CS 314 Principles of Programming Languages

Lecture 21

Zheng Zhang

Department of Computer Science
Rutgers University

Wednesday 23rd November, 2016
Project 2 and Homework 7 released.

HW 7 due 11/30 Wednesday, 11:59pm EST. Project 2 due 12/4 Sunday, 11:59pm EST.

Midterm grade calculation policy change. Originally midterm is 25% and final comprises 35% of the overall grade. Now we will take the better score of your midterm and final exam in percentage and use it to calculate 60% of the overall grade as long as you take the final exam. Midterm grades will be released earlier next week.
Recursion in lambda calculus

Does this make sense?

\[ f \equiv \ldots f \ldots \]

In lambda calculus, such an equation does not define a term. How to find a \( \lambda \)-term that does “satisfy” the recursive definition?
Example

Example:

\[\text{add} \equiv \lambda mn.\]

\[(\text{cond } m (\text{add} (\text{succ } m) (\text{pred } n)) (\text{isZero? } n))\]

Just to make things easier to read, we will write instead:

\[\text{add} \equiv \lambda mn.\]

\[\text{if } (\text{isZero? } n) \text{ then } m \text{ else } (\text{add} (\text{succ } m) (\text{pred } n))\]

This is not a valid definition of a \(\lambda\)-term. What about this one?

\[\text{add} \equiv \lambda f. (\lambda mn.\]

\[\text{if } (\text{isZero? } n) \text{ then } m \text{ else } (f (\text{succ } m) (\text{pred } n)))\]

Claim: The fixed point of the above function is what we are looking for.
Function fixed points

The fixed points of a function $g$ is the set of values

$$fix_g = \{ x | x = g(x) \}.$$

Examples:

<table>
<thead>
<tr>
<th>function $g$</th>
<th>$fix_g$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\lambda x.6$</td>
<td>${ 6 }$</td>
</tr>
<tr>
<td>$\lambda x.(6 - x)$</td>
<td>${ 3 }$</td>
</tr>
<tr>
<td>$\lambda x.((x*x) + (x-4))$</td>
<td>${-2, 2}$</td>
</tr>
<tr>
<td>$\lambda x.x$</td>
<td>entire domain of $f$</td>
</tr>
<tr>
<td>$\lambda x.(x+1)$</td>
<td>${ }$</td>
</tr>
</tbody>
</table>

Zheng Zhang 5 CS@Rutgers University
Function fixed points

Is there a $\lambda$–term $Y$ that “computes” a fixed point of a function $F = \lambda f.(\ldots f\ldots)$, i.e., $YF = F(YF)$?

YES. $Y$ is called the fixed point combinator.

$$Y \equiv \lambda f.((\lambda x.f(x x)) (\lambda x.f(x x)))$$

$$YF = ((\lambda f.((\lambda x.f(x x)) (\lambda x.f(x x)))) F)$$
$$= (\lambda x.F(x x)) (\lambda x.F(x x))$$
$$= F( (\lambda x.F(x x)) (\lambda x.F(x x)))$$
$$= F(YF)$$
The Y–combinator

Example:

\[ F \equiv \lambda f. (\lambda m n. \text{if (isZero? } n) \text{ then } m \text{ else } (f \text{ (succ } m) \text{ (pred } n))) \]

\[
((YF) \ 3 \ 2) =
\]

\[
(((\lambda f. ((\lambda x. f(x \ x)) (\lambda x. f(x \ x)))) F) \ 3 \ 2) =
\]

\[
((F((\lambda x. F(x \ x)) (\lambda x. F(x \ x)))) 3 \ 2) =
\]

\[
((\lambda m n. \text{if (isZero? } n) \text{ then } m \text{ else } ((\lambda x. F(x \ x)) (\lambda x. F(x \ x))) \text{ (succ } m) \text{ (pred } n))) 3 \ 2) =
\]

if (isZero? 2) then 3 else

\[
(((\lambda x. F(x \ x)) (\lambda x. F(x \ x))) \text{ (succ } 3) \text{ (pred } 2)) =
\]

\[
((\lambda x. F(x \ x)) (\lambda x. F(x \ x))) 4 \ 1) =
\]

if (isZero? 1) then 4 else

\[
(((\lambda x. F(x \ x)) (\lambda x. F(x \ x))) \text{ (succ } 4) \text{ (pred } 1)) =
\]

\[
((\lambda x. F(x \ x)) (\lambda x. F(x \ x))) 5 \ 0) =
\]

if (isZero? 0) then 5 else

\[
(((\lambda x. F(x \ x)) (\lambda x. F(x \ x))) \text{ (succ } 5) \text{ (pred } 0)) = 5
\]
The Y–combinator example (cont.)

Note:

- Informally, the Y–combinator allows us to get as many copies of the recursive procedure body as we need. The computation “unrolls” recursive procedure calls one at a time.
- This notion of recursion is purely syntactic.
We can express all computable functions in our $\lambda$-calculus. However, nobody “programs” in lambda calculus. For that we have more “convenient” functional languages.

All computable functions can be express by the following two combinators, referred to as $S$ and $K$:

- $K \equiv \lambda xy.x$
- $S \equiv \lambda xyz.xz(yz)$

Combinatory logic is as powerful as Turing Machines.
Free and bound variables

Function \((\text{lambda} \ (x) \ e)\) “binds” variable \(x\) in “body” \(e\). You can think of this as a declaration of variable \(x\) with scope \(e\).

\[
( (\text{lambda} \ (y) \ (y \ z)) \ y )
\]

- Binding occurrence
- Bound occurrence
Lexical (static) scoping

The occurrence of a variable matches the lexically closest binding occurrence. An occurrence of a variable without a matching binding occurrence is called free.

A variable can occur free and bound in an expression.

⇒ Conceptually, we only substitute free occurrences of the formal arguments in the function body when computing a function application!
Substituting bound occurrences of a formal argument is called capturing.
Environments and Closures

Environments:

> Defer substitution by recording the bindings for the variables we would substitute in a data structure called an environment. If we need the value that a variable denotes, we just look it up in the environment.

An environment is a finite map from variables to values

\[ \rho \in Env = Variables \rightarrow Values \]

Closures

This is what you get if you define a function value in our mzscheme interpreter:

> (lambda (x) (+ x a))
  #<procedure>
> (define test (lambda (x) (+ x a)))
> 
> (test 1)
  reference to undefined identifier: a
Pair the environment with a function (lambda abstraction). The environment must contain values for all free variables of the function. The function can only be evaluated in its attached environment, making capturing (i.e., replacing the “wrong” parameter) impossible. Such a pairing is called a **closure**.

A closure is a pair consisting of an environment and a lambda abstraction

\[ cl \in \text{Closure} = \{ \langle \lambda, \rho \rangle \mid \text{FreeVar}(\lambda) \subseteq \text{DOM}(\rho) \} \]

**Closures can be used to implement lexical scoping.** They represent lexically scoped **function values**.
How To Apply a Closure?

How to apply a closure value to actual argument values?

1. Let $c_v$ be the closure value $\langle (\lambda(x) \ e), \rho \rangle$.

2. Apply $c_v$ to a value $a_v$ as follows:

   Evaluate the body $e$ of the function in the environment $\rho$ of the closure **extended** by the mapping of the formal parameter $x$ to the actual value $a_v$ ($\rho[x \rightarrow a_v]$).

$$
((\lambda(x)
\quad ((\lambda(z) ((\lambda(x)(z \ x)) \ 3)) \ (\lambda(y)(+ \ x \ y)))) \ 1)
$$

---

<table>
<thead>
<tr>
<th>closure interpreter</th>
</tr>
</thead>
<tbody>
<tr>
<td>${}$</td>
</tr>
<tr>
<td>$((\lambda(z)$</td>
</tr>
<tr>
<td>$\quad ((\lambda(x)(z \ x)) \ 3))$</td>
</tr>
<tr>
<td>$\quad (\lambda(y)(+ \ x \ y))$</td>
</tr>
<tr>
<td>$(((\lambda(x)$</td>
</tr>
<tr>
<td>$\quad (z \ x)) \ 3)$</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>$((\lambda(x)$</td>
</tr>
<tr>
<td>$\quad (z \ x))$</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>$((\lambda(x)$</td>
</tr>
<tr>
<td>$\quad (+ \ x \ y))$</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

$4$
Next Lecture

Things to do:

▶ Homework problem set 7 and project 2 posted.

Next time:

▶ Parallel and concurrent programming