COMPSCI 1JC3

Introduction to Computational Thinking

Fall 2017

06 Algebraic Data Types

William M. Farmer

Department of Computing and Software
McMaster University

October 18, 2017

Admin

- Midterm 1 will be held on Friday at 19:00–21:00 pm.
  - Testing rooms:

    MDCL 1102 (students Aksamit to Khanna).
    MDCL 1105 (students Lenko to Zhou).
  - 30 multiple choice questions.
    - Covers everything up to the end of Week 05.
    - Will be electronically marked.
    - Bring some HB pencils with you.
  - Two-stage format.
- Discussion sessions this week:
  - Wednesday: More on operating systems (Ch. 4 of CT)
  - Thursday: Review session for Midterm Test 1.
- Office hours: To see me please send me a note with times.
- Are there any questions?

Advice

Try to isolate what you don't understand!

- Formulate questions to ask in the lectures and tutorials.
- Formulate questions to ask on Avenue.
- Formulate questions for the Drop-In Centre.
- Formulate questions for Thursday's review session.

Review

1. Operating systems.
2. Kernel of an operating system.
3. System calls.
4. System programs.
5. Open source software.
6. Graphical vs. command line interfaces.

Creating New Types in Haskell

- A very important part of programming is choosing or creating the right types for the task at hand.
- There are two main ways of creating types in Haskell:
  1. Give a new name to an old type.
  2. Create a new type of new values.

Synonym Types

- A synonym type is a new name for an old type.
- In Haskell, a synonym type definition has the form:

  `type` *new-name* `=` *old-type*

- The type *new-name* can be used any place where the type *old-type* can be used.
- Example:

```
type Vector = (Double,Double,Double)
```


Algebraic Types

- An algebraic data type (algebraic type for short) is a new type of new values formed as a "sum" of "products".
- In Haskell, the definition of an algebraic type has the form:

$$\begin{array}{lll} \texttt{data}\ t = & C_1\ t_1^1\ \cdots\ t_{m_1}^1 \\ & \mid\ C_2\ t_1^2\ \cdots\ t_{m_2}^2 \\ & \vdots \\ & \mid\ C_n\ t_1^n\ \cdots\ t_{m_n}^n \end{array}$$

  where:
  - t is the name of the new type.
  - $C_1, \ldots, C_n$ are value constructors that create new values.
  - The t_j^is are types that may include t itself.
  - $m_1, \ldots, m_n \geq 0$.
- Functions can be defined on the new type using pattern matching with respect to the value constructors.

Sum and Product Types

- A sum type is an algebraic type that has more than one constructor.
- Example:

```
data Bool = False | True
```

- A product type is an algebraic type that has one constructor and the same structure as a tuple type.
- Example:

```
data Point = MakePoint Float Float
```

  which has the same structure as the tuple type

```
type Point = (Float,Float)
```

Enumeration Types

- An enumeration type is an algebraic type that enumerates a finite set of new values.
- The definition of an enumeration type has the form:

  data t = C_1 | C_2 | $\cdots$ | C_n

- Example:

  ```
  data Bool = False | True
  ```


Example: Bool

```
import Prelude hiding (Bool, False, True)

data Bool = False | True deriving (Show)

implies :: Bool -> Bool -> Bool

True `implies` False = False
_ `implies` _        = True
```


Example: Days of the Week

```
data WeekDay = Sunday
             | Monday
             | Tuesday
             | Wednesday
             | Thursday
             | Friday
             | Saturday
             deriving (Show)

meaning :: WeekDay -> String

meaning Sunday = "sun's day"
meaning Monday = "moon's day"
meaning Tuesday = "Tiw's day"
meaning Wednesday = "Woden's day"
meaning Thursday = "Thor's day"
meaning Friday = "Frige's day"
meaning Saturday = "Saturn's day"
```


Recursive Types

- A recursive type (or inductive type) is an algebraic type whose defined type is included in the constructor's types.
- Examples:

  ```
  data Nat
    = Zero
    | Suc Nat

  data ListInteger
    = Nil
    | Cons Integer ListInteger

  data BinTreeFloat
    = Leaf Float
    | Branch BinTreeFloat Float BinTreeFloat
  ```

Example: Nat

```
data Nat
  = Zero
  | Suc Nat
  deriving (Show)

natPlus :: Nat -> Nat -> Nat

x `natPlus` Zero     = x
x `natPlus` (Suc y)  = Suc (x `natPlus` y)

natTimes :: Nat -> Nat -> Nat

x `natTimes` Zero     = Zero
x `natTimes` (Suc y)  =
  x `natPlus` (x `natTimes` y)
```


Types with Parameters

- An algebraic type can define a type constructor that has types as parameters.
- Examples:

```
data List a
  = Nil
  | Cons a (List a)

data BinTree a
  = Leaf a
  | Branch (BinTree a) a (BinTree a)

data Maybe a
  = Just a
  | Nothing
```


Example: List and Maybe

```
import Prelude hiding (Maybe, Just, Nothing)

data List a
  = Nil
  | Cons a (List a)
  deriving (Show)

data Maybe a
  = Just a
  | Nothing
  deriving (Show)

head2 :: List a -> Maybe a

head2 Nil           = Nothing
head2 (Cons x y)    = Just x
```


Example: BinTree

```
data BinTree a
  = Leaf a
  | Branch (BinTree a) a (BinTree a)
  deriving (Show)

binTreeNodes :: BinTree a -> Integer

binTreeNodes (Leaf _)         = 1
binTreeNodes (Branch s _ t)   =
  (binTreeNodes s) + 1 + (binTreeNodes t)

binTreeSum :: Num a => BinTree a -> a

binTreeSum (Leaf x)         = x
binTreeSum (Branch s x t)   =
  (binTreeSum s) + x + (binTreeSum t)
```

Algebraic Types as Languages

- An algebraic type A defines a new language L of expressions.
  - L is infinite when A is recursive.
- The expressions of L are in a one-to-one correspondence with the values of A.
  - The expressions of L serve as literals for the values of A.
- Functions over A can be defined using pattern matching on the different forms of expressions of L.
  - At least one pattern is needed for each constructor of A.