Script:Higher-order functions in Haskell

Headline

Lecture "Higher-order functions in Haskell" as part ofCourse:Lambdas in Koblenz

Summary

Higher-order functions are functions that take functions as arguments or return functions as results. Much of the expressiveness and convenience of functional programming is a consequence of the status of functions to be first-class citizens. In this lecture, we focus on higher-order functions for list processing, e.g., the map function. We also look at important related concepts such as partial application of functions or lambda abstractions for constructing anonymous functions.

Video

Some version of the lecture has been recorded.

Higher order functions in Haskell

Concepts

- Higher-order function
- Partial application
- Polymorphism
- Map function
- Fold function
- Filter function
- Zip function
- Uncurrying
- Currying
- List comprehension
- Lambda abstraction

Languages

- Language:Haskell

Contributions

- haskellEngineer: No higher-order functions
- haskellList: Leverage map and sum
- haskellLambda: Leverage anonymous functions
- haskellProfessional: Richer demonstration

Metadata

- Course:Lambdas in Koblenz
- Script:Data modeling in Haskell

Anonymous function

Headline

A function without a name based on lambda abstraction

Metadata

- http://www.haskell.org/haskellwiki/Anonymous_function
- http://en.wikipedia.org/wiki/Lambda_calculus
- Vocabulary:Functional programming
- Concept

Currying

Headline

Transformation of a function on multiple arguments to take one at a time

Illustration

In Language:Haskell, functions are typically in curried form. Consider, for example, the signature of addition:

```
> :t (+)
(+) :: Num a => a -> a -> a
```

Curried form is generally convenient as it directly enablespartial application. For instance, we can define an increment function, just by passing one argument to addition:

```
inc :: Int -> Int
inc = (+) 1
```

If we were to define addition in uncurried form, then this would look as follows:

```
add :: Num a => (a, a) -> a
add (x,y) = x+y
```

We can easily perform currying:

```
add' :: Num a => a -> a -> a
add' x y = add (x,y)
```

In fact, the currying transformation can be represented by the followinghigher-order function:

```
curry :: ((a, b) -> c) -> a -> b -> c
curry f a b = f (a,b)
```

(We would need such a curry(ing) function for each number of arguments.) Thus, we can express the currying transformation as follows:

```
add'' :: Num a => a -> a -> a
add'' = curry add
```

Relationships

See uncurrying for the dual transformation.

Citation

(http://en.wikipedia.org/wiki/Currying, 18 May 2013)

In mathematics and computer science, currying is the technique of transforming a function that takes multiple arguments (or a tuple of arguments) in such a way that it can be called as a chain of functions, each with a single argument (partial application). It was originated by Moses SchÃ¶nfinkel ... and later re-discovered by Haskell Curry.

Metadata

- http://en.wikipedia.org/wiki/Currying
- Concept

Higher-order function

Headline

A Function that takes as an argument or returns a function

Illustration

A higher-order function is a function that takes functions as arguments or returns functions as results. Consider the following Language:Haskell function which applies a given argument function twice:

```
twice :: (x -> x) -> x -> x
twice f = f . f
```

For instance:

```
> twice (+1) 40
42
```

twice is clearly a higher-order function in that its first argument is of a function type "x -> x"*twice* is actually also a higher-order function in that it composes a function that it returns as result, namely "f . f". The function composition operator "." is another higher-order function, which is obvious from its type, again:

```
> :t (.)
(.) :: (b -> c) -> (a -> b) -> a -> c
```

For what it matters, the type itself already reveals that the second function is applied before the first function because of how type "a" of the third argument equals with the argument type of the second function, etc. Now it would be straightforward to define function composition (except that we do not need to do, as it is predefined evidently):

```
(.) f g x = f (g x)
```

See list processing functions such as map, fold, and filter for more examples of higher-order functions. See the concept of currying as well.

Metadata

- Vocabulary:Functional programming
- http://en.wikipedia.org/wiki/Higher-order_function
- Concept

List comprehension

Headline

A language construct for list processing

Illustration

Mapping over a list

Suppose you want to map over a list of numbers to increment each number. The following list comprehension implements this requirement:

```
> [ x+1 | x <- [1,2,3,4,5] ]
[2,3,4,5,6]
```

The expression after the bar "|" is the *generator* for elements *x*. The expression before the bar computes the element of the resulting list from the generated elements. The Map function actually models such element-wise mapping. Thus, the list comprehension could also be expressed as follows:

```
> map (\x -> x + 1) [1,2,3,4,5]
[2,3,4,5,6]
```

Filtering a list

Suppose you want to filter a list of numbers to only keep odd numbers. The following list comprehension implements this requirement:

```
> [ x | x <- [1,2,3,4,5], odd x ]
[1,3,5]
```

There are two expressions after the bar. The first one is a *generator* for elements *x*, the second is a *guard* to impose a condition of generated elements. The Filter function actually models filtering according to a guard (i.e., a predicate). Thus, the list comprehension could also be expressed as follows:

```
> filter odd [1,2,3,4,5]
[1,3,5]
```

Multiple generators and guards

List comprehensions can use multiple generators and guards, the scope of the generators extends to the subsequent generators and guards. For instance, we may collect all combinations of pairs of elements drawn from two separate lists as follows:

```
> [ (x,y) | x <- [1,2,3], y <- ['a','b'] ]
[(1,'a'),(1,'b'),(2,'a'),(2,'b'),(3,'a'),(3,'b')]
```

Let us compute the sum of all possible pairs with elements drawn from one list, where a guard is applied to reject pairs of identical elements:

```
> let sample = [1,2,3,4,5]
e> [ x + y | x <- sample, y <- sample, x /= y ]
[3,4,5,6,3,5,6,7,4,5,7,8,5,6,7,9,6,7,8,9]
```

Metadata

- Language construct

Partial application

Headline

Apply a function to some but not all arguments

Illustration

Here is an example of "non-partial" application of addition inLanguage:Haskell:

```
> 41 + 1
42
```

In this example, we actually increment 41. Let us thus define the increment function as a partial application of addition so that we can model the same computation as follows:

```
> let inc = (+) 1
> inc 41
42
```

It is important that "+" was surrounded by "(...)" because "+" is an infix operator and we need to use in an prefix manner when aiming at partial application. The notation "(+)" does indeed produce a prefix operator.

In Haskell, sections for infix operators correspond to a special form of partial application. A section applies an infix operator to one of its two operands by using parenthesization in the following way:

```
> let inc = (+1)
> inc 41
42
```

In this example, we have applied "+" to its second operand. For what it matters (and because addition is commutative), we could also define the increment function in terms of a section for the first operand:

```
> let inc = (1+)
> inc 41
42
```

Discussion

In languages with type-level functions such as parametrized type synonyms or data types, e.g., Language:Haskell, partial application makes sense at the type level as well.

Metadata

- Vocabulary:Functional programming
- http://en.wikipedia.org/wiki/Partial_application
- http://www.haskell.org/haskellwiki/Section_of_an_infix_operator

- Concept

Polymorphism

Headline

The ability of program fragments to operate on elements of several types

Illustration

Consider the type of list append inLanguage:Haskell:

```
(++) :: [a] -> [a] -> [a]
```

This type signature uses a type variable *a* to express that list append is polymorphic in the element type *a*. The operation can be applied for as long as the element type of both operand lists for an append are the same. We also speak of parametric polymorphism in this case.

Consider the type of addition inLanguage:Haskell:

```
(+) :: Num a => a -> a -> a
```

This type signature uses a type constraint on the operand type of addition to express that only "suitable" types (i.e., type-class instances of *Num*) can be used for addition. We also speak of type-class polymorphism or more generally ofbounded polymorphism in this case. Languages with subtyping may also use types in a subtyping hierarchy for bounds.

Metadata

- http://en.wikipedia.org/wiki/Polymorphism (computer science)
- http://en.wikipedia.org/wiki/Polymorphism in object-oriented programming
- Vocabulary:Programming language
- Concept

Uncurrying

Headline

Transformation of a function to take its multiple arguments at once

Illustration

In Language:Haskell, functions are typically in curried form. Consider, for example, the signature of addition:

```
> :t (+)
(+) :: Num a => a -> a -> a
```

Thus, addition takes one argument at a time. If we were to define addition in uncurried form, then this would look as follows:

```
add :: Num a => (a, a) -> a
add (x,y) = x + y
```

In fact, the uncurrying transformation can be represented by the followinghigher-order function:

```
uncurry :: (a -> b -> c) -> (a, b) -> c
uncurry f (a,b) = f a b
```

(We would need such an uncurry(ing) function for each number of arguments.) Thus, we can express the uncurrying transformation as follows:

```
add' :: Num a => (a, a) -> a
add' = uncurry (+)
```

Relationships

See currying for the dual transformation.

Citation

(http://en.wikipedia.org/wiki/Currying, 18 May 2013)

Uncurrying is the dual transformation to currying, and can be seen as a form of defunctionalization. It takes a function *f*(*x*) which returns another function *g*(*y*) as a result, and yields a new function *fâ€²*(*x*,*y*) which takes a number of additional parameters and applies them to the function returned by function *f*.

Metadata

- http://en.wikipedia.org/wiki/Currying
- Concept

Zip function

Headline

Map a tuple of sequences into a sequence of tuples

Illustration

In Language:Haskell, the basic *zip* function is of the following type:

```
> :t zip
zip :: [a] -> [b] -> [(a, b)]
```

Thus, the *zip* function takes two lists of possibly different element types and returns a list of pairs. Here is an illustrative application of *zip* to build pairs of characters with associated ASCII codes:

```
> zip ['A','B','C'] [65,66,67]
[('A',65),('B',66),('C',67)]
```

There exist further variation on zipping. For instance, there is also a*zipWith* function which applies an argument function to pairs rather than constructing pairs. For instance, consider two lists of operands for addition; pairwise addition can be applied as follows:

```
> :t zipWith
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
> zipWith (+) [1,2,3] [4,5,6]
[5,7,9]
```

For what it matters, the basic *zip* function is defined as follows:

```
zip :: [a] -> [b] -> [(a, b)]
zip = zipWith (,) -- apply pair construction
```

Metadata

- Common function
- Higher-order function
- https://en.wikipedia.org/wiki/Convolution_(computer_science)
- http://stackoverflow.com/questions/1115563/what-is-zip-functional-programming

Course:Lambdas in Koblenz

Headline

Introduction to functional programming at the University of Koblenz-Landau

Schedule of latest edition

- Lecture First steps
- Lecture Basic software engineering
- Lecture Searching and sorting
- Lecture Basic data modeling
- Lecture Higher-order functions
- Lecture Type-class polymorphism
- Lecture Functors and friends
- Lecture Functional data structures
- Lecture Unparsing and parsing
- Lecture Monads

Additional lectures

- Lecture Generic functions

Metadata

- http://softlang.wikidot.com/course:fp

Functional programming

Headline

The functional programming paradigm

Illustration

Consider the following definition of the factorial function inLanguage:Haskell:

```
-- A recursive definition of the factorial function
factorial n =
  if n==0
    then 1
    else n * factorial (n-1)
```

This definition describes the computation of factorial in terms of basic arithmetic operations ("functions") and the recursive application of the factorial function itself. There are no variables that are assigned different values over time. This situation is representative of the functional programming paradigm.

For comparison, consider the following definition of the factorial function inLanguage:Java:

```
  // An imperative definition of the factorial function
  public static int factorial(int n) {
    int result = 1;
    for (int i=n; i>1; i--)
      result = result * i;
    return result;
  }
```

A *result* variable is used in a loop to aggregate the product. Also, the loop uses a variable*i* to iterate from *n* down to *1*. Arguably, the recursive formulation is also straightforward in Java, but Java with its emphasis on variables and assignment as well as mutable data structures and encapsulation of state in objects does not encourage functional programming.

Citation

(http://en.wikipedia.org/wiki/Functional_programming, 14 April 2013)

In computer science, functional programming is a programming paradigm that treats computation as the evaluation of mathematical functions and avoids state and mutable data. It emphasizes the application of functions, in contrast to the imperative programming style, which emphasizes changes in state. ... Functional programming has its roots in lambda calculus, a formal system developed in the 1930s to investigate computability, the Entscheidungsproblem, function definition, function application, and recursion. Many functional programming languages can be viewed as elaborations on the lambda calculus.

Metadata

- http://en.wikipedia.org/wiki/Functional_programming
- Programming paradigm
- Vocabulary:Programming
- Lambda calculus

Language:Haskell

Headline

The functional programming language Haskell

Details

There are plenty of Haskell-based contributions to the 101project. This is evident from corresponding back-links. More selective sets of Haskell-based contributions are organized in themes: Theme:Haskell data, Theme:Haskell potpourri, and Theme:Haskell genericity.

Metadata

- http://www.haskell.org/
- http://en.wikipedia.org/wiki/Haskell_(programming_language)
- Functional programming language

Filter function

Headline

A higher-order function for filtering elements of a list

Illustration

Let's filter a given list of numbers to only retain the odd numbers. InLanguage:Haskell, we would use the filter function as follows:

```
Prelude> filter odd [1,2,3,4,5]
[1,3,5]
```

That is, *filter* is applied to a predicate (i.e., a function to return a Boolean) and a list;*filter* returns the list of those elements that satisfy the predicate. The higher-order function *filter* can be defined as follows:

```
-- Define filter via pattern matching
filter :: (a -> Bool) -> [a] -> [a]
filter p [] = []
filter p (x:xs) = if p x then x : ys else ys
  where
    ys = filter p xs
```

For what it matters, the *filter* function could also be defined in terms of thefold function, thereby reusing a recursion scheme as opposed to the explicitly recursive definition above.

```
-- Define filter via foldr
filter' p = foldr f []
  where
    f x = if p x then (:) x else id
```

Further, we can also define the *filter* function in terms of list comprehension; in this manner we hint at the meaning of list comprehensions because, in fact, list comprehensions correspond to syntactic sugar whose elimination also involves the Filter function.

```
-- Define filter via list comprehension
filter'' :: (a -> Bool) -> [a] -> [a]
filter'' p xs = [ x | x <- xs, p x]
```

Metadata

- Common function
- Higher-order function
- http://en.wikipedia.org/wiki/Filter_(higher-order_function)

Lambda abstraction

Headline

The language construct for anonymous functions according to the lambda calculus

Illustration

Suppose you want to increment all numbers in a list. Let us useLanguage:Haskell for illustration:

```
-- Sample input
sample = [1,3,5,7,11,13,17]
-- Expected output
expected = [2,4,6,8,12,14,18]
```

We shall use the map function to increment all elements of the input. We may apply the map function to a suitable increment function as follows:

```
-- Use a plain increment function
output = map inc sample
  where
    inc :: Int -> Int
    inc x = x + 1
```

Arguably, we may want to omit an explicit declaration of the increment function as its definition is trivial, not much insight can be gained from its definition, and no further reuse is intended (as we assume here). Thus, we may use an anonymous function instead of an extra increment function:

```
-- Use an anonymous increment function
output' = map (\x -> x + 1) sample
```

Thus, increment is expressed inline by an anonymous function that takes one argument*x* which is mapped to *x + 1*. Arguably, a Haskell programmer may actually also consider the following form where addition is refined into increment by means of partial application, possibly even using Haskell's specific section notation:

```
-- Use partial application of (+) for increment
output'' = map ((+) 1) sample

-- Use a section for increment
output''' = map (+1) sample
```

All these attempts compute the expected result.

Metadata

- http://www.haskell.org/haskellwiki/Lambda_abstraction
- http://en.wikipedia.org/wiki/Lambda_calculus
- Language construct
- Vocabulary:Functional programming

List processing

Headline

Processing lists using appropriate idioms

Metadata

- http://en.wikibooks.org/wiki/Haskell/List_processing
- Vocabulary:Functional programming
- Concept

Map function

Headline

A higher-order function to apply an argument function to all elements of a list

Illustration

Let's map over a given list of numbers to increment them. InLanguage:Haskell, we would use the map function as follows:

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

That is, *map* is applied to a function and a list; *map* returns a list of the same length as the input list which each element "transformed" by the argument function. The higher-order function *map* can be defined as follows:

```
-- Define map via pattern matching
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs
```

For what it matters, the *map* function could also be defined in terms of thefold function, thereby reusing a recursion scheme as opposed to the explicitly recursive definition above.

```
-- Define map via foldr
map' :: (a -> b) -> [a] -> [b]
map' f = foldr ((:) . f) []
```

Further, we can also define the *map* function in terms of list comprehension; in this manner we hint at the meaning of list comprehensions because, in fact, list comprehensions correspond to syntactic sugar whose elimination also involves the Map function.

```
-- Define map via list comprehension
map'' :: (a -> b) -> [a] -> [b]
map'' f xs = [ f x | x <- xs ]
```

Metadata

- Common function
- Higher-order function
- Vocabulary:Functional programming
- http://en.wikipedia.org/wiki/Map_(higher-order_function)