CMPUT325: Applications of λ-calculus

Dr B. Price and Dr. R Greiner

18th October 2004

Introduction

- λ-calculus models calculation using only 2 rules
- Can only represent functions and application
- Explicit datastructures and control structures: `absent!`
- How can λ-calculus implement any calcluation?
- Possible to formalize data and control as function application
- Standard idioms map high-level data structures and control into λ-C expressions

Abstract Numbers I

- The concept of number can be built up from “0”, "successor"

  i.e., `1=successor(0), 2=successor(1), 3=successor(2)` ...
- Let $\sigma(n)$ ≡`successor(n)` ... so $1 = \sigma(0)$, $3 = \sigma(\sigma(\sigma(0))$, ...
- Numbers ≈ a sequence of function applications
- Addition is a short-hand for composing successor

  2+1 $\equiv \sigma(\sigma(\ \sigma(0)\ )) \equiv$ 3
- "Zero" is called the additive identity.
  - $\Rightarrow \quad n + 0 = n \quad$ for any n

Abstract Numbers II

- Successor and addition have natural inverses:
- Predecessor is the number before the current one.
  Let $\pi(n)$ be the predecessor of n.
  - $b = \sigma(a) \Rightarrow \pi(b) = a$
- Substraction is the inverse of addition: a+b=c ⇒ c-b=a
- Multiplication can be defined in terms of addition;
  and division as inverse of multiplication
- Negative numbers and real-numbers can be dervied from
  addition and division
- First, we need to define the successor function and zero

λ-Calculus Numbers

- Church found idioms with the desired properties:
  - Each number $\approx$ 2-argument functions
  - 0 $\equiv$ (λs | (λz | z)) $\equiv$ (λs z | z)
  - Successor $\sigma(n) \equiv$ (λx | (λ s z | s (x s z))) ⟨n⟩
    - Always returns 2-arg function (λ s z | s (⟨n⟩ s z))
    - Note: (λ s z | s (⟨n⟩ s z)) applies ⟨n⟩ to 2 function-constants
    - Application "copies" body of number into new function
- The successor of zero:

$\sigma(0) \equiv$ (λx s z | s (x s z)) (λs z| z)
Free vars get bound in (λs z| z)? No - no free vars!
→ (λ s z | s ((λs z | z) s z))
→ (λ s z | s ((λs z | z)) s z))
→ (λ s z | s z) $\equiv$ (λ s z | (s z))

Successor

- The successor of one:

$\sigma(1) \equiv$ (λx s z | s (x s z)) (λ s z | (s z))
$\equiv$ (λ s z | s ((λ s z | (s z)) s z))
$\equiv$ (λ s z | s ((λ s z | (s z)) s z))
$\equiv$ (λ s z | s (s z)) $\equiv$ (s z | (s (s z)))

- The successor of two:

$\sigma(2) \equiv$ (λx s z | s (x s z)) (λ s z | (s (s z)))
$\equiv$ (λx s z | s ((λ s z | (s (s z))) s z))
$\equiv$ (λx s z | s ((λ s z | (s (s z))) s z))
$\equiv$ (λx s z | s (s (s z)))
$\equiv$ (λx s z | (s (s (s z))))

Addition

- (+ m n) ≡ (λ x y | (λs z | x s (y s z))) ⟨m⟩ ⟨n⟩
  - Returns a 2-arg function: (λs z | ⟨m⟩ s (⟨n⟩ s z))
  - ⟨n⟩ applied to s z (copies body into new function)
  - ⟨m⟩ is applied to s and copy of ⟨n⟩
    - A total of ⟨m⟩ successors are composed onto ⟨n⟩
- Addition of 1+1

```
(λx y | (λs z | x s (y s z) ) )
                   (λ s z | s z)    (λ s z | s z)
≡ (λs z | (λ s z | s z) s ((λ s z | s z) s z) )
≡ (λs z | (λ s z | s z) s s z)
≡ (λs z | s s z)
```


Successor as Addition

- Check: $\sigma(n)$=(+ 1 n)

```
(λx y | (λs z | x s (y s z) ) ) (λ s z | s z)
≡ (λx y | (λs z | (λ s z | s z) s (y s z) ) )
≡ (λx y | (λs z | (λ s z | s z) s (y s z) ) )
≡ (λy | (λs z | s (y s z) ) )
```

- ... equivalent to our definition of successor !
  (λx s z | s (x s z))

Multiplication

- (* m n) ≡ (λ x y (λs | x (y s))) ⟨m⟩ ⟨n⟩
  - ⟨n⟩ passed as 1^{st} argument to number ⟨m⟩=(λsz|...)
  - Body of ⟨n⟩ is copied once for each successor op in ⟨m⟩
- (* 3 2)

```
(λ x y (λs | x (y s)))
     (λs z | s (s (s z))) (λs z | s (s z))
≡(λs | (λs z | s (s (s z))) ((λs z | s (s z)) s))
≡(λs | (λs z | s (s (s z))) ((λs z | s (s z)) s))
≡(λs | (λs z | s (s (s z))) (λz | s (s z)))
≡(λs | (λs z | s (s (s z))) (λz | s (s z)))
≡(λs | (λz |
   (λz | s (s z))((λz | s (s z))((λz | s (s z)) z)))
≡(λs z | s (s (s (s (s (s z))))))
```


Multiplication by Zero

- (* 0 m)

```
(λ x y | (λs | x (y s))) (λs z| z) ⟨m⟩
≡(λ y | (λs | (λs z| z) (y s))) ⟨m⟩
≡(λ y | (λs | (λs z| z) (y s))) ⟨m⟩
≡(λ y | (λs | (λz | z))) ⟨m⟩
≡(λ y | (λs z | z)) ⟨m⟩
```

- Take any number m = (λs z | s s ... s z)

```
(λ y (λs z|  z)) m
≡ (λs z|  z)
```

Predecessor and Subtraction

- For n>0, predecessor returns the integer before n, otherwise it returns zero
- There is no way to take apart a λ-calculus expression easily (Predecessor is not simply removing an 's' from the body of a Church number)

```
PREDECESSOR≡
  n (λag|(a(λbc|c))(⟨successor⟩(a(λbc|c))))
                                      (λg|00) (λab|a)
```

- Applies a `function` that maps (x,y) to (y,y+1) to the pair `(0,0)` n times
- Results in the pair (n-1,n)
- The `left number`, n-1 is the predecessor
- (- m n) ≡(λmn | n⟨predecessor⟩m)

Boolean Expressions

- Define the boolean values
  - True: `T = (λ c d | c)`
    - Returns its first argument
  - False: `F ≡(λ c d | d)`
    - Returns its second argument
- Boolean functions

```
(not m) ≡ (λx | x F T)
        ≡ (λx | x (λc d| d) (λc d|c) )
```

- Example `(not t)`

```
≡ (λx | x (λc d| d) (λc d|c) ) (λc d|c)
≡ (λc d|c) (λc d| d) (λc d|c)
≡ (λc d|c) (λc d| d) (λc d|c)
≡ (λc d| d)  ≡ F
```

Boolean Expressions

- (and m n) ≡(λx y | x y F)
  - So if x is F, will return 2^{nd} arg F
  - Otherwise if x is T will return 1^{st} arg y

```
(and T F)
≡ (λx y| x y F) (λcd|c) (λcd|d)
≡ ((λcd|c) (λcd|d) F)
≡ ((λcd|c) (λcd|d) F)
≡ (λcd|d)
(and F T)
≡ (λx y| x y F) (λcd|d) (λcd|c)
≡ (λcd|d) (λcd|c) F
≡ (λcd|d) (λcd|c) F
≡ F ≡ (λcd|d)
```


OR, ZEROP and Math Predicates

- OR(⟨F⟩, ⟨G⟩) is true if ⟨F⟩ is true or ⟨G⟩ is true

  OR(x)≡(λwz|wTz)

- ZEROP(n) returns true if n=0

  zerop(n)≡(λx|x F not F)

- Relations on integers

  x≥y≡zerop(x-y)
  x<y ≡ not x≥y
  x=y ≡x≥y AND y≥x

Conditional

- If P then M else N $\equiv$ (λuvw | uvw) PMN
- P is a function returning true T or false F
- Recall, T returns first argument, F returns second
- Example: (IF T M N)

```
≡ (λuvw | uvw) TMN
≡ (λuvw | uvw) TMN
≡ (λvw | Tvw) MN
≡ (λvw | (λcd|c) vw) MN
≡ (λvw | v) MN
≡ M
```


Lists

- Cons cell `(M . N)` represented as 1 arg function `(λz| z M N)`
- Cons operator: (cons M N) $\equiv$(λx y `(λ z | z x y)`) M N
- First: (car m) $\equiv$ (λx | x T) m where T$\equiv$(λcd|c)
- Rest: (cdr m) $\equiv$(λx | x F) m where F$\equiv$(λcd|d)
- Example: (car (λ z | z M N))

```
≡ (λx | x T) (λ z | z M N)
≡ (λ z | z M N) T
≡ (T M N) ≡ ((λcd|c) M N) ≡ M
```

- Example: (cdr (λz| z M N))

```
≡ (λx | x F) (λ z | z M N)
≡ (λ z | z M N) F
≡ (F M N) ≡ ((λcd|d) M N) ≡ N
```

Alternative Definition of Numbers I

- Define numerals as recursive lists
  - $0 \equiv (\lambda x|\ x)$
  - $\sigma(n) \equiv$(1+ n) $\equiv$(cons F n) for all $n \geq 0$ where F$\equiv(\lambda$cd|d)
  - $\pi(n) \equiv$(1- n) $\equiv$(cdr n) $\equiv (\lambda$x | x F)
  - (zerop n) $\equiv$ (first n) $\equiv (\lambda$x | x T)
- Examples

$\sigma(n) \equiv$ (cons F $\cdot$)
$\equiv (\lambda$x y (λ z | z x y)) F
$\equiv (\lambda$y (λz | z F y))

Alternative Definition of Numbers II

1 $\equiv \sigma(0) \equiv$ (cons F 0)
$\equiv (\lambda$y | (λz| z F y)) (λx| x)
$\equiv (\lambda$z| z F (λx| x)))
$\equiv$ [F 0]

2 $\equiv\sigma(1) \equiv$ (cons F 1) $\equiv$ (cons F (cons F 0))
$\equiv (\lambda$y | (λz| z F y)) (λz| z F (λx| x)))
$\equiv (\lambda$z| z F (λz| z F (λx| x))))
$\equiv$ [F F 0]

3 $\equiv\sigma(1) \equiv$ (cons F 1) $\equiv$ (cons F (cons F 0))
$\equiv (\lambda$y | (λz| z F y))
(λz| z F (λz| z F (λx| x))))
$\equiv (\lambda$z| z F
(λz| z F (λz| z F (λx| x)))))
$\equiv$ [F F F 0]

Alternate Definition of Plus

- Analysis of examples of (+ m n)

```
(+ [0] [0])    → [0] Easy
(+ [0] [F 0]) → [F 0] Easy
(+ [F 0] [F 0]) → [F F 0]
  ≡(+ [0] [F F 0]) Made easy
(+ [F F 0] [F 0])
  ≡ (+ [0] [F F F 0]) →[F F F 0] Made easy
```

- Leads to a recursive definition of plus (+ m n)

```
plus≡(λx y |
        (zerop x)   ;; T returns first arg!
          y
          (plus (pred x) (succ y)) )  m n
```

- Recursive plus is self-referential
- Need a technique for recursion

Recursion in λ-Calculus

- Recursion reuses same code repeatedly. Earlier we saw

```
(λ x | x x) (λ x | x x)
→ (λ x | x x)  (λ x | x x)
```

- The expression (λ x | x x) is preserved indefinitely!
- Let R be a function. What does this variation do?

```
(λ x | R (x x)) (λ x | R (x x))
→ R ( (λ x | R (x x)) (λ x | R (x x)) )
```

- Note:
  - 1 copy of R "peeled" off
  - Self-replicating expression preserved:
    (λ x | R (x x)) (λ x | R (x x))

Fixed-Point Combinator

- A fixed point for a function is an argument whose image is itself

  `x is a fixed-point of f, if f(x)=x`

- The square function has 2 fixed points $0^2 = 0$ and $1^2 = 1$
- A fixed-point combinator finds the fixed point of a function.
- Let `Y` be a fixed-point combinator. `Y(square)=1`
  - Because `square(1)=1`
  - By definition: `square(Y(square))=Y(square)`
- In general: a fixed-point combinator is a function `Y` with the property `F(Y(F)) = Y(F)` for all functions `F`

λ-Calculus Fixed-Point Combinator I

- Define a function: R≡(λ f | ⟨body⟩)
- Define a fixed-point combinator

  Y ≡ (λy | (λx | y (x x)) (λx | y (x x)))

- Apply fixed-point combinator to R to find a fixed-point

  (Y R)≡(λy | (λx | y (x x)) (λx | y (x x))) R
  $\xrightarrow{\beta}$(λx | R (x x)) (λx | R (x x))

- Denote fixed-point: ⟨YR⟩≡(λx | R (x x)) (λx | R (x x))

λ-Calculus Fixed-Point Combinator II

- Evaluating ⟨YR⟩

```
⟨YR⟩≡(λ x | R (x x)) (λ x | R (x x))
 β
 → R ( (λ x | R (x x)) (λ x | R (x x)) )
    R      ((λx|R (x x)) (λx|R (x x)))
 function      self-replicating form
 ≡ R ⟨YR⟩
```

- Evaluate ⟨YR⟩ whenever we need a copy of R
- ⟨YR⟩ is an R factory

Specific Fixed-Point Combinators

- Combinator we use was discovered by Haskell B. Curry

```
Y ≡ (λy | (λx | y (x x)) (λx | y (x x)))
```

- Combinator discovered by Alan Turing

```
Θ = (λx|(λy|(y (x x y))) (λx|(λy|(y (x x y)))))
```

- This one works for applicative order reduction

```
Θ_V = (λh| (λx|(h (λy|(y (x x y)))))
           (λx|(h (λy|(y (x x y))))) ))
```

Recursion with Haskell Combinator

- Define: R≡(λ f y | ⟨body⟩)
- **Normal order** eval of combinated R to argument m

```
⟨YR⟩ m    ;; what happen's when we eval ⟨YR⟩?
≡ R ⟨YR⟩ m
```

- What's next step in Normal order reduction?
- Do leftmost apply: Apply R to its arguments
  - R's 1^{st} arg f : self-replicating combinated function ⟨YR⟩
  - R's 2^{nd} arg y : m
  - R "performs calculations on" its argument y=m
  - R evaluates to either:
    - a single value, say n, for a base case
    - "copy" of itself stored in f applied to a reduced value, ⟨YR⟩ m' for recursive case

Recursion Example: Non-recursive

- Ignoring f arg, what does this "sort-of" compute?

```
F = (λf n | (zerop n)          ;; T returns 1st arg
              1
              (* n (f (1- n))) )
```

- Basically, factorial: f(0)=1, f(1)=1, f(2)=2, f(3)=6 ...
- Let d be a dummy function constant:

```
F d 0 →  1
For m>0  F d m
→  (* n (d (1- n)))
```

- d undefined!

Factorial Example: Base Case

```
F ≡ (λf n| (zerop n)   ;; T returns first arg
                 1
                 (* n (f (1- n))) )
```

- Use Haskell combinator Y to make F recursive

```
(Y F)
≡ (((λ y | (λ x | y (x x)) (λ x | y (x x)))
   (λf n|(zerop n) 1 (* n (f (1- n)))) )
```

$\xrightarrow{\beta}$ ⟨YF⟩ ;; long expr with 2 copies of F

- Factorial example: base case

⟨YF⟩ 0 ;; what happens when ⟨YF⟩ is eval'd?

$\xrightarrow{\beta}$ F ⟨YF⟩ 0

$\xrightarrow{\beta}$ 1

Factorial Example: Recursive Case

```
F ≡ (λf n| (zerop n)   ;; T returns first arg
                 1
                 (* n (f (1- n))) )
```

- Factorial example: recursive case
(Roughly in partly applicative order...)

⟨YF⟩ 1

$\xrightarrow{\beta}$ F ⟨YF⟩ 1

$\xrightarrow{\beta}$ (* 1 (⟨YF⟩ (1- 1))

$\xrightarrow{\beta}$ (* 1 (F ⟨YF⟩ (1- 1))

$\xrightarrow{\beta}$ (* 1 1)

$\xrightarrow{\beta}$ 1

Plus Example: Recursive Solution

```
(+ [F F 0] [F 0])
  ≡ (+ [0] [F F F 0]) →[F F F 0]
```

```
P≡(λp x y |
      (zerop x)
         y
         (p (pred x) (succ y)))
```

$(\texttt{Y P}) \xrightarrow{\beta} \langle\texttt{YP}\rangle$

$\langle\texttt{YP}\rangle\ \texttt{1 2}$

$\xrightarrow{\beta}\ \texttt{P}\ \langle\texttt{YP}\rangle\ \texttt{1 2}$

$\xrightarrow{\beta}\ \langle\texttt{YP}\rangle\ \texttt{0 3}$

$\xrightarrow{\beta}\ \texttt{P}\ \langle\texttt{YP}\rangle\ \texttt{0 3} \xrightarrow{\beta}\ \texttt{3}$

Summation Example: Recursive Solution

```
Key idiom: sum(n)=n+sum(n-1)
```

```
S≡(λs n | (zerop n) 0 (+ n (s (1- n))))
```

$(\texttt{Y S}) \xrightarrow{\beta} \langle\texttt{YS}\rangle$

$\langle\texttt{YS}\rangle\ \texttt{1}$

$\xrightarrow{\beta}\ \texttt{S}\ \langle\texttt{YS}\rangle\ \texttt{1}$

$\xrightarrow{\beta}\ \texttt{(+ 1 (}\langle\texttt{YS}\rangle\texttt{ (1- 1)))}$

$\xrightarrow{\beta}\ \texttt{(+ 1 (S }\langle\texttt{YS}\rangle\texttt{ 0))}$

$\xrightarrow{\beta}\ \texttt{(+ 1 0)} \xrightarrow{\beta}\ \texttt{1}$