CSE 505

Lecture #9

October 1, 2012

Lambda Calculus

Expr ::= Var | λ Var. Expr | (Expr Expr)

Key Concepts:

- Bound and Free Occurrences
- Substitution, Reduction Rules: α, β, η
- Confluence and Unique Normal Form
- Nontermination and Leftmost Reduction
- Church-Rosser Property (==>* and <==>*)
- Encoding Things in Lambda Calculus
- Recursion and Fixed-Point Operator (Y)

Recursion and Fixed-points

fact = λn.(((if (is0 n)) 1) ((mult n) (fact (pred n)))

t = λf. λn.(((if (is0 n)) 1) ((mult n) (f (pred n))))

Why does the fixed-point of t capture f?

Fixed point g has the property: g = (t g)

g = λn.(((if (is0 n)) 1) ((mult n) (g (pred n)))

Y = λf. (λx.(f (x x)) λx.(f (x x)))

In Lambda Calculus, Y is fixed-point operator, because (for any t):

(Y t) <==>* (t (Y t))

Proof:

(Y t) = (λf. (λx.(f (x x)) λx.(f (x x))) t)

==> (λx.(t (x x)) λx.(t (x x)))

==> (t (λx.(t (x x)) λx.(t (x x))))

<==> (t (Y t))

Least Fixed Point

Consider: letrec f(n) = if n=0 then 0 else f(n);

$$\text{Fixed-point } f1(n) = \begin{cases} 0, & \text{if } n=0 \\ 1, & \text{if } n \geq 0 \end{cases}$$

$$\text{Fixed-point } f2(n) = \begin{cases} 0, & \text{if } n=0 \\ 2, & \text{if } n \geq 0 \end{cases}$$

$$\text{Least fixed-point } g(n) = \begin{cases} 0, & \text{if } n=0 \\ ?, & \text{if } n \geq 0 \end{cases}$$

Typed Lambda Calculi

Thus far, we have studied the untyped lambda calculus, i.e., no types associated with vars.

There are two well-known calculi:

- the simply-type lambda calculus
- the second-order (polymorphic) lambda calculus

Interesting, adding types causes all lambda expressions to terminate! Cannot have (x x).

Lambda Calculus in PLs

Lisp: (lambda (f g) (lambda (x) (f (g x))))

Javascript: function(f g) {return function (x)
return f(g(x))}}

Python: comp = lambda f,g: lambda x: f(g(x))
double = lambda x:x*2
triple = lambda y:y*3
comp(double, triple)(100)
= 600

Functional languages such as Scheme, ML and Haskell also support lambda calculus.

Types in PLs

- Concrete and Abstract types
- Strong Typing
- Type equivalence and security
- Exceptions
- Polymorphism
- Type inference
- Advanced Type Systems

What is a 'type'?

Simple Answer: A type is a set of values.

e.g. type int = { ..., -2, -1, 0, 1, 2 ...}

Better Answer: A type is a set + operations.

e.g. type int = { ..., -2, -1, 0, 1, 2 ...} together with operations such as +, -, *, etc.

That is, a type is more like an algebra.

(Even better characterization is possible.)

What is a 'type' (cont'd)

Consider type int = { ..., -2, -1, 0, 1, 2 ...} with operations such as +, -, *, etc.

Are numerals ..., -2, -1, 0, 1, 2 ... necessary in defining int?

Can't we use ..., -10, -01, 0, 01, 10, 11, ... ?

Choice of literals not crucial ... We can define the values of a type using operations called constructors. But literals are useful in practice.

A type is a set of operations!

Abstract Data Type

zero: int
succ: int → int
+ : int * int → int
* : int * int → int
% : int * int → int
...
< : int * int → int

Refined int type

exception overflow,
underflow, dividebyzero
zero: int
succ: int → int
+ : int * int → int
* : int * int → int
% : int * int → int
...
< : int * int → bool

Why use types?

Three important reasons:

-faster execution:
e.g. for (i=1; i++; i<=n) { ... x * y ... }

-better program readability/documentation:
e.g. void f(x) { ... } vs void f(Tree x) { ... }

-early error checking:
e.g. int x; void f(string y){ ... }; ... f(x) ...

Strongly Typed Language

Popular Answer:
"every variable has a type declared for it"

Above criterion not sufficient (or even necessary!):

int i, j, k; ... $\boxed{\text{i := j ** k}}$...

Somewhat Better Answer:
"A language is strongly typed if compiler can perform all type-checking"

Strong Typing Defined

By a static (source code) analysis of the program:

1. the type of every expression can be unambiguously determined; and
2. all type equivalence tests can be unambiguously decided.

Pascal Type System

Unstructured
built-in: integer, real, char, boolean
user-defined: enumerations and subranges

Structured
user-defined: record, array, set, file, ↑

Note: Pascal does not have abstract types. Algol 68 preceded Pascal, and was the first language to introduce user-defined types ('modes').

Polymorphic Types: Motivation

Pascal is an important milestones in PLs, because of its type system.

But Pascal has first-order types, also known as monomorphic types.

Many modern PLs have higher-order types, or polymorphic types. They also support abstract data types, a concept that was not fully developed when Pascal was invented.

Polymorphic Types: Motivation

Pascal is an important milestones in PLs, because of its type system.

But Pascal has first-order types, also known as monomorphic types.

Many modern PLs have higher-order types, or polymorphic types. They also support abstract data types, a concept that was not fully developed when Pascal was invented.

Why 'Mono' Types not Enough

```
int length(l) {
        int len = 0;
        while (l !== nil) {
                len := len + 1;
                l := l.next;
        }
        return len;
    }
```

Desired type of length is: $(\forall t)$ t List → int

Polymorphic Types (cont'd)

Another example:

```
List append(l1, l2) {
        if (l1 = nil) return l2 else
         return cons(l1.val,
                        append(l1.next, l2));
}
```

Desired type of append is:

$(\forall t)$ t List * t List $\to$ t List

Polymorphic Types (cont'd)

The generic sorting procedure:

void sort(var a: array[t]) { ... }

might be given the type:

$(\forall t)$ t array $\to$ void

This type is not quite correct for sort.

Robin Milner

- Inventor of ML
- Introduced polymorphic types and type inference
- Received ACM Turing Award

Characteristics of ML

- Strongly typed language (with type inference):
  - (parametric) polymorphic types
  - concrete type
  - abstract types
- Higher-order functional language:
  - expression-oriented (like Lisp)
  - rule-based definitions, with pattern matching
  - higher-order functions (output can also be function)
  - static scoping, with nested function definitions
- Modular language:
  - signatures, structures, and functors

ML Type System

τ ::= int | real | bool | string | unit
| τ * ... * τ (N-tuple)
| { f_1:τ, ..., f_k : τ} (Record)
| τ list
| $\tau \to \tau$ (Function Type)
| ' id (Type variable)
| '' id
| user_defined_concrete_type
| user_defined_abstract_type

Overloaded op'r

negative literal

Type Name	Operations	Sample Literals
int	+, -, *, div, mod, =, >, >=, <>, ...	... ~2, ~1, 0, 1, 2 ...
real	+, -, *, /, =, >, >=, <>, ...	~3.14, 0.0, 3E2, 9.99 ...
bool	andalso, orelse, if then else, =, <>, ...	true, false
string	size, ^, =, <>, ...	"", "abcd", ...
unit	=, <>	()

'short-circuit' op'r

ML is a Functional Language

- Program = set of function definitions
- No imperative features, especially no assignments and updating.
- Computation is essentially the reduction of expression to a value.
- Keywords: functions, types, values, expressions, reduction strategies, ...

Type Inference

Examples:

fun f(x) = x + 1; ⇨ int → int

fun f(x,y) = (x+y, y*2.5); ⇨ real*real → real*real

fun f(x,y) = x*x + y ⇨ Unresolved overloaded operators + and *

Resolving overloaded operators

fun f(x:int, y) = x*x + y;

fun f(x, y:int) = x*x + y;

fun f(x, y):int = x*x + y;

fun f(x, y) = x*x + y:int;

if-then-else

if <expr$_1$> then <expr$_2$> else <expr$_3$>

$(\forall t)$ **bool * t * t → t**

Examples:

fun f(x,y) = if x > y then x*2 else y*2; ⇨ f: int*int → int

fun f(x,y) = if x > y then x*x else y*y; ⇨ Unresolved overloaded ops

Recursive Definitions

fact: int→int

```
fun fact(n) =
    if n=0
    then 1
    else n*fact(n-1);
```

n:int

agrees with fact: int->int

Another recursive definition

```
fun h(x,y) = if x then y else h(y,x);
```

x:bool y:bool

Hence, h: bool*bool → bool

Type inference is a process of propagating type information (bottom-up, top-down, and side-ways) in order to determine the types of all identifiers – variables or functions.

Type Inference - Remarks

In the absence of overloaded operators, types for all identifiers in an ML program can be unambiguously determined without any type declarations (assuming no type errors).

Local Vars & Nested Scopes

```
let <local_value_bindings>
 in  <expression>
end
```

```
let    val a = 10;
       val b = f(a)
in     let val a  = 20;
           val c  = g(a, b)
        in   (a+b) / (a - c)
       end
end
```



```
fun quad(a,b,c)=
    let  val disc = b*b - 4.0*a*c;
         val den = 2.0*a
      in
         if disc = 0.0 then ~b/den else
         if disc > 0.0 then
           let val  sq   = sqrt(disc)
            in ((sq-b)/den,  (~sq-b)/den)
           end
         else "imaginary"
    end;
```


Concrete Data Types

```
datatype root = imaginary
              | one of real
              | two of real*real;
```

Types for the constructors:

```
imaginary : root
  one : real -> root
  two : real * real -> root
```



```
fun quad(a,b,c)=
    let val disc = b*b - 4.0*a*c;
        val den = 2.0*a
     in
        if disc = 0.0 then one(-b/den) else
        if disc > 0.0 then
              let val  sq = sqrt disc
               in two((sq-b)/den,
                        (-sq-b)/den)
              end
        else imaginary
    end;
```

Almost correct, except for equality test

```
fun quad(a,b,c)=
        let val disc = b*b - 4.0*a*c;
             val den = 2.0*a;
             fun realeq(x,y) = abs(x-y) < 0.0001
         in
             if realeq(disc,0.0) then one(-b/den) else
             if disc > 0.0 then
                         let val  sq = sqrt disc
                          in two((sq-b)/den,
                                     (-sq-b)/den)
                        end
             else imaginary
       end;
```

val quad = fn: real * real * real -> root

Using 'quad'

```
- quad(1.0, 5.0, 6.0);
  val it = two(~2.0, ~3.0) : root

- quad(1.0, 2.0, 1.0);
  val it = one(~1.0) : root

- quad(1.0, 1.0, 1.0);
  val it = imaginary : root
```


Pattern Matching

```
fun nroots(imaginary) = 0
  | nroots(one(x)) = 1
  | nroots(two(x,y)) = 2;
```

root -> int

int -> int

```
fun fib(1) = 1
  | fib(2) = 1
  | fib(n) = fib(n-1) + fib(n-2);
```


Polymorphic Datatypes

```
datatype 'a list =   [ ]    |    ::  of  'a  *  'a list;
```

cons

Constructors:

[ ] :   'a list

:: :   'a * 'a list → 'a list

List Constants (literals)

- [1, 2, 3] = 1 :: 2 :: 3 :: [ ]
  : int list
- ["abc", "def"] = "abc" :: "de" :: [ ]
  : string list
- [[1, 2],[3]] = ((1::2::[]) :: (3::[]) :: [])
  : int list list

[1, "apple", 3.14] → badly typed list

[1, [2], [1,2,3]] → badly typed list

Polymorphically Typed Functions

```
fun length([ ])   = 0
  | length(h::t) = 1 + length(t);
```

What is the type of length?

- Output Type: int, since 0:int, and this agrees with the second case 1 + length(t);

- Input Type: the terms [ ] and h::t have types 'a list and 'b list, but a=b since they must be compatible.

Hence the type of length: 'a list → int.

Polymorphic Functions (cont'd)

```
fun member(x, [])      = false
  | member(x, h::t) =  if x=h
                        then true
                        else member(x,t)
```

"a * "a list → bool

Note: The following is incorrect:

Nonlinear Pattern Disallowed!

```
fun member(x, [])      = false
  | member(x, x::t) =  true
  | member(x, y::t) =  false
```

More examples

```
fun mystery([ ])   = []
  | mystery(h::t) = h @ mystery(t)
```

Mystery type analysis: 'a list * 'a list ->'a list

- input and output types must be lists;
- h @ mystery(t) indicates that h must be a list;

mystery: 'a list list -> 'a list

A simple tree datatype

```
datatype 'a tree=
          leaf of  'a
        | node of 'a tree  * 'a tree
```

Sample tree literals:

leaf(true)

node(leaf(1), leaf(2))

node(node(leaf(3.5), leaf(4.5)), leaf(1.5))

node(node(leaf("Ada"), leaf("C")),
     node(leaf("Java"), leaf("Prolog")))

'a tree → int

anonymous variable

```
fun depth(leaf(_)) = 0
  | depth(node(t1, t2)) =
        let val d1 = depth(t1);
            val d2 = depth(t2)
        in  if d1>d2
            then 1+d1
            else 1+d2
        end;
```


The Quicksort Algorithm

```
fun qsort([])   =  []
  | qsort(h::t) =
          let val (l, r) = partition(h, t)
          in qsort(l) @ [h] @ qsort(r)
          end;
```

append

```
fun partition(pivot, [ ]) = ([ ], [ ])
  | partition(pivot, h::t) =
      let val (l, r) = partition(pivot, t)
       in  if h < pivot
           then (h::l,  r)
           else (l,  h::r)
      end;
```

unresolved overloaded op'r


```
fun partition(pivot, [ ]) = ([ ], [ ])
  | partition(pivot:int, h::t) =
      let val (l, r) = partition(pivot, t)
       in  if h < pivot
           then (h::l,  r)
           else (l,  h::r)
      end;

fun qsort([])   =  []
  | qsort(h::t) = let val (l, r) = partition(h, t)
                  in qsort(l) @ [h] @ qsort(r)
                  end;
```

must specify type

int list → int list