Notes for Lecture 11 (Fall 2022 week 5, part 3): Polymorphism

Jana Dunfield

October 5, 2022

The code for this lecture is in `lec11.hs`.

1 Polymorphism

Some Haskell functions are polymorphic ("many forms"): they work with arguments (and/or results) of more than one type.

(Some languages also call this "polymorphism", some call it "generics".)

Consider the lambda that returns its argument:

```
(\x -> x)
```


This very small function doesn't do anything with `x` except return it, so it doesn't care what type `x` has.

We can write


```
(\x -> x) 4
```

and get 4 back, or write


```
(\x -> x) False
```

and get False back, or (recalling lec10 on higher-order functions)

```
(\x -> x) not
```

which returns the built-in function `not`.

Since the result is a function, Haskell will not print it, but we can check that it is indeed `not` by asking for its type

```
*Lec11> :type (\x -> x) not
(\x -> x) not :: Bool -> Bool
```

and by asking what it returns given False or True:

```
*Lec11> ((\x -> x) not) False
True
*Lec11> ((\x -> x) not) True
False
```

(We usually can't completely, or "exhaustively", test a function. But there are only two possible values of type `Bool`, `False` and `True`. So we can try all of them. This kind of testing is not always reliable in other languages, but it is reliable in Haskell.)

If we ask for the type of `(\x -> x)`, we get something that needs explanation:

```
*Lec11> :type (\x -> x)
(\x -> x) :: r -> r
```

What type is `r`? It stands for any type, but what exactly does that mean?

I think the best way to read the above type is to add the following, which I hope shows up correctly (it is 2021 but we still can't rely on computers displaying the right symbol):

```
∀r (r -> r)
```

If it doesn't show up, or if you don't remember what ∀ means, this says:

```
for all r, r -> r
```

Another reading is: Give me a type `r`, then I am a function of type `r -> r`. Another: Give me a type `r`, and an argument of type `r`, and I return a result of type `r`.

I find it annoying that H [illegible] e logic, the formulas


```
R -> R
```

and


```
∀r (r -> r)
```

don't mean the same thing.

Haskell decides which types are literal names ("constants", in the logical sense) and which types are "variables" ("type variables") based on how they're written:

`r`	begins with a lowercase letter, so it is a type variable
`R`	begins with an uppercase letter, so it is a type name
`Int`	begins with an uppercase letter, so it is a type name
`a32`	begins with a lowercase letter, so it is a type variable
`aKBZX`	begins with a lowercase letter, so it is a type variable

When I see the type `r -> r`, I mentally rewrite it to

```
∀r (r -> r)
```

For types with multiple type variables, like

```
(a -> b) -> a -> b
```

we add "for all"s for every type variable:

```
∀a ∀b ((a -> b) -> a -> b)
```

You may recall from logic that the order of "for all" quantifiers doesn't matter, as long as they're all grouped together, so it doesn't matter whether we think of it as

```
∀a ∀b ((a -> b) -> a -> b)
```

or as

```
∀b ∀a ((a -> b) -> a -> b)
```

Exercise 1. Jumping ahead a little: Copy the following two lines into a file and try to write an expression to replace `undefined`:

```
hmm :: (a -> b) -> a -> b
hmm f x = undefined
```

Think about what you have, and what you are trying to return: `hmm` takes `f :: a -> b`, and `x :: a`, and you want to return something of type `b`.

Lec11 ended by asking you to comment out the type declaration for `mymap`, which was:

```
-- mymap f [3, 2, 1] == [f 3, f 2, f 1]
--  where f is a function of type In
mymap :: (Integer -> Integ
```


As we can see by asking Haskell `:type mymap`, Ha


```
(t -> t1) -> [t] -> [t1]
```

Both `t` and `t1` begin with a lowercase letter, so they are type variables and we should read the type as

```
∀t ∀ t1 ((t -> t1) -> [t] -> [t1])
```

The names `t` and `t1` aren't the ones I would have chosen, so I've declared the same type with different type variable names:

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

This type declaration says:

For all types 'a' and 'b',
given a function from 'a' to 'b',
and a list whose elements are of type 'a',
`mymap` returns a list whose elements are of type 'b'.

We originally declared `mymap` to take a function `Integer -> Integer`, and a list of Integers, and return a list of Integers. But the two lines of code in `mymap` don't do anything integer-related! Any integer operations will be done by the function `Integer -> Integer`.

Since `mymap` itself doesn't do anything that requires the elements of the argument to be integers, `mymap` can have a much more general type.

First, let's check that we don't lose anything by declaring a more general type for `mymap`. For example, can we still multiply elements by 9?

```
multiply_list_by_9 :: [Integer] -> [Integer]
multiply_list_by_9 = mymap (\y -> y * 9)
```

`lec11.hs` loaded, so the answer should be yes (try an example if you like). And this makes sense if we think of a type as a logical formula: the logical formula

$$P(I, I)$$

is an instance of the formula

$$\forall a\ \forall b\ P(a, b)$$

In predicate logic, if we assume $\forall a\ \forall b\ P(a, b)$, we can prove $P(I, I)$ using $\forall$-elimination. The assumption $\forall a\ \forall b\ P(a, b)$ is more powerful than $P(I, I)$, because it says P holds for all (a, b), not only for (I, I). Likewise, the type `(a -> b) -> [a] -> [b]` is more powerful than the type `(Integer -> Integer) -> [I`

Some examples of usi


```
strictly_positive :: [
strictly_positive = mymap (\x -> x > 0)
-- strictly_positive [-3, 1, 4, 0] == [False, True, True, False]
```


Exercise 2. Exercise: figure out what the type of `negate_elems` e, then check your answer using `:type`.

```
negate_elems = mymap not
-- negate_elems [False, False, True, False] == [True, True, False, True]
```

Exercise 3. Exercise: figure out what the type of `addAtoZ` should be, then check your answer using `:type`.

```
addAtoZ = mymap (\s -> "A" ++ s ++ "Z")
-- addAtoZ ["b", "aaa", "A", "QU", "uu"] == ["AbZ", "AaaaZ", "AAZ", "AQUZ", "AuuZ"]
```

You might have noticed that we've used various kinds of lists: `strictly_positive` takes a list of Integers (type `[Integer]`), and returns a list of Booleans (type `[Bool]`). Lists are, therefore, polymorphic data types.

We can declare our own polymorphic data types.

This is a binary tree that stores keys only at the leaves:

```
data Tree a = Branch (Tree a) (Tree a)
            | Leaf a
            deriving Show
```

If we ask for the types of Branch and Leaf, we get

```
*Lec11> :type Branch
Branch :: Tree a -> Tree a -> Tree a
*Lec11> :type Leaf
Leaf :: a -> Tree a
```

In my mind, I add "for all 'a'" to these types:

```
Branch :: ∀a (Tree a -> Tree a -> Tree a)
Leaf   :: ∀a (a -> Tree a)
```

Thinking of ∀-elimination, I see that I can use Leaf to create trees (very small trees) whose leaves have different kinds of keys in them:

```
*Lec11> :type Leaf True
Leaf True :: Tree Bool
*Lec11> :type Leaf (False, True)
Leaf (False, True) :: Tree (Bool,Bool)
*Lec11> :type Leaf 4
Leaf 4 :: Num a => Tree a
```


The last one is a little com e a provided that 'a' is a numeric type.

1.1 More 204 stuff, skip if you want

Again, I read it as ∀a `(Num a => Tree a)`:

> "For all types 'a', if 'a' is numeric,
> then I have type "`Tree a`".

Aside: In 204 (if you took it with me, anyway), we translated English sentences like

> "Every ghost is a cat"

to

$$\forall x(G(x) \rightarrow C(x))$$

which is literally, "For all x, if x is a Ghost, then x is a Cat."

The type ∀a `(Num a => Tree a)` has a similar structure, and the => can still be read as an implication, but in Haskell types we're always talking about "what something is": the type `Num a => Tree a` is about `Leaf 4`. The type ∀a `(Num a => Tree a)` doesn't mean "for all a, if a is a `Num` then a is a `Tree`"; it means "for all a, if a is a `Num` then `Leaf 4` is a `Tree a`."

1.2 Back from 204

We can even create trees whose leaves contain other trees:

```
*Lec11> :type Leaf (False, Leaf 'c')
Leaf (False, Leaf 'c') :: Tree (Bool, Tree Char)
```

This is a tree (consisting of a single leaf) where the key stored in that leaf is a pair of a Boolean and a tree that stores characters.

Exercise 4. Write an expression of type `Tree (Bool, Tree Char)` that uses the Branch constructor at least twice.

When we talk about polymorphic data types in type declarations, they have to be "instantiated": `Tree` by itself is not a type.

```
*Lec11> (Leaf 3) :: Tree
<interactive>:58:13:
    Expecting one more argument to 'Tree'
    Expected a type, but 'Tree' has kind '* -> *'
    In an expression type signature: Tree
    In the expression: (Leaf 3) :: Tree
    In an equation for '
```


This is another GHC err etty clear once you know that Ha another type, like Integer or Bool, and produces a type.

`Tree Integer`	tree with integer keys
`Tree Bool`	tree with boolean keys

If this made sense to you, you can skip the following. If you really want to know what "`* -> *`" means, read on:

The second line, "`Expected a type, but 'Tree' has kind '* -> *'`", is probably less clear. "`*`" means "a type"; "`* -> *`" is a function that takes a type ("`*`") and returns a type ("`*`"). Tree will give us an actual type if we give it a type:

`Tree Char`	tree with character keys

In ordinary English, "type" and "kind", as nouns, are pretty much synonyms. Haskell uses "kind" in the error message because talking about "the type of a type" would be confusing. Types classify data; kinds classify types.

Exercise 5. Describe, in English, the elements of the type `Tree (String, String)`.

Exercise 6. Try to guess what `mymap Leaf` does. Then check your guess by typing

```
mymap Leaf [True, True, False]
```

Is `mymap Leaf` any different from `mymap (\k -> Leaf k)`?