First-class Functions (Continuation)

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

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
?
let x = 3 in
?
f 5
```

Scoping Again

- What's going on?

We create a Closure!

[]
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

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

- Lookup f in the environment

Environment from the closure!

- 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:

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

- 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
?
let x = 3 in
?
f 5
```

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

  $[x \mapsto 2]$

  let f = \y -> x + y in

  $[f \mapsto (\backslash y \to x + y, [x \mapsto 2]),\ x \mapsto 2]$

  let x = 3 in

  $[x \mapsto 3,\ f \mapsto (\backslash y \to x + y, [x \mapsto 2]),\ x \mapsto 2]$

  f 5

We create a Closure!

Evaluation

- How to evaluate the following expression?

  $[x \mapsto 3, f \mapsto (\backslash y \to x + y, [x \mapsto 2]), x \mapsto 2]$
  f 5

- Lookup f in the environment

Environment from the closure!

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

- evaluate x + y with $[y \mapsto 5, x \mapsto 2]$

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.

Introduction to Recursion

Resources

Lecture covers:

- Chapter 5 of “Anatomy of Programming Languages”

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

Top-Level Functions

- The language with top-level functions allowed us to define recursive functions, such as:

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

- The function environment is built before any evaluation takes place.
- Thus when evaluating an expression all top-level functions are available in the environment.

First-class Functions and Recursion

- With the current implementation first-class functions we can model top-level functions

- However we lose the ability to define recursive functions.

- Consider, for example:

$$\textbf{let}\ fact = \lambda n \to \textbf{if}\ n \equiv 0\ \textbf{then}\ 1\ \textbf{else}\ n * fact\ (n-1)$$
$$\textbf{in}\ fact\ (10)$$

What happens here?

First-class Functions and Recursion

- Recall the definition of evaluation for first class functions for declarations:

```
evaluate (Declare x exp body) env =
  evaluate body newEnv
    where newEnv = (x, evaluate exp env) : env
```

- The declared variable x is on scope in body, but not on exp!

Recalling Scope

Scope

- The scope of a variable is the portion of text of a program in which a variable is defined. For example:

Scope of y

```
let y = 7 in
  let x = 3 in
     5 + (let x = 2 in x + y) * x
```

Scope

- The scope of a variable is the portion of text of a program in which a variable is defined. For example:

```
let y = 7 in
  let x = 3 in
    5 + (let x = 2 in x + y) * x
```

Scope of y

Scope when y is not a recursive declaration!

Scoping

- The story about scoping was a little simplified as it does not account for the possibility of recursive definitions. That is, we assumed that the following was not allowed:

```
var x = e; body     // where e uses the x being defined

let x = e in body  // where e uses the x being defined
```

We will now talk about definitions that allow this.

Recursive Declarations

- To allow recursive functions we need to allow declarations to be recursive.
- However this can cause problems. Consider the following example:

$$\mathbf{let}\ x = x + 1\ \mathbf{in}\ x$$

What should be the result of this?

Recursive Declarations

- To allow recursive functions we need to allow declarations to be recursive.

- However this can cause problems. Consider the following example:

$$\mathbf{let}\ x = x + 1\ \mathbf{in}\ x$$

One way to
interpret this definition is as an
infinite loop!

Recursive Declarations

- evaluate let x = x + 1 in x with []
- evaluate x + 1 with [x -> x + 1]
- evaluate (x + 1) + 1 with [x -> x + 1]
- evaluate ((x + 1) + 1) + 1 with [x -> x + 1]
- …

Recursive Definitions in Haskell

- Consider the following Haskell definition:

```
ones = 1 : ones
```

What should be the result of this?

Recursive Definitions in Haskell

- Consider the following Haskell definition:

```
> ones = 1 : ones
> [1,1,1,1,1,…]
```

An infinite list of ones!

Recursive Definitions in Haskell

- evaluate ones = 1 : ones with []
- evaluate 1 : ones with [ones -> 1 : ones]
- evaluate 1 : (1 : ones) with [ones -> 1 : ones]
- evaluate 1 : (1 : (1 : ones)) with [ones -> 1 : ones]
- ...

Recursive Definitions in Haskell

- Consider the following Haskell definition:

```
> numbers = 1 : map (\n -> n + 1) numbers
```

What should be the result of this?

Recursive Definitions in Haskell

- Consider the following Haskell definition:

```
> numbers = 1 : map (\n -> n + 1) numbers
```

The list of all natural numbers!

- What is the difference between:

```
> loop = 1 + loop

> numbers = 1 : map (\n -> n + 1) numbers
```

Are the two definitions useless?

Recursive Definitions in Haskell

- What is the difference between:

```
> loop = 1 + loop

> numbers = 1 : map (\n -> n + 1) numbers
```

Are the two definitions useless?

No! Because of Haskell is lazy (uses call-by-name) the second definition makes sense!
We can lazily compute the natural numbers.

Question

- Define a function that gives you the first 100 natural numbers using numbers and take:

```
take :: Int -> [a] -> [a]
```

Answer

- Define a function that gives you the first 100 natural numbers using numbers and take:

```
nat100 = take 100 numbers
```

When is a recursive definition good?

- It is not always easy to determine if a value will loop infinitly or not.
- Rule of thumb: if the recursive variable is used within a data constructor (e. g. :) or inside a function then it will probably not loop.

Using Results of Functions as Arguments

- An interesting use of recursion and laziness is the ability to use the result of a function as an argument to the function itself.