Advanced features in Haskell

Anders Møller
amoeller@cs.au.dk

Overview

- Building abstractions using **higher-order** functions, **polymorphic** functions, and **parameterized** types
- Structuring programs using **type classes** and **modules**
- **Lazy evaluation** – the evaluation strategy used in Haskell
- **Proving properties** of programs in a purely functional language
- **Side-effects** in a purely functional language

Building abstractions using higher-order functions

```
listsum [] = 0
listsum (x:xs) = x + listsum xs

listprod [] = 1
listprod (x:xs) = x * listprod xs
```

- The two functions perform a **similar traversal** of the list, but with different operations being performed
- Let us make an **abstraction** of the traversal! (like the visitor pattern)

Building abstractions using higher-order functions

```
fold op init [] = init
fold op init (x:xs) = x `op` fold op init xs

listsum = fold (+) 0
listprod = fold (*) 1
```

- `fold op init [x1,x2,...,x42]` ⇒ `(x1 `op` (x2 `op` ... (x42 `op` init)...))`
- Now the traversal is **separated** from the operations
- `listsum` and `listprod` are now defined more concisely and **without recursion**
- This technique is **very common** in functional programming for all kinds of data structures

Polymorphic functions (parametric polymorphism)

```
map f [] = []
map f (x:xs) = f x : map f xs
```

- What is the type of **map**?
- Possible answers:

```
map :: (Integer -> Integer) ->
       [Integer] -> [Integer]
map :: (Integer -> String) ->
       [Integer] -> [String]
map :: ([String] -> [Integer]) ->
       [[String]] -> [[Integer]]
```

- They are all correct!

5

Type schemes

```
map f [] = []
map f (x:xs) = f x : map f xs
```

- The **principal** type of map:

  map :: (a -> b) -> [a] -> [b]

  for *any* **a** and **b** ← "type variables"

- Type schemes are sometimes written with universal quantifiers:

  ∀a,b: (a -> b) -> [a] -> [b]

- In Java: `List<B> map(Map<A,B> f, List<A> zs)`

6

Quiz!

We earlier defined `fold` as follows:

```
fold op init [] = init
fold op init (x:xs) = x `op` fold op init xs
```

What is the (principal) **type** of `fold`?

Hint: fold is a **polymorphic**, **higher-order** function

Hint 2: the type must have the form ... -> ... -> ... -> ...

7

Types of constructors

- Constructors are a special kind of functions

```
data Expr = Const Float
          | Add Expr Expr
          | Neg Expr
          | Mult Expr Expr

x = eval (Add (Const 3.0) (Const 4.0))
```

- `Const :: Float -> Expr`
- `Add :: Expr -> Expr -> Expr`
- `Neg :: Expr -> Expr`
- `Mult :: Expr -> Expr -> Expr`

8

Building abstractions using parameterized types

```
abstract class Expr {}

class Const extends Expr {
  Float x;
  Const(Float x) {this.x=x;}
}

class Add extends Expr {
  Expr e1,e2;
  Add(Expr e1,Expr e2) {this.e1=e1;
                        this.e2=e2;}
}

…
```

9

Building abstractions using parameterized types

- Type definitions can be parameterized

```
data Expr = Const Float
          | Add Expr Expr
```

```
data Expr a = Const a
            | Add (Expr a) (Expr a)
            | Neg (Expr a)
            | Mult (Expr a) (Expr a)

type FloatExpr = Expr Float
```

- Now `Expr` is a *parameterized* type
  - it takes a type as "argument" and "returns" a type
- Like generic classes in Java!

10

Building abstractions using parameterized types

```
abstract class Expr<A> {}

class Const<A> extends Expr<A> {
  A x;
  Const(A x) {this.x=x;}
}

class Add<A> extends Expr<A> {
  Expr<A> e1,e2;
  Add(Expr<A> e1,Expr<A> e2) {this.e1=e1;
                              this.e2=e2;}
}

…
```

11

Terminology review

- **Higher-order function**: a function that takes another function as argument and/or returns one as result
- **Polymorphic function**: a function that works with arguments of many possible types
- **Type scheme**: a type that involves type variables
  - the type of a polymorphic function is a type scheme
- **Parameterized type**: a type that takes another type as "argument" and "returns" a type
  - their constructors are often polymorphic functions

12

Overview

- Building abstractions using higher-order functions, polymorphic functions, and parameterized types
- **Structuring programs using type classes and modules**
- Lazy evaluation – the evaluation strategy used in Haskell
- Proving properties of programs in a purely functional language
- Side-effects in a purely functional language

13

Qualified types and type classes

- In the type schemes we have seen, the type variables are **universally quantified**
  (example: `map :: (a -> b) -> [a] -> [b]`)
- ...in other words, the definition of `map` can assume *nothing* of the corresponding input
- What is the (principal) type of `qsort`?
  - we want it to work on **any list whose elements are comparable**,
  - but require nothing else
- The solution: qualified types!

14

The type of `qsort`

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

```
qsort :: Ord a => [a] -> [a]
```

- The type variable a is now **qualified** with the **type class** `Ord`
- Now `qsort` works with any list whose elements are instances of the `Ord` type class!

15

Type classes and instances

```
class Ord a where
  (<) :: a -> a -> Bool
  (>=) :: a -> a -> Bool
```

defines a *type class* named `Ord`

```
data Person = Student Name Score
type Name = String
type Score = Integer

lowerScore :: Person -> Person -> Bool
lowerScore (Student n1 s1) (Student n2 s2) = s1 < s2
```

```
instance Ord Person where
  x < y = lowerScore x y
  x >= y = not (lowerScore x y)
```

makes `Person` an *instance* of `Ord`

(The actual `Ord` type class in the standard prelude is a little longer)

16

More about type classes

- Haskell's type class mechanism has some parallels to **Java's interface classes**
- Ad-hoc polymorphism (also called **overloading**)
  - for example, the < and >= operators are overloaded
  - the `instance` declarations control how the operators are implemented for a given type

17

Standard type classes

- `Ord` – used for totally ordered data types
- `Show` – like an interface for Java's `toString()`
- `Eq` – equality (==, /=)
- `Num` – functionality common to all kinds of numbers
- ...

18

Example: equality on booleans

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

```
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
```

```
instance Eq Bool where
  True == True = True
  False == False = True
  _ == _ = False
  x /= y = not (x == y)
```

(how is **`not`** defined?)

19

Quiz!

1. In what sense is `qsort` **polymorphic**?
2. What are **qualified** types used for?

20

Modules

- Modularization features provide
  - **encapsulation**
  - **reuse**
  - **abstraction**

- A module **requires** and **provides** functionality
- In Haskell:

```
module Calculator (Expr,eval,run_gui) where
import Math
import Graphics
…
```

21

Overview

- Building abstractions using higher-order functions, polymorphic functions, and parameterized types
- Structuring programs using type classes and modules
- **Lazy evaluation – the evaluation strategy used in Haskell**
- Proving properties of programs in a purely functional language
- Side-effects in a purely functional language

22

Call-by-value vs. call-by-need

```
switch n e1 e2
   | n > 0 = e1
   | otherwise = e2
```

- How is `switch m (f x) (g y)` evaluated?
- In Java:
  (where the expression looks like `switch(m,f(x),g(y))`)
  - *first* evaluate each argument
  - *then* evaluate the function body with the argument values in place of the formal parameters
- In Haskell:
  - start evaluating the function body
  - evaluate the arguments "**by need**"

23

Lazy evaluation

```
switch n e1 e2
   | n > 0 = e1
   | otherwise = e2
```

- How is `switch m (f x) (g y)` evaluated?
- *Evaluation is **lazy**!*
  - e.g. `(f x)` is only evaluated at runtime if actually needed
  - if the body uses the argument multiple times, it is only evaluated once

24

Streams – a typical use of laziness

- A *producer* of an **infinite stream** of integers:

```
fib = 1:fib2
fib2 = 1:fib3
fib3 = add fib fib2
  where add (x:xs) (y:ys) = (x+y) : add xs ys
```

- A *consumer* of an infinite stream of integers:

```
consumer stream n =
  if n == 1 then show head
  else show head ++ ", " ++ consumer tail (n-1)
  where head:tail = stream
```

- `consumer fib 10` ⇒
  "1, 1, 2, 3, 5, 8, 13, 21, 34, 55"

25

Drawbacks of lazy evaluation

- Compared to eager evaluation:
  - More difficult to reason about performance (time and space)
  - Runtime overhead

26

Overview

- Building abstractions using higher-order functions, polymorphic functions, and parameterized types
- Structuring programs using type classes and modules
- Lazy evaluation – the evaluation strategy used in Haskell
- **Proving properties of programs in a purely functional language**
- Side-effects in a purely functional language

27

Proving properties of programs

- Last week we saw two definitions of 'reverse':

```
reverse1 [] = []
reverse1 (x:xs) = reverse1 xs ++ [x]
```

```
reverse2 zs = rev2 [] zs
    where rev2 acc [] = acc
          rev2 acc (x:xs) = rev2 (x:acc) xs
```

- How do we **prove** that these two definitions are functionally equivalent?

28

Proof by induction

- We want to show: ∀*as*: `reverse1` *as* = `reverse2` *as*
  i.e. ∀*as*: `reverse1` *as* = `rev2 []` *as*
- The functions work by recursion, so let's try the proof technique of **induction**!
- During the recursion, *acc* can be anything so we need to generalize the statement slightly:
  ∀*as*,*acc*: `reverse1` *as* ++ *acc* = `rev2` *acc as*

```
reverse1 [] = []
reverse1 (x:xs) = reverse1 xs ++ [x]
```

```
reverse2 zs = rev2 [] zs
    where rev2 acc [] = acc
          rev2 acc (x:xs) = rev2 (x:acc) xs
```

29

Reasoning in a purely functional language

What to learn from the proof:

- Reasoning in a purely functional language means that
  - we do not have to consider potential side-effects
    - modification of global state
    - exceptions
    - aliasing
    - …
  - we do not have to consider order or duplication of execution steps
- ... and this is of course also useful when reasoning less formally

30

Overview

- Building abstractions using higher-order functions, polymorphic functions, and parameterized types
- Structuring programs using type classes and modules
- Lazy evaluation – the evaluation strategy used in Haskell
- Proving properties of programs in a purely functional language
- **Side-effects in a purely functional language**

31

Side-effects in a purely functional language

NOTICE:

- The following four slides do not go into the same level of detail as the rest of the lecture
- The intention of showing them is to indicate why Haskell is more than just a toy programming language...

32

Side-effects in a purely functional language

- We really need side-effects in practice!
  - I/O (communication with the outside world)
  - exceptions
  - mutable state
  - ...
- *How can such **imperative** features be incorporated in a **purely functional** language?*

33

The `IO` type class

- **Action**: a special kind of value
  - e.g. read from keyboard or write to a file
  - must be ordered in a well-defined manner for program execution to be meaningful
- **Command**: expression that evaluates to an action
- `IO` *`T`*: the type of a command that yields a value of type *`T`*
  - `getLine :: IO String`
  - `putStr :: String -> IO ()`
- Sequencing IO operations:

```
(>>=) :: IO a -> (a -> IO b) -> IO b
```

34

Example of command sequencing

- First read a string from input, then write a string to output:

```
getLine >>= \s -> putStr ("Simon says: " ++ s)
```

- An alternative, more convenient syntax:

```
do s <- getLine
   putStr ("Simon says: " ++ s)
```

- This looks very "imperative", but all the side-effects are controlled via the `IO` type class!

35

Monads – the magic of Haskell

- `IO` is merely an instance of the more general type class **`Monad`**
- Another application of `Monad` is for simulating mutable state
- The concept of *monads* comes from a branch of mathematics known as category theory...

36

Summary

- **Higher-order functions**, **polymorphic functions**, and **parameterized types** are useful for building abstractions
- **Type classes** and **modules** are useful mechanisms for structuring programs
- **Lazy evaluation** allows programming with infinite data structures
- Reasoning about program behavior is often easier when the language is **purely functional**
- Using **monads**, we can control **side-effects** in a purely functional language

37