Agenda

Lambda Calculus

Church-Turing Hypothesis

- Computers vs. Programs
  - They have independent existence.
  - Programs pre-date (at least digital) computers – Church's lambda calculus (1935) about the same time as Turing Machines (1936)
- Church-Turing Hypothesis
  - "Computability" is defined by Turing-Computability
- Practical Programming
  - TM ⇔ RAM ⇔ Von. Nuemann Computers ⇔ Assembly Language ⇔ C Language
  - Thus TM model serves as foundation for "imperative languages" like C.
  - There is a similar foundation for functional programming: Lambda Calculus.

Church-Turing Equivalence

- Church-Turing equivalence:
  - Lambda Calculus ⇔ (Partial) Recursive Functions ⇔ Turing Machines
  - Refer to Lewis & Papadimitriou or Aho,Hopcroft,Ullman for proof of: Recursive Functions ⇔ Turing Machines
  - To prove:
    Lambda Calculus ⇔ (Partial) Recursive Functions

Lambda Calculus

- Recursive functions include simple notions of:
  - Numeric Functions, Names (for functions), Recursion, Iteration (i.e. Minimization)
- Lambda calculus (LC) includes (only):
  - Functions and Names (i.e variables)
- In LC,
  - Everything is a function (variables also denote functions)
  - Functions take functions as arguments and return functions as values

Lambda Calculus – Abstraction

- Consider the following example:
  - $(\lambda x.\ 3x^2+1)\ 5 = 3*5^2 + 1 = 76$
- This is an expression –
  - in particular the application of a function on a value: $\lambda x.\ 3x^2+1$ is a function.
- In LC, a function is known as an abstraction:
  - $3*5^2 + 1$, $3*2^2 + 1$, $3*10^2 + 1$ are all concrete instances of the abstraction $3x^2+1$ over variable x.
  - So, we refer to $\lambda x.\ 3x^2+1$ as an abstraction and
  - we refer to x as a bound variable
    - because the abstraction binds the variable to the expression.

Lambda Calculus – Bound Variable

- The notion of a bound variable is analogous to that in calculus or predicate logic
  - If we write $\int f(x,y)\, dx$,
    - then x is a bound variable (e.g. it is bound to the integral)
    - but y is not bound – it is referred to as a free variable.
  - Similarly, if we write $\forall x.\ P(x,y)$
    - then x is a bound variable (e.g. it is bound to the quantifier)
    - but y is a free variable
- Thus, in LC
  - If we write $(\lambda x.\ x+y)$
    - then x is a bound variable and y is a free variable.
  - The scope of this x is the abstraction (in which it is bound).

Lambda Calculus - Syntax

- Back to the first example,
  - $(\lambda x.\ 3x^2+1)\ 5$ is an application of the abstraction $(\lambda x.\ 3x^2+1)$ on the value 5.
- So, in LC, there are two essential constructs
  - Abstraction which is of the form $(\lambda x.\ M)$ where M is in LC
  - Application which is of the form $M\ N$ where M and N are in LC
- Formally, LC syntax is defined by the grammar
  - Term → Var | (λ Var. Term) | Term Term

Lambda Calculus - Semantics

- The semantics of LC is obtained by defining the meaning of how application is evaluated.
- Considering the old example
  - $(\lambda x.\ 3x^2+1)\ 5 = 3*5^2 + 1 = 76$
    - I.e the abstraction is opened up: $3x^2+1$
    - The value 5 is substituted for x: $3*5^2 + 1$
    - Continue…
  - To generalize, $(\lambda x.\ M)\ N$ is evaluated by
    - Replacing occurrences of x by N in M
- Formally, $(\lambda x.\ M)\ N = M\ [\ N / x\ ]$
  - where T[N/x] is T in which var. x is replaced by N

Lambda Calculus - Substitution

- Consider more examples of substitution and application:

  a. $(x^2+x+1)\ [N/x] = N^2 + N + 1$

  b. $(x+ ((\lambda x.\ x+ 3)\ 5))\ \ [N/x] = N + ((\lambda x.\ x+ 3)\ 5)$

  Observe that the scope of the outer x is occluded by the inner abstraction, which introduces a new binding.

- So, M [N / x] refers to M where all free occurrences of x have been replaced with N:
  - In example (b) above the inner occurrence of x (as in x + 3) is not free.

Lambda Calculus – Substitution [2]

- When we evaluate M [N/x],
  - N may be any LC term – in particular, N may contain free occurrences of (some) variables bound in M:
    - e.g. (λ y. x y) [y / x]
  - In such cases, blind replacement would be inappropriate.
    - But bound variables can be consistently renamed – e.g.
    - (λ y. x y) = (λ z. x z)

Lambda Calculus – Substitution [3]

- So, we define the substitution M [N / x] as follows:
  1. replace all free occurrences of x in M with N if the free variables of N are not bound in M
  2. otherwise apply the renaming rule on M until the first case applies

Lambda Calculus – Substitution [4]

- Examples
  - $(x\ x)\ [\ y\ /\ x] = (y\ y)$
  - $(x\ y)\ [\ y\ /\ x] = (y\ y)$
  - $(x\ y)\ [(\lambda\ x.\ x\ x)\ /\ x] = (\lambda\ x.\ x\ x)\ y$
  - $(\lambda\ x.\ x\ x)\ [\ y\ /\ x\ ] = (\lambda\ x.\ x\ x)$
  - $(\lambda\ y.\ y\ y)\ [\ y\ /\ x\ ] = (\lambda\ y.\ y\ y)$
  - $(\lambda\ y.\ y\ x)\ [\ (y\ z)\ /\ x\ ] = (\lambda\ w.\ w\ x)\ [\ (y\ z)\ /\ x\ ]$
    $= (\lambda\ w.\ w\ (y\ z))$

Lambda Calculus

- In our examples we were using numbers like 3 and 5, and operators like + and *, but
  - LC syntax allows only variables or functions or applications.
- However, it turns out that this rudimentary syntax is enough to reconstruct all these elements.
  - i.e. LC is Turing-equivalent.
- We proceed to construct booleans, conditionals, numbers, numeric operations and eventually recursion.

Power of Lambda

- Booleans:
  - True ≡ $(\lambda x. (\lambda y. x))$
  - False ≡ $(\lambda x. (\lambda y. y))$
- What is the motivation?
  - Efficacy of Data representations are based on their use.
  - True and False are values useful in evaluating conditionals.
  - Conditionals take two expressions (say a then part and an else part) and choose one of the two.
    - True chooses the then part
    - False chooses the else part

Power of Lambda [2]

- Conditional (IF is defined as follows)
  - B E1 E2 where B is boolean
- Consider
  - True E1 E2 = (True E1) E2

    $= ((\lambda x.\ (\lambda y.\ x))\ E1)\ E2$

    $= (\lambda y.\ E1)\ E2$

    $= E1$
  - False E1 E2 = (False E1) E2

    $= ((\lambda x.\ (\lambda y.\ y))\ E1)\ E2$

    $= (\lambda y.y)\ E2$

    $= E2$

Power of Lambda [3]

- Formally
  - IF $\equiv$ (λb. (λe1. (λe2. b e1 e2)))
- Other boolean operations can also be constructed:
  - NOT $\equiv$ (λb. b False True)
  - Check this out:
    - NOT True = ?
    - NOT False = ?

Power of Lambda [4]

- Pairing
  - [M, N] $\equiv$ (λb. IF b M N)
  - FIRST $\equiv$ (λp. p True)
  - SECOND $\equiv$ (λp. p False)
- Lists
  - Obtain by nested pairing.

Lambda Calculus - Review

- Syntax

  Abstraction **Application**

  - Term → Var | (λ Var. Term) | Term Term

- Semantics
  - Rule for Application: **β-reduction**
  - $(\lambda x.\ M)\ N = M\ [\ N\ /\ x\ ]$
  - where T[N/x] is T in which var. x is replaced by N

    substitution

  Evaluation is repeated application, because everything in LC is a function!

Lambda Calculus – Review

- substitution M [N / x] :
  1. replace all free occurrences of x in M with N if the free variables of N are not bound in M
  2. otherwise apply the renaming rule on M until the first case applies
- Substitution requires renaming
  - Rule for renaming:
  - $(\lambda x.\ M) = (\lambda y.\ N)$ where y does not appear in M and N is M[y/x]

α**-equivalence**

Lambda Calculus – Review

- Church Booleans:
  - True ≡ $(\lambda x.\ (\lambda y.\ x))$
  - False ≡ $(\lambda x.\ (\lambda y.\ y))$
- Conditional (IF is defined as follows)
  - B E1 E2 where B is boolean

Lambda Calculus

- Points of Notation:
  - Application is left-associative
    - i.e. P Q R is actually (P Q) R
  - Currying
    - (λx y. M) is same as (λx. λy. M)
  - [Theoretical aside: Currying is in effect equivalent to s-m-n Theorem on Turing machines - Refer to Hopcroft, Ullman]

Church Lists

- Pairing
  - $[M, N] \equiv (\lambda b.\ \text{IF } b\ M\ N)$
  - $\text{FIRST} \equiv (\lambda p.\ p\ \text{True})$
  - $\text{REST} \equiv (\lambda p.\ p\ \text{False})$
  - E.g.

$$\text{FIRST } [M,N] \equiv (\lambda p.\ p\ \text{True})\ (\lambda b.\ \text{IF } b\ M\ N)$$

$$\Rightarrow (\lambda b.\ \text{IF } b\ M\ N)\ \text{True}$$

$$\Rightarrow \text{IF True } M\ N$$

$$\Rightarrow M$$

Church Lists

- Lists are obtained by nested pairing:
  - e.g. The list (a,e,i,o,u) is encoded as [a,[e,[i,[o,u]]]]

- Getting the Nth element:
  - Get2 $\equiv$ (λl. FIRST (REST l))
  - Get5 $\equiv$ (λl. FIRST (REST (REST (REST (REST l)))))

Church Numerals

- Numerals (close to unary notation):
  - 0 ≡ λx. x
  - N+1 ≡ [False, N]
    - E.g. 4 = (False, False, False, False, 0)
- (Elementary) Arithmetic
  - We need Succ, Pred, Zero? to start:
    - Succ N ➔ N+1
    - Pred N+1 ➔ N
    - Zero? 0 ➔ True and Zero? N+1 ➔ False

Church Numerals

- Succ $\equiv \lambda$n. [False, n]
- Pred $\equiv$ REST
- Zero? $\equiv \lambda$n. n True
  - E.g.
  - Zero? $\underline{0}$ = (λn. n True) (λn. n)
    $\rightarrow$ (λn. n) True
    $\rightarrow$ True
  - Zero? $\underline{N+1}$ = (λn. n True) [False, $\underline{N}$]
    $\rightarrow$ [False, $\underline{N}$] True
    = (λb. b False $\underline{N}$) True
    $\rightarrow$ True False $\underline{N}$
    $\rightarrow$ False

Repetition

- Recursion: To introduce recursion, we introduce a simple non-terminating construct.
  - Omega $\equiv$ (λx. x x) (λx. x x)
  - Omega ➔ ??
- But Omega isn't useful as it simply repeats itself.
- To make it useful, we introduce a function parameter:
  - Omega' $\equiv$ (λx. f (x x)) (λx. f (x x))
  - Omega' ➔ ??

Repetitive Application

- Omega' = (λx. f (x x)) (λx. f (x x))

  ➔ f (λx. f (x x)) (λx. f (x x))

  = f Omega'

  ➔ f f Omega'

  …
- How do we control/terminate this?
  - Make an abstraction out of Omega'
  - Call it Y
    - Y = (λf. (λx. f (x x)) (λx. f (x x)))

Controlled Repetition

- What does Y do?
  - Y g = $(\lambda f.\ (\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x)))\ g$
  - $\rightarrow (\lambda x.\ g\ (x\ x))\ (\lambda x.\ g\ (x\ x))$
  - $\rightarrow g\ (\lambda x.\ g\ (x\ x))\ (\lambda x.\ g\ (x\ x))$
  - $\rightarrow g\ (Y\ g)$

Recursive Function – Example

Church representation of 1

- How do we use it?
  - Consider the factorial function (not exactly in LC):
  - fact $\equiv$ (λn. IF (zero? n) $\underline{1}$ (prod n (fact (pred n))))
  - This isn't permissible because
    - we are defining *fact* but using it as a free variable in the definition.
  - To circumvent this, introduce another abstraction: Define F as
  - F $\equiv$ (λfact. (λn. IF (zero? n) $\underline{1}$ (prod n (fact (pred n)))))
  - Now consider, Y F ➔ ??

Recursive Function – Example

- Y F $\underline{m}$ = (λf. (λx. f (x x)) (λx. f (x x))) F $\underline{m}$
- → F (Y F) $\underline{m}$
- = (λfact. (λn. IF (zero? n) $\underline{1}$ (prod $\underline{n}$ (fact (pred $\underline{n}$))))) (Y F) $\underline{m}$
- → (λn. IF (zero? n) $\underline{1}$ (prod $\underline{n}$ ((Y F) (pred $\underline{n}$)))) $\underline{m}$
- → IF (zero? $\underline{m}$) $\underline{1}$ (prod $\underline{m}$ ((Y F) (pred $\underline{m}$)))
- = IF (zero? $\underline{m}$) 1 (prod $\underline{m}$ ((Y F) $\underline{m-1}$)))

Note: $\underline{m}$ is the Church representation of the numeral m

- Why is this correct? Proof by induction:
- Base Case: m=0 returns 1
- Hypothesis: Y F m-1 returns (m-1)! (for m > 0)
- Step: Y F m returns m * (m-1)! (for m > 0)

Recursion

- In general, given a function f,
  - X is known as a fix-point of f, if   f X = X
- Y computes fix-point for any f because,
  - Y f = f (Y f)
- Fix-Point theorem says:
  - For any recursive function f, a fix point exists (as witnessed by Y f).
- Now we can write other arithmetic functions such as add, mult and so on recursively.
  - Exercise!

Fix-points and Recursion [1]

- Consider the following (partial) functions:
  - f0(n) ≡ if (n=0) 1 else ⊥
  - f1(n) ≡ if (n=0) 1 else if (n<=1) n*f0(n-1) else ⊥
  - f2(n) ≡ if (n=0) 1 else if (n<=2) n*f1(n-1) else ⊥
  - f3(n) ≡ if (n=0) 1 else if (n<=3) n*f2(n-1) else ⊥
  - …
  - fj(n) ≡ if (n=0) 1 else if (n<=j) n*fj-1(n-1) else ⊥
- Observation: The infinite union of all fj for j ε ℕ is the *factorial* function.
  - A function is a set of ordered pairs (such that first elements are unique).
    - Therefore the union is a well-defined operation.
  - Each fj is consistent with fk for all k < j and all n < k.
    - Therefore the resulting union is also a function.
- Note:
  - Recursion captures this infinite union.

Fix-points and Recursion [2]

- Simplified form:
  - $f^0(n) \equiv$ if $(n=0)$ 1 else $\perp$
  - $f^1(n) \equiv$ if $(n=0)$ 1 else $n*f^0(n-1)$
  - $f^2(n) \equiv$ if $(n=0)$ 1 else $n*f^1(n-1)$
  - $f^3(n) \equiv$ if $(n=0)$ 1 else $n*f^2(n-1)$
  - …
  - $f^j(n) \equiv$ if $(n=0)$ 1 else $n*f^{j-1}(n-1)$
- And
  - $fact = \bigcup_{j \in \mathbb{N}} f^j$

Fix-points and Recursion [3]

- Simplified form using LC notation:
  - $f^0 \equiv \lambda n.\ (= n\ 0)\ 1\ \perp$
  - $f^1 \equiv \lambda n.\ (= n\ 0)\ 1\ (*\ n\ (f^0\ (\text{pred}\ n)))$
  - $f^2 \equiv \lambda n.\ (= n\ 0)\ 1\ (*\ n\ (f^1\ (\text{pred}\ n)))$
  - $f^3 \equiv \lambda n.\ (= n\ 0)\ 1\ (*\ n\ (f^2\ (\text{pred}\ n)))$
  - …
  - $f^j \equiv \lambda n.\ (= n\ 0)\ 1\ (*\ n\ (f^{j-1}\ (\text{pred}\ n)))$
- Each f^j is defined from its predecessor f^{j-1}
  - Can we automate this construction?
  - Consider

$$F \equiv \lambda f.\ \lambda n.\ (= n\ 0)\ 1\ (*\ n\ (f\ (\text{pred}\ n)))$$

  - Then, $F\ f^j = f^{j+1}$

Fix-points and Recursion [4]

- Given the definition
  - $F \equiv \lambda f.\ \lambda n.\ (= n\ 0)\ 1\ (*\ n\ (f\ (pred\ n)))$
- And the fact
  - $F\ f^j = f^{j+1}$
- We can ask, what is the fix-point of F?
  - F fact = fact where fact = $\cup_{j\ \varepsilon\ \mathbb{N}} f^j$
- Thus if you define the meta-function F for any iterative sequence of functions f^j,
  - Then the fix-point of F is equivalent to the recursive function defining ($\cup f^j$)

Fix-points and Recursion [4]

- Recall that Y computes the fix-point for any given function.
- we can define F as
  - F $\equiv$ λf. λn. (= n 0) 1 (* n (f (pred n)))
- And obtain its fix-point (Y F).

Partial Recursive Functions

- Recursive Functions –
  - The smallest set of numeric functions including initial functions and closed under composition, primitive recursion, and minimalization.
  - See Induction definition on the next slide.

Partial Recursive Functions

- Constant functions (for each n and k) are p.r:
  - F(x1 x2 … xk) = <u>n</u>
- Successor function is p.r.
- Projection functions (for each k and i) are p.r:
  - P(x1 x2 … xk) = xi
- If h, g1, g2, … gm, are p.r then f (as defined below) is p.r:
  - f(x1,x2,…,xk) = h(g1(x1,…xk), g2(x1,…xk),…gm(x1,…xk))
- If χ and ψ are p.r., then so is φ, where :

  $\varphi(0, n) = \chi(n)$

  $\varphi(m+1, n) = \psi(\varphi(m, n), m, n)$

if χ is p.r, then so is φ, where :

$\varphi(m) = \min_n [\ \chi(m, n)=0\ ]$

Lambda Calculus vs. Partial Recursive Functions

- To prove that lambda calculus is as powerful:
  1. Prove that initial functions are λ-definable.
  2. Prove that if g, h1, h2, … are λ-definable, then f is λ-definable where f x = g(h1(x), h2(x), …)
  3. Prove that if χ and ψ are λ-definable, then so is φ, where :

     $\varphi(0, n) = \chi(n)$

     $\varphi(m+1, n) = \psi(\varphi(m, n), m, n)$
  4. Prove that if χ is λ-definable, then so is φ, where :

     $\varphi(m) = \min_n [\ \chi(m, n)=0\ ]$

Lambda Calculus vs. Partial Recursive Functions

- Proof of 1 (obvious):
  - Constant functions (for each n and k):
    - λx1 x2 … xk. <u>n</u>
  - Successor (already defined)
  - Projection functions (for each k and i):
    - λx1 x2 … xk. xi
- Proof of 2:
  - Composition (for each m and k):
    - λh g1 g2 … gm. λx1 x2 … xk.
      h (g1 x1 x2 … xk) (g2 x1 x2 … xk) … gm(x1 x2 … xk)

Partial Recursive Functions in LC

- Proof of 3:
- (Try to)Define φ by:

λmn. IF (zero? m) (χ n) (ψ (φ (pred m) n) (pred m) n)

- But this is recursive, so define F by

F $\equiv$ λ φmn IF (zero? m) (χ n) (ψ (φ (pred m) n) (pred m) n)

- By Fix-Point Theorem,
  - φ = Y F.

Partial Recursive Functions in LC

- Proof of 4:
- Define φ by:

  λn. (λm. IF (zero? (χ m n)) n (φ (succ n) m))
- But this is recursive, so define F by

F $\equiv$ λ φ. λn. (λm. IF (zero? (χ m n)) n (φ (succ n) m))
- By Fix-Point Theorem,
  - φ = Y F

Reduction and Evaluation

- Single step of application is known as reduction (in particular as β-reduction):
  - $(\lambda x.\ M)\ N = M\ [\ N\ /\ x\ ]$
- Evaluation of a lambda term is achieved by repeated reductions:
  - $(\lambda n.\ n\ (\lambda x.\ \lambda y.\ x))\ (\lambda b.\ b\ (\lambda x.\ \lambda y.\ y)\ \underline{N})$

    ➔ $(\lambda b.\ b\ (\lambda x.\ \lambda y.\ y)\ \underline{N})\ (\lambda x.\ \lambda y.\ x)$

    ➔ $(\lambda x.\ \lambda y.\ x)\ (\lambda x.\ \lambda y.\ y)\ \underline{N}$

    ➔ $(\lambda y.\ (\lambda x.\ \lambda y.\ y))\ \underline{N}$

    ➔ $(\lambda x.\ \lambda y.\ y)$

Reduction and Evaluation

- When does evaluation step?
  - When there is no more reduction possible.
- Reduction of a term is possible only
  - if it is an application i.e. of the form u v
  - and u evaluates to an abstraction
- A term of the form
  - U V where U is an abstraction

  is known as reducible expression

  i.e. a redex

Reduction and Evaluation

- $(\lambda n.\ n\ (\lambda x.\ \lambda y.\ x))\ (\lambda b.\ b\ (\lambda x.\ \lambda y.\ y)\ \underline{N})$
  - ➔ $(\lambda b.\ b\ (\lambda x.\ \lambda y.\ y)\ \underline{N})\ (\lambda x.\ \lambda y.\ x)$
  - ➔ $(\lambda x.\ \lambda y.\ x)\ (\lambda x.\ \lambda y.\ y)\ \underline{N}$
  - ➔ $(\lambda y.\ (\lambda x.\ \lambda y.\ y))\ \underline{N}$
  - ➔ $(\lambda x.\ \lambda y.\ y)$

This expression cannot be reduced further – this is known as a ***normal form***

Reduction and Evaluation

- Evaluation may not terminate
  - $(\lambda x.\ x\ x)\ (\lambda x.\ x\ x)$

$\rightarrow (\lambda x.\ x\ x)\ (\lambda x.\ x\ x)$

$\rightarrow (\lambda x.\ x\ x)\ (\lambda x.\ x\ x)$

This expression does not ever result in a normal form.

Reduction Order

- Evaluate the expression
  - (λx. y) ((λx. x x) (λx. x x))
- Observations?

Reduction Order

- (Normal Order) Theorem:
  - If a lambda expression s can be reduced to a normal form, then the reduction sequence arising from s by reducing the left-most outermost redex will always result in normal form.

Church-Rosser Theorem

- If a term t reduces to s1 and t also reduces to s2, then there exists a term u such that both s1 and s2 reduce to u.

- Corollary:
  - Normal forms if they exist are unique (upto α-equivalence) for a given expression.