

Algebraic Datatypes

Product Types

Dr. Mattox Beckman

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN
DEPARTMENT OF COMPUTER SCIENCE

- ▶ 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.

```

1 Prelude> x = 10 :: Integer
2 Prelude> y = "Hi"
3 Prelude> :t x
4 x :: Integer
5 Prelude> :t y
6 y :: [Char] -- [Char] is a synonym for String
7 Prelude> p = (x,y)
8 Prelude> :t p
9 p :: (Integer, [Char])
    
```

Projection Functions

- ▶ We have projection functions:

```

1 Prelude> :t fst
2 fst :: (a, b) -> a
3 Prelude> :t snd
4 snd :: (a, b) -> b
5 Prelude> fst p
6 10
7 Prelude> snd p
8 "hi"
    
```

n-tuples

- ▶ We have *n*-tuples:

```

1 Prelude> let p4 = (10, "hi", \x -> x + 1, (2,3))
2 Prelude> :t p4
3 p4
4 :: (Num t, Num a, Num t1, Num t2) =>
5   (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: $(a + bi) \times (c + di) = ac - bd + (ad + bc)i$

```

1 cadd (a,b) (c,d) = (a + c, b + d)
2 cmul (a,b) (c,d) = (a * c - b * d,
3                   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?

```

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

Record Type Definitions

Record Syntax

```
data Name = Name { field :: type [, field :: type ...] }
```

```

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

- ▶ To create an element of type `Complex`, you have two choices.

1. Treat the constructor as a function:


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


```
1 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.

```
1 Main> re c
2 10.54
3 Main> im c
4 34.2
```

Here are our complex number functions:

```
1 cadd x y = Complex { re = re x + re y
2                   , im = im x + im y }
3 cmul x y = Complex { re = re x * re y - im x * im y
4                   , 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.

```
1 data Person = Person { fname :: String
2                       , lname :: String
3                       , age  :: Int }
4                       deriving (Show,Eq)
5
6 people = [ Person "Bilbo" "Baggins" 111,
7            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.