Lists and Tuples

Composite Data Types in Haskell

Prof Hans Georg Schaathun

Høgskolen i Ålesund

18th January 2016

Outline

Scalar Data Types

1. Int, Integer
2. Double
3. Bool
4. Char, String

Composite Data

Larsgårdsveien 2
6009 Ålesund

Composite Data

Larsgårdsveien 2
6009 Ålesund

1. Street name (String)
2. House number (Integer)
3. Post code (Integer)
4. Post town (String)

Composite Data

Larsgårdsveien 2
6009 Ålesund

1. Street name (String)
2. House number (Integer)
3. Post code (Integer)
4. Post town (String)

```
type Address = (String,Integer,Integer,String)

home = ("Larsgårdsveien",2,6009,"Ålesund")
```

Tuples of Tuples

Example (Tuples)

```
type Person = (String,String,Bool)
type Address = (String,Integer,Integer,String)
type Customer = (Person,Address)
```

Tuples of Tuples

Example (Tuples)

```
type Person = (String,String,Bool)
type Address = (String,Integer,Integer,String)
type Customer = (Person,Address)
```

Example (Tuple functions)

```
getAddress ::  Customer -> Address
getAddress c = snd c
```

Tuples of Tuples

Example (Tuples)

```
type Person = (String,String,Bool)
type Address = (String,Integer,Integer,String)
type Customer = (Person,Address)
```

Example (Tuple functions)

```
getAddress ::  Customer -> Address
getAddress c = snd c
```

Definition (`snd` function)

```
snd ::  (a,b) -> b
snd (_,y) = y
```

Pattern Matching

Example

```
showPerson ::  Person -> String
showPerson (x,y,_) = x ++ " " ++ y ++ "\n"
```

Pattern Matching

Example

```
showPerson ::  Person -> String
showPerson (x,y,_) = x ++ " " ++ y ++ "\n"
```

Example

```
showAddress ::  Address -> String
showAddress (x,y,_,_) = x ++ " " ++ show y ++
"\n"
```

Pattern Matching

Example

```
showPerson ::  Person -> String
showPerson (x,y,_) = x ++ " " ++ y ++ "\n"
```

Example

```
showAddress ::  Address -> String
showAddress (x,y,_,_) = x ++ " " ++ show y ++
"\n"
```

Example

```
showCustomer ::  Customer -> String
showCustomer (p,a) = showPerson p ++ showAddress
a
```

Another example

Example

```
addPair ::  (Integer,Integer) -> Integer
addpair (x,y) = x + y
```

Outline

Algebraic Data Types

Type alias	`type Person = (String,String,Bool)`
Genuinly new type	`data Person = Person String String Bool`

Algebraic Data Types

Type alias	`type Person = (String,String,Bool)` `me = ("John", "Doe", True)`
Genuinly new type	`data Person = Person String String Bool` `me = Person "John" "Doe" True`

Outline

Lists versus tuples

Tuples	
Fixed length	
Any combination of types	
`item = ("Oranges", 5)`	
`type Customer = (Person,Address)`	

Lists versus tuples

Tuples	Lists
Fixed length	Variable length
Any combination of types	One type for all elements
`item = ("Oranges", 5)`	`goods = ["Oranges", "Bananas", "Apples"]`
`type Customer = (Person,Address)`	`[Customer]`

Lists definitions

Some lists of type `[Integer]`

1. `[2,3,5,7,11]`
2. `[1..10]`
3. `[0,5..100]`
4. `[10,8..0]`
5. `[]`
6. `[1..]`

Lists definitions

Some lists of type `[Integer]`

1. `[2,3,5,7,11]`
2. `[1..10]`
3. `[0,5..100]`
4. `[10,8..0]`
5. `[]` (empty)
6. `[1..]` (infinite)

Functions on lists

```
let l = [2,3,5,7,11]
l!!3
head l
tail l
l ++ [13,17,19]
0:l
```

Functions on lists

```
let l = [2,3,5,7,11]
l!!3                    → 7
head l
tail l
l ++ [13,17,19]
0:l
```

Functions on lists

```
let l = [2,3,5,7,11]
l!!3                    → 7
head l                  → 2
tail l
l ++ [13,17,19]
0:l
```

Functions on lists

```
let l = [2,3,5,7,11]
l!!3                    → 7
head l                  → 2
tail l                  → [3,5,7,11]
l ++ [13,17,19]
0:l
```

Functions on lists

```
let l = [2,3,5,7,11]
l!!3                  → 7
head l                → 2
tail l                → [3,5,7,11]
l ++ [13,17,19]       → [2,3,5,7,11,13,17,19]
0:l
```

Functions on lists

```
let l = [2,3,5,7,11]
l!!3                  → 7
head l                → 2
tail l                → [3,5,7,11]
l ++ [13,17,19]       → [2,3,5,7,11,13,17,19]
0:l                   → [0,2,3,5,7,11]
```

The String is a List

1. `['a','c'..'m']`

The String is a List

1. `['a','c'..'m']`
    - `"acegikm"`

The String is a List

1. `['a','c'..'m']`
   - `"acegikm"`
2. `"Hello" ++ ", " ++ "John"`
   - List concatenation used on strings
3. `head "Hello"`
4. `tail "Hello"`

List comprehension

```
— let l = [1..10]
```

List comprehension

- `let l = [1..10]`
- Set comprehension in mathematics
  - $\{2x | x = 1, \ldots, 10\}$

List comprehension

- `let l = [1..10]`
- Set comprehension in mathematics
  - $\{2x | x = 1, \ldots, 10\}$
  - $\{2x | x \in \{1, \ldots, 10\}\}$

List comprehension

- `let l = [1..10]`
- Set comprehension in mathematics
  - $\{2x | x = 1, \ldots, 10\}$
  - $\{2x | x \in \{1, \ldots, 10\}\}$
- List comprehension in Haskell
  - `[ 2*x | x <- [1..10] ]`
  - `[ 2*x | x <- l ]`

List comprehension

— `let l = [1..10]`

— Set comprehension in mathematics

- $\{2x | x = 1, \ldots, 10\}$
- $\{2x | x \in \{1, \ldots, 10\}\}$

— List comprehension in Haskell

- `[ 2*x | x <- [1..10] ]`
- `[ 2*x | x <- l ]`

— `let l = [1..20]`

— `[ x | x <- l, x `mod` 2 = 0 ]`

— `[ x | x <- l, x `mod` 2 = 0, x > 3 ]`

Outline

Summary

- Three types of composite data types
  1. Tuples
  2. Lists
  3. Algebraic data types
- Function definitions with pattern matching
  - patterns give access to constituent elements