CS 314 Principles of Programming Languages

Lecture 20

Zheng Zhang

Department of Computer Science
Rutgers University

Friday 18th November, 2016
Class Information

- Midterm grades will be released next week.
- Class schedule change next week. That is, before Thanksgiving break, Tuesday is Thursday, Wednesday is Friday. Your recitation schedule might be changed.
- Final exam: December 21, 4:00pm to 7:00pm. Talk to me if you have any conflict.
- HW 7 and project 2 will be released this weekend.
Programming in lambda calculus

The lambda calculus has very few constructs and it is therefore easy to reason about it.

Remember: Computation in the lambda calculus is a sequence of applications of reduction rules (mostly $\beta$-reductions).

Logical constants and operations (incomplete list):

- $\textbf{true} \equiv \lambda a.\lambda b. a$
- $\textbf{false} \equiv \lambda a.\lambda b. b$

- $\textbf{cond} \equiv \lambda m.\lambda n.\lambda p. (p \ m \ n)$
- $\textbf{not} \equiv \lambda x.(x \ false \ true)$
- $\textbf{and} \equiv \textbf{homework}$
- $\textbf{or} \equiv \lambda x.\lambda y. (x \ true \ y)$
What about data structures?

\textit{data structures:}

pairs can be represented as

\[ [M \cdot N] \equiv \lambda z.(z M N) \]

\begin{align*}
\text{build} & \equiv \lambda x.\lambda y.\lambda z.(z x y) \\
\text{first} & \equiv \lambda x.(x \text{ true}) \\
\text{second} & \equiv \lambda x.(x \text{ false})
\end{align*}
Programming in lambda calculus

What about arithmetic constants and operations?

Church Numerals:

\begin{align*}
0 & \equiv \lambda fx.x \\
1 & \equiv \lambda fx.(f \ x) \\
2 & \equiv \lambda fx.(f \ (f \ x)) \\
& \ldots \\
n & \equiv \lambda fx.(f\underbrace{(f(\ldots(fx)\ldots))}_{n \ times}) & \equiv \lambda fx.(f^n x)
\end{align*}
Programming in lambda calculus

The natural number $n$ is represented as a function that applies a function $f$ $n$–times recursively to its argument $x$.

\begin{align*}
\textbf{succ} & \equiv \lambda m. (\lambda fx. (f (m f x))) \\
\textbf{add} & \equiv \lambda mn. (\lambda fx. ((m f) (n f x))) \\
\textbf{mult} & \equiv \lambda mn. (\lambda fx. (m (n f) x)) \\
\textbf{isZero?} & \equiv \lambda m. (m (\lambda x. \text{false}) \text{ true})
\end{align*}
Examples:

\[(\text{mult } 2 \ 3) =
((\lambda mn. (\lambda fx. ((m (n f)) x))) \ 2 \ 3) =
\]

\[\lambda f_0 x_0. ((2 \ \begin{array}{c} \boxed{3 \ f_0} \end{array}) x_0) =
\]

\[\lambda f_0 x_0. ((2 \ ((\lambda fx. (f (f (f x)))) f_0)) x_0) =
\]

\[\lambda f_0 x_0. ((2 \ (\lambda x. (f_0 (f_0 (f_0 x))))) x_0) =
\]

\[\lambda f_0 x_0. ((2 \ (\lambda x_1. (f_0^3 x_1))) x_0) =
\]

\[\lambda f_0 x_0. ((\lambda x. ((\lambda x_1. (f_0^3 x_1))) (\lambda x_1. (f_0^3 x_1))) x_0) =
\]

\[\lambda f_0 x_0. ((\lambda x. ((\lambda x_1. (f_0^3 x_1))) (f_0^3 x)) x_0) =
\]

\[\lambda f_0 x_0. ((\lambda x. (f_0^3 (f_0^3 x))) x_0) =
\]

\[\lambda f_0 x_0. (f_0^3 (f_0^3 x_0)) =
\]

\[\lambda fx. (f^6 x) = 6
\]
Programming in lambda calculus

Examples:

\((\text{isZero? } 0) =\)
\(\left((\lambda f x.x) \ (\lambda y.\text{false})\right) \text{ true} =\)
\((\lambda x.\text{true}) \text{ true} = \text{true}\)

\((\text{isZero? } n) \quad \text{where } n > 0 ?\)
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:
\[
\text{add} \equiv \lambda mn. \\
\quad (\text{cond } m (\text{add} (\text{succ } m) (\text{pred } n)) \ (\text{isZero? } n))
\]

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

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

This is not a valid definition of a \( \lambda \)-term. What about this one?
\[
\text{add} \equiv \lambda f. (\lambda mn. \\
\quad \text{if } (\text{isZero? } n) \text{ then } m \text{ else } (f (\text{succ } m) (\text{pred } n)))
\]

**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>$\lambda x.((x \times x) + (x-4))$</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>
Function fixed points

Is there a λ–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 ≡ \lambda f. (\lambda mn.
  if (isZero? n) then m else (f (succ m) (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 mn. if (isZero? n) then m else
  (((\lambda x.F(x x)) (\lambda x.F(x x))) (succ m) (pred n))) 3 2) =
if (isZero? 2) then 3 else
  (((\lambda x.F(x x)) (\lambda x.F(x x))) (succ 3) (pred 2)) =
(F((\lambda x.F(x x)) (\lambda x.F(x x)))) 4 1) =
((\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))) (succ 4) (pred 1)) =
(F( (\lambda x.F(x x)) (\lambda x.F(x x)))) 5 0) =
((\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))) (succ 5) (pred 0)) = 5
Next Lecture

Things to do:

▶ Homework problem set 7 will be posted soon.
▶ Project 2 (Scheme) will be posted this weekend; start programming in Scheme!

Next time:

▶ lambda calculus
▶ defining interpreters