From Löb's Theorem to Spreadsheet Evaluation

Dan Piponi

February 2024 (originally November 2006)

As I've mentioned in the past, sometimes you can write useful Haskell code merely by writing something that type checks successfully. Often there's only one way to write the code to have the correct type. Going one step further: the Curry-Howard isomorphism says that logical propositions corresponds to types. So here's a way to write code: pick a theorem, find the corresponding type, and find a function of that type.

So I'm looking at the books on my shelf and there's *The Logic of Provability* by Boolos. It's about a kind of modal logic called provability logic in which $\Box a$ roughly means "a is provable". One of the axioms of this logic is a theorem known as Löb's theorem.

Before getting onto Löb's Theorem, I should mention *Haskell's Paradox*[1]. It goes like this:

Let S be the proposition "If S is true, Santa Claus exists".

Suppose

$$S \text{ is true.}$$

Then

$$\text{If } S \text{ is true, Santa Claus exists.}$$

So, still assuming our hypothesis, we have

$$S \text{ is true and if } S \text{ is true, Santa Claus exists.}$$

[1]Usually known by Haskell Curry's last name.

Therefore

S is true and if S is true, Santa Claus exists.

And hence

Santa Claus exists.

In other words, assuming S is true, it follows that Santa Claus exists. In otherwords, we have proved

If S is true then Santa Claus exists.

regardless of the hypothesis. But that's just a restatement of S so we have proved

S is true.

and hence that

Santa Claus exists.

Fortunately we can't turn this into a rigorous mathematical proof. We can try. We have to use some kind of Gödel numbering scheme to turn propositions into numbers and then if a proposition has number g, we need a function True so that True(g)=1 if g is the Gödel number of something true and 0 otherwise. But because of Tarski's proof of the indefinability of truth, we can't do this. On the other hand, we can replace True with Provable, just like in Gödel's incompleteness theorem. If we do this, the above argument (after some work) turns into a valid proof - in fact, a proof of Löb's theorem. Informally it says that if it is provable that "P is provable implies P" then P is provable. We did something similar above with $P =$ "Santa Claus exists". In other words

$$\Box(\Box P \to P) \to P$$

So I'm going to take that as my theorem from which I'll derive a type. But what should $\Box$ become in Haskell? Let's take the easy option, let it be anything:

```
import Maybe

class Lob a where
    lob :: a (a x -> x) -> a x
```

So if a is of type class Lob, then there is a function lob that is of the appropriate type with a playing the role of the provability predicate.

So now to actually find an instance of this.

Suppose a is some kind of container. The argument of `lob` is a container of functions. They are in fact functions that act on the return type of `lob`. So we have a convenient object for these functions to act on, we coprophagistically feed the return type of `lob` to each of the elements of the argument in turn. Haskell, being a lazy language, doesn't mind. Here's a possible implementation:

```
instance Lob [] where
    lob x = map (\a -> a (lob x)) x
```

Informally you can think of it like this: the parts are all functions of the whole and `lob` resolves the circularity. Anyway, when I wrote this, I had no idea what `lob` did.

So here's one of my first examples:

```
test1 = [\x -> x!!1, length]
```

`lob test1` is `[2, 2]`. We have set the first element to equal the second one and the second one is the length of the list. Even though element 0 depends on element 1 which in turn depends on the size of the entire list containing both of them, this evaluates fine. Note the neat way that the first element refers to something outside of itself, the next element in the list. To me this suggests the way cells in a spreadsheet refer to other cells. So with that in mind, here is an instance for `Num` I found on the web. (I'm sorry, I want to credit the author but I can't find the web site again):

```
instance Show (x -> a)
instance Eq (x -> a)

instance (Num a,Eq a) => Num (x -> a) where
    fromInteger = const . fromInteger
    f + g = \x -> f x + g x
    f * g = \x -> f x * g x
    negate = (negate .)
    abs = (abs .)
    signum = (signum .)
```

With these definitions we can add, multiply and negate `Num` valued functions. For example:

```
f x = x * x
g x = 2 * x+1

test2 = (f + g) 3
```

Armed with that we can define something ambitious like the following:

```
test3 = [ (!!5), 3, (!!0) + (!!1), (!!2) * 2,
          sum . take 3, 17]
```

Thing of it as a spreadsheet with `(!!n)` being a reference to cell number `n`. Note the way it has forward and backward references. And what kind of spreadsheet would it be without the `sum` function? To evaluate the spreadsheet we just use `lob test`. So `lob` is the spreadsheet evaluation function.

Who'd have guessed that a useful function like this is what we'd get from Löb's theorem?

1 Afterword

See Kenneth Foner's Functional Pearl Getting a quick fix on comonads.