\mathit{foldr}\ (\circ)\ \mathit{id}\ [\mathit{id}, \mathit{const}, (\texttt{"c"} +\!\!+)]$$

we write

$$\mathit{id} \circ \mathit{const} \circ (\texttt{"c"} +\!\!+)$$

In summary, we can define the next version of Skip-Halt-Print commands as follows:

$$\begin{aligned} & \mathbf{type}\ \mathit{Program} = \mathit{String} \to \mathit{String} \\ & \mathit{skip}, \mathit{halt} :: \mathit{Program} \\ & \mathit{skip} \ \ = \mathit{id} \\ & \mathit{halt} \ \ = \mathit{const}\ \texttt{""} \\ & \mathit{print} \quad :: \mathit{String} \to \mathit{Program} \\ & \mathit{print} = (+\!\!+) \end{aligned}$$

Now, our programs look like this:

$$\mathit{print}\ \texttt{"hello"} \circ \mathit{skip} \circ \mathit{print}\ \texttt{"world"} \circ \mathit{halt}$$

To execute them, we apply them to an empty string, for example:

$$(\mathit{print}\ \texttt{"hello"} \circ \mathit{skip} \circ \mathit{print}\ \texttt{"world"} \circ \mathit{halt})\ \texttt{""} = \texttt{"helloworld"}$$

We need to accordingly adjust the bodies of our contexts:

```
type Cont = Program → Program

fizz, buzz, base :: Int → Cont
fizz  n | n ‘mod‘ 3 ≡ 0 = λx → print "fizz" ∘ x ∘ halt
        | otherwise     = id
buzz n  | n ‘mod‘ 5 ≡ 0 = λx → print "buzz" ∘ x ∘ halt
        | otherwise     = id
base n                  = λx → x ∘ print (show n)

fizzbuzz :: Int → String
fizzbuzz n = (base n ∘ fizz n ∘ buzz n) skip ""
```

Notice that ∘ is now overloaded: it composes both programs from commands (as in the bodies of functions *fizz*, *buzz*, and *base*) and contexts (as in the body of *fizzbuzz*).

Inlining

The truth is that we do not need to implement the entire Skip-Halt-Print language to solve FizzBuzz – our three contexts suffice. Thus, we inline the definitions of *base*, *skip*, *halt*, and *print* in *fizzbuzz*. We also put *fizz* and *buzz* as local definitions, so that we don't have to pass *n* around:

```
fizzbuzz :: Int → String
fizzbuzz n = (fizz ∘ buzz) id (show n)
where
  fizz | n ‘mod‘ 3 ≡ 0 = λx → const ("fizz" ++ x "")
       | otherwise     = id
  buzz | n ‘mod‘ 5 ≡ 0 = λx → const ("buzz" ++ x "")
       | otherwise     = id
```

Final polishing

As the last step, we abstract over the divisor and the printed message in *fizz* and *buzz*:

```
fizzbuzz :: Int → String
fizzbuzz n = (test 3 "fizz" ∘ test 5 "buzz") id (show n)
  where
    test d s x | n ‘mod‘ d ≡ 0 = const (s ++ x "")
               | otherwise     = x
```

What is going on in this program? The (higher-order) function *test* has the following, longish type:

$$test :: Int \to String \to (String \to String) \to String \to String$$

To understand its logic, it is convenient to name the last argument and rewrite the function to the following equivalent definition:

$$\begin{aligned} test\ d\ s\ x\ v \mid & n \text{ `}mod\text{` } d \equiv 0 = s +\!\!+ x\ \texttt{""} \\ \mid & otherwise \qquad = \qquad x\ v \end{aligned}$$

The argument $v :: String$ represents the default value of the function (originally set by the function *fizzbuzz* to the string representation of n), while $x :: String \to String$ represents a continuation – the rest of the computation parametrised by a new default value. If the modulus test fails, we change neither the continuation nor the default value. If the test succeeds, we print out the string s, but also change the default value to the empty string, so that the string representation of n is not printed out.

Exercises

Exercise 3. Prove that for $f :: t \to s \to s$ and $a :: s$, the following equality holds:

$$foldr\ f\ a\ xs = foldr\ (\circ)\ id\ (fmap\ f\ xs)\ a$$

Exercise 4. In the "Inlining" step, we silently performed some cleaning-up. In reality, a bald inlining of *base* and *skip* in *fizzbuzz* yields

$$((\lambda x \to x \circ (+\!\!+)\ (show\ n)) \circ fizz \circ buzz)\ id\ \texttt{""}$$

Show that it is equal to $(fizz \circ buzz)\ id\ (show\ n)$.

Exercise 5. Adjust the final solution to the FizzBuzzHissHowl problem. Do you have to go through the entire derivation once more, or is the final solution modular?

Summary

To solve a trivial problem, we went through a bit of a hassle: formal language design and semantics, embedded DSLs, interpreters, higher-order abstract syntax to implement contexts, algebra of programming in the form of reasoning about folds. One might also argue that the obtained solution is not too intuitive. Do we really need such heavy artillery to solve FizzBuzz?

Though this tutorial is not meant to be dead serious and is mostly a pretext for some fun with the functional programming technologies listed above – also, going through this derivation might be a risky move during a job interview –

there is a small point it wants to convey: **Functional programmers! Remember higher-order functions!** They are your tool to express programs with non-trivial structure, to follow the information-flow more closely, to dynamically build your programs in runtime. A harsh, cantankerous functional programming pedagogue might say that they are such a basic tool that the final FizzBuzz program shouldn't appear complicated at all (and could be easily written by hand) if one knows their paradigm.

Closing remarks

The Skip-Halt-Print language is based on Edsger W. Dijkstra's Skip-Abort, which can be found in his textbook *A Discipline of Programming* [2, Chapter 4] (I am grateful to Tomasz Wierzbicki for the reference.) However, one needs to be aware of a difference in semantics between HALT and ABORT: the former peacefully ends the computation (like `return` in *C*-like languages), while the latter atrociously breaks it (like Haskell's *error* function).

I would also like to thank Jeremy Gibbons for his comments. The first idea for this tutorial sparkled in my head after Laurence E. Day's Facebook post:

> I can write doctoral level Haskell without so much as missing a beat,
> but I'd have a genuinely hard time writing a FizzBuzz program in Java.

The main point of this tutorial is that FizzBuzz is **not at all trivial**, but, many thanks to higher-order functions in Haskell, solvable. I don't know about Java.

References

[1] Wikipedia. `http://en.wikipedia.org/wiki/Fizz_buzz`.

[2] Edsger W. Dijkstra. **A discipline of programming**. Prentice-Hall series in automatic computation, Prentice-Hall, Incorporated (1976).