

Notes for Lecture 8 (Fall 2022 week 4): 'data' types

Jana Dunfield

September 25, 2022

1 Continuing discussion of recursive functions

See `lec8.py`.

2 'data' types

Haskell code for this lecture is in `lec8.hs`.

Type synonyms give names for existing types. To create something new, somewhat similarly to declaring a class in an object-oriented language, we use the 'data' keyword:

```
{-
```

```
  3. defining new data types ("data ...")
```

```
Time to start declaring our own types:
```

```
This says: Building is a data type
```

```
The values of this type are:
```

```
  Goodwin, WalterLight, BeamishMunroe, Dupuis
```

```
-}
```

```
data Building = Goodwin
```

```
              | WalterLight
```

```
              | BeamishMunroe
```

```
              | Dupuis
```

```
              -- | lowercase    data constructors must start with uppercase
```

```
              deriving Show
```

(This example feels strange; I haven't been inside a campus building in a while.)

Each of the things after the `=` is a *data constructor*, often called just a *constructor*. At a high level, the 'data' declaration says:

- this is what a Building is:
 - Goodwin is a Building
 - WalterLight is a Building
 - BeamishMunroe is a Building
 - Dupuis is a Building
- also, please print the constructor names when you need to ("`deriving Show`")

(Having to write “deriving Show” gets pretty annoying. It might annoy me even more than it annoys you.)

In an object-oriented language, we might define `Building` by declaring an abstract class `Building` and deriving the four classes `Goodwin`, `WalterLight`, ... as subclasses of `Building`. There are many differences between using a class and using a Haskell ‘data’ type; one important difference is that when we use ‘data’, we are defining the entire type at once. We have to list *all* the possible constructors. In most object-oriented languages, we can keep defining new subclasses (possibly scattered throughout various source files).

We have actually seen a ‘data’ type already:

```
data Bool = True
          | False
          deriving Show    -- this line is actually more complicated
```

(You can do other things with “deriving” than tell Haskell to print the constructor names, but I don’t want to get into that right now.)

If we type a constructor name into GHCi, it returns it:

```
*Lec7> Goodwin
Goodwin
```

That doesn’t seem very interesting. This is a little more interesting:

```
has_elevator :: Building -> Bool
```

```
has_elevator Goodwin = True
has_elevator WalterLight = False
has_elevator BeamishMunroe = True
has_elevator Dupuis = True
```

This function tells us whether a given building has an elevator.

■ Remark 1. Building history digression:

According to the stories I’ve heard, Walter Light does not have an elevator because every floor is connected to every floor of Goodwin, which does have an elevator. Goodwin was supposed to have two elevators, and in fact has two elevator shafts. The second elevator was supposed to be installed when Walter Light was completed, but they ran out of money building Walter Light, so we have one elevator for two buildings.

When it’s safe for you to be in Goodwin Hall again—whenever that is—look for evidence of the second elevator shaft on the second floor of Goodwin.

How does the function work? It might be surprising that Haskell is okay with this function: it seems to define itself four times, sort of like a function that uses guards—but `has_elevator` doesn’t use guards.

Actually, `has_elevator` uses pattern matching, but instead of only “lining up” the components of tuples, Haskell is comparing the argument to the pattern. Here’s what Haskell does when we step `has_elevator WalterLight`. (See the next page, to keep it all on one page.)

```
has_elevator WalterLight
=> ????
```

Line up the FIRST CLAUSE, `has_elevator Goodwin = ...`
with the expression:

```
has_elevator Goodwin
has_elevator WalterLight
```

Does `WalterLight` match the pattern `Goodwin`?
Equivalently: Is there any way to make

`WalterLight` equal to `Goodwin`?

Answer: no.

Move on to the SECOND CLAUSE, `has_elevator WalterLight = ...`
Line up with the expression:

```
has_elevator WalterLight    -- second clause
has_elevator WalterLight    -- expression
```

Is there any way to make

`WalterLight` equal to `WalterLight`?

Yes, sure, they're identical.

The pattern `WalterLight` matched the argument `WalterLight`,
so we step to the right-hand side of

```
has_elevator WalterLight = False
```

```
has_elevator WalterLight
=> False
```

Pattern matching also works with “more than one argument”. Even though `my_and` is technically a function of one argument that returns a function of type `Bool -> Bool`, Haskell lets us write `my_and` as if it had two arguments. And Haskell lets us pattern-match on both at the same time.

```
my_and :: Bool -> Bool -> Bool
my_and True True = True
my_and True False = False
my_and False True = False
my_and False False = False
```

Similar to `has_elevator`, Haskell tries to match arguments against patterns. Unlike `has_elevator`, Haskell matches two things at once. For example, if we step

```
my_and False True
```

we will first line up

```
my_and True  True = True
my_and False True
```

which doesn't match (the first argument False is definitely not equal to the pattern True). Since it doesn't match, we move on to the second clause.

```
my_and True  False = False
my_and False True
```

Again, it doesn't match (neither argument matches!), so we move on to the third clause, which matches:

```
my_and False True = False
my_and False True
```

and step to the right-hand side of the clause, which is False.

As my example steppings have suggested, Haskell always does pattern matching *in order*. We can take advantage of this to define 'and' more concisely:

```
another_and :: Bool -> Bool -> Bool
another_and True True = True
another_and _ _ = False
```

Anything matches the wildcard pattern `_`, so if we apply `another_and` to arguments that are not both True, we will match the patterns in the second clause and return False.

The function `something` in `lec7.hs` does something like this, but is more complicated:

```
something :: (Building, Building) -> Bool
something (Dupuis, WalterLight) = False
something (Dupuis, _) = True
something (_, WalterLight) = True
something (_, _) = False
```

Experiment in GHCi with `something` applied to various pairs of Buildings, and see if you follow how Haskell is deciding which clause matches.

The code ends with an exercise:

```
-- adjacent
-- True iff buildings are _directly_ adjacent.
-- For example, Goodwin and WalterLight are adjacent,
-- but WalterLight and BeamishMunroe are not because
-- you have to go through Goodwin.
--
-- "Building map": || and === show direct connections
--
```

```
-- Dupuis
--   ||
-- BeamishMunroe===Goodwin===WalterLight
--
-- (I think there's a secret locked one-way door from Dupuis to Goodwin.
-- It doesn't count.)
adjacent :: Building -> Building -> Bool
adjacent b1 b2 = undefined
```

It's a little annoying to write because adjacency is a symmetric relation, so

```
adjacent Goodwin WalterLight = True
```

isn't enough to model the connection; we also want to return True for adjacent WalterLight Goodwin.

3 lec8.hs

Some more material is in lec8.hs.

4 Trees

The Building type from Lecture 8 is similar to an enum in Java or C. Haskell's data declarations can do more than that.

Recall that Goodwin, WalterLight, etc. are *data constructors*, often called just *constructors*. Each constructor of the Building type took no arguments: Goodwin is a Building, with no other information needed.

```
data Building = Goodwin
              | WalterLight
              | BeamishMunroe
              | Dupuis
              | Dunning
              deriving Show    -- tell Haskell to print Goodwin as Goodwin, etc.
```

In this section, we define a type Tree that is not like an enum type:

```
data Tree = Empty
          | Branch Tree Integer Tree
          deriving Show
```

This says: a Tree is either

- Empty, or
- Branch l k r where l is a Tree, k is an Integer, and r is a Tree.

(The line “deriving Show” tells Haskell to allow itself to print Trees.)

The meaning I intend (which Haskell doesn’t know, it only knows what’s in the Tree declaration) is that

- Empty represents a *leaf* (containing no information), and
- Branch l k r represents a *branch* whose left child is the tree l, containing an integer key k, and whose right child is the tree r.

By itself, the word Branch is a constructor but not a tree: we need to know the children of the branch, and the integer key being stored.

Branch Empty 9 Empty is a tree, because we have given three *arguments* to Branch. In fact, the type Haskell gives to Branch is the type of a function:

```
Branch :: Tree -> Integer -> Tree -> Tree
      ^^^^      ^^^^^^^      ^^^^      ^^^^
      first    second    third    result
      argument argument argument
```

As we have seen, Haskell will not print functions, so if you enter Branch by itself into GHCi, it will not print an expression. If you ask GHCi for the type of Branch, you will get the type above.

As with built-in functions and functions we define, we can *partially apply* a constructor:

```
*Lec9> :type Branch Empty
Branch Empty :: Integer -> Tree -> Tree
*Lec9> :type Branch Empty 3
Branch Empty 3 :: Tree -> Tree
*Lec9> let empty_3 = Branch Empty 3
*Lec9> empty_3 Empty
Branch Empty 3 Empty
*Lec9>
```

The function empty_3 is “waiting” for its last argument, the right child.

Remark 2. You can declare something (empty_3) within GHCi by writing `let` before the declaration. Entering code longer than one line is annoying, but for one-line expressions I sometimes find it easier to do within GHCi, rather than doing it in a file and loading the file.

The rest of this section is in `lec8.hs`.