Pemrograman Fungsional

Komputasi, Ekspresi, Datatype

Ade Azurat

Kamis, 21 September 2020

UNIVERSITAS INDONESIA
Veritas, Probitas, Iustitia
FAKULTAS ILMU KOMPUTER

Review Diskusi Pekan 01 Scele

- Bahasa lain dalam rumpun deklaratif

    Prolog

    SQL

Review Diskusi Pekan 01 Scele

- Bahasa lain dalam paradigma fungsional
  - Erlang
  - Elixir
  - Miranda
  - Lisp
  - Clojure
  - F#
  - Javascript

Review Diskusi Pekan 01 Scele

- GraphQL – SQL, declarative?
  - GraphQL termasuk declarative seperti SQL
  - Belum akan sepenuhnya menggantikan microservices

Review Diskusi Pekan 01 Scele

- Lambda di Java
  - Anonymous Inner Class
  - Stream API

Review Diskusi Pekan 01 Scele

- Goodbye OO?
  - No silver bullet
  - Masing-masing punya kelebihan
  - Strukturisasi dan abstraksi dari OO masih membantu
  - Selalu kritis!

Agenda dan Learning Objective

- Agenda
  - Model komputasi
  - Ekspresi, values, type
  - Datatype
- Learning Objective
  - Memahami model komputasi Fungsional. Diberikan sebuah program, dapat menjelaskan cara proses eksekusinya
  - Memahami konsep ekspresi sebagai bagian dari proses pembuatan algoritma. Diberikan permasalahan, dapat menyusun ekspresi sebagai penyusun algoritma dan penyelesaian masalah
  - Memahami representasi datatype. Diberikan permodelan masalah, dapat menyusun abstract data type yang berkesesuaian.

Computation by Calculation

- Computation by calculation is a simple concept that everyone should be familiar with, since it's not unlike ordinary arithmetic calculation. For example:

```
3 * (9 + 5)
  ➔ 3 * 14
  ➔ 42
```

- However, since we want computers to perform these tasks, we are also interested in issues such as efficiency:

```
3 * (9 + 5)
  ➔ 3*9 + 3*5
  ➔ 27 + 3*5
  ➔ 27 + 15
  ➔ 42
```

- Same answer, but the former was *more efficient* than the latter, since it took a fewer number of steps.

Abstraction

- We are also interested in *abstraction:* the process of recognizing a repeating pattern, and capturing it succinctly in one place instead of many.
- For example:

  ```
  3*(9+5)        4*(6+2)        7*(8+1)        ...
  ```

  This repeating pattern can be captured as a *function:*

  ```
  easy x y z = x*(y+z)
  ```

  Then each instance can be replaced by:

  ```
  easy 3 9 5    easy 4 6 2    easy 7 8 1    ...
  ```

- We can also perform calculations with *symbols*. For example, we can *prove* that `easy a b c = easy a c b`:

  ```
  easy a b c
  ➔ a*(b+c)          { unfold }
  ➔ a*(c+b)          { commutativity of + }
  ➔ easy a c b       { fold }
  ```

Expressions, Values, and Types

- The objects on which we calculate are called *expressions*.
- When no more unfolding (of either a primitive or user-defined function) is possible, the resulting expression is called a *value*.
- Every expression (and therefore every value) has a *type*.
  (A type is a collection of expressions with common attributes.)
- We write **`exp :: T`** to say that expression **`exp`** has type **`T`**.
- Examples:
  - Atomic expressions:
    `42 :: Integer`, **`'a' :: Char`**, **`True :: Bool`**
  - Structured expressions:
    `[1,2,3] :: [Integer]` - a *list* of integers
    `('b',4) :: (Char,Integer)` – a *pair* consisting of a character and an integer
  - Functions:

```
(+)  :: Integer -> Integer -> Integer
easy :: Integer -> Integer -> Integer -> Integer
```

Abstraction

- Our derivation of the function `easy` is a good example of the use of the *abstraction principle*: separating a repeating pattern from the particular instances in which it appears. In particular, the example demonstrates *functional abstraction.*
- *Naming* is an even simpler kind of abstraction:

```
let pi = 3.14159
in 2*pi*r1 + 2*pi*r2
```

- *Data abstraction* is the use of data structures to store common values on which common operations may be applied in an abstract manner.
- The “circle areas” example from the text demonstrates all three kinds of abstraction.

"Circle Areas" Example

- Original program:

```
totalArea = pi*r1^2 + pi*r2^2 + pi*r3^2
```

- A more abstract program:

```
totalArea =
  listSum [circArea r1, circArea r2, circArea r3]

listSum []     = 0
listSum (a:as) = a + listSum as

circArea r = pi*r^2
```

- The new program is longer than the old – in what ways is it better?
  - The code for the area of a circle has been isolated (using functional abstraction), thus minimizing errors and facilitating change and reuse.
  - The number of circles has been generalized (via data abstraction) from three to an arbitrary number, thus anticipating change and reuse.

Evaluation: Proof by Calculation

- Proof that the new `totalArea` is equivalent to the old:

```
listSum [circArea r1, circArea r2, circArea r3]
```
→ { unfold listSum }
```
circArea r1 + listSum [circArea r2, circArea r3]
```
→ { unfold listSum }
```
circArea r1 + circArea r2 + listSum [circArea r3]
```
→ { unfold listSum }
```
circArea r1 + circArea r2 + circArea r3 + listSum []
```
→ { unfold listSum }
```
circArea r1 + circArea r2 + circArea r3 + 0
```
→ { unfold circArea (three places) }
```
pi*r1^2 + pi*r2^2 + pi*r3^2 + 0
```
→ { simple arithmetic }
```
pi*r1^2 + pi*r2^2 + pi*r3^2
```

Defining New Datatypes

- The ability to define new data types in a programming language is important.
- Kinds of data types:
  - *enumerated types*
  - *records* (or *products*)
  - *variant records* (or *sums*)
  - *recursive types*
- Haskell's `data` declaration provides these kinds of data types in a uniform way that abstracts away from their implementation details, by providing an abstract interface to the newly defined type.
- Before looking at the example from Chapter 2, let's look at some simpler examples.

The Data Declaration

- Example of an *enumeration* data type:

```
data Day = Sun | Mon | Tue | Wed | Thu | Fri | Sat
     deriving Show
```

- The names `Sun` **through** `Sat` are constructor constants (since they have no arguments) and are the *only* elements of the type.

- For example, we can define:

```
valday :: Integer -> Day
valday 1 = Sun
valday 2 = Mon
valday 3 = Tue
valday 4 = Wed
valday 5 = Thu
valday 6 = Fri
valday 7 = Sat

Hugs> valday 4
Wed
```

Constructors & Patterns

- Constructors can be matched by *patterns*.
- For example :

```
dayval :: Day -> Integer
dayval Sun = 1
dayval Mon = 2
dayval Tue = 3
dayval Wed = 4
dayval Thu = 5
dayval Fri = 6
dayval Sat = 7

Hugs> dayval Wed
4
```

Other Enumeration Data Type Examples

```
data Bool = True | False  -- predefined in Haskell
     deriving Show
data Direction = North | East | South | West
     deriving Show
data Move = Paper | Rock | Scissors
     deriving Show

beats :: Move -> Move
beats Paper    = Scissors
beats Rock     = Paper
beats Scissors = Rock

Hugs> beats Paper
Scissors
```

Variant Records

- More complicated data types:

```
data Tagger = Tagn Integer | Tagb Bool
```

- These constructors are not constants – they are functions:

```
Tagn :: Integer -> Tagger
Tagb :: Bool    -> Tagger
```

- As for all constructors, something like "`Tagn 12`"
    - Cannot be simplified
      (and thus, as discussed in Chapter 1, it is a *value*).
    - Can be used in patterns.

Example functions on Tagger

```
number  (Tagn n) = n
boolean (Tagb b) = b

isNum (Tagn _) = True
isNum (Tagb _) = False

isBool x = not (isNum x)

Hugs> :t number
number :: Tagger -> Integer

Hugs> number (Tagn 3)
3

Hugs> isNum (Tagb False)
False
```

Another Variant Record Data Type

```
data Temp = Celsius  Float
          | Fahrenheit  Float
          | Kelvin  Float
```

- We can use patterns to define functions over this type:

```
toKelvin (Celsius c)    = Kelvin (c + 272.0)
toKelvin (Fahrenheit f) =
      Kelvin ( 5/9*(f-32.0) + 272.0 )
toKelvin (Kelvin k)     = Kelvin k
```

Finally: the **Shape** Data Type from the Text

- The **Shape** data type from Chapter 2 is another example of a variant data type:

```
data Shape = Rectangle Float Float
           | Ellipse Float Float
           | RtTriangle Float Float
           | Polygon [(Float,Float)]
deriving Show
```

- The last line – "**deriving Show**" – tells the system to build a **show** function for the type **Shape** (more on this later).
- We can also define functions yielding refined shapes:

```
circle, square :: Float -> Shape
circle radius = Ellipse radius radius
square side   = Rectangle side side
```

Functions over `shape`

- Functions on shapes can be defined using pattern matching.

```
area :: Shape -> Float
area (Rectangle s1 s2)   =  s1*s2
area (Ellipse r1 r2)     =  pi*r1*r2
area (RtTriangle s1 s2) = (s1*s2)/2
area (Polygon (v1:pts)) =  polyArea pts
   where polyArea :: [(Float,Float)] -> Float
         polyArea (v2:v3:vs) = triArea v1 v2 v3 +
                                 polyArea (v3:vs)
         polyArea _          = 0
```

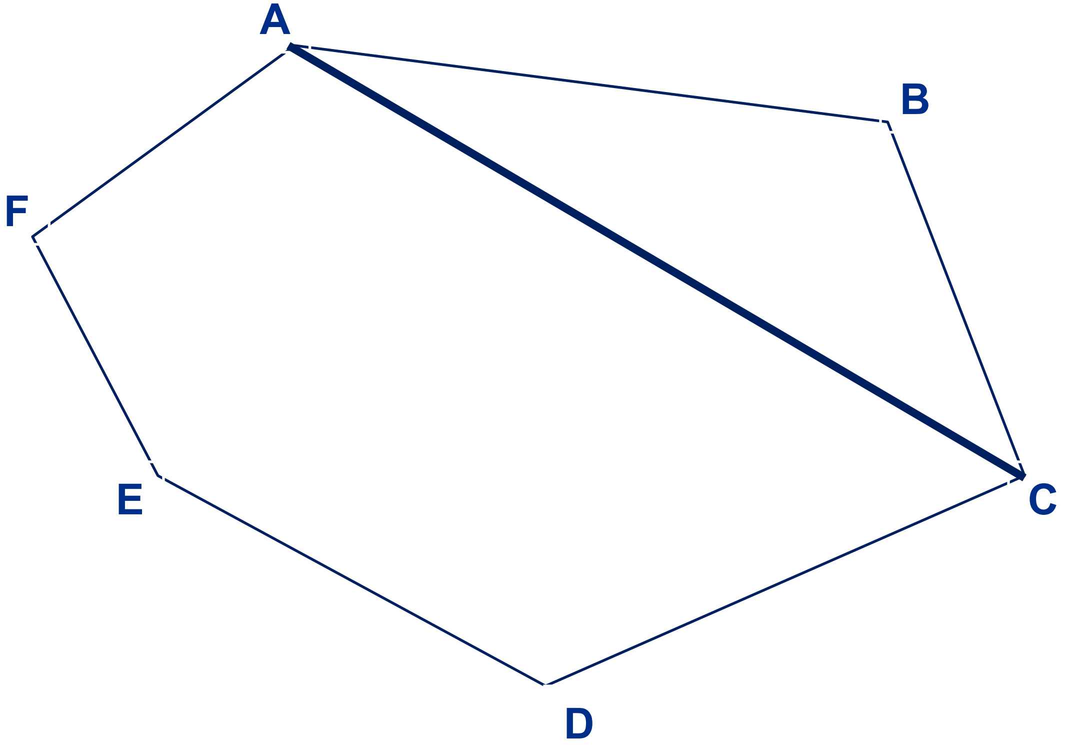
Note use of auxiliary function.

Note use of nested patterns.

Note use of wild card pattern (which matches anything).

Algorithm for Computing Area of Polygon

totalArea = area (triangle [A,B,C]) + area (polygon[A,C,D,E,F])



TriArea

```
triArea v1 v2 v3 =
  let a = distBetween v1 v2
      b = distBetween v2 v3
      c = distBetween v3 v1
      s = 0.5*(a+b+c)
  in sqrt (s*(s-a)*(s-b)*(s-c))


distBetween (x1,y1) (x2,y2)
   = sqrt ((x1-x2)^2 + (y1-y2)^2)
```

Selamat Belajar dan Berlatih!

Silahkan baca bab 1-3 buku Haskell School of Expression

atau bab 1-4 buku Haskell The Craft of Functioal Programming