CSE 505

Lecture #9

October 1, 2012

Lambda Calculus

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

Key Concepts:

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

Lecture 9: 10/1/2012

CSE 505 / Jayaraman

Recursion and Fixed-points

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

$$t = f. \ n.(((if (is0 n)) \ 1) \ ((mult n) (f (pred n))))$$

Why does the fixed-point of t capture f?

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

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

Lecture 9: 10/1/2012

SE 505 / Jayarama

$$Y = f. (x.(f(x x)) x.(f(x x)))$$

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

$$(Y \ t) <==>^* (t \ (Y \ t))$$

Proof:

$$(Y t) = (f. (x.(f (x x)) x.(f (x x))) t)$$

$$==> (x.(t (x x)) x.(t (x x)))$$

$$==> (t (x.(t (x x)) x.(t (x x))))$$

$$<=> (t (Y t))$$

Least Fixed Point

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

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

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

Least fixed-point
$$g(n) = \begin{cases} 0, & \text{if } n=0 \\ 2, & \text{if } n=0 \end{cases}$$

Lecture 9: 10/1/2012

CSE 505 / Jayaraman

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).

Lecture 9: 10/1/2012

CSE 505 / Jayaraman

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, $\ensuremath{\mathsf{ML}}$ 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

Lecture 9: 10/1/2012

CSF 505 / Javaraman

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.

Lecture 9: 10/1/2012

CSE 505 / Jayaraman

A type is a set of operations! Zero: int succ: int → int +: int * int → int *: int * int → int %: int * int → int ... <: int * int → int Lecture 9: 10/1/2012

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

Lecture 9: 10/1/2012

CSE 505 / Jayaraman

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 $f(\text{string y})\{\dots\};\dots f(x)\dots$

Lecture 9: 10/1/2012

CSE 505 / Ja

Strongly Typed Language

Popular Answer:

"every variable has a type declared for it"

Above criterion not sufficient (or even necessary!):

int i, j, k; ...
$$i := j ** k$$
..

Somewhat Better Answer:

"A language is strongly typed if compiler can perform all type-checking"

Lecture 9: 10/1/2012

CSF 505 / Javarams

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.

Lecture 9: 10/1/2012

15

CSE 505 / Jayaraman

Pascal Type System

Unstructured

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

Structured

user-defined: record, array, set, file, 1

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

Lecture 9: 10/1/2012

CSE 505 / Jayaraman

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.

Lecture 9: 10/1/2012

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.

Lecture 9: 10/1/2012

CSE 505 / Jayarama

Why 'Mono' Types not Enough int length(l) { int len = 0; while (I!== nil) { len := len + 1; I := l.next; } return len; } Desired type of length is: (∀t) t List → int Lecture 9: 101/2012 19 CSE 505 / Jayaraman

```
Polymorphic Types (cont'd)

Another example:

List append(I1, I2) {

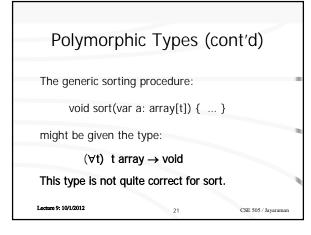
if (I1 = nil) return I2 else
return cons(I1.val,
append(I1.next, I2));
}

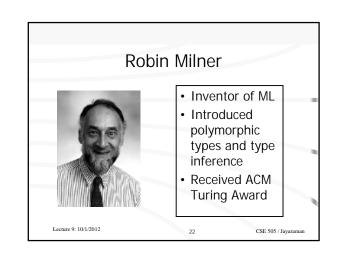
Desired type of append is:

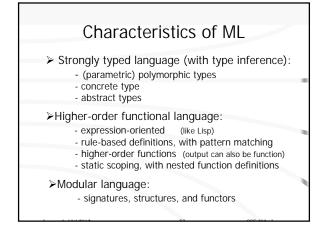
(∀t) t List * t List → t List

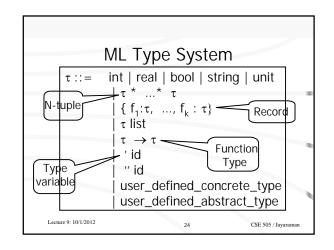
Locture 9: 10/1/2012

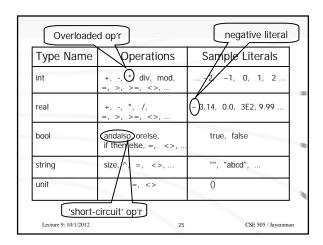
20 CSE 505 / Jayaraman
```

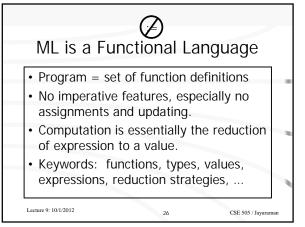


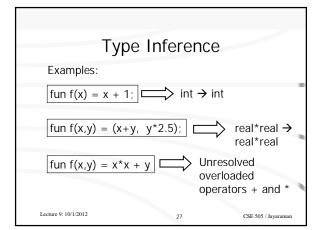


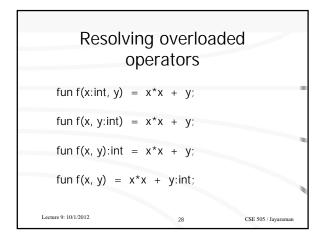


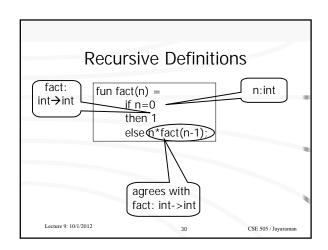


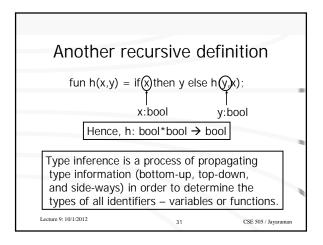


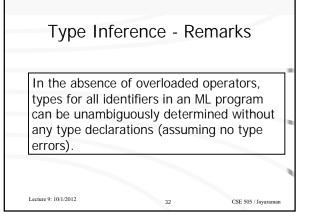








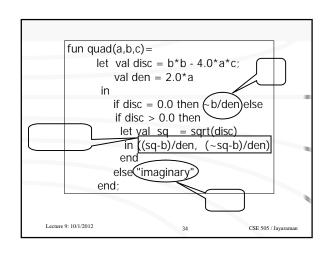




```
Local Vars & Nested Scopes

| 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 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

Lecture 9: 101/2012

37 CSE 505 / Jayaraman
```

```
Using 'quad'

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

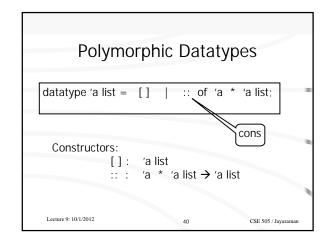
- quad(1.0, 2.0, 1.0);
val it = one(~1.0) : root

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

Lecture 9: 10/1/2012

38

CSE 505 / Jayaraman
```



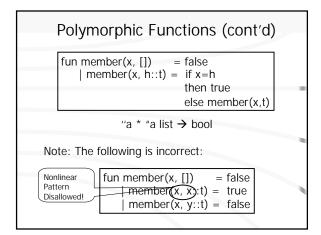
```
List Constants (literals)

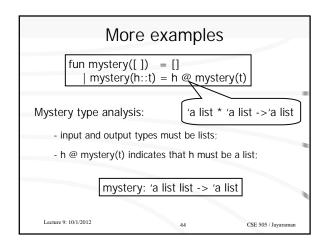
• [1, 2, 3] = 1 :: 2 :: 3 :: []
: int list
• ["abc", "def"] = "abc" :: "de" :: []
: string list
• [[1, 2],[3]] = ((1::2::[]) :: (3::[]) :: [])
: int list list

[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.





```
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(node(leaf("Java"), leaf("Prolog")))

Lecture 9: 10/1/2012
45
CSE 505 / Jayuraman
```

```
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;

Lecture 9: 101/2012

46

CSE 505 / Jayaraman
```

```
The Quicksort Algorithm
fun qsort([]) = []
 | qsort(h::t) =
             let val (I, r) = partition(h, t)
              in qsort(I) @ [h] @ qsort(r)
                                              append
   fun partition(pivot, []) = ([], [])
      | partition(pivot, h::t) =
          let val (I, r) = partition(pivot, t)
          in if h pivot
              then (h::I, T)
                                          esolved
              else (l, h::r)
                                   overloaded op'r
          end;
                          47
```

```
fun partition(pivot, []) = ([], [])
| partition(pivot:int, h::t) =
| let val (I, r) = partition(pivot, t)
| in if h < pivot
| then (h::I, r) | must
| specify
| type
| fun qsort([]) = []
| qsort(h::t) = let val (I, r) = partition(h, t)
| in qsort(I) @ [h] @ qsort(r)
| end;
| int list → int list
| Lecture 9: 10/1/2012 | 48 | CSE 505 / Jayaraman
```