Lists and Tuples

Composite Data Types in Haskell

Prof Hans Georg Schaathun

Høgskolen i Ålesund

February 2, 2015

hials

Outline

hials

Scalar Data Types

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

hials

Composite Data

- An adress consists of
  1. Street name
  2. House number
  3. Post code
  4. Post town
- How do we represent an address?
- Tuples

hials

Composite Data

- An adress consists of
  1. Street name (String)
  2. House number (Integer)
  3. Post code (Integer)
  4. Post town (String)
- How do we represent an address?
- Tuples

hials

Custom Types

- An address
  - `home = ("Larsgårdsveien",2,6025,"Ålesund")`
  - `home :: (String,Integer,Integer,String)`
- Declare a new type alias
  - `type Address = (String,Integer,Integer,String)`
- Type aliases is a convenience for readability

hials

Important

- Type name starts with a capital letter
- Function names start lower-case
- `type` defines an alias
- Type checking does not distinguish between `Address` and (String,Integer,Integer,String)

hials

Tuples of Tuples

Tuples can be made of any types

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

hials

Programming with Tuples

- A tuple is a single object
  - `getAddress :: Customer -> Address`
  - `getAddress c = snd c`
  - `snd` returns the second element of a pair
- In fact, `snd` is defined as
  - `snd :: (a,b) -> b`
  - `snd (_,y) = y`
  - This is pattern matching with tuples

hials

Pattern Matching

- `showCustomer :: Customer -> String`
- `showCustomer (p,a) =`
  `showPerson p ++ showAddress a`
- `showPerson :: Person -> String`
- `showPerson (x,y,_) = x ++ " " ++ y ++ "\n"`
- `showAddress :: Address -> String`
- `showAddress (x,y,) = x ++ " " ++ show y ++ "\n"`

hials

Pattern Matching

- `showCustomer :: Customer -> String`
- `showCustomer (p,a) =`
  `showPerson p ++ showAddress a`
- `showPerson :: Person -> String`
- `showPerson (x,y,_) = x ++ " " ++ y ++ "\n"`
- `showAddress :: Address -> String`
- `showAddress (x,y,) = x ++ " " ++ show y ++ "\n"`

hials

Pattern Matching

- `showCustomer :: Customer -> String`
- `showCustomer (p,a) =`
  `showPerson p ++ showAddress a`
- `showPerson :: Person -> String`
- `showPerson (x,y,_) = x ++ " " ++ y ++ "\n"`
- `showAddress :: Address -> String`
- `showAddress (x,y,) = x ++ " " ++ show y ++ "\n"`

hials

Another example

Pattern Matching

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

hials

Outline

hials

Algebraic Data Types

- We can declare new types `type`
  - `type Person = (String,String,Bool)`
  - composite types using tuples
  - type aliases, not new types
- Genuinly new types is possible
  - Algebraic data types
  - `data Person = Person String String Bool`
  - Objects are created with the constructor `Person`
  - `me = Person John Doe True`
- We will consider algebraic data types next week

hials

Outline

hials

The limitation of tuples

- Each tuple type has a fixed length
  - **`(Integer,Integer)`** is a different type from **`(Integer,Integer,Integer)`**
- What if you need a list of customers, with unbounded length?
  - then lists is the answer
- **`[Customer]`** is a list type
  - arbitrary number of customers

hials

Lists versus tuples

- `item = ("Oranges", 5)`
- `goods = ["Oranges","Bananas","Apples"]`

	Tuples	Lists
Length	Fixed length	Variable length
Constituent Types	Any combination	One type for all elements

hials

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..]`

hials

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)

hials

Functions on lists

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

hials

Functions on lists

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

hials

Functions on lists

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

hials

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
```

hials

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
```

hials

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]
```

hials

The String is a List

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

hials

The String is a List

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

hials

The String is a List

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

hials

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 ]`

hials

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 ]`

hials

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 ]`

hials

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 ]`

hials

List comprehension with conditions

- `let l = [1..20]`
- `[ x | x <- l, x `mod` 2 = 0 ]`
- `[ x | x <- l, x `mod` 2 = 0, x > 3 ]`

hials

Outline

hials

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

hials