Notes for Lecture 13 (Fall 2022 week 6, part 2): Polymorphism: ‘Maybe’ type

Jana Dunfield

October 16, 2022

The code for this lecture is in `lec13.hs`.

1 Polymorphism, continued

As seen in lec11, Haskell lets us define ‘data’ types that are generic (polymorphic). Here is another example of that:

```
data Sequence a = Last a
                | Elem a (Sequence a)
```


This defines a type with two constructors: `Last` contains something of type ‘a’, and `Elem` contains something of type ‘a’ and another `Sequence`. For example, a value of the type `Sequence Bool` would be


```
Elem False (Elem True (Last False))
```

If we think of `Elem` as ‘cons’ (`:`), and `Last` as ‘nil’ (`[]`), this is almost the same as a Haskell list. But it’s not quite the same! A list can be empty; an empty list has no elements. Something of type `Sequence a`, however, must contain at least one element: the smallest `Sequence a` is `Last e` where `e` has type `a`. (Perhaps confusingly, we can write


```
Last
```

but it will have type `a -> Sequence a`; it is a function that waits for something of type `a` and then builds a one-element sequence.)

To see how to work with `Sequence`s, let’s rewrite `mymap` (from lec12.hs) to work on sequences instead of lists. The definition of `mymap` was:

```
mymap :: (a -> b) -> [a] -> [b]
mymap f []       = []
mymap f (x : xs) = (f x) : (mymap f xs)
```

To convert this, I start by renaming the function and changing its type declaration. `[a]` is a list of `a`s; `Sequence a` is a sequence of `a`s:

```
seqmap :: (a -> b) -> Sequence a -> Sequence b
seqmap f []       = []
seqmap f (x : xs) = (f x) : (seqmap f xs)
```

This won't compile, because the clauses that define `seqmap` still use lists. Next, I rewrite the second clause. `x : xs` becomes `Elem x xs`, and similarly in the right-hand side.

```
seqmap :: (a -> b) -> Sequence a -> Sequence b
seqmap f []          = []
seqmap f (Elem x xs) = Elem (f x) (seqmap f xs)
```

This still won't compile because the first clause uses a list. We can actually delete the first clause: a `Sequence` can't be empty, so there is no case corresponding to the empty list `[]`.

```
seqmap :: (a -> b) -> Sequence a -> Sequence b
seqmap f (Elem x xs) = Elem (f x) (seqmap f xs)
```

Attempting to test `seqmap` will show that we're not done:

```
*Lec13> seqmap (\x -> x + 1) (Elem 2 (Last 10))
Elem 3 *** Exception: lec13.hs:18:1-47:
           Non-exhaustive patterns in function seqmap
```


Haskell tried to pattern-match with the argument `Last 10`, but since the only pattern is `(Elem x xs)`, Haskell "fell off" the end of the function. "Non-exhaustive" means "incomplete": we didn't write a pattern that matches `Last ...`, so our definition of `seqmap` is incomplete.

To fix this, we need to add a clause whose pattern is `Last y`:

```
seqmap :: (a -> b) -> Sequence a -> Sequence b
seqmap f (Elem x xs) = Elem (f x) (seqmap f xs)
seqmap f (Last y)    = undefined
```


I want to apply the given function `f` to the last element as well as the earlier elements. (Functions like `map`, `mymap`, `seqmap` are meant to be widely useful; maybe there would be a situation where I'd want to apply a function to every element *except* the last one, but if I needed to do that, I would write another function to do it.)

So, given `Last y`, I return `Last (f y)`.

```
seqmap :: (a -> b) -> Sequence a -> Sequence b
seqmap f (Elem x xs) = Elem (f x) (seqmap f xs)
seqmap f (Last y)    = Last (f y)
```

Now, I can call `seqmap`:

```
*Lec13> seqmap (\x -> x + 1) (Elem 2 (Last 10))
Elem 3 (Last 11)
```

We have now defined a type that is very similar to lists, but that doesn't allow "nothing": something of type `Sequence a` must have at least one thing of type `a` in it.

2 'Maybe' type

2.1 What's it for?

Let's turn this around. What if we want to represent "nothing"?

Software often needs to represent the absence of information. For example, if we are writing backup software, we probably want to store the time of the last backup. But if we haven't yet backed up, no such time exists. Assuming we have a type `Time`[1] We could represent this in Haskell:

```
type Time = Integer
data BackupTime = Never
                | BackedUp Time
```

A less clear alternative would be to represent the last backup by `Time` alone, and have a value of zero represent "never". This is prone to error: unless we remember to check for a value of zero, our backup system will probably display something like "Last backed up: on Jan 1 1970". (It also can't represent the situation where we really did last do a backup on January 1st, 1970, but that situation is extremely unlikely.)

If we *do* remember to check, we can display "Last backed up: never", but if we used the `BackupTime` type, we can write


```
displayBackupTime :: BackupTime -> String
displayBackupTime Never = "never"
displayBackupTime (BackedUp time) = "on " ++ (show time)
    -- this is going to display an integer, which is not readable,
    --  but I don't want to bother figuring out how to do it
```


Needing to represent "nothing" is so common that Haskell defines a type with a constructor `Nothing` to represent nothing, and a constructor `Just` that takes one argument:

```
data Maybe a = Nothing
             | Just a
```

Instead of defining `BackupTime`, we could use the type `Maybe Time`: `Nothing` would represent "never", and `Just t` would represent "backed up at time `t`".

2.2 Comparison to other languages

Many languages have something that represents "nothing":

```
Python  has  None

  Java  has  null

     C  has  NULL

Pascal  has  nil
```

[1] I won't try to use the actual Haskell time library. For simplicity, imagine that `Time` is "seconds since 00:00, January 1, 1970".

Or, at least, these languages can represent "nothing" in some circumstances:

- In Java, any reference to an `Object` (including any descendant of the `Object` class) can be `null`, but types like `int` cannot be null.
- In C, pointers can be `NULL` but non-pointer types (like `int`, `bool`, `char`) can't be `NULL`.

So you get "nothingness" for some types, but not others.

Programming in languages that have features like `None` and `null` is tricky, because you have to be careful that when you think you have something—say, an IP address—you actually have an IP address, and not `None` or `null`.

It's difficult to be careful all the time.

The computer scientist Tony Hoare, who put `null` into the language Algol-W in 1965, called it a "billion-dollar mistake":

> "My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years."
>
> `https://en.wikipedia.org/wiki/Tony_Hoare`

When I program in a language with `null`, I always worry that things I think I see—arguments to methods, instance variables—aren't really there; they could be `null`. So I have to check whether they're `null` before I do anything with them.

In Haskell, `null`-ness is signalled by the `Maybe` type. If a function takes an `ArithExpr`, I don't need to check if it's really an `ArithExpr`. I get to represent "nothing" only when I need to.

2.3 The sad story of the NULL licence plate

Some of the problems around "null" values can't exactly be blamed on the programming language. For example:

`https://www.wired.com/story/null-license-plate-landed-one-hacker-ticket-hell/`

I think the problem here is the use of a string, which is information, to represent the absence of information. If Haskell were as popular as Java, we might have the same problem with a licence plate that read "NOTHING".

2.4 Finding in trees

`lec13.hs` includes a function `find1` that searches for a key in a binary tree type `Tree`.

```
data Tree = Empty
          | Branch Tree (Integer,String) Tree
          deriving (Show, Eq)
```

If the key is not found, `find1` returns `Nothing`. If the key is found, `find1` returns the associated `String`.