LECTURE 05

Theory and Design of PL (CS 538)

February 05, 2020

DEFINING NEW TYPES

WHY USE CUSTOM TYPES?

- Better describe what programs should “mean”
  - Is this integer measuring length, or weight?
  - Use the compiler to do these basic checks

SANITIZING INPUT

- In Haskell: `newtype` declaration

```
newtype CheckedStr   = Safe String
newtype UncheckedStr = Unsafe String
```

- Suppose: have some way to check strings

```
checkString :: UncheckedStr -> CheckedStr
```

SANITIZING INPUT

- Compiler makes sure you don't forget to check!

```
processSafeStr :: CheckedStr -> Output
processSafeStr = ...

mysteryStr :: UncheckedStr
mysteryStr = ...

processSafeStr (checkString mysteryStr) -- OK
processSafeStr (mysteryStr)             -- Compiler complains!
```

WHY USE CUSTOM TYPES?

- Support more richer data
  - Not just integers, booleans, and functions
  - Lists, trees, maps, etc.

THREE KEY INGREDIENTS

1. Name of type, and parameters
    - Simple: `char` for character
    - Complex: `[a]` for list of elements of same type
2. Some way to *make* things of this type
    - Package up parts into a data of the new type
    - Also called *constructors*
3. Some way to *use* things of this type
    - Use data packaged inside things of this type
    - Also called *destructors*

EXAMPLE: PRODUCTS

ALSO KNOWN AS TUPLES

- Wrap up several pieces of data into one
- Just one option: must contain all data

```
data Pair a b = MkPair a b
```

- *Type variables* `a` and `b`: can stand for any type

```
(MkPair 1 True) :: Pair Int Bool
```

USING TUPLES

- Given tuple, pattern match to extract data

```
fstPair :: Pair a b -> a
fstPair (MkPair x _) = x

sndPair :: Pair a b -> b
sndPair (MkPair _ y) = y
```

- Note: still need to put the constructor `MkPair`

TYPES WITH PARAMETERS

- Pair is an example of a *parametric type*
- Any two types `a` and `b` give a type `Pair a b`
- Can require parameters to be the same:

```
data SamePair a = MkSamePair a a

(MkSamePair 1 3)          :: SamePair Int
(MkSamePair True False) :: SamePair Bool

-- Not allowed: (MkSamePair 1 False)
```

FANCIER PRODUCTS: RECORDS

- Sometimes we want to work with large tuples:

```
data Person = MkPerson
    String      -- Name
    Bool        -- Is employed?
    Bool        -- Is married?
    Int         -- Age
    String      -- Address
```

- Very annoying (and error-prone) to work with:

```
getName   (MkPerson name _   _ _ _) = name
getEmploy (MkPerson _    emp _ _ _) = emp
...
```

RECORD SYNTAX

- Haskell provides *record syntax* for these tuples

```
data Person = MkPerson
     { name     :: String     -- Name
     , employed :: Bool       -- Is employed?
     , married  :: Bool       -- Is married?
     , age      :: Int        -- Age
     , address  :: String     -- Address
     }
```

- Automatically generates accessor functions:

```
name     :: Person -> String
employed :: Person -> Bool
...
```

BUILDING RECORDS

- Standard syntax for building a new record:

```
defaultPerson  :: Person
defaultPerson  = MkPerson
    { name     = "John Doe"
    , employed = True
    , married  = False
    , age      = 30
    , address  = "123 Main Street, Anytown, WI"
    }
```

USING RECORDS

- Standard syntax for updating records:

```
-- Keep all fields the same, except for name and address:
defaultPerson' = defaultPerson
     { name = "Jane Doe"
     , address = "456 Main Street, Anytown, WI }
```

- Can pattern match on selected fields

```
getNameAddress :: Person -> (String, String)
getNameAddress (MkPerson { name = n, address = a }) = (n, a)
```

EXAMPLE: SUMS

ALSO KNOWN AS ENUMS

- Basic idea: choice between different options
- Example: a type `Color`

```
data Color = Red | Green | Blue
```

- Can pack additional data with each option:

```
data Time = HoursMinutes Int Int | Minutes Int
```

BUILDING ENUMS

```
data Time = HoursMinutes Int Int | Minutes Int
```

- First label in each option is a *data constructor*
- Two constructors: `HoursMinutes` and `Minutes`
- Can make a `Time` in exactly two ways:
  - `HoursMinutes 11 59 :: Time`
  - `Minutes 1800 :: Time`

EXTRACTING DATA

- Pattern match: give program to run for each option

```
whatColorBellPepper :: Color -> String
whatColorBellPepper Red   = "It is red."
whatColorBellPepper Green = "It is green!"
whatColorBellPepper Blue  = "It is blue?"
```

- Can also match on data inside different options

```
whatTime :: Time -> String
whatTime (HoursMinutes m h) = (show m) ++ ":" ++  (show h)
whatTime (Minutes m)        = (show m) ++ " min. past midnight"
```

EXAMPLE: MAYBE

BUILDING MAYBES

- A `Maybe a` is either nothing, or an `a`

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

- To make something of this type, use constructors

```
noValue :: Maybe Int
noValue = Nothing

someValue :: Maybe Int
someValue = Just 13
```

UNWRAPPING MAYBES

- Given a maybe, describe how to handle both cases
- Compiler complains if `Nothing` case isn't handled

```
printMaybe :: Maybe Int -> String
printMaybe Nothing = "No value here :("
printMaybe (Just x)  = "Got a value: " ++ (show x)
```

USE: OPTIONAL VALUES

- Contains an actual value, or nothing (is "null")
- `Nothing` is usually indicates failure
- For instance: lookup function

```
findIndex :: (a -> Bool) -> [a] -> Maybe Int
-- findIndex p returns (Just index) if element satisfying p
-- findIndex p returns Nothing if no element satisfies p
```

EXAMPLE: EITHER

BUILDING EITHERS

- `Either` is just a sum with two type parameters:

```haskell
data Either a b = Left a | Right b

-- Auto-generated: Left  :: a -> Either a b
-- Auto-generated: Right :: b -> Either a b
```

- Use `Left` or `Right` to create an `Either a b`

UNWRAPPING EITHERS

- Just like for `Maybe`, do a case analysis:

```
doubleRight :: Either Int Int -> Int
doubleRight (Left  x) = x
doubleRight (Right y) = y + y
```

USE: ERROR-HANDLING

- Either normal value, or an error
- Convention
  - Right is normal case, holds result value
  - Left is error case, includes error info

```
safeModulo :: Int -> Int -> Either String Int
safeModulo m n
    | n == 0 = Left "Error: Modulo by zero!"
    | n /= 0 = Right (n `mod` m)
```

INDUCTIVE DATATYPES

GENERALIZE A BIT

- All the types we have seen so far are *inductive types*
- Basic pattern:
  - Some type parameters (maybe zero)
  - Some number of constructors
  - Unwrap values by matching on constructor
- Inductive: data may be of the type being defined!

NATURAL NUMBERS

- Either zero, or one plus another natural number

```
data Nat = Zero | Succ Nat
-- Succ short for "successor"
```

- As always, operate by pattern matching on cases

```
addNats :: Nat -> Nat -> Nat

-- 0 + n' = n'
addNats Zero     n' = n'

-- (1 + n) + n' = 1 + (n + n')
addNats (Succ n) n' = Succ $ addNats n n'
```

LISTS

- Either empty list, or an element plus another list
- Takes a type parameter `a`: type of list elements

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

maybeHead :: List a -> Maybe a
maybeHead Nil          = Nothing
maybeHead (Cons x xs) = Just x
```

BINARY TREES

- Either leaf, or node with data plus two child trees

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

swap :: Tree a -> Tree a
swap Leaf          = Leaf
swap (Node x l r)  = Node x (swap r) (swap l)
```