CS4450/7450
AoPL, Chapter 4: Functions
Principles of Programming Languages

Dr. William Harrison

University of Missouri

November 7, 2016

Announcements

- We're continuing with William Cook's online textbook, *Anatomy of Programming Languages*. It is available here. We're in Chapter 4.
- All programming languages have some notion of function—it's key to *procedural abstraction* that supports reusability.
- This chapter answers the question: what are functions?
  - "Top Level Functions": functions as they occur in non-functional languages like C
  - "First Class Functions": functions as they occur in functional languages like Haskell and JavaScript.

Outline for section 1

Top-level Functions

- Discussing extension to interpreter for *top-level functions*; full code is here.
- Languages like C only allow function declarations at the **top level**.
    - In C, a program is a list of function declarations followed by a main function.
- Here's an example in JavaScript:

```
function power(n, m) {
  if (m == 0)
    return 1;
  else
    return n * power(n, m - 1);
}

function main() {
  return power(3, 4);
}
```

Concrete & Abstract Syntax for Top-level Functions

- Concrete syntax for Function Declarations and Calls:

  `function` *func_name*(*param_name*, . . . , *param_name*) { *body_expr* }
  func_name(*expression*, . . . , *expression*)

Concrete & Abstract Syntax for Top-level Functions

- Concrete syntax for Function Declarations and Calls:

  `function` *func_name*(*param_name*, ..., *param_name*) { *body_expr* }
  func_name(*expression*, ..., *expression*)

- Function environments:

```
type FunEnv   = [(String, Function)]
data Function = Function [String] Exp
```

Concrete & Abstract Syntax for Top-level Functions

- Concrete syntax for Function Declarations and Calls:

  `function` *func_name*(*param_name*, ..., *param_name*) { *body_expr* }
  func_name(*expression*, ..., *expression*)

- Function environments:

```
type FunEnv   = [(String, Function)]
data Function = Function [String] Exp
```

- Programs (function env with main expression):

```
data Program = Program FunEnv Exp
```

Concrete & Abstract Syntax for Top-level Functions

- Concrete syntax for Function Declarations and Calls:

  `function` *func_name*(*param_name*, ..., *param_name*) { *body_expr* }
  func_name(*expression*, ..., *expression*)

- Function environments:

```
type FunEnv   = [(String, Function)]
data Function = Function [String] Exp
```

- Programs (function env with main expression):

```
data Program = Program FunEnv Exp
```

- Extension to expressions:

```
data Exp = ...
         | Call String [Exp]
```

Evaluating Top-level Functions

- Here are the changes to the interpreter:

```
execute :: Program -> Value
execute (Program funEnv main) = evaluate main [] funEnv

evaluate :: Exp -> Env -> FunEnv -> Value
                          ...
evaluate (Call fun args) env funEnv
                = evaluate body newEnv funEnv
  where Function xs body = fromJust (lookup fun funEnv)
        newEnv = zip xs [evaluate a env funEnv | a <- args]
```

Evaluating Top-level Functions

- Here are the changes to the interpreter:

```
execute :: Program -> Value
execute (Program funEnv main) = evaluate main [] funEnv

evaluate :: Exp -> Env -> FunEnv -> Value
                      ...
evaluate (Call fun args) env funEnv
                = evaluate body newEnv funEnv
  where Function xs body = fromJust (lookup fun funEnv)
        newEnv = zip xs [evaluate a env funEnv | a <- args]
```

- The steps in evaluating a function call:
  1. Look up the func def by name `lookup fun funEnv`, to get the func's param list `xs` and `body`
  2. Evaluate the actual arguments to get a list of values

     `[evaluate a env funEnv | a <- args]`
  3. Create `newEnv` by zipping the param names with actual arg values.
  4. Evaluate the function `body` in the new environment `newEnv`

Outline for section 2

First-class Functions

- A **first-class value** is a value that can be used like any other value. Specifically:
    1. A value is first class if it can be passed to functions,
    2. returned from functions, and
    3. represented in a variable binding.

First-class Functions

- A **first-class value** is a value that can be used like any other value. Specifically:
  1. A value is first class if it can be passed to functions,
  2. returned from functions, and
  3. represented in a variable binding.
- For example, functions are first-class values in Haskell

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

First-class Functions

- A **first-class value** is a value that can be used like any other value. Specifically:
  1. A value is first class if it can be passed to functions,
  2. returned from functions, and
  3. represented in a variable binding.
- For example, functions are first-class values in Haskell

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

- Functions are not first-class values in C:

```
int foo (int x) { ... }
```

  How would you write `map` in C?

Outline for section 3

λ Notation

- Go ahead and read section 4.3 of AoPL—it is entirely review.

Outline for section 4

Representing Environments as Functions

- Type of environments

```
type EnvF = String -> Maybe Value
-- Ex:
envF1 "x"    = Just (IntV 3)
envF1 "y"    = Just (IntV 4)
envF1 "size" = Just (IntV 10)
envF1 _      = Nothing
-- Empty environment
emptyEnv _ = Nothing
```

Representing Environments as Functions

- Type of environments

```
type EnvF = String -> Maybe Value
-- Ex:
envF1 "x"    = Just (IntV 3)
envF1 "y"    = Just (IntV 4)
envF1 "size" = Just (IntV 10)
envF1 _      = Nothing
-- Empty environment
emptyEnv _ = Nothing
```

- Environment lookup is just application;

```
x2 = envF "x"
```

Representing Environments as Functions

- Type of environments

```
type EnvF = String -> Maybe Value
-- Ex:
envF1 "x"    = Just (IntV 3)
envF1 "y"    = Just (IntV 4)
envF1 "size" = Just (IntV 10)
envF1 _      = Nothing
-- Empty environment
emptyEnv _ = Nothing
```

- Environment lookup is just application;

```
x2 = envF "x"
```

- Extending an environment:

```
-- we called this "tweek" in previous slides
bindF :: String -> Value -> EnvF -> EnvF
bindF var val env = \testVar -> if testVar == var
                                then Just val
                                else env testVar
```

Representing Environments as Functions

- Question:

```
(bindF "x" (IntV 9) (bindF "x" (IntV 0) emptyEnv)) "x" = ???
```

Representing Environments as Functions

- Question:

```
(bindF "x" (IntV 9) (bindF "x" (IntV 0) emptyEnv)) "x" = ???
```

- Rewriting an earlier interpreter with functional environments:

```
evaluateF :: Exp -> EnvF -> Value
evaluateF (Literal v) env      = v
evaluateF (Unary op a) env
      = unary op (evaluateF a env)
evaluateF (Binary op a b) env
      = binary op (evaluateF a env) (evaluateF b env)
evaluateF (Variable x) env     = fromJust (env x)
evaluateF (Declare x exp body) env = evaluateF body newEnv
  where newEnv = bindF x (evaluateF exp env) env
```

Representing Environments as Functions

- Question:

```
(bindF "x" (IntV 9) (bindF "x" (IntV 0) emptyEnv)) "x" = ???
```

- Rewriting an earlier interpreter with functional environments:

```
evaluateF :: Exp -> EnvF -> Value
evaluateF (Literal v) env      = v
evaluateF (Unary op a) env
      = unary op (evaluateF a env)
evaluateF (Binary op a b) env
      = binary op (evaluateF a env) (evaluateF b env)
evaluateF (Variable x) env     = fromJust (env x)
evaluateF (Declare x exp body) env = evaluateF body newEnv
  where newEnv = bindF x (evaluateF exp env) env
```

- Read Section 4.4.4 (Currying) and skip Section 4.4.5 (Church Numerals).

Outline for section 5

Contrast and Compare

- In a language w/o first-class functions:

```
int foo (int x, int y) { ... } -- top level only
    ...
ans = foo(2,3);                -- must be fully applied
```

Contrast and Compare

- In a language w/o first-class functions:

```
int foo (int x, int y) { ... } -- top level only
    ...
ans = foo(2,3);                -- must be fully applied
```

  - ∴ the abstract syntax for Top Level Functions:

```
data Exp = ...
         | Call String [Exp]    -- complete arg list
```

Contrast and Compare

- In a language w/o first-class functions:

```
int foo (int x, int y) { ... } -- top level only
    ...
ans = foo(2,3);                -- must be fully applied
```

  - ∴ the abstract syntax for Top Level Functions:

```
data Exp = ...
         | Call String [Exp]    -- complete arg list
```

- With first-class functions:

```
let
    foo = \ x y -> ...         -- local function
in
    foo 2                      -- partial application
```

Contrast and Compare

- In a language w/o first-class functions:

```
int foo (int x, int y) { ... } -- top level only
    ...
ans = foo(2,3);                -- must be fully applied
```

  - ∴ the abstract syntax for Top Level Functions:

```
data Exp = ...
         | Call String [Exp]    -- complete arg list
```

- With first-class functions:

```
let
    foo = \ x y -> ...         -- local function
in
    foo 2                      -- partial application
```

  - ∴ the abstract syntax for First Class Functions:

```
data Exp = ...
         | Function  String Exp -- 1st-class func
         | Call      Exp Exp    -- partial app
```

Evaluating First Class Functions

- Starting with 4.5.2 first, will discuss 4.5.1 second.
- Discussing extension to interpreter for *first-class functions*; full code is here.
- Contrast & Compare:

```
data Exp = Literal   Value
            ...
         | Declare   String Exp Exp
         | Call      String [Exp]

data Exp = Literal   Value
            ...
         | Declare   String Exp Exp
         | Function  String Exp        -- new
         | Call      Exp Exp           -- changed
```

Closures

- Defn. A **closure** combines a function expression with an environment. It is a value expressed by a function expression.

Closures

- Defn. A **closure** combines a function expression with an environment. It is a value expressed by a function expression.
- Here's a new value:

```
data Value = IntV  Int
           | BoolV Bool
           | ClosureV String Exp Env
```

Closures

- Defn. A **closure** combines a function expression with an environment. It is a value expressed by a function expression.
- Here's a new value:

```
data Value = IntV  Int
           | BoolV Bool
           | ClosureV String Exp Env
```

- New evaluation clause:

```
evaluate (Function x body) env = ClosureV x body env
```

Closures

- Defn. A **closure** combines a function expression with an environment. It is a value expressed by a function expression.
- Here's a new value:

```
data Value = IntV  Int
           | BoolV Bool
           | ClosureV String Exp Env
```

- New evaluation clause:

```
evaluate (Function x body) env = ClosureV x body env
```

- Calling closures:

```
evaluate (Call fun arg) env = evaluate body newEnv
  where ClosureV x body closeEnv = evaluate fun env
        newEnv = (x, evaluate arg env) : closeEnv
```

Outline for section 6

Section 4.6: Environment Closure Diagrams

Go ahead and skip this section altogether.

Outline for section 7

Parameter Passing Conventions

- A **parameter passing convention** describes how arguments to functions are evaluated.

Parameter Passing Conventions

- A **parameter passing convention** describes how arguments to functions are evaluated.
- Ex. in "`(\ x -> e) (2+3)`", is "`2+3`" evaluated before the call to `(\ x -> e)` or only when its value is needed?

Parameter Passing Conventions

- A **parameter passing convention** describes how arguments to functions are evaluated.
- Ex. in "`(\ x -> e) (2+3)`", is "`2+3`" evaluated before the call to `(\ x -> e)` or only when its value is needed?
- Call-by-Value (CBV) evaluates the argument expression first, producing a value, which is then added to the environment.

Parameter Passing Conventions

- A **parameter passing convention** describes how arguments to functions are evaluated.
- Ex. in "`(\ x -> e) (2+3)`", is "`2+3`" evaluated before the call to `(\ x -> e)` or only when its value is needed?
- Call-by-Value (CBV) evaluates the argument expression first, producing a value, which is then added to the environment. Call-by-Name (CBN) only evaluates the argument when the value is needed to complete the computation.

Eager (CBV) languages vs. Lazy (CBN) languages

Consider the following Standard ML program:

```
bomb : int -> int
fun bomb n = bomb n ;
fun one x  = 1;
```

Eager (CBV) languages vs. Lazy (CBN) languages

Consider the following Standard ML program:

```
bomb : int -> int
fun bomb n = bomb n ;
fun one x  = 1;
```

- What happens if we evaluate "`one (bomb 99)`"?

Eager (CBV) languages vs. Lazy (CBN) languages

Consider the following Standard ML program:

```
bomb : int -> int
fun bomb n = bomb n ;
fun one x  = 1;
```

- What happens if we evaluate "`one (bomb 99)`"?
- Eager language (e.g., ML, Scheme, C, ...): Non-termination. Why?
- Lazy language (e.g., Haskell): Produces `1`. Why?

Eager (CBV) Evaluation

To evaluate application `(rator rand)`

Eager (CBV) Evaluation

To evaluate application `(rator rand)`

1. Evaluate `rator`
   - Produces a function value f
   - I.e., a closure: `ClosureV x e env`

Eager (CBV) Evaluation

To evaluate application `(rator rand)`

1. Evaluate `rator`
   - Produces a function value f
   - I.e., a closure: `ClosureV x e env`
2. Evaluate `rand`
   - Produces a value v

Eager (CBV) Evaluation

To evaluate application `(rator rand)`

1. Evaluate `rator`
   - Produces a function value *f*
   - I.e., a closure: `ClosureV x e env`
2. Evaluate `rand`
   - Produces a value *v*
3. Apply *f* to *v*
   - evaluate `e` in extended environment $(\mathtt{x}, v) : \mathtt{env}$

Implementing Eager Evaluation

```
evaluate (Declare x exp body) env =
   case evaluate exp env of
      VException -> VException
      v          -> evaluate body ((x,v) : env)

evaluate (Call fun arg) env =
  case evaluate arg env of
     VException -> VException
     v -> case evaluate fun env of
            ClosureV name body denv -> evaluate body ((name,v) : denv)
```

Implementing Eager Evaluation

```
evaluate (Declare x exp body) env =
   case evaluate exp env of
      VException -> VException
      v          -> evaluate body ((x,v) : env)

evaluate (Call fun arg) env =
  case evaluate arg env of
     VException -> VException
     v -> case evaluate fun env of
            ClosureV name body denv -> evaluate body ((name,v) : denv)
```

- The lines `VException -> VException` need not be included. I put them here to dovetail with the text.
- The "`case evaluate arg env of`" and "`case evaluate fun env of`" force the evaluation of `fun` and `arg`

Lazy (CBN) Evaluation

To evaluate application `(rator rand)` in environment `rho`

Lazy (CBN) Evaluation

To evaluate application `(rator rand)` in environment `rho`

1. Evaluate `rator`
   - Produces a function value f
   - I.e., a closure: `ClosureV x e env`

Lazy (CBN) Evaluation

To evaluate application `(rator rand)` in environment `rho`

1. Evaluate `rator`
   - Produces a function value f
   - I.e., a closure: `ClosureV x e env`
2. Apply f to `evaluate rator rho`
   - evaluate `e` in extended environment (`x`, `evaluate rator rho`) : `env`

Outline for section 8

Summary

```
data Exp = Literal   Value
         | Unary     UnaryOp Exp
         | Binary    BinaryOp Exp Exp
         | If        Exp Exp Exp
         | Variable  String
         | Declare   String Exp Exp
         | Function  String Exp
         | Call      Exp Exp
  deriving (Eq, Show)

type Env = [(String, Value)]

evaluate :: Exp -> Env -> Value
     ...         ...            ...

evaluate (Function x body) env = ClosureV x body env
evaluate (Declare x exp body) env = evaluate body newEnv -- CBN
  where newEnv = (x, evaluate exp env) : env
evaluate (Call fun arg) env = evaluate body newEnv       -- CBN
  where ClosureV x body closeEnv = evaluate fun env
        newEnv = (x, evaluate arg env) : closeEnv
```