Lambda-Calculus

Vitaly Shmatikov

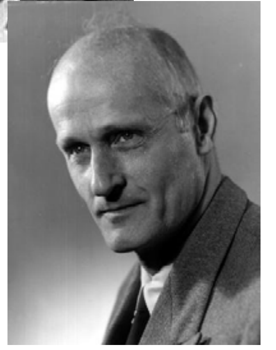
Reading Assignment

- Mitchell, Chapter 4.2

Lambda Calculus

- Formal system with three parts
  - Notation for function expressions
  - Proof system for equations
  - Calculation rules called reduction
- Additional topics in lambda calculus
  - Mathematical semantics
  - Type systems

History



- Origins: formal theory of substitution
  - For first-order logic, etc.
- More successful for computable functions
  - Substitution $\rightarrow$ symbolic computation
  - Church/Turing thesis
- Influenced design of Lisp, ML, other languages
- Important part of CS history and foundations

Why Study Lambda-Calculus?

- Basic syntactic notions
  - Free and bound variables
  - Functions
  - Declarations
- Calculation rule
  - Symbolic evaluation useful for discussing programs
  - Used in optimization (inlining), macro expansion
    - Correct macro processing requires variable renaming
  - Illustrates some ideas about scope of binding
    - Lisp originally departed from standard lambda calculus, returned to the fold through Scheme, Common Lisp

Expressions and Functions

- Expressions

  $x + y \qquad\qquad x + 2*y + z$

- Functions

  $\lambda x.\ (x+y) \qquad\qquad \lambda z.\ (x + 2*y + z)$

- Application

  $(\lambda x.\ (x+y))\ 3 \qquad = \ 3 + y$

  $(\lambda z.\ (x + 2*y + z))\ 5 \qquad = \ x + 2*y + 5$

Parsing: $\lambda x.\ f\ (f\ x) = \lambda x.(\ f\ (f\ (x))\ )$

Higher-Order Functions

- Given function f, return function f ∘ f

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

- How does this work?

  $(\lambda f.\ \ \lambda x.\ f\ (f\ x))\ (\lambda y.\ y+1)$

  $=\ \ \lambda x.\ (\lambda y.\ y+1)\ ((\lambda y.\ y+1)\ x)$

  $=\ \ \lambda x.\ (\lambda y.\ y+1)\ (x+1)$

  $=\ \ \lambda x.\ (x+1)+1$

Same result if step 2 is altered

Same Procedure (ML)

- Given function f, return function f ∘ f

  fn f => fn x => f(f(x))

- How does this work?

  (fn f => fn x => f(f(x))) (fn y => y + 1)

  = fn x => ((fn y => y + 1) ((fn y => y + 1) x))

  = fn x => ((fn y => y + 1) (x + 1))

  = fn x => ((x + 1) + 1)

Same Procedure (JavaScript)

- Given function f, return function f ∘ f

  function (f) { return function (x) { return f(f(x)); } ; }

- How does this work?

```
(function (f) { return function (x) { return f(f(x)); } ;  })
                (function (y) { return y + 1; })

function (x) { return (function (y) { return y + 1; })
                          ((function (y) { return y + 1; }) (x)); }

function (x) { return (function (y) { return y +1; }) (x + 1); }

function (x) { return ((x + 1) + 1); }
```

Declarations as "Syntactic Sugar"

```
function f(x) {
    return x+2;
}
f(5);
```

$$(\underbrace{\lambda f.\ \ f(5)}_{\text{block body}})\ \ (\underbrace{\lambda x.\ x+2}_{\text{declared function}})$$

Free and Bound Variables

- Bound variable is a “placeholder”
  - Variable x is bound in $\lambda x.\ (x+y)$
  - Function $\lambda x.\ (x+y)$ is same function as $\lambda z.\ (z+y)$
- Compare

  $\int x+y\ dx \ = \ \int z+y\ dz \qquad \forall x\ \ P(x) = \forall z\ \ P(z)$
- Name of free (i.e., unbound) variable matters!
  - Variable y is free in $\lambda x.\ (x+y)$
  - Function $\lambda x.\ (x+y)$ is <u>not</u> same as $\lambda x.\ (x+z)$
- Occurrences
  - y is free and bound in $\lambda x.\ ((\lambda y.\ y+2)\ x) + y$

Reduction

- Basic computation rule is β-reduction

$$(\lambda x.\ e_1)\ e_2 \quad \rightarrow \quad [e_2/x]e_1$$

where substitution involves renaming as needed (why?)

- Reduction
  - Apply basic computation rule to any subexpression
  - Repeat
- Confluence
  - Final result (if there is one) is uniquely determined

Renaming Bound Variables

- Function application

$$\underbrace{(\lambda f.\ \ \lambda x.\ f\ (f\ x))}_{\text{apply twice}}\ \ \underbrace{(\lambda y.\ y+x)}_{\text{add } x \text{ to argument}}$$

- Substitute "blindly" – do you see the problem?

$$\lambda x.\ [(\lambda y.\ y+x)\ ((\lambda y.\ y+x)\ x)]\ \ =\ \ \lambda x.\ x+x+x$$

- Rename bound variables

$$(\lambda f.\ \ \lambda z.\ f\ (f\ z))\ \ (\lambda y.\ y+x)$$

$$=\ \ \lambda z.\ [(\lambda y.\ y+x)\ ((\lambda y.\ y+x)\ z))]\ \ =\ \ \lambda z.\ z+x+x$$

Easy rule: always rename variables to be distinct

Main Points About Lambda Calculus

- λ captures the “essence” of variable binding
  - Function parameters
  - Declarations
  - Bound variables can be renamed
- Succinct function expressions
- Simple symbolic evaluator via substitution
- Can be extended with
  - Types, various functions, stores and side effects...

What is a Functional Language?

- “No side effects”
- Pure functional language: a language with functions, but without side effects or other imperative features

No-Side-Effects Language Test

Within the scope of specific declarations of $x_1, x_2, \ldots, x_n$, all occurrences of an expression e containing only variables $x_1, x_2, \ldots, x_n$, must have the same value.

```
begin
    integer x=3; integer y=4;
    5*(x+y)-3
    ...             // no new declaration of x or y //
    4*(x+y)+1
end
```

Backus’ Turing Award

- John Backus: 1977 Turing Award
  - Designer of Fortran, BNF, etc.
- Turing Award lecture
  - Functional programming better than imperative programming
  - Easier to reason about functional programs
  - More efficient due to parallelism
  - Algebraic laws
    - Reason about programs
    - Optimizing compilers

Reasoning About Programs

- To prove a program correct, must consider everything a program depends on
- In functional programs, dependence on any data structure is explicit (why?)
- Therefore, it's easier to reason about functional programs
- Do you believe this?

Quicksort in Haskell

- Very succinct program

```
qsort [] = []
qsort (x:xs) = qsort elts_lt_x ++ [x]
                               ++ qsort elts_greq_x
   where elts_lt_x = [y | y <- xs, y < x]
         elts_greq_x = [y | y <- xs, y >= x]
```

- This is the <u>whole thing</u>
  - No assignment – just write expression for sorted list
  - No array indices, no pointers, no memory management, ...

Compare: Quicksort in C

```
qsort( a, lo, hi ) int a[], hi, lo;
{ int h, l, p, t;
  if (lo < hi) {
       l = lo; h = hi; p = a[hi];
       do {
           while ((l < h) && (a[l] <= p)) l = l+1;
           while ((h > l) && (a[h] >= p)) h = h-1;
           if (l < h) { t = a[l]; a[l] = a[h]; a[h] = t; }
        } while (l < h);
        t = a[l]; a[l] = a[hi]; a[hi] = t;
        qsort( a, lo, l-1 );
        qsort( a, l+1, hi );
    }
}
```

Case Study

[Hudak and Jones, Yale TR, 1994]

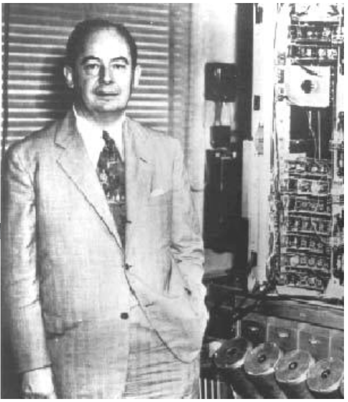
- Naval Center programming experiment
  - Separate teams worked on separate languages

Language	Lines of code	Lines of documentation	Development time (hours)
(1) Haskell	85	465	10
(2) Ada	767	714	23
(3) Ada9X	800	200	28
(4) C++	1105	130	–
(5) Awk/Nawk	250	150	–
(6) Rapide	157	0	54
(7) Griffin	251	0	34
(8) Proteus	293	79	26
(9) Relational Lisp	274	12	3
(10) Haskell	156	112	8

Some programs were incomplete or did not run

  - Many evaluators didn't understand, when shown the code, that the Haskell program was complete. They thought it was a high-level partial specification.

Von Neumann Bottleneck



- Von Neumann
  - Mathematician responsible for idea of stored program
- Von Neumann bottleneck
  - Backus' term for limitation in CPU-memory transfer
- Related to sequentiality of imperative languages
  - Code must be executed in specific order

```
function f(x) { if (x<y) then y = x; else x = y; }
g( f(i), f(j) );
```

Eliminating VN Bottleneck

- No side effects
  - Evaluate subexpressions independently
    - function f(x) { return x<y ? 1 : 2; }
    - g(f(i), f(j), f(k), ...);
- Good idea but ...
  - Too much parallelism
  - Little help in allocation of processors to processes
  - ...
- Effective, easy concurrency is a hard problem