Modern Programing Languages

Introduction to Haskell (4)

Wamberto Vasconcelos
w.w.vasconcelos@abdn.ac.uk

2016–2075

Plan of Lecture

1. Algebraic Types in Haskell
2. Modules in Haskell
3. Abstract Data Types in Haskell
4. Summary of the Properties of Haskell
5. Assorted Info

Algebraic Types in Haskell (1)

- The Haskell type system supports all the complex types provided by other, more traditional, languages:

Conventional Type	Haskell Type
named types	type synonyms
arrays	lists
strings	list of char
records	tuples or algebraic types
enumerated types	algebraic types
variant records	algebraic types
pointer-based structures	algebraic types

Algebraic Types in Haskell (2)

- An algebraic type definition is:

  data TypeName = Constr_1 | Constr_2 | ... | Constr_n

- Type names and constructors must begin with capital letters.
- The simplest form of algebraic type consists of a set of alternative constructors. This defines an enumerated type, e.g.,

```haskell
data Temp     = Cold | Hot
data Season   = Spring | Summer | Autumn | Winter
```

- To define functions over such types we use pattern matching. For example:

```haskell
weather ::  Season -> Temp
weather Summer  = Hot
weather _       = Cold   -- don't care pattern
```

Algebraic Types in Haskell (3)

- Each base type `Int`, `Float`, `Bool` and `Char` has its own equality, ordering, enumeration and show/read functionalities (classes).
- When we introduce a new type we might expect these classes.
- These classes can be supplied by the system if we ask for them.
- To inform Haskell, we introduce `deriving`:

```
data Season =  Spring | Summer | Autumn | Winter
               deriving (Eq,Ord,Enum,Show,Read)
```

- The order is that in which constructors were listed in the definition. For example, `Spring < Summer < Autumn < Winter`
- We can then use expression of the form `[Spring..Autumn]`

Algebraic Types in Haskell (4)

- Constructors also provide an explicit label for elements of the type.
- We can also use other existing types, such as `Int`, `Bool`, etc:

```
data People = Person Name Age
type Name = [Char]
type Age = Int
```

- Some example of elements of type `People` are:

```
Person "Mary Poppins" 34
Person "Sommerset Maughan" 54
```

Algebraic Types in Haskell (5)

- Functions over algebraic types are defined using pattern matching. For instance,

```
showPerson ::  People -> String
showPerson (Person st n) =
                 st ++ " age:  " ++ show n
```

- Given the definition above, then we have:

```
showPerson (Person "Dillbert" 34)
 ⇝ "Dillbert age:  34"
```

- N.B.: When defining functions over algebraic types, it is important to ensure that cases for all constructors have been defined.

Algebraic Types in Haskell (6)

- The general form of the algebraic type definitions is:

  $$
  \begin{array}{l}
  \texttt{data}\ \mathrm{TypeName} = \\
  \qquad \mathrm{Constructor}_1\ \mathrm{Type}_{[1,0]}\ \ldots\ \mathrm{Type}_{[1,k_1]}\ \mid \\
  \qquad \mathrm{Constructor}_2\ \mathrm{Type}_{[2,0]}\ \ldots\ \mathrm{Type}_{[2,k_2]}\ \mid \\
  \qquad \mathrm{Constructor}_n\ \mathrm{Type}_{[n,0]}\ \ldots\ \mathrm{Type}_{[1,k_n]}
  \end{array}
  $$

- Algebraic types are more than "fancy types"!!
- Algebraic types can be used recursively to define more complex data structures such as trees.

Algebraic Types in Haskell (7)

- For instance, an algebraic type for mathematical expressions:

```
data Expr =
        Value Int | Add Expr Expr | Sub Expr Expr
```

- The expression `(2 + 3) - 4` is represented in type `Expr` as:

```
Sub (Add (Value 2) (Value 3)) (Value 4)
```

- Again, we use pattern matching to manipulate values of this type.
- A simple evaluator for the expressions above is:

```
eval ::  Expr -> Int
eval (Value v)   = v
eval (Add e1 e2) = (eval e1) + (eval e2)
eval (Sub e1 e2) = (eval e1) - (eval e2)
```

Algebraic Types in Haskell (8)

- Algebraic types become even more powerful when combined with polymorphic type variables. Example – different binary trees:

```
data Tree a = Nil | Node a (Tree a) (Tree a)
     deriving (Eq,Ord,Show)
type IntTree = Tree Int
type MyTree = Tree ([Char], Int)
```

- We can then use pattern matching to define general functions over these structures:

```
countNodes ::  Tree a -> Int
countNodes Nil = 0
countNodes (Node _ left right)
    = (countNodes left) + (countNodes right) + 1
```

Modules in Haskell (1)

- A module consists on a number of definitions (types, functions, etc.).
- A module has a precisely defined interface, describing what it exports (that is, offers to users).
- Other modules import definitions that are on offer.
- Advantages:
  - A large system can be broken up and developed independently;
  - The system can be compiled separately (more efficient);
  - Encourages reuse;

Modules in Haskell (2)

- Each module is named via a header:

```
module Ant where
data Ants  = ...
anteater x = ...
```

- To import a module:

```
module Bee where
import Ant
beeKeeper = ...
```

- All visible definitions from module `Ant` can be used in `Bee`.
- We can control what is to be exported, though:

```
module Bee (beeKeeper, anteater) where ...
```

Abstract Data Types in Haskell (1)

- Abstract Data Types (ADT's) are well known in software engineering as a mechanism for hiding the implementation of complex data types.
- For example, suppose we wish to implement a stack data type. We would need to support the following operations:
  - create a new (empty) stack;
  - add a new item to the top of a stack (push);
  - remove the top element from the stack (pop);

Abstract Data Types in Haskell (2)

- Users of ADTs do not need to know how operations are implemented.
- All that they need to know is the name of the operations and the argument values that they require, that is, the interface or signature of the data type.
- In Haskell ADT's are defined as modules:

```
module Stack
( Stack,
  emptyStack,    -- Stack
  push,          -- Int -> Stack -> Stack
  pop,           -- Stack -> Stack
) where ...
```

- Only the items in brackets are visible.

Abstract Data Types in Haskell (3)

- This is all that any user of the ADT `Stack` need be told.
- However, the implementor of the ADT must next state how the stack will be represented and manipulated:

```
module Stack (Stack,emptyStack,push,pop) where
data Stack = MyStack [Int]
emptyStack ::  Stack
emptyStack = MyStack []
push ::  Int -> Stack -> Stack
push x (MyStack xs) = (MyStack xs++[x])
pop ::  Stack -> Stack
pop (MyStack []) = (MyStack [])
pop (MyStack (x:xs)) = (MyStack xs)
```

Abstract Data Types in Haskell (4)

- As implementors of the ADT, we are free to change the definitions of these functions, or even the underlying representation of the data type, without affecting the other programs which make use of it...
- providing we keep the same signature and meaning for each function!!

Summary of the Properties of Haskell

- Haskell is a declarative, high-level language.
- Haskell is lazily evaluated – we can compute with infinite data structures in finite time and space.
- Haskell provides a rich, but high-level, set of data types. This separates computation over complex structures from the dangers of direct pointer manipulation and memory management.
- Haskell supports polymorphic functions. This results in highly generic and reusable programs.
- Haskell supports higher-order programming. This means that we can capture commonly used idioms and abstractions for easy reuse, and more concise programs.