An introduction to functional programming using Haskell

Anders Møller
amoeller@cs.au.dk

Haskell

- The most popular purely functional, lazy programming language
- "Functional programming language":
  - a program is a collection of mathematical functions
- "Purely functional":
  - all variables refer to immutable, persistent values
  - that is, new values can be created, but existing ones cannot be modified!
- "Lazy":
  - expressions are evaluated “by need”

Example: QuickSort in Java

```java
void qsort(int[] a, int lo, int hi) {
    if (lo < hi) {
        int l = lo;
        int h = hi;
        int p = a[lo];
        do {
            while (l < h && a[l] <= p) l = l+1;
            while (h > l && a[h] >= p) h = h-1;
            if (l < h) {
                int t = a[l];
                a[l] = a[h];
                a[h] = t;
            }
        } while (l < h);
        int t = a[h];
        a[h] = a[lo];
        a[lo] = t;
        qsort(a, lo, h-1);
        qsort(a, h+1, hi);
    }
}
```

Example: QuickSort in Haskell

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

The program is the specification!

List operations:

- [] the empty list
- : adds an element x to the head of a list xs
- ++ concatenates two lists xs and ys
- x:y:z abbreviation of x:(y:(z:[]))
Overview

- Definitions, functions, expressions, and values
- Strong typing

- Recursive data structures
- Higher-order functions

- Programming styles, performance
- Reasoning about program behavior

Definitions, functions, expressions, and values

- Function application can be written without parentheses
- Functions are reminiscent of methods in Java, but ...

Evaluation through pattern matching and expression rewriting

\[
\begin{align*}
\text{len} & \text{[]} = 0 \\
\text{len} (x:xs) &= \text{len} \text{ } xs + 1 \\
\text{numbers} &= [7,9,13] \\
n &= \text{len} \text{ } \text{numbers}
\end{align*}
\]

- \( n \Rightarrow \text{len} \text{ } \text{numbers} \Rightarrow \text{len} [7,9,13] \Rightarrow \text{len} 7:[9,13] \Rightarrow \text{len} [9,13] + 1 \Rightarrow \text{len} 9:[13] + 1 + 1 \Rightarrow \text{len} 13:[1] + 1 + 1 \Rightarrow \text{len } [1] + 1 + 1 + 1 \Rightarrow 0 + 1 + 1 + 1 \Rightarrow 1 + 1 + 1 \Rightarrow 2 + 1 \Rightarrow 3 \)

- Actual evaluation may proceed in a different order, but the resulting value is the same

let expressions

\[
\begin{align*}
\text{len} & \text{[]} = 0 \\
\text{len} (x:xs) &= \text{len} \text{ } xs + 1 \\
\text{numbers} &= [7,9,13] \\
n &= \text{let} \text{ } \text{len} [1] = 0 \\
\text{len} (x:xs) &= \text{len} \text{ } xs + 1 \\
n &= \text{len} \text{ } \text{numbers}
\end{align*}
\]
where clauses

```haskell
len [] = 0
len (x:xs) = len xs + 1

numbers = [7,9,13]
n = len numbers
  where len [] = 0
        len (x:xs) = len xs + 1

numbers = [7,9,13]
n = let len [] = 0
    in len numbers

numbers = [7,9,13]
n = len numbers
  where len [] = 0
        len (x:xs) = len xs + 1
```

case expressions

```haskell
len [] = 0
len (x:xs) = len xs + 1

numbers = [7,9,13]
n = len numbers
  where len [] = 0
        len (x:xs) = len xs + 1

len ys = case ys of
  [] -> 0
  x:xs -> len xs + 1

numbers = [7,9,13]
n = len numbers
```

Wildcard pattern

```haskell
len [] = 0
len (x:xs) = len xs + 1

numbers = [7,9,13]
n = len numbers
  where len [] = 0
        len (x:xs) = len xs + 1

numbers = [7,9,13]
n = let len [] = 0
    in len numbers

numbers = [7,9,13]
n = len numbers
  where len [] = 0
        len (x:xs) = len xs + 1
```

Pattern matching, if-then-else, and guards

- Multiple function definitions with the same function name
- case expressions
- if-then-else expressions:
  ```haskell
  if e1 then e2 else e3 = case e1 of
    True -> e2
    False -> e3
  ```
- Guards:
  ```haskell
  sign x | x > 0 = 1
  | x == 0 = 0
  | x < 0 = -1
  ```
Iteration and recursion

- We don't need loops when we have recursion
- Divide and conquer!

```
len [] = 0
len (x:xs) = len xs + 1
```

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

Layout of Haskell programs

```
numbers = [7,9,13]
n = let {len [] = 0
         len (x:xs) = len xs + 1
         len numbers
      }
```

```
numbers = [7,9,13]
n = let len [] = 0
    len (x:xs) = len xs + 1
    len numbers
```

```
numbers = [7,9,13]
n = let {len [] = 0; len (x:xs) = len xs + 1}
    len numbers
```

Quiz!

Write a function `rev` that reverses a given list

for example,

```
rev [7,9,13] ⇒ [13,9,7]
```

(don't worry about performance for now...)

```
{-
  numbers = [7,9,13]
  n = len numbers
-}

Hint: recall the definition of `len`:

```
len [] = 0
len (x:xs) = len xs + 1
```
Overview

- Definitions, functions, expressions, and values
- **Strong typing**
  - Recursive data structures
  - Higher-order functions
  - Programming styles, performance
  - Reasoning about program behavior

Types

- **Integer, String, Float, Char, ...** – base types
- **[X]** – a list of X values
- **X → Y** – a **function** from X values to Y values
- **(X, Y, Z)** – a **tuple** of an X value, a Y value, and a Z value
- ...

```haskell
len :: [Integer] → Integer
len [] = 0
len (x:xs) = len xs + 1

numbers :: [Integer]
n = len numbers
```

- **Every expression has a type**
- The Haskell **static type checker** makes sure you are not mixing apples and bananas

Type errors at compile-time

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

- Can you spot the error?
- The Hugs tool:
  - **ERROR:** "example.hs":42 - Cannot infer instance
    - *** Instance : Num [a]***
    - *** Expression : qsort***
- The error is found, but the quality of type error messages is not always great...
Type inference

- A type annotation is a contract between the author and the user of a function definition
- In Haskell, writing type annotations is optional
  - use them to express the intended meaning of your functions
  - omit them if the meaning is "obvious"
- (Compare this with Java!)
- When omitted, Haskell finds the type for you!
- ...in fact, the best possible type!
  (also called, the principal type)
- More about type inference later...

Overview

- Definitions, functions, expressions, and values
- Strong typing

- Recursive data structures
- Higher-order functions
- Programming styles, performance
- Reasoning about program behavior

Type definitions

```haskell
data Color = Red | Green | Blue

f Red = 1
f Green = 2
f Blue = 3
```

```haskell
data Shape = Rectangle Float Float
           | Circle Float

area (Rectangle x y) = x * y
area (Circle r) = pi * r * r
```

- Red, Green, Blue, Rectangle, and Circle are called constructors

Constructors vs. pattern matching

- Constructors are special functions that construct values
  example:
  ```haskell
  Rectangle 3.0 4.0
  ```
  constructs a Shape value

- Pattern matching can be used to "destruct" values
  example:
  ```haskell
  getX (Rectangle x y) = x
  ```
  defines a function that can extract the x component of a Rectangle value
Recursive data structures

- Type definitions can be recursive!

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

eval (Const c) = c
eval (Add e1 e2) = eval e1 + eval e2
eval (Neg e) = - eval e
eval (Mult e1 e2) = eval e1 * eval e2
```

Type synonyms

- Type synonyms are just abbreviations...

```haskell
type String = [Char]
type Name = String
data OptionalAddress = None | Addr String
type Person = (Name,OptionalAddress)
```

Naming requirements

- The naming style we have been using is mandatory:
  - **Type** names and constructor names begin with an *upper-case* letter (e.g. Expr or Rectangle)
  - **Value** names begin with a *lower-case* letter (e.g. qsort or x)

Quiz!

Given the type definition

```haskell
data TreeNode = Leaf Integer
  | Branch TreeNode TreeNode
```

write a function `leafSum` that computes the sum of the leaf values for a given TreeNode structure
Overview

- Definitions, functions, expressions, and values
- Strong typing
- Recursive data structures
- Higher-order functions
- Programming styles, performance
- Reasoning about program behavior

Higher-order functions

- Functions are first-class values! (unlike methods in Java...)
- Functions can take functions as arguments and return functions

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

```
numbers = [7,9,13]
inc x = x + 1
more_numbers = map inc numbers
```

- Function application associates to the left:
  \( f \ x \ y = (f \ x) \ y \)

Currying

- \( add_A :: (\text{Integer, Integer}) \to \text{Integer} \)
- \( add_A (x,y) = x + y \)
- \( add_B :: \text{Integer} \to \text{Integer} \to \text{Integer} \)
- \( add_B \ x \ y = x + y \)
- \( inc = add_B \ 1 \)

- \( add_A \) takes a pair of integers as argument and returns their sum
- \( add_B \) takes one integer as argument and returns a function that takes another integer as argument and returns their sum
- This trick is named after Haskell B. Curry (1900-1982)

Anonymous functions

- A \( \lambda \)-abstraction is an anonymous function
- Traditional syntax:
  \( \lambda \ x . \ exp \) where \( x \) is a variable name and
  \( exp \) is an expression that may use \( x \)
  \( (\lambda \) is like a quantifier)
- Haskell syntax:
  \( \ \backslash \ x \ \to \ exp \)
  \( inc \ x = x + 1 \)
  \( \Rightarrow inc = \ \backslash x \ \to \ x + 1 \)
  \( add \ x \ y = x + y \)
  \( \Rightarrow add = \ \backslash x \ \to \ \backslash y \ \to \ x + y \)
**Infix operators**

- Infix operators (e.g. + or ++) are just binary functions!
  \[ x + y = (+) \times y \]

- Binary functions can be written with an infix notation
  \[ \text{add}\ x\ y = x \text{add}\ y \]

- Operator precedence and associativity can be specified with "fixity declarations"

---

**Haskell standard prelude**

- A library containing commonly used definitions
- Examples:
  \[ [] ++ ys = ys \]
  \[ (x:xs) ++ ys = x : (xs ++ ys) \]
  \[ \text{True} \land x = x \]
  \[ \text{False} \land _ = \text{False} \]
  \[ \text{type String} = \text{[Char]} \]
  \[ \text{data Bool} = \text{False | True} \]

- The core of Haskell is quite small
- Theoretically, everything can be reduced to the \(\lambda\)-calculus...

---

**List comprehensions**

- Lists are pervasive in Haskell...
  \[ \text{lt} = \{ y | y \leftarrow xs, y < x \} \]

- means the same as
  \[ \text{lt} = \text{concatMap}\ f\ xs \]
  \[
  \text{where}
  f\ y | y < x = [y]
  \text{otherwise} = []
  \]

- (concatMap is defined in the standard prelude)

---

**An operation on lists: zip**

- zip takes two lists as input (Curry style) and returns a list of pairs
  \[ \text{zip}\ [1,2,3] [\text{['a','b','c']}]
  \]
  evaluates to
  \[ [(1,'a'),(2,'b'),(3,'c')] \]

- How is zip defined?
  \[ \text{zip}\ (x:xs)\ (y:ys) = (x,y) : \text{zip}\ xs\ ys\]
  \[ \text{zip}\ [ ]\ ys = [ ]\]
  \[ \text{zip}\ xs\ [ ] = [ ]\]
Quiz!

Explain why the following 3 definitions have the same meaning – and describe what they do:

- \((f \cdot g) x = f (g x)\)
- \((.) f g x = f (g x)\)
- \(f \cdot g = \lambda x \rightarrow f (g x)\)

Overview

- Definitions, functions, expressions, and values
- Strong typing
- Recursive data structures
- Higher-order functions
- Programming styles, performance
- Reasoning about program behavior

Syntactic redundancy

<table>
<thead>
<tr>
<th>Expression style</th>
<th>Declaration style</th>
</tr>
</thead>
<tbody>
<tr>
<td>each function is defined as one expression</td>
<td>each function is defined as a series of equations</td>
</tr>
<tr>
<td><code>let</code></td>
<td><code>where</code></td>
</tr>
<tr>
<td><code>\lambda</code></td>
<td>arguments on left-hand-side of <code>=</code></td>
</tr>
<tr>
<td><code>case</code></td>
<td>function-level pattern matching</td>
</tr>
<tr>
<td><code>if</code></td>
<td>guards</td>
</tr>
</tbody>
</table>

A note on performance

- Being in math world doesn’t mean that we can ignore performance...
- Assume that we define `reverse` by

  ```haskell
  reverse [] = []
  reverse (x:xs) = reverse xs ++ [x]
  ```

- Recall that `++` is defined by

  ```haskell
  [] ++ ys = ys
  (x:xs) ++ ys = x : (xs ++ ys)
  ```

- Computing `reverse zs` takes time \(O(n^2)\) where \(n\) is the length of `zs`
A faster reverse using an accumulator

reverse zs = rev [] zs
  where rev acc [] = acc
  rev acc (x:xs) = rev (x:acc) xs

- reverse [7,9,13] ⇒ rev [] [7,9,13] ⇒
  rev [13,9,7] [] ⇒ [13,9,7]

- Computing reverse zs now takes time O(n)
  where n is the length of zs

- Moral: you have to know how programs are evaluated

Referential transparency

- Purely functional means that evaluation has no side-effects
  - a function maps input to output and does nothing else, just like a "real mathematical function"

- Referential transparency: "equals can be substituted for equals"
  - can disregard order and duplication of evaluation

Overview

- Definitions, functions, expressions, and values
- Strong typing
- Recursive data structures
- Higher-order functions
- Programming styles, performance
- Reasoning about program behavior

Referential transparency

- Easier for the programmer (and the compiler) to reason about program behavior
- Example:
  \[ f(x) \times f(x) \]
  always has the same behavior as
  \[ \text{let } z = f(x) \text{ in } z \times z \]
  (which is not true for Java!)
- More about proving properties of programs later...
<table>
<thead>
<tr>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Haskell is purely functional</td>
</tr>
<tr>
<td>- Recursive data structures</td>
</tr>
<tr>
<td>- Pattern matching</td>
</tr>
<tr>
<td>- Higher-order functions</td>
</tr>
<tr>
<td>- Strong typing</td>
</tr>
</tbody>
</table>

– advanced features in the next lecture 😊