Introduction to the $\lambda$-Calculus

Part II

CS209 - Functional Programming

Dr. Greg Lavender
Department of Computer Science
Stanford University

iszero Predicate

- Test for zero
  - $\text{iszero} = \lambda n.n(\lambda x.F)T$
  - $\text{iszero}_0 = (\lambda n.n(\lambda x.F)T) \lambda sz.z$
    => $(\lambda sz.z)(\lambda x.F)T$
    => $T$
  - $\text{iszero}_S0 = (\lambda n.n(\lambda x.F)T) \lambda sz.s(z)$
    => $(\lambda sz.s(z))(\lambda x.F)T$
    => $(\lambda x.F)T$
    => $F$
  - convince yourself that $\text{iszero}_S n$ always returns $F$
Ordered Pairs

- Lambda expressions for ordered pairs
  - $\text{fst} = \lambda p.p \, T = \lambda p.p \, (\lambda t.f.t)$
  - $\text{snd} = \lambda p.p \, F = \lambda p.p \, (\lambda t.f.f)$
  - $(E_1, E_2) = \lambda f.f \, E_1 \, E_2$
  - $\text{fst} (E_1, E_2) = (\lambda p.p \, T) \, (E_1, E_2)$
    $\Rightarrow (E_1, E_2) \, T$
    $\Rightarrow (\lambda f.f \, E_1 \, E_2) \, T$
    $\Rightarrow T \, E_1 \, E_2$
    $\Rightarrow (\lambda t.f.t) \, E_1 \, E_2$
    $\Rightarrow E_1$
  - Similarly for $\text{snd} (E_1, E_2)$

n-Tuples

- Tuples are defined in terms of pairs
  - $(E_1, E_2, \ldots, E_n) = (E_1, (E_2, (\ldots (E_{n-1}, E_n) \ldots)))$
  - $E^1 = \text{fst} \, E$
  - $E^2 = \text{fst}(\text{snd} \, E)$
  - $E^i = \text{fst}(\text{snd}(\ldots (\text{snd} \, E) \ldots)))$ if $i < n$
  - $E^n = \text{snd}(\ldots (\text{snd} \, E) \ldots))$
    - $n$-1 applications of $\text{snd}$
  - $(E_1, E_2, \ldots, E_n)^2 = (E_1, (E_2, (\ldots)))^2$
    $\Rightarrow \text{fst} \, (\text{snd}(E_1, (E_2, (\ldots))))$
    $\Rightarrow \text{fst} \, (E_2, (\ldots))$
    $\Rightarrow E_2$
  - Prove that $(E_1, E_2, \ldots, E_n) \Rightarrow E_i$ for $1 \leq i \leq n$
Predecessor function

- Predecessor is subtraction by 1
  - BUT, pred 0 => 0
  - pred = \( \lambda n f x. \text{snd}(n \text{prefn} f) \cdot (T, x) \)
    - where prefn is defined as:
      - prefn = \( \lambda p. (F, (\text{fst} p \to \text{snd} p \cdot (f \cdot (\text{snd} p))) \)
    - where \((E \to E_1 \mid E_2)\) is syntactic sugar for \((\text{test } E \cdot E_1 \cdot E_2)\)
  - Show that:
    - pred(succ n) => n
    - pred 0 => 0

Fixed points

- A “fixed point” is a value \( x \) in the domain of a function that is the same in the range \( f(x) \).
- Every value in the domain of the identity function is a fixed point
  - \( \lambda x. x = x \)
- can you think of others?
  - factorial(1) = 1
  - fibonacci(0) = 0, fibonacci(1) = 1
  - square(0) = 0, square(1) = 1
  - abs(x) = x, if x >= 0
  - sin(0) = 0
- Functionals may also have fixed points
  - \( D_x(e^x) = e^x \)
Some Computability Theory

- **Gödel numbering**
  - every program is represented by a finite string of symbols:
    - \( P = \text{int main() \{ printf ("Hello world\n"); \} } \)
  - a general algorithm can be defined that converts any program into a unique natural number \( e \), called the *Gödel number* of the program
    - \( \text{gödel}(P) = e \)
  - for any Turing machine \( M \) representing a program, we can assign it a Gödel number \( e \) and denote the Turing machine that computes that program by \( M_e \)
  - Let \( U \) be a Turing machine that using inputs \( e \) and \( x \) computes \( M_e(x) \):
    - call \( e \) the program and \( x \) the input to program \( e \)
    - \( U = \text{if (e is a program) then } M_e(x) \text{ else output } 0 \)
    - \( U \) is thus a universal Turing machine
    - this is the key idea that led John von Neumann to "invent" the stored program concept used in modern computers

Gödel Numbering Function

- How do we write a Gödel numbering function?
  - a program is just a sequence of characters from some alphabet. For example, the ASCII alphabet has 128 characters:
    ```
    map chr [0..127] =>
    "\NUL\SOH\STX\ETX\ETB\CAN\EM\SUB\ESC\FS\GS\RS\US
    !"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJKLMNOPQRSTUVWXYZ[\]\]^_`abcdefghijklmnopqrstuvwxyz{|}~\DEL"
    ```
  - define a \( n \)-degree polynomial where each coefficient is the integer value of the corresponding ASCII character and "\( x \)" is the size of the alphabet:
    - \( P(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0 \)
    - for any given program text convert it to an array of ASCII values and use those values as the coefficients \( a_n \ldots a_0 \)
    - then evaluate the polynomial with \( x = 128 \) using Horner's rule
Gödel Numbering Function

- Here is an example in Haskell

```haskell
godel :: String -> Integer
godel p = horner 128 (ascii p)
```

```haskell
main() {
 printf("Hello, world\n"); }
```

> 9496774814982586627079243442307576668681324862865981775787352
> 7659021316462083581

Kleene's Fixed Point Theorem

- Also known as Kleene's recursion theorem
  - let $\phi_n = \lambda x. U(e,x)$
  - For every computable function $f$ there is a number $n$ such that $\phi_n = \phi(f(n))$
- Corollary
  - There is a Gödel number $n$ such that $\phi_n$ is the constant function with output $n$
  - Hence, $n$ is the Gödel number of a "self-reproducing" program.
    - i.e., a Turing machine whose program, denoted by Gödel number $n$, does nothing on any input except print its own code, i.e., the string $\text{ungodel}(n)$
  - This is the idea behind a "quine"
    - the name quine was first used by Hofstadter in his book Gödel, Escher, Bach in honor of the logician W.V.O. Quine
    - Omega $= (\lambda x.xx)(\lambda x.xx)$ is an example of a self-reproducing program in the lambda calculus
  - compare this to the following in Scheme and C:

```scheme
((lambda (x) `(,x ',x)) `(lambda (x) `(,x ',x)))
```

```c
main(a){
 printf("a="main(a){
 printf(a=%c%s%c,34,a,34);}",34,a,34);}
```

> 623284745206023142292120031582029650201082718881172293919483215307138944
> 7075616437663706123166240102409248112023870086939648708460245757
Recursive Definition in the $\lambda$-calculus

- **How do we define recursive expressions?**
  - will this work?
    - $\text{mult} \equiv \lambda m \ n. (\text{iszero} m \rightarrow 0 | \text{add} n \ (\text{mult} \ (\text{pred} m) \ n))$
  - No! There is a problem with this definition
    - we cannot define $\text{mult}$ in terms of itself without some way to handle the self-referential naming -- recursive self-reference introduces a challenge
    - most programming languages do this automagically, but in the pure $\lambda$-calculus, we have to come up with a syntactic way to allow recursive definitions
    - recall that in Scheme, "letrec" is a special syntactic form for writing recursive definitions (ML has "val rec")
    - we have to come up with some way in the lambda calculus to express recursion syntactically without defining a function directly in terms of itself
  - We need a special mathematical device called a "fixed point operator" that works for any function defined in the lambda calculus

---

Defining Recursive Functions

- **Fixed Point Theorem for $\lambda$-calculus**
  - For all $F$, there exists an $X$ such that $F(X) = X$
  - Proof:
    - let $W = \lambda x. F(xx)$ and let $X = WW$
    - then $X = WW = \lambda x. F(xx)W \Rightarrow F(WW) = F(X)$
    - $X = WW$ is called a fixed point of $F$

- **Fact**: we can generate a fixed point for any function $F$
  - Let $F = \lambda y. y$
  - $W = \lambda x. F(xx) = \lambda x. (\lambda y. y)(xx)$
  - Proof that $X = WW$ is a fixed point for $\lambda y. y$
    - $X = WW = (\lambda x. (\lambda y. y)(xx))(\lambda x. (\lambda y. y)(xx))$
    - $\Rightarrow (\lambda y. y)((\lambda x. (\lambda y. y)(xx))(\lambda x. (\lambda y. y)(xx)))$
    - $= F(X)$
    - note that $(\lambda y. y)((\lambda x. (\lambda y. y)(xx))(\lambda x. (\lambda y. y)(xx))) \Rightarrow (\lambda x. xx)(\lambda x. xx)$
    - so $(\lambda x. xx)(\lambda x. xx)$ is a fixed point for $\lambda y. y$, i.e., $\lambda x. x$

- **Homework**: Generate a fixed point for $F = \lambda x. y . y$
Fixed Point Operators

- Consider a fixed point operator \( \text{Fix} \)
  - \( \text{Fix} \) \( F = F (\text{Fix} \ F) \)
  - \( \text{Fix} \) applied to any function \( F \) gives \( F \) and repeats \( F \) by applying \( \text{Fix} \) to \( F \) one more time
  - there are many such fixed point operators
- The fixed point operator commonly defined in the \( \lambda \)-calculus is called the (lazy) \( Y \) combinator
  - \( Y = (\lambda f. ((\lambda x.f(xx)) (\lambda x.f(xx)))) \)
  - \( Y \ E = (\lambda f. ((\lambda x.f(xx)) (\lambda x.f(xx)))) E \)
    \( \Rightarrow (\lambda x.E(xx)) (\lambda x.E(xx)) \)
    \( \Rightarrow E ((\lambda x.E(xx)) (\lambda x.E(xx))) \)
  - since \( Y \ E \Rightarrow E (Y \ E) \Rightarrow E (E (Y \ E)) \Rightarrow E (E (E (Y \ E)) ..) \)
    - i.e., \( Y \) is a fixed point operator that when applied to any expression \( E \) applies \( E \) to a copy of itself repeatedly (i.e., recursively)
    - the "recursion" only terminates if \( E \) has a terminating condition
- Homework: evaluate \( Y \lambda x.x \)

Using the \( Y \) combinator

- Any expression of the form
  - \( f \ x_1 ... x_n = E \)
    is called recursive if \( f \) occurs free in \( E \)
  - if you want: \( f \ x_1 ... x_n = ~~~~ f ~~~~ \) then define
    - \( F = Y (\lambda f \ x_1 ... x_n. ~~~~ f ~~~~) \)
    - for example:
      - \( \text{mult} = \lambda m \ n. (\text{iszero} m \rightarrow 0 \mid \text{add} n \ (\text{mult} \ (\text{pred} \ m) \ n)) \)
      - \( \text{Becomes} \)
        - \( \text{multfn} = \lambda f \ m. \ (\text{iszero} m \rightarrow 0 \mid \text{add} n \ (f \ (\text{pred} \ m) \ n)) \)
        - \( \text{mult} = Y \ \text{multfn} \)
        - \( \text{mult} \ x \ y = (Y \ \text{multfn}) \ x \ y \)
          \( \Rightarrow \text{multfn} \ (Y \ \text{multfn}) \ x \ y \)
          \( \Rightarrow \text{multfn} \ \text{mult} \ x \ y \)
          \( \Rightarrow (\lambda f \ m. \ (\text{iszero} m \rightarrow 0 \mid \text{add} n \ (f \ (\text{pred} \ m) \ n))) \ \text{mult} \ x \ y \)
          \( \Rightarrow (\text{iszero} \ x \rightarrow 0 \mid \text{add} y \ (\text{mult} \ (\text{pred} \ x) \ y)) \)
          \( \Rightarrow (\text{iszero} \ x \rightarrow 0 \mid \text{add} y \ ((Y \ \text{multfn}) \ (\text{pred} \ x) \ y)) \)
          \( \Rightarrow ... \)
“Fix” in Haskell

- This works because of lazy evaluation.
  - if Haskell had eager evaluation, \( f (\text{fix } f) \) would never terminate:
    \[
    f (\text{fix } f) \Rightarrow f f (\text{fix } f) \Rightarrow f f f (\text{fix } f) \Rightarrow \ldots
    \]

\[
\text{fix } f = f (\text{fix } f)
\]

\[
\text{fact} :: \text{Integer} \rightarrow \text{Integer}
\]

\[
\text{fact} = \text{fix} (\lambda f \ n \rightarrow \text{if } n == 0 \text{ then } 1 \text{ else } n * \text{fact}(n-1))
\]

\[
\text{fact } 3 \Rightarrow (\lambda f \ n \rightarrow \text{if } n == 0 \text{ then } 1 \text{ else } n * \text{fact}(n-1)) (\text{fix } f) 3
\]

\[
= (\text{if } 3 == 0 \text{ then } 1 \text{ else } 3 * (\text{fix } f)(3-1))
\]

\[
= (3 * ((\lambda f \ n \rightarrow \text{if } n == 0 \text{ then } 1 \text{ else } n * \text{fact}(n-1)) (\text{fix } f)(3-1)))
\]

\[
= \ldots
\]

\[
= (3 * (2 * (1 * 1)))
\]

\[
= 6
\]

What about Eagerly Evaluated Scheme?

- We can’t use the (lazy) \( \mathbf{Y} \) combinator in Scheme because it would not terminate under applicative order (eager) evaluation:

\[
(\text{define } \mathbf{Y} (\lambda f (\lambda x ((\lambda x (f (\lambda y ((x x) y)))) (\lambda x (f (\lambda y ((x x) y)))))())
\]

\[
(\text{define T} (\lambda f ((\lambda x (f (\lambda y ((x x) y)))) (\lambda x (f (\lambda y ((x x) y))))))
\]

\[
(\text{define fact} (\lambda g (\lambda n (\text{if} (\text{zero?} n) 1 (* n (g (- n 1)))))
\]

\[
5/24/09
\]

15