Chapter 8

Declaring Types and Classes

Connor Baker

Compiled on December 23, 2019 at 6:09pm

8.1 Type declarations

We can introduce new names for types that we already have by means of the `type` keyword. As an example, if we wanted to think of an ordered pair of integers as some position, we could do so by writing

```
type Pos = (Int, Int)
```

and we might declare a transformation as a function which maps a `Pos` to another `Pos`

```
type Trans = Pos -> Pos
```

Note: the name of the newly declared type must be capitalized.

`type` declarations are not allowed to be recursive. As an example, `type Tree = (Int, [Tree])` is not allowed. Recursive types may be declared using the `data` mechanism, which is discussed in the next section.

`type` declarations can be parameterized by other types. If we wanted a more generic two-tuple instead of one specifically meant for integers, we could write

```
type Pair a = (a,a)
```

Additionally, we can use more than one parameter. Suppose we wanted a type which is a list of keys associated with values:

```
type Assoc k v = [(k,v)]
```

We could define a function which returns the first value that is associated with a key

```
find :: Eq k => k -> Assoc k v -> v
find k t = head [v | (k',v) <- t, k == k']
```

`find` is a function which takes a key (which must be a type which derives `Eq`), a list of key-value pairs, and returns a value. It works by taking the first element of the list of values which match the key used as an argument.

8.2 Data declarations

A new type, as opposed to a synonym for an existing type, can be declared by specifying its values using the `data` mechanism.

If we wanted to define a new type which describes the cardinal directions, we could do

```
data Move = North | South | East | West
```

The pipe (`|`) symbol is read as *or* and the new values of the type are called *constructors*. The names of new constructors must begin with a capital letter, just like the name of the new types. Additionally, the same constructor name cannot be used in multiple types.

With pattern matching, we can define functions which apply a move to a position, which apply a sequence of moves to a position, and which apply the reverse direction of a move.

```
move :: Move -> Pos -> Pos
move North (x,y) = (x,y+1)
move South (x,y) = (x,y-1)
move East  (x,y) = (x+1,y)
move West  (x,y) = (x-1,y)

moves :: [Move] -> Pos -> Pos
moves [] p = p
moves (m:ms) p = moves ms (move m p)

rev :: Move -> Move
rev North = South
rev South = North
rev East  = West
rev West  = East
```

Note: If you try these in GHCi add `deriving Show` to the end of the data declaration. Without it, GHCi can't display values of the new type.

Constructors can have arguments. As an example, consider the `Shape` data type:

```
data Shape = Circle Float | Rect Float Float
```

These constructors are used to define functions on `Shape`s. As examples, consider the following:

```
square :: Float -> Shape
square n = Rect n n

area :: Shape -> Float
area (Circle r) = pi * r ^ 2
area (Rect x y) = x * y
```

The constructors `Circle` and `Rect` are considered *constructor functions* (because they take arguments) which map `Float`s to `Shape`s.

The difference between functions and constructor functions is that the latter have no defining equations and exist solely for the purpose of building pieces of data. As an example, `negate 1.0` can be evaluated to `-1.0`; `Circle 1.0` is fully evaluated and cannot be further simplified. The expression `Circle 1.0` is a piece of data in the same way that `1.0` is.

Data declaration can also be parameterized. Consider the following type from `Prelude`:

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

This type is thought of as being values of a type `a` which may either fail or succeed. As an example, let's consider the same versions of `div` and `head` (recall that `div` will not work if the divisor is `0` and that `head` will not work on the empty list):

```
safediv :: Int -> Int -> Maybe Int
safedif _ 0 = Nothing
safediv m n = Just (div m n)
```

```
safehead :: [a] -> Maybe a
safehead [] = Nothing
safehead xs = Just (head xs)
```

8.3 Newtype declarations

Types with a single constructor with a single argument can be declared with the `newtype` mechanism. As an example, a natural number type could be defined as

```
newtype Nat = N Int
```

The single constructor `N` takes an `Int` as an argument – it is up to the library to ensure that it is non-negative.

Let's look at how `newtype` compares with these two alternative definitions of the natural number type

```
type Nat = Int

data Nat = N Int
```

Using `newtype` instead of `type` means that `Nat` and `Int` are different types instead of synonyms; this ensures that the type system does not allow using one in place of the other.

Using `newtype` instead of `data` brings a performance benefit. `newtype` constructors (like `N`) don't incur any costs when the programs are evaluated since they are automatically removed by the compiler once type-checking has completed.

Note: In general, `newtype` provides better type safety than `type` and better performance than `data`.

8.4 Recursive types

As mentioned in the first section, `type` declarations can't be recursive. However, both `data` and `newtype` can be recursive. Consider a recursive definition of the natural numbers

```
data Nat = Zero | Succ Nat
```

Then the values of type `Nat` correspond to natural numbers – `Zero` represents the number zero and `Succ` represents the successor function (as defined on the naturals, that's `(1+)`).

We can define the following conversion functions:

```
nat2int :: Nat -> Int
nat2int Zero     = 0
nat2int (Succ n) = 1 + nat2int n

int2nat :: Int -> Nat
int2nat 0 = Zero
int2nat n = Succ (int2nat (n-1))
```

We can define addition on the `Nat` type as

```
add :: Nat -> Nat -> Nat
add Zero n = n
add (Succ m) n = Succ (add m n)
```

This definition draws on the idea that the addition of two natural numbers takes place by moving the `Succ` constructors from the first number until they exhausted, at which point the `Zero` at the end is replaced by the second number. As an example, let's see what happens with `add 2 1`:

```
add (Succ (Succ Zero)) (Succ Zero)
=   {applying add}
Succ (add (Succ Zero) (Succ Zero))
=   {applying add}
Succ (Succ (add Zero (Succ Zero))
=   {applying add}
Succ (Succ (Succ Zero))
```

As another example, consider the following definition of the `List` type:

```
data List a = Nil | Cons a (List a)
```

A value of type `List a` is either `Nil` (the empty list) or of the form `Cons x xs` for some values `x` of type `a` and `xs` of type `List a`, which represent a non-empty list.

We can re-define the `len` function so that it operates on our own `List`:

```
len :: List a -> Int
len Nil        = 0
len (Cos _ xs) = 1 + len xs
```

Recursion makes representing some data structures, like binary trees, fairly simple:

```
data BinTree a = Leaf a | Node (BinTree a) a (BinTree a)
```

The second constructor can be interpreted as the left child, the value, and the right child of a node in the tree.

We can define the tree

```
      5
     / \
    /   \
   /     \
  3       7
 / \     / \
1   4   6   9
```

as

```
t :: BinTree Int
t = Node (Node (Leaf 1) 3 (Leaf 4)) 5
         (Node (Leaf 6) 7 (Leaf 9))
```

Consider a function that searches such a tree and reports whether a value exists within it (where `l`, `v`, and `r` are the left node, value, and right node, respectively):

```
occurs :: Eq a => a -> Tree a -> Bool
occurs x (Leaf v)     = x == v
occurs x (Node l v r) = x == v || occurs x l || occurs x r
```

As an added benefit of lazy evaluation, if either of the first two conditions are true, then the remaining conditions are not evaluated. In the worst case `occurs` traverses the entire tree.

Consider a function which flattens a tree to a list:

```
flatten :: Tree a -> [a]
flatten (Leaf x) = [x]
flatten (Node l x r) = flatten l ++ [x] ++ flatten r
```

If applying this function to a tree yields a sorted list, then the tree is called a search tree.

Applying `flatten` to our tree yields `[1,3,4,5,6,7,9]` so we do have a search tree. Since that is the case, we can re-write `occurs` to take advantage of the fact that we have a binary search tree:

```
occurs :: Ord a => a -> Tree a -> Bool
occurs x (Leaf v) = x == v
occurs x (Node l v r)
  | x == v = True
  | x <  v = occurs x l
  | x >  v = occurs x r
```

This version only traverses one path down the tree instead of testing every node.

We can declare many different types of trees. We can have trees that only have data in their leaves,

```
data Tree a = Leaf a | Node (Tree a) (Tree a)
```

trees that have data only in their nodes,

```
data Tree a = Leaf | Node (Tree a) a (Tree a)
```

trees that have different data types in their leaves and nodes,

```
data Tree a b = Leaf a | Node (Tree a b) b (Tree a b)
```

or trees that have a list of sub-trees

```
data Tree a = Node a [Tree a]
```

8.5 Class and instance declarations

Classes are different from types and are defined by the **`class`** mechanism. As an example, the class `Eq` of equality types is declared in the standard prelude as

```
class Eq a where
  (==), (/=) :: a -> a -> Bool
  x /= y = not (x == y)
```

We interpret this as meaning that a type `a` is an instance of the class `Eq` if and only if it supports the equality and inequality operators defined within the class. In actuality, because we have a *default definition* for the `/=` operator, instances of `Eq` only require a definition for the `==` operator.

Type classes were created as a structured way to express "ad-hoc polymorphism", which is essentially the technical term for overloaded functions. In the example `Eq` class given above, if we have a complex data structure like a tree, our definition of the `==` operator will be different from one which compares two `Pair`s. Making a type an instance of `Eq` is a promise to the compiler that that type has a specialized definition of the `==` operator. Additionally, default definitions can be overridden by an instance if (as the word default implies).

As another example, we can make the type `Bool` into an instance of `Eq` as follows:

```
instance Eq Bool where
  False == False = True
  True  == True  = True
  _     == _     = False
```

Only types which are declared by the `data` and `newtype` mechanisms can be made into instances of classes.

Classes can be extended to form new classes. As an example consider the class `Ord`, which contains values which have a total ordering. It is defined as an extension of `Eq`:

```
class Eq a => Ord a where
(<), (<=), (>), (>=) :: a -> a -> Bool
min, max             :: a -> a -> a

min x y
  | x <= y    = x
  | otherwise = y

max x y
  | x <= y    = y
  | otherwise = x
```

As illustrated by the definition of the class `Ord`, instances must be instances of `Eq` and support six additional operators.

We can have `Bool` implement `Ord` by adding support for the additional operators which do not have a default implementation:

```
instance Ord Bool where
False < True = True
_     < _    = False

-- We can define all of these operators in terms of < and ==
-- just by switching the order of the operands and using ||
b <= c = (b < c) || (b == c)
```

```
b >= c = c <= b
b >  c = c < b
```

Derived instances

When new types are declared, it's usually appropriate to make them instances of a few built-in classes. We can do this by using the `deriving` mechanism:

```
data Bool = False | True
              deriving (Eq, Ord, Show, Read)
```

Haskell then automagically implements the required definitions for those classes.

```
> False == False
True

> False < True
True

> show False
"False"
```

You might wonder how the compiler could decide how to implement an ordering on `Bool`. In the case of deriving instances of `Ord`, the ordering of the constructors is determined by their position in its declaration. If we had defined `Bool` as `data Bool = True | False` then we would have `True < False == True`.

When constructors have arguments, the types of those arguments *must* be instances of derived classes. Recall our definitions of `Shape` and `Maybe`:

```
data Shape = Circle Float | Rect Float Float

data Maybe a = Nothing | Just a
```

Deriving `Shape` as an equality type requires that the type `Float` is also an equality type (which is the case here). The same requirement holds for `Maybe`: if `Maybe a` derives `Ord`, then `a` must also derive `Ord` (which means that we have created a class constraint on the accepted parameters).

Just like with lists and tuples, `Shape`'s values with constructors have their arguments ordered lexicographically. Assuming that `Shape` derives `Ord`, we have

```
> Rect 1.0 4.0 < Rect 2.0 3.0
True

> Rect 1.0 4.0 < Rect 1.0 3.0
False
```

8.6 Tautology checker

The remainder of this chapter contains two extended programming exercises. The first exercise is to develop a function that decides if simple logical propositions are always true (a *tautology*).

Consider a language of propositions built up from basic values (`False`,`True`) and variables (`A`, `B`, `...`, `Z`) using negation (¬), conjunction (∧), implication ($\implies$), and parentheses. As an example, the following are propositions:

$A \wedge \neg A$

$(A \wedge (A \implies B)) \implies B$

The meaning of the logical operators can be defined using truth tables.

A	$\neg A$
F	T
T	F

A	B	$A \wedge B$
F	F	F
F	T	F
T	F	F
T	T	T

A	B	$A \implies B$
F	F	T
F	T	T
T	F	F
T	T	T

They can be combined to evaluate the results of more complex propositions.

A	$A \wedge \neg A$
F	F

A	$A \wedge \neg A$
T	F

A	B	$(A \wedge (A \implies B)) \implies B$
F	F	T
F	T	T
T	F	T
T	T	T

We see that the first is in fact always false (a *contradiction*) while the second is a tautology.

Our first step should be to define a data type to represent our propositions.

```
data Prop = Const Bool
          | Var Char
          | Not Prop
          | And Prop Prop
          | Imply Prop Prop
```

We might define our propositions as

```
p1 :: Prop
p1 = And (Var 'A') (Not (Var 'A'))

p2 :: Prop
p2 = Imply (And (Var 'A') (Imply (Var 'A') (Var 'B'))) (Var 'B')
```

To evaluate the proposition we need to know the value of each of its variables. We can declare a substitution as a lookup table which associates variable names with logical values, re-using the `Assoc` type we introduced earlier in the chapter:

```
type Subst = Assoc Char Bool
```

As an example, the substitution `[('A', False), ('B', True)]` assigns the variable `A` to `False` and the variable `B` to `True`. With the ability to perform substitutions, we can define a function which evaluates propositions given a substitution.

```
eval :: Subst -> Prop -> Bool
eval _ (Const b)   = b
eval s (Var x)     = find x s
eval s (Not p)     = not (eval s p)
eval s (And p q)   = eval s p && eval s q
eval s (Imply p q) = eval s p <= eval s q
```

As a digression, let us consider why we can implement the logical operator for implication, $\implies$, with the ordering we get for free on logical values.

We know that we have four cases for implication:

A	B	$A \implies B$
F	F	T
F	T	T
T	F	F
T	T	T

Note that the implication operator is only false when the truthiness of A is strictly greater than the truthiness value of B (we take advantage of the ordering `False < True` defined in the `Bool` type).

A proposition is a tautology if it is true for all possible substitutions. To make progress towards being able to evaluate whether a proposition is a tautology, we need a function that yields a list of all the variables in a proposition.

```
vars :: Prop -> [Char]
vars (Const _)   = []
vars (Var x)     = [x]
vars (Not p)     = vars p
vars (And p q)   = vars p ++ vars q
vars (Imply p q) = vars p ++ vars q
```

Note: `vars` doesn't provide any sort of mechanism for removing duplicates – that's okay since we can do it later on.

Our goal now is to create a function that returns a list containing every possible set of combinations of true and false. An example of the desired use might be (formatted for clarity):

```
> bools 3
[[False, False, False]
,[False, False, True]
,[False, True, False]
,[False, True, True]
,[True, False, False]
,[True, False, True]
,[True, True, False]
,[True, True, True]]
```

We can think of this function as counting in binary (where `[False, False, False]` corresponds to the binary number `000`, `[False, False, True]` to `001`, and so on). Alternatively, we can observe the relation to the mathematical definition of the power set, where the truth value at a given index determines whether we include that element in the power set.

It is known that the power set of a set A with n elements has 2^n elements. To construct the power set of a set A, one can build a binary tree as they traverse A. For each element of A, split every leaf of the tree – the left branch

represents skipping the current element while the right branch represents including the element in some set in the power set. This choice – inclusion or exclusion – is performed for every set in the set. By the multiplication principle, given n elements, we have 2^n possible choices.

```
bools :: Int -> [[Bool]]
bools 0 = [[]]
bools n = map (False :) bss ++ map (True :) bss
  where
    bss = bools (n - 1)
```

With `bools`, we can now construct a function which performs all possible substitutions,

```
substs :: Prop -> [Subst]
substs p = map (zip vs) (bools (length vs))
  where
    vs = rmdups (vars p)
```

where `rmdups` is as defined in Chapter 7.

As an example,

```
> substs p2
[[('A',False),('B',False)],
[('A',False),('B',True)],
[('A',True),('B',False)],
[('A',True),('B',True)]]
```

Lastly, we need to build a function which checks to see whether a proposition is tautologous.

```
isTaut :: Prop -> Bool
isTaut p = and [eval s p | s <- substs p]
```

8.7 Abstract Machine

The second extended example builds a machine which can operate on arithmetic expressions built from integers and the addition operator, yielding an integer.

```
data Expr = Val Int | Add Expr Expr

value :: Expr -> Int
value (Val n)   = n
value (Add x y) = value x + value y
```

Tracing the evaluation of `(2 + 3)+ 4`, we have

```
value (Add (Add (Val 2) (Val 3)) (Val 4))
={ applying value }
value (Add (Val 2) (Val 3)) + value (Val 4)
={ applying the first value }
(value (Val 2) + value (Val 3)) + value (Val 4)
```

```
={ applying the first value }
(2 + value (Val 3)) + value (Val 4)
={ applying the first value }
(2 + 3) + value (Val 4)
={ applying the first + }
5 + value (Val 4)
={ applying value }
5 + 4
={ applying + }
9
```

The order of evaluation for the operands of `value` is not specified. In fact, the order of evaluation is left up to Haskell. If we want to enforce an order of evaluation, we can do so by defining an *abstract machine* for expressions, which would specify the process of evaluating an expression.

To do this, we begin by defining a type of *control stacks* for our abstract machine. The control stacks are a list of operations to be performed after the current evaluation has finished.

```
type Cont = [Op]

data Op = EVAL Expr | ADD Int
```

A function which evaluates an expression in the context of a control stack can be defined as

```
eval :: Expr -> Cont -> Int
eval (Val n)   c = exec c n
eval (Add x y) c = exec x (EVAL y : c)
```

Picking apart `eval`, we have to possible paths through the function. In the first case, we have an integer (which cannot be further evaluated) and we begin executing the control stack. In the second case, we have an expression. We evaluate the first argument, `x`, first, placing `EVAL y` on the control stack so that `y` is evaluated after the evaluation of `x` is complete.

Our function which performs the actual evaluation, `exec`, can be defined as

```
exec :: Cont -> Int -> Int
exec []           n = n
exec (EVAL y : c) n = eval y (ADD n : c)
exec (ADD n : c)  m = exec c (n + m)
```

We can interpret `exec` by looking at the different cases we cover.

- Given an empty control stack, return the integer as the result
- Given a control stack where the first element is a postponed evaluation (`EVAL`), evaluate it and put the integer argument on the stack to signal that it should be added to the result of the evaluation of `y`
- Given a control stack where the first element is an `ADD` operation, evaluation of both arguments must have been completed and we can continue executing the remaining control stack

The last step in constructing our abstract machine is to construct a function which evaluates an expression to an integer by invoking `eval` with the given expression and an empty stack:

```
value :: Expr -> Int
value e = eval e []
```

Hutton remarks:

> The fact that our abstract machine uses two mutually recursive functions, eval and exec, reflects the fact that it has two modes of operation, depending upon whether it is being driven by the structure of the expression or the control stack. To illustrate the machine, here is how it evaluates (2 + 3) +4:
>
> ```
> value (Add (Add (Val 2) (Val 3)) (Val 4))
> ={ applying value }
> eval (Add (Add (Val 2) (Val 3)) (Val 4)) []
> ={ applying eval }
> eval (Add (Val 2) (Val 3)) [EVAL (Val 4)]
> ={ applying eval }
> eval (Val 2) [EVAL (Val 3), EVAL (Val 4)]
> ={ applying eval }
> exec [EVAL (Val 3), EVAL (Val 4)] 2
> ={ applying exec }
> eval (Val 3) [ADD 2, EVAL (Val 4)]
> ={ applying eval }
> exec [ADD 2, EVAL (Val 4)] 3
> ={ applying exec }
> exec [EVAL (Val 4)] 5
> ={ applying exec }
> eval (Val 4) [ADD 5]
> ={ applying eval }
> exec [ADD 5] 4
> ={ applying exec }
> exec [] 9
> ={ applying exec }
> 9
> ```
>
> Note how eval proceeds downwards to the leftmost integer in the expression, maintaining a trail of the pending right-hand expressions on the control stack. In turn, exec then proceeds upwards through the trail, transferring control back to eval and performing additions as appropriate.

8.8 Chapter Remarks

The abstract machine example is derived from Hutton and Wright's "Calculating an Exceptional Machine."

The type of control stacks used are a special case of the zipper data structure for traversing values of recursive types, covered in greater detail in Huet's, "The Zipper."