First-class Functions

2015/2016 2nd Semester

CSIS0259 / COMP3259
Principles of Programming Languages

Resources

Lecture covers:

- Chapter 4 of “Anatomy of Programming Languages”

http://www.cs.utexas.edu/~wcook/anatomy/anatomy.htm

Top-level Functions

- Our latest extension to the language was to allow top-level functions:

```
function absolute(x) {
  if (x > 0) then x else -x
}

absolute(-3)
```

Top-level functions are defined outside expressions.

Many languages familiar to you support top-level functions only (example C).

Top-level Functions

- Top level-functions allow us to create functions, but we cannot pass functions as arguments or return functions
- In other words, we cannot have function values!

First-Class Functions

- When a language supports functions as values, then we say that the language has first-class functions.
- Being first class, means that not only we can create functions, but also to pass function as arguments and return functions.
- The language with top-level functions does not have first class-functions.

IMPORTANT: Home Reading

- Especially for those students not taking the Functional Programming class, please read the following chapter:

http://learnyouahaskell.com/higher-order-functions

This chapter will improve your understanding of first-class functions (and Haskell)!

Languages with first-class functions

- Today essentially all major programming languages support first-class functions:
  - All functional languages support it of course:
    - Haskell, ML, OCaml, Scala, Scheme, Racket …
  - Most mainstream languages recently added support for it:
    - Java 8, C#, C++11
  - Scripting languages have had this feature for a while
    - Python, Ruby, Javascript

Why First-Class Functions?

Why First-Class Functions?

- First class functions are great for reuse.

- First class functions offer a compact notation to pass code around as arguments.

- See, for example, the following use-case from the Java 8 tutorial:

http://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html#use-case

Language with First-Class Functions

- In a language with first class functions we can have functional (or lambda) expressions:

```
(\x -> x + 1), (\x y -> max x y), map (\x -> ord x) l
```

Haskell

```
function(x) {return x+1;}, function(x,y) {return max(x,y);}
```

Javascript

Lambda Calculus

- The Lambda calculus provides an extremely elegant foundation for first-class functions

Alonzo Church

Only 3 kinds of expressions in the lambda calculus:

var	**variable**
exp exp	**application**
\var -> exp	**function**

Example:

(\x -> x) y

Lambda Calculus and Programming Languages

- The lambda calculus provides the foundations of modern programming languages

- Haskell is directly based on the lambda calculus

Lambda Expression

- A lambda expression allows us to declare an anonymous function: a function with no name. For example:

```
\x -> if (x > 0) then x else -x
```

function argument (called “x”)

body expression

Lambda Expression

- A lambda expression allows us to declare an anonymous function: a function with no name. For example:

```
\x -> if (x > 0) then x else -x
```

- Here is an equivalent named function:

```
absolute x = if (x > 0) then x else -x
```

Lambda Expression

- A lambda expression allow us to easily create first-class functions: functions that can be passed as arguments or returned. For example

```
map (\x -> x + 1) [1..10]
```

Lambda Expression

- A lambda expression allow us to easily create first-class functions: functions that can be passed as arguments or returned. For example

```
map (\x -> x + 1) [1..10]
```

- Here is an equivalent expression using a named function:

```
let add x = x + 1 in map add [1..10]
```

Lambda Expression

- A lambda expression allow us to easily create first-class functions: functions that can be passed as arguments or returned. For example

```
add x = \y -> x + y
```

returns a function

```
map (add 5) [1..10]
```

Declarations and Lambdas

- Functions in Haskell are implemented as lambda terms

- Consider the following Haskell definition:

$$f\ (x) = x * 2$$

- This is equivalent to:

$$f\ x = x * 2$$
$$f = \lambda x \to x * 2$$

Lambda Calculus in Haskell

- Rule of function arguments:

$$name\ var = body \qquad \equiv \qquad name = \lambda var \rightarrow body$$

We can transform a function with 1 argument into a function value!

Functions with 2 arguments or more

- Consider the following Haskell definition:

  ```
  max x y = if (x > y) then x else y
  ```

- The same transformation applies here

  ```
  max x = \y -> if (x>y) then x else y

  max = \x -> \y -> if (x>y) then x else y
  ```

How to interpret these definitions?

Functions with 2 arguments or more

- Consider the following Haskell definition:

```
max x y = if (x > y) then x else y
```

two argument function

- The same transformation applies here

```
max x = \y -> if (x>y) then x else y
```

one argument function, returning a function expression

```
max = \x -> \y -> if (x>y) then x else y
```

function expression

Types: Functions with 2 arguments or more

```
max :: Int -> Int -> Int
max x y = if (x > y) then x else y

max :: Int -> (Int -> Int)
max x = \y -> if (x>y) then x else y

max :: (Int -> (Int -> Int))
max = \x -> \y -> if (x>y) then x else y
```

Types: Functions with 2 arguments or more

Function types are right-associative. Therefore all definitions have the same type:

```
max :: Int -> Int -> Int
max x y = if (x > y) then x else y
```

```
max :: Int -> Int -> Int
max x = \y -> if (x>y) then x else y
```

```
max :: Int -> Int -> Int
max = \x -> \y -> if (x>y) then x else y
```

Question

- Which of the following types are equivalent:

Type 1	Type 2
(Int -> Int) -> Int -> Int	(Int -> Int) -> (Int -> Int)
Int -> (Int -> Int) -> Int	Int -> Int -> Int -> Int
(a -> (a -> a)) -> a -> a	(a -> a -> a) -> (a -> a)

Answer

- Which of the following types are equivalent:

Type 1	Type 2
(Int -> Int) -> Int -> Int	(Int -> Int) -> (Int -> Int)
Int -> (Int -> Int) -> Int	Int -> Int -> Int -> Int
(a -> (a -> a)) -> a -> a	(a -> a -> a) -> (a -> a)

The types in the first and last rows are equivalent

Sections

- In Haskell, there is syntactic sugar to make some lambda functions even shorter. Instead of:

```
map (\x -> x + 1) [1..10]
```

- We can write the code with a section:

```
map (+1) [1..10]
```

- Here, the compiler does the following expansion:

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

Function Equality

- Another useful equality to know is:

```
f x = g x   <==>   f = g
```

so, instead of writing:

```
add1 xs = map (\x -> x + 1) xs
```

we can write

```
add1 = map (\x -> x + 1)
```

Question

- What do the following expressions return? Can you explain their result?

```
zipWith (+) [1..10] [1..10] ==> ?

map (/2) [1..10] ==> ?

map (2/) [1..10] ==> ?
```

Answers

- Answers:

zipWith (+) [1..10] [1..10] ==> [2,4,6,…]

map (/2) [1..10] ==> [0.5,1.0,1.5, …]

map (2/) [1..10] ==> [2.0,1.0,0.666, …]

Curried vs Uncurried functions

- Consider the following two Haskell definitions

```
max x y = if (x > y) then x else y

max (x,y) = if (x > y) then x else y
```

Are these definitions equivalent?

Curried vs Uncurried functions

- A curried function is a function where arguments can be partially applied

```
max x y = if (x > y) then x else y
```

- An uncurried function is a function where arguments must be passed all at once

```
max (x,y) = if (x > y) then x else y
```

Functions as Data

- When functions are values, they can also be used to define data

- For example, we can create an alternative representation of environments based on functions:

```
type EnvF = String -> Maybe Value
```

- With EnvF we can implement operations such as

```
empty :: EnvF
lookup :: String -> EnvF -> Maybe Value
insert :: String -> Value -> EnvF -> EnvF
```

Functions as data

The key idea is that a value of type EnvF represents the lookup function of an environment. For example:

```
env :: EnvF
env = insert “x” 5 empty
```

To lookup a value in env all is needed is to apply it:

```
env “x”     ===> Just 5
env “y”     ===> Nothing
```

Implementing Functions as Data

Functions as data

- The lookup operation is very simple:

```
lookup :: String -> EnvF -> Maybe Value
lookup s f = f s
```

Functions as data

- The empty operation defines what happens when we lookup an empty environment:

```
empty :: EnvF
empty s = Nothing
```

In other words, looking up an empty environment always returns Nothing.

Functions as data

- The insert operation defines what happens when we lookup a variable s2 in an environment with one entry (s1 |-> v1) and a remaining environment e:

```
insert :: String -> Value -> EnvF -> EnvF
insert s1 v1 f =
  \s2 -> if (s1 == s2) then Just v1 else f s2
```

s2 is what to lookup

Revisiting Functions and Declarations

- The language with declarations allows us to write expressions such as

  var x = 5; x   (JavaScript)

  let x = 5 in x   (Haskell)

A declaration is quite similar to a function (except that it does not allow arguments).

How to add functions to a language with declarations?

Revisiting Functions and Declarations

How to add functions to a language with declarations?

1. One option is to add functions independently of variable declarations (we did this before for top-level functions).

2. Another option is to add first class functions and get top-level functions for free!

How?

Revisiting Functions and Declarations

Named functions with declarations and lambda expressions in JavaScript:

```
var absolute = function(x) {
   if (x > 0)
      return x;
   else
      return -x;
};
absolute(-3)
```

Revisiting Functions and Declarations

Named functions with declarations and lambda expressions in Haskell:

```
let absolute = \x -> if (x > 0) then x else -x
in absolute(-3)
```

Declarations & Lambdas

- It turns out that even declarations are not needed with lambda's: we can implement declarations in terms of lambdas and function application

How?

Declarations & Lambdas

- It turns out that even declarations are not needed with lambda's: we can implement declarations in terms of lambdas and function application

```
let x = exp in body
===>
(\x -> body) exp
```

Declarations & Lambdas

- For example

```
let x = 5 in x + 8
===>
(\x -> x + 8) 5
```

Both expressions return 13

Scoping Again

- Consider the (Haskell) program:

```
let x = 2 in
let f y = x + y in
let x = 3 in
f 5
```

What's the result?

Scoping Again

- Consider the (Haskell) program:

```
let x = 2 in
let f y = x + y in
let x = 3 in
f 5
```

In Haskell result is 7

Scoping Again

- What's going on?

[]
let x = 2 in
[x ↦ 2]
let f = \y -> x + y in
?
let x = 3 in
f 5

What happens here?

Scoping Again

- What's going on?

[]

let x = 2 in

[x ↦ 2]

let f = \y -> x + y in

[f ↦ (\y -> x + y,[x ↦ 2]), x ↦ 2]

let x = 3 in

[x ↦ 3, f ↦ (\y -> x + y, [x ↦ 2]), x ↦ 2]

f 5

We create a Closure!

Closure

- Closure: A closure is a combination of a function expression and an environment.
- Closures preserve the bindings that existed at the point when the function was defined.

Evaluation

- What's going on?

```
[]
let x = 2 in
[x ↦ 2]
let f = \y -> x + y in
[f ↦ (\y -> x + y, [x ↦ 2]), x ↦ 2]
let x = 3 in
[x ↦ 3, f ↦ (\y -> x + y, [x ↦ 2]), x ↦ 2]
f 5
```

How to evaluate?

Evaluation

- How to evaluate the following expression?

  [x ↦ 3, f ↦ (\y -> x + y, [x ↦ 2]), x ↦ 2]
  f 5

Environment
from the closure!

- Lookup f in the environment

- evaluate (\y -> x + y) 5 with [x ↦ 2]

- evaluate x + y with [y ↦ 5, x ↦ 2]

Evaluation of functions

- The rule to evaluate lambda expressions is:

```
(\var -> body) exp

===>

substitute var |-> (evaluate exp) in body
```

Implementing First-Class Functions

Implementation

Updating expressions:

$$\begin{aligned}&\textbf{data}\ Exp = ...\\&\quad \mid Function\ String\ Exp -- new\end{aligned}$$

- The function constructor represents a first-class function:

\x -> exp

Implementation

Updating values:

$$
\begin{aligned}
&\textbf{data}\ \mathit{Value} = \mathit{IntV}\ \mathit{Int} \\
&\quad\mid \mathit{BoolV}\ \mathit{Bool} \\
&\quad\mid \mathit{ClosureV}\ \mathit{String}\ \mathit{Exp}\ \mathit{Env} - - \mathit{new} \\
&\quad\textbf{deriving}\ (\mathit{Eq}, \mathit{Show})
\end{aligned}
$$

- The ClosureV constructor represents a closure:

(\x -> exp, env)

function

environment at function definition

Implementation

- Evaluating functions:

$$evaluate\ (Function\ x\ body)\ env = ClosureV\ x\ body\ env - -\ new$$

- Evaluating a function simply creates a closure with the function and the current environment

Implementation

- Evaluating function calls/application:

$$
\begin{aligned}
&evaluate\ (Call\ fun\ arg)\ env = evaluate\ body\ newEnv --\ changed \\
&\quad \textbf{where}\ ClosureV\ x\ body\ closeEnv = evaluate\ fun\ env \\
&\qquad newEnv = (x, evaluate\ arg\ env) : closeEnv
\end{aligned}
$$

- We first evaluate the fun expression, which should return a closure
- Then we create a suitable new environment (newEnv) from the environment in the closure
- Finally we evaluate the body of the function in the closure using newEnv

Design Choices

Scoping Strategies

- Historically there have been two different scoping strategies for first-class functions:
  - Static (or Lexical) Scoping: The free variables in a function definition are bound to the environment at the point of the definition.
  - Dynamic Scoping: The free variables in a function definition are bound at the function call point.

Scoping Strategies

- Consider the program:

```
let x = 2 in
let f = \y -> x + y in
let x = 3 in
f 5
```

- Free variables of `\y -> x + y`:

  {x}

Dynamic Scoping

- Example:

```
[]
let x = 2 in
[x ↦ 2]
let f = \y -> x + y in
[f ↦ \y -> x + y, x ↦ 2]
let x = 3 in
[x ↦ 3, f ↦ \y -> x + y, x ↦ 2]
f 5
```

Just store the lambda expression

Evaluation

- How to evaluate the following expression?

  [x ↦ 3, f ↦ \y -> x + y, x ↦ 2]
  f 5

- Lookup f in the environment

- evaluate (\y -> x + y) 5 with [x ↦ 3, f ↦ \y -> x + y, x ↦ 2]

- evaluate x + y with [y ↦ 5, x ↦ 3, f ↦ \y -> x + y, x ↦ 2]

Evaluation

- evaluate (\y -> x + y) 5 with [x ↦ 3, f ↦ \y -> x + y, x ↦ 2]
- evaluate x + y with [y ↦ 5, x ↦ 3, f ↦ \y -> x + y, x ↦ 2]
- result is 8
- free variables of \y -> x + y are bound to the environment at the call point

Dynamic Scoping

- Pros
  - Easy to implement
- Cons
  - Problematic for reasoning about programs. Binding the free variables at the call point makes it very difficult to reason about programs

Static Scoping

- Example:

```
[]
let x = 2 in
[x ↦ 2]
let f = \y -> x + y in
[f ↦ (\y -> x + y,[x ↦ 2]), x ↦ 2]
let x = 3 in
[x ↦ 3, f ↦ (\y -> x + y, [x ↦ 2]), x ↦ 2]
f 5
```

We create a
Closure!

Evaluation

- How to evaluate the following expression?

  [x ↦ 3, f ↦ (\y -> x + y, [x ↦ 2]), x ↦ 2]
  f 5

Environment
from the closure!

- Lookup f in the environment
- evaluate (\y -> x + y) 5 with [x ↦ 2]
- evaluate x + y with [y ↦ 5, x ↦ 2]

Evaluation

- result is 7
- free variables of `\y -> x + y` are bound to the environment at the point of the function definition

Static Scoping

- Pros
  - Easy to reason about programs.
- Cons
  - We need closures for the implementation

Scoping Strategies

- Generally speaking it has been accepted that static scoping is a superior strategy.
- Essentially all modern programming languages use static scoping.

Evaluation Strategies

- In languages with declarations, functions or first-class functions there are a few different design options when it comes to evaluation.

  - Call-by-value: In call-by-value expressions, such as parameters of functions or variable initialisers, are always evaluated before being added to the environment.

  - Call-by-name: In call-by-name an expression is not evaluated until it is needed.

Evaluation Strategies

- Consider the programs:

```
var x = longcomputation; 3

(\x -> 3) longcomputation
```

Call-by-value

- Consider the programs:

expression is evaluated!

```
var x = longcomputation; 3

(\x -> 3) longcomputation
```

- In call-by-value parameters or initializers are always evaluated.

- For some programs this can be wasteful

Call-by-name

- Consider the programs:

expression is
not evaluated!

```
var x = longcomputation; 3

(\x -> 3) longcomputation
```

- In call-by-name the expressions are added to the environment unevaluated.

- This can avoid some computations that are not needed

Call-by-name

- Consider the programs:

expression is
evaluated twice!

```
var x = longcomputation; x + x

(\x -> x + x) longcomputation
```

- In call-by-name the expressions are added to the environment unevaluated.

- However call-by-name, if implemented naively, can lead to redundant computation

Exceptions

- In a language with exceptions the result of a program may depend on the evaluation strategy. For example:

  ```
  var x = 3 / 0; 7
  ```

- Results in an exception in a call-by-value language;

- Results in 7 in a call-by-name language.