CSCI 740 - Programming Language Theory

Lecture 7

Programming in the λ-calculus

Instructor: Hossein Hojjat

September 15, 2017

Recap

- A λ-calculus expression is defined as

$$\begin{array}{rcll} e & ::= & x & \text{variable} \\ & | & \lambda x.e & \text{abstraction} \\ & | & e\ e & \text{application} \end{array}$$

α-conversion

$$\lambda x.e \ \rightarrow_\alpha \ \lambda y.e[x \mapsto y] \qquad \text{if } y \notin \mathsf{FV}(e)$$

β-reduction

$$(\lambda x.e)\ e' \ \rightarrow_\beta \ e[x \mapsto e']$$

η-conversion

$$\lambda x.(e\ x) \ \rightarrow_\eta \ e \qquad \text{if } \ x \notin \mathsf{FV}(e)$$

Review: Church Booleans

$$\begin{aligned}
\text{TRUE} &\triangleq \lambda x.\lambda y.x \\
\text{FALSE} &\triangleq \lambda x.\lambda y.y \\
\text{IF} &\triangleq \lambda b.\lambda e_t.\lambda e_f.b\ e_t\ e_f \\
\text{NOT} &\triangleq \lambda b.b\ \text{FALSE}\ \text{TRUE} \\
\text{AND} &\triangleq \lambda b_1.\lambda b_2.b_1\ b_2\ \text{FALSE} \\
\text{OR} &\triangleq \lambda b_1.\lambda b_2.b_1\ \text{TRUE}\ b_2
\end{aligned}$$

Encoding Pairs

The pair ADT:

- There is 1 constructor (taking 2 arguments) and 2 selectors
- 1st selector returns the 1st arg passed to the constructor
- 2nd selector returns the 2nd arg passed to the constructor

$$\text{FST} = \lambda p.p\ \text{TRUE}$$
$$\text{SND} = \lambda p.p\ \text{FALSE}$$
$$\text{PAIR} = \lambda xy.\lambda z.(z\ x\ y)$$

Exercise: Reduce FST(PAIR x y)

Review: Church Numerals

- Church numerals: encode a number n as a function that takes s and z and applies s to z, n times

$$
\begin{array}{ll}
c_0 & \lambda s.\lambda z.z \\
c_1 & \lambda s.\lambda z.s\ z \\
c_2 & \lambda s.\lambda z.s\ (s\ z) \\
c_3 & \lambda s.\lambda z.s\ (s\ (s\ z)) \\
\ldots &
\end{array}
$$

Review: Church Numerals

- Church numerals: encode a number n as a function that takes s and z and applies s to z, n times

$$
\begin{array}{lll}
c_0 & & \lambda s.\lambda z.z \\
c_1 & & \lambda s.\lambda z.s\ z \\
c_2 & & \lambda s.\lambda z.s\ (s\ z) \\
c_3 & & \lambda s.\lambda z.s\ (s\ (s\ z)) \\
\ldots & & \\
\mathsf{SUCC} & \triangleq & \lambda n.\lambda s.\lambda z.s\ (n\ s\ z)
\end{array}
$$

- successor: take a Church numeral and return a Church numeral that (when called) applies s one more time

Review: Church Numerals

- Church numerals: encode a number n as a function that takes s and z and applies s to z, n times

$$
\begin{array}{lll}
c_0 & & \lambda s.\lambda z.z \\
c_1 & & \lambda s.\lambda z.s\ z \\
c_2 & & \lambda s.\lambda z.s\ (s\ z) \\
c_3 & & \lambda s.\lambda z.s\ (s\ (s\ z)) \\
\ldots & & \\
\text{SUCC} & \triangleq & \lambda n.\lambda s.\lambda z.s\ (n\ s\ z) \\
\text{PLUS} & \triangleq & \lambda n_1.\lambda n_2.\lambda s.\lambda z.n_1\ s\ (n_2\ s\ z)
\end{array}
$$

- plus: take two "numbers" and return a "number" that uses one number as the zero argument for the other

Exercise

- Define multiplication in the λ-calculus

Exercise

- Define multiplication in the λ-calculus

Solution.

- Recall:

$$c_{n_1} \quad \triangleq \quad \lambda s.\lambda z.\underbrace{s\ (\ldots s(s\ z)\ldots)}_{n_1 \text{ applications}}$$

- To multiply by c_{n_2} we need to replace every s with n_2 copies of s
- That's exactly what the representation of m already does!

$$\text{TIMES} \quad \triangleq \quad \lambda n_1.\lambda n_2.\lambda s.\lambda z.n_1\ (n_2\ s)\ z$$

Exercise

- Define multiplication in the λ-calculus

Solution.

- Recall:

$$c_{n_1} \quad \triangleq \quad \lambda s.\lambda z.\underbrace{s\ (\ldots s(s\ z)\ldots)}_{n_1 \text{ applications}}$$

- To multiply by c_{n_2} we need to replace every s with n_2 copies of s
- That's exactly what the representation of m already does!

$$\text{TIMES} \quad \triangleq \quad \lambda n_1.\lambda n_2.\lambda s.\lambda z.n_1\ (n_2\ s)\ z$$

- Can you simplify TIMES with η-conversion?

Exercise

- Define a λ-calculus term ISZERO such that: ISZERO n returns TRUE if n is ZERO, and returns FALSE otherwise

Exercise

- Define a λ-calculus term ISZERO such that: ISZERO n returns TRUE if n is ZERO, and returns FALSE otherwise

Solution.

- $$\text{ISZERO} = \lambda n.n\ (\lambda x.\text{FALSE})\ \text{TRUE}$$
- λx.FALSE is a function that throws away its argument and always returns FALSE

Exercise

- Define a λ-calculus term ISZERO such that: ISZERO n returns TRUE if n is ZERO, and returns FALSE otherwise

Solution.

- $$\text{ISZERO} = \lambda n.n\ (\lambda x.\text{FALSE})\ \text{TRUE}$$
- λx.FALSE is a function that throws away its argument and always returns FALSE

- Reduce the following expressions:
  - ISZERO c_0
  - ISZERO c_1

Predecessor

- Increasing a number is easy:
  just wrap a few more function applications!
- How can we compute c_{n-1} from c_n?
- Build a series of pairs of the form (c_{n-1}, c_n) and take c_{n-1} from the pair at the end
- Start by defining PZero $= (c_0, c_0) =$ PAIR $c_0\ c_0$
- Define PSucc $= \lambda p.$PAIR (SND p) (SUCC (SND p))
  - Ignores the first item of the pair
  - Moves the second to the first position
  - Puts the successor of the second element into the second position
- In general: c_n PSucc PZero $= (c_{n-1}, c_n)$ for $n > 0$

$$\text{PRED} \quad \triangleq \quad \lambda n.\text{FST}(n\ \text{PSucc PZero})$$

Recursive Functions

- How can we write recursive functions like factorial?
- In Scala, we can write the factorial function recursively as:

```
def fact(n:Int):Int = if (n==0) 1 else n*fact(n-1)
```

- Can we define something like this?

$$\text{FACT} \triangleq \lambda n.\text{IF } (\text{ISZERO } n)\ c_1\ (\text{TIMES } n\ (\text{FACT } (\text{PRED } n)))$$

- In slightly more readable notation:

$$\text{FACT} \triangleq \lambda n.\textbf{if } (n = 0) \textbf{ then } 1 \textbf{ else } n\ \times \text{FACT } (n-1)$$

- Doesn't work in the λ-calculus: the functions are all anonymous

Recursive Functions

$$\text{FACT} \quad = \quad \lambda n.\textbf{if } (n = 0) \textbf{ then } 1 \textbf{ else } n \ \times \text{FACT } (n-1)$$

- It is possible to express recursion as a function

$$\begin{array}{lll} \text{FACT} & = & (\lambda n. \cdots \text{FACT} \cdots) \\ & \leftarrow_\beta & (\lambda f.(\lambda n. \cdots f \cdots)) \ \text{FACT} \\ & = & H \ \ \text{FACT} \end{array}$$

- Factorial function FACT is a fixpoint of the (non-recursive) function H
- Fixpoint of a function is an element that is mapped to itself by the function

$$H \quad = \quad \lambda f.\lambda n.\textbf{if } (n = 0) \textbf{ then } 1 \textbf{ else } n \ \times f(n-1)$$

Fix-Points

- Fixpoint of a function: $f(x) = x$
- Examples of fixpoints:

f	fixpoint
$\lambda x.5$	
$\lambda x.(8 - x)$	
$\lambda x.(x^2 - 2x + 2)$	
$\lambda x.x$	
$\lambda x.x + 1$	

Fix-Points

- Fixpoint of a function: $f(x) = x$
- Examples of fixpoints:

f	fixpoint
$\lambda x.5$	5
$\lambda x.(8 - x)$	
$\lambda x.(x^2 - 2x + 2)$	
$\lambda x.x$	
$\lambda x.x + 1$	

Fix-Points

- Fixpoint of a function: $f(x) = x$
- Examples of fixpoints:

f	fixpoint
$\lambda x.5$	5
$\lambda x.(8 - x)$	4
$\lambda x.(x^2 - 2x + 2)$	
$\lambda x.x$	
$\lambda x.x + 1$	

Fix-Points

- Fixpoint of a function: $f(x) = x$
- Examples of fixpoints:

f	fixpoint
$\lambda x.5$	5
$\lambda x.(8 - x)$	4
$\lambda x.(x^2 - 2x + 2)$	1,2
$\lambda x.x$	
$\lambda x.x + 1$	

Fix-Points

- Fixpoint of a function: $f(x) = x$
- Examples of fixpoints:

f	fixpoint
$\lambda x.5$	5
$\lambda x.(8 - x)$	4
$\lambda x.(x^2 - 2x + 2)$	1,2
$\lambda x.x$	every value
$\lambda x.x + 1$	

Fix-Points

- Fixpoint of a function: $f(x) = x$
- Examples of fixpoints:

f	fixpoint
$\lambda x.5$	5
$\lambda x.(8 - x)$	4
$\lambda x.(x^2 - 2x + 2)$	1,2
$\lambda x.x$	every value
$\lambda x.x + 1$	no value

Fixpoint Theorem

Theorem:

Every lambda expression e has a fixed point p such that $(e\ p) \leftrightarrow_\beta p$

Proof:

Let:

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

Now consider:

$$\begin{array}{rcll} p = Y\ e & \rightarrow_\beta & (\lambda x.e\ (x\ x))\ (\lambda x.e\ (x\ x)) \\ & \rightarrow_\beta & e\ \big((\lambda x.e\ (x\ x))(\lambda x.e\ (x\ x))\big) \\ & = & e\ p \end{array}$$

Fixpoint Theorem

Theorem:

Every lambda expression e has a fixed point p such that $(e\ p) \leftrightarrow_\beta p$

Proof:

Let:

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

Now consider:

$$\begin{array}{rcl} p = Y\ e & \rightarrow_\beta & (\lambda x.e\ (x\ x))\ (\lambda x.e\ (x\ x)) \\ & \rightarrow_\beta & e\ \big((\lambda x.e\ (x\ x))(\lambda x.e\ (x\ x))\big) \\ & = & e\ p \end{array}$$

- Function Y is called "Y combinator"
- It was discovered by Haskell Curry
- It can always be used to find a fixed point of an arbitrary lambda expression

$$\forall e : Y\ e \quad \leftrightarrow_\beta \quad e\ (Y\ e)$$

How does Y work?

- Recall the non-terminating expression

$$\Omega \;=\; (\lambda x.x\ x)(\lambda x.x\ x)$$

- Ω loops endlessly without doing any productive work
- Note that $(x\ x)$ represents the body of the "loop"
- Simply define Y to take an extra parameter f and put it into the loop

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

- So Y just inserts some productive work into the body of Ω
- Y: The function that takes a function f and returns $f(f(f(f(\cdots))))$

Using the Y Combinator

- Recall that the fixpoint theorem states that $Y\ e\ \leftrightarrow_\beta e\ (Y\ e)$
- For any constant function $\lambda x.k$ (where k is a constant):

$$Y\ (\lambda x.k)\ \rightarrow\ (\lambda x.k)\ (Y\ (\lambda x.k)) \rightarrow_\beta k$$

Exercise

- Consider $f = \lambda x.x + 1$ where f has no fixed point
- Then compute $Y\ f$