Product Types

Dr. Mattox Beckman

University of Illinois at Urbana-Champaign
Department of Computer Science

Algebraic Datatypes

- We want to be able to build new types by combining existing types.
- Two ways to do it:
  - *Product* types: tuples and records
  - *Sum* types: disjoint types
    a.k.a. tagged unions, disjoint unions, etc.

Objectives!

Objectives:

- Explain what a *product type* is.
- Use *pairs* and *records* to model various structures: dictionaries, databases, and complex numbers.

Tuples

- An *n*-tuple is an ordered collection of *n* elements.
- If $n = 2$ we usually call it a pair.

```
Prelude> x = 10 :: Integer
Prelude> y = "Hi"
Prelude> :t x
x :: Integer
Prelude> :t y
y :: [Char]  -- [Char] is a synonym for String
Prelude> p = (x,y)
Prelude> :t p
p :: (Integer, [Char])
```

Projection Functions

- We have projection functions:

```
Prelude> :t fst
fst :: (a, b) -> a
Prelude> :t snd
snd :: (a, b) -> b
Prelude> fst p
10
Prelude> snd p
"hi"
```

n-tuples

- We have *n*-tuples:

```
Prelude> let p4 = (10,"hi",\x -> x + 1, (2,3))
Prelude> :t p4
p4
:: (Num t, Num a, Num t1, Num t2) =>
    (t, [Char], a -> a, (t1, t2))
```

Example

- Complex numbers have the form $a + bi$, where $i \equiv \sqrt{-1}$.
- Addition: $(a + bi) + (c + di) = (a + c) + (b + d)i$
- Multiplication: $(a + bi) \times (c + di) = ac - bd + (ad + bc)i$

```
cadd (a,b) (c,d) = (a + c, b + d)
cmul (a,b) (c,d) = (a * c - b * d,
                    a * d + b * c)
```

We could use tuples to represent complex numbers, like this. (Hint: What are the types of these functions?) Why might this be a bad idea?

```
Prelude> :t cadd
cadd :: (Num t, Num t1) => (t, t1) -> (t, t1) -> (t, t1)
```

Record Type Definitions

Record Syntax

$$\texttt{data}\ \mathit{Name} = \ \mathit{Name}\ \{ \mathit{field} :: \mathit{type}\ [, \mathit{field} :: \mathit{type} \ldots] \}$$

```
data Complex = Complex { re :: Float, im :: Float }
               deriving (Show,Eq)
```

- To create an element of type `Complex`, you have two choices.
  1. Treat the constructor as a function:

     ```
     c = Complex 10.54 34.2
     ```
  2. Specify the field names:

     ```
     c = Complex { re = 10.54, im = 34.2 }
     ```

Each of the field names becomes a function in Haskell. By default, *field names must be unique*, but Haskell 8.X lets you override this.

Haskell creates the field selector functions automatically.

```
Main> re c
10.54
Main> im c
34.2
```

Here are our complex number functions:

```
cadd x y = Complex { re = re x + re y
                   , im = im x + im y }
cmul x y = Complex { re = re x * re y - im x * im y
                   , im = re x * im y + re y * im x }
```

Example: Database Records

- Records are often used to model database-like data.
- Example: we want to store first name, last name, and age.

```
data Person = Person { fname :: String
                     , lname :: String
                     , age :: Int }
              deriving (Show,Eq)

people = [ Person "Bilbo" "Baggins" 111,
           Person "Harry" "Potter" 19 ]
```

- The `deriving (Show,Eq)` allows us to be able to print and test for equality.

Some Things to Try

- An *associative list* is a representation of a dictionary that uses a list of key-value pairs. They were commonly used in functional languages. Example: `[("emergency",911),("jenni",8675309)]`
- Write a function `add` that takes a key, a corresponding value, and an associative list, and returns a new one with the items inserted. For extra fun, have it keep the keys in a sorted order.
- Write a function `mylookup` that takes a key and an associative list and returns the corresponding value. This function will not behave well if the key is not in the list!
- Instead of tuples, try defining a record type with `Key` and `Value` fields, and use that instead.