Advanced Functional Programming, 1DL450

Lecture 7, 2012-11-19

Cons T Åhs

Haskell Basics

- Named after Haskell Curry (1900-1982)
- A well crafted and designed pure functional programming language
  - strongly based on λ-calculus and properties of it
    - partial application is trivial
  - polymorphically staticially typed
    - type inference and checking at compile time catches problems early
    - types are an important and natural part of a program
    - type variables essential
  - lazy evaluation
    - evaluation on demand, at most once
    - elegant expression

Values and Types

```
            17  :: Integer
           'c'  :: Char
         "foo"  :: String
       [1,2,3]  :: [Integer]
['1', '2', '3'] :: [Char]
(1, '1', "foo") :: (Integer, Char, String)
          succ :: Integer -> Integer
```

- Fairly standard syntax for primitive instances and types
- Note syntax for for tuples
- Typing is strict so you can't have a list of mixed types

Functions

```
succ :: Integer -> Integer
succ n = n + 1

fac :: Integer -> Integer
fac 0 = 1
fac n = n * fac (n - 1)

listlen :: [a] -> Integer
listlen [] = 0
listlen (x:xs) = 1 + listlen xs
```

- Functions are written as equations
- Several clauses can be used
- Note inconsistent patterns for lists
- types for functions can be included, but no harm in not doing so
  - the compiler will infer types and complain
  - adding types adds readability
- Note type variable for `listlen` - a list of any type

Types

```
data State = On | Off
data NamedColor =
   Black | Red | Green | Blue | White | Cyan | Magenta | Yellow

data ColorSpec = RGB (Integer, Integer, Integer) | Named NamedColor

data Color = RGBA (Integer, Integer, Integer, Integer)

to_color (Named nc) = to_color (name_to_color nc)
to_color (RGB (r, g, b)) = RGBA (r, g, b, 255)
```

```
name_to_color Black = RGB(0, 0, 0)
name_to_color White = RGB(255, 255, 255)
  ...
```

- Create your own types by enumerating constants and constructors
  - Constants and constructors start with uppercase
- A type used in another type has to be wrapped in a constructor - why?

Polymorphic Types

```
data Pair a = Pr a a
data Tuple a b = Tpl a b
data Fruit = Apple | Orange | Pear | Banana

type PriceMap = Tuple Fruit Integer
type String = [Char]

type Map a b = [(a, b)]
```

- `Pair` requires that both arguments are of the same type
- `Tuple` can pair two different types
- Synonyms for types can be defined for convenience

Recursive Types

```
data Tree a = Empty | Node a (Tree a) (Tree a)

Empty :: Tree a
Node :: a -> Tree a -> Tree a -> Tree a

depth :: Tree a -> Integer
depth Empty = 0
depth (Node x left right) = 1 + max (depth left) (depth right)

data Sexpr a = Leaf a | Cons (Sexpr a) (Sexpr a)

traverse (Leaf x) = [x]
traverse (Cons car cdr) = (traverse car) ++ (traverse cdr)
```

- Constructors have types - they are functions
  - Note that `Empty` by itself does not form a complete type
- Use function clauses corresponding to type definition
- Haskell also has ++ for list concatenation

List Comprehensions

```
[(x,y)| x <- [1,2,3], y <- [2,3,4], x*y < 6]
[(1,2),(1,3),(1,4),(2,2)]

qsort [] = []
qsort (x:xs) =
   qsort [y | y <- xs, y < x] ++ [x] ++ qsort [y | y <- xs, y >= x]
```

- Haskell has list comprehensions, very similar to Erlang, but take care to distinguish between one or two vertical bars

Functions

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

Integer -> Integer -> Integer
add x y = x + y

mapcar :: (a -> b) -> [a] -> [b]
mapcar f []     = []
mapcar f (x:xs) = f x : mapcar f xs
```

```
inclist :: [Integer] -> [Integer]
inclist = mapcar (add 1)
```

- Note difference between types of `sum` and `add`
- Functions are curried in the normal case - this is useful
- Partial application is trivial
- We can define functions with knowing about number of arguments or mentioning arguments
- Program can become very compact - but not always too readable
- Haskell has a built in `map`

Anonymous Functions

```
mapex1 = map (\x -> x + 1)

mapex2 = map (\x y -> x + y)
```

- Very nice notation for anonymous functions - almost optimal
- What do the above expressions return?

```
mapex1 :: [Integer] -> [Integer]

mapex2 :: [Integer] -> [Integer -> Integer]
```

Infix and Prefix

```
*Main> map (+3) [1,2,3]
[4,5,6]

*Main> map (3+) [1,2,3]
[4,5,6]

*Main> map ("the " ++) ["table","language","larch tree"]
["the table","the language","the larch tree"]

*Main> map (++" jam") ["blueberry", "traffic", "monster"]
["blueberry jam","traffic jam","monster jam"]
```

- Even infix operators can be applied partially
- Note that for a non commutative operator order matters

Infix and Prefix

```
[] @@ ys = ys
(x:xs) @@ ys = x : (xs@@ys)

x +++ y = x - y
x ++- y = x - y

infixr 5 +++
infixl 5 ++-



*Main> 2 +++ 3 +++ 4
3
*Main> 2 ++- 3 ++- 4
-5
```

- Apart from the built in infix operators, you can define your own
- Infix operators are built from non alphanumeric characters
- Take care regarding associativity
- Extensive use of infix operators might make for compact, but difficult to read programs - be careful.
  - Good names are easier to understand than symbols

Functional Composition

```
-- same as built in operator . (a period)
compose f g = \x -> f (g x)


*Main> (fac . length) "foo"
6
```

- Function composition is easy and useful
- Syntax rather anonymous - easy to not see it
- Composition is not commutative
- What is the type of functional composition?

Lazy evaluation

```
-- ok definition, but will not return
away x = away x


*Main> length [away 10, fib 100, fib 1000]
3
```

- We get a “correct” answer immediately despite having a list of one undefined element and two very long running computations
- Haskell is lazy, i.e., computes a value only when needed
  - none of the elements in the list are actually computed
  - computation is non strict, i.e., functions getting undefined arguments might still return a well defined answer
- Lazy evaluation
  - efficient since we evaluate a value at most once
  - can be surprising since evaluation order is non intuitive
- How can lazy evaluation be implemented?

Lazy & infinite data structures

```
ones = 1 : ones

fromn n = n : fromn (n + 1)

squares = map (\x -> x*x) (fromn 0)

evensquares = filter even squares

oddsquares = [x | x <- squares, odd x]

*Main> take 10 evensquares
[0,4,16,36,64,100,144,196,256,324]

*Main> take 10 oddsquares
[1,9,25,49,81,121,169,225,289,361]
```

- Since we don't evaluate until a value is asked for there is no harm in describing infinite data structures
  - avoid certain operations, such as asking for the length of them
- This magic is everywhere, i.e., we don't need to do anything special do use it
  - `filter` and list comprehension
- How much space does `ones` and `squares` take?

Lazy & infinite data structures

```
sumlists (x:xs) (y:ys) = (x+y):sumlists xs ys
sumlists    xs     ys  = []

-- an infinite list of fibonacci numbers
fibs = 0 : 1 : sumlists fibs (tail fibs)

*Main> take 15 fibs
[0,1,1,2,3,5,8,13,21,34,55,89,144,233,377]

*Main> take 15 (filter even fibs)
[0,2,8,34,144,610,2584,10946,46368,196418,832040,3524578,14930352,6
3245986,267914296]

*Main> take 15 (filter odd fibs)
[1,1,3,5,13,21,55,89,233,377,987,1597,4181,6765,17711]
```

- Programming with (infinite) streams of data rather finite lists
- Life is simpler since we can ignore the pesky base cases :-)
  - on the other hand, streams might end so we should take care of them

Lazy & infinite data structures

```
zip1 (x:xs) (y:ys) = (x, y) : zip1 xs ys
zip1    xs     ys  = []

map2 f (x:xs) (y:ys) = (f x y) : map2 f xs ys
map2 f xs ys = []

zip2 = map2 (\x y -> (x, y))

fibs2 = 0 : 1 : [x+y | (x,y) <- zip fibs2 (tail fibs2)]
```

```
map22 f xs ys = [f x y | (x, y) <- zip xs ys]

fibs3 = 0 : 1 : map22 (\x y -> x + y) fibs3 (tail fibs3)
```

- zip is useful, taking two streams/lists and constructing one stream/list of them
- some alternative ways of defining an infinite stream of fibonacci numbers

Lazy & infinite data structures

```
sieve n = filter (\m -> not ((mod m n) == 0))

sieves (n:ns) = n : sieves (sieve n ns)

primes = sieves ([2..])
```

- Notation [n..m] gives a list of integers n to m (inclusive)
- Notation [n..] gives a list of integers from n upwards
- Only three lines of code for the list of all prime numbers!

Pattern matching

- Similar to ML
  - Recall example for colour
  - no repeated variables in patterns
  - no match “non exhaustive” warning - runtime error instead
- case very similar to Erlang (but less powerful pattern matching)
  - must be type correct, i.e., all clauses return the same type

```
take 0 x = []
take n [] = []
take n (x:xs) = x : take (n-1) xs

-- equivalent definition
take n xs =
  case (n, xs) of
    (0,  _)    -> []
    (_, [])    -> []
    (m, y:ys) -> y : take (n-1) ys
```

Correct IF!

```
if e1 then e2 else e3

-- equivalent to
case e1 of
  True  -> e2
  False -> e3
```

- `if` is correct, but a bit verbose
- λ is quite neat and compact, but they still mess up `if`..

Local variables

```
accreverse l =
  let rev [] a = a
      rev (x:xs) a = rev xs (x:a)
  in rev l []
```

- `let` similar to `LET` in Lisp, but can be used for recursive functions
  - no distinction between values and functions (as in Lisp)

Local variables

```
quiksort p [] = []
quiksort p (x:xs) =
   (quiksort p lesser) ++ [x] ++ (quiksort p greater)
   where lesser  = filter (not . (p x)) xs
         greater = filter (p x) xs
```

- `where` gives values after main expression, a backwards `let`

Layout matters!

- Grouping of expressions (in, e.g., `let`, `where`, `case`) and closing of definitions etc is determined by the layout
- The column where you write is thus important
  - relation to previous (sub) parts of expression is important
- Sigh, yet another language making this mistake
  - Bad Idea in the 60s for FORTRAN and PL/1
  - Still a Bad Idea
- You have been warned.