CS538 Lecture 14:
ML Typechecking and Type Inference

April 1997

James R. Larus
Computer Sciences Department
University of Wisconsin–Madison
larus@cs.wisc.edu

(Based on notes by Satish Chandra)

Outline:

- Overview of core ML

Why Typechecking?

- Operators must be applied to valid arguments:
  - + operates on pair of ints (or pair of reals)
  - 3 + true is meaningless
- In assignment, **`var := expr`**, types of **`var`** and **`expr`** must match
- In procedure call, types of actual arguments must match formal parameters
  - sqrt(2.0) is OK
  - sqrt(2) causes a run-time error
- Function must produce consistent result

```
fun bad(0) = 0
  | bad(1) = 1.0;
```

Dynamic Typing, Static Typing, Strong Typing

- *Dynamic typing*: typechecking is performed at run-time
  - Values carry types
  - e.g. Lisp and Smalltalk
- *Static typing*: typechecking is performed at compile time
  - Locations are typed
  - e.g. Pascal, C, C++
- *Strong typing*: static typing and guarantee: NO type errors at run time
  - e.g. ML

Why are Types Important?

- Prevent programming errors
- Fortran has no pointers
  - Simulate pointers with integers
  - A linked list can be formed as:

```
integer NODEVAL(MAX)
integer NEXT(MAX)
... NODEVAL(NEXT(i)) = j ...
call myproc(NODEVAL, NEXT)
...
subroutine myproc(LINKS, VALUES)
end
```

  - C types prevents pitfall

```
struct node {
  int value;
  struct node *next}
```

Run-time Checking is also Important

- C tagged unions:

```
typedef struct node {
  int tag;
  union {
    double dval;
    int ival;
    char *pval;
  } uval;
} node;

node n;

switch n.tag {
case INT: print_int_val(n.uval.ival); break;
case DOUBLE: print_int_val(n.uval.ival); break;
/* trouble, but type system does not complain */
```


Run-Time Checking, cont'd

- ML prevents this type of error

```
datatype node = DVAL of real | IVAL of int | CHARVAL of
string;

case node
    of IVAL(v) => print_int(v)
    | DVAL (v) => print_int(v)
    ...
```

Type System Holes

- Typecasting in C

```
attach_mem(char **p, int nbytes)
{
        *p = (char *) malloc(nbytes);
}
char *p = NULL;
attach_mem((char **) p, 100);
```

- It is possible to write correct code in Fortran and C.
  - The problem is that these languages' type systems let some errors pass undetected at compile time

ML Type Inference

- Values

```
- [1,2,3];
val it = [1,2,3] : int list

- [(1,true), (2, false), (3, true)];
val it = [(1,true),(2,false),(3,true)] : (int * bool) list

- [(3 > 4), (4 < 1)];
val it = [false,false] : bool list

- [(3 > 4), 4];
  Error: operator and operand don't agree (tycon mismatch)
  operator domain: bool * bool list
  operand:         bool * int list
```


Copyright © 1996–97 by James R. Larus. All rights reserved.

ML Type Inference, cont'd

- Functions

```
- fun twice x = 2 * x;
val twice = fn : int -> int

- fun fact n =  if n = 0 then 1 else n * fact(n - 1);
val fact = fn : int -> int

- fun add(a,b) = a+b;
overloaded variable cannot be resolved: +

- fun add(a,b:int) = a+b;
val add = fn : int * int -> int

- fun add a b:int = a+b;
val add = fn : int -> int -> int

- add 3;
val it = fn : int -> int

- it(4);
val it = 7 : int
```


Polymorphic Functions

- Functions that operate on more than one type of object

```
- fun isnil L = if L = [] then true else false;
val isnil = fn : 'a list -> bool
```

  - 'a (read: quote a) is a type variable
  - Type variable in function type indicates that function is polymorphic
  - ML infers that type of L ('a) does not affect operation of function
  - **`isnil`** never looks at values in list
  - **`isnil([2,3]), isnil([true, false]), isnil([])`**

Copyright © 1996–97 by James R. Larus. All rights reserved.

Polymorphic Functions, cont'd

- More examples

```
- fun id x = x;
val id = fn : 'a -> 'a

- fun append(nil, l) = l
  |   append(fst::rst, l) = fst::append(rst,l);
val append = fn : 'a list * 'a list -> 'a list
```

- Why not `'a list * ' b list -> 'c list?`

```
- fun eithernil(l1, l2) =
    if isnil(l1) orelse isnil(l2) then true else false;
val eithernil = fn : 'a list * 'b list -> bool

- fun f a b = a::[b];
val f = fn : 'a -> 'a -> 'a list

- map isnil [[1,2], [3,4], []];
val it = [false,false,true] : bool list

- map;
val it = fn : ('a -> 'b) -> 'a list -> 'b list
```


Polymorphic Functions, cont'd

```
- fun pow [] = [[]]
  |   pow(h::t) =  let val pset = pow t
                   in append((map (cons h) pset), pset)
                  end
val pow = fn : 'a list -> 'a list list

- pow([1,2]);
val it = [[1,2],[1],[2],[]] : int list list

- fun comp(f,g)(x) = f(g(x));
val comp = fn : ('a -> 'b) * ('c -> 'a) -> 'c -> 'b

- fun comp f g x = f(g(x));
val comp = fn : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b
```

ML Type Inference

- Milner's type inference

  1. Know types of literals and predefined functions and operators, such as **`hd`**, **`tl`**, **`::`**
  2. All instances of same id (under scoping rules) have same type
  3. In expression $\mathbf{e_1 = e_2}$, typeof($\mathbf{e_1}$) = typeof($\mathbf{e_2}$)
  4. In conditional expression **`if`** $\mathbf{e_1}$ **`then`** $\mathbf{e_2}$ **`else`** $\mathbf{e_3}$
     - typeof($\mathbf{e_1}$) = **`bool`**
     - typeof($\mathbf{e_2}$) = typeof($\mathbf{e_3}$), which is also type of whole expression
  5. In call **`(f x)`**, if **`f`** has type **`'a->'b`**
     - typeof(**`x`**) = **`'a`** and type of whole expression is **`'b`**
  6. In tuple $\mathbf{(e_1, e_2, \ldots e_n)}$, if typeof($\mathbf{e_1}$) = $\mathbf{'a_1}$, typeof($\mathbf{e_2}$) = $\mathbf{'a_2}$, and so on
     - type of whole tuple is $\mathbf{('a_1 * 'a_2 * \ldots 'a_n)}$

Inference, cont'd

  7. In list $\mathbf{[v_1, v_2, \ldots v_n]}$, if typeof($\mathbf{v_1}$) = $\mathbf{'a_1}$, typeof($\mathbf{v_2}$) = $\mathbf{'a_2}$, and so on,
     - $\mathbf{'a_1}$ = $\mathbf{'a_2}$ ... and type of list is $\mathbf{'a_1}$ **`list`**
  8. In function definition **`fun f x = expr`**, if typeof(**`x`**) = **`'a`** and typeof(**`expr`**) = **`'b`**
     - typeof(**`f`**) is **`'a->'b`**

- Trivia: These rules are not an algorithm per se
  - They form a formal system (a.k.a. proof system)
  - The type checking algorithm (called algorithm-W) is essentially a theorem prover

Type Inference Example

```
- fun f(p,q) = if p then 1 else q;
```

- Rule 8. Assume **f** is **'a->'b**
- Rule 6. **'a** is some **('a1 * 'a2)**
  - therefore typeof(**p**) = **'a1**
  - typeof(**q**) = **'a2**
- Rule 4. **p** should be bool
  - therefore **'a1** = **bool**
- Rule 4. typeof(**1**) should equal typeof(**q**)
  - therefore **'a2** = **int**
  - also, 'b = int
- Rule 8. f is **(bool * int) -> int**

Examples, cont'd

```
- fun len l = if null(l) then 0 else succ(len (tl l));
```

- Know:
  - **tl: 'a list -> 'a list**
  - **null: 'a list -> bool**
  - **succ: int->int**
- What if function had "**then 1**" instead?
  - Milner's algorithm tries to find Most General Type (MGT)
  - MGT is a type such that any other "correct" type for an expression can be obtained by substituting a type into a type variable
  - **int list -> int** is a correct type for **len**, but **'a list -> int** is more general, because substituting the type variable **'a** with **int** obtains former type

Perform Type Inference

- Perform a tree walk on parse tree, keeping set of assumptions about type of variables encountered
  - Crucial part is enforcing typeof(e1) = typeof(e2)
  - Current assumptions about types of e1 and e2 must be reconciled (technically called *Unify*)
  - Unification often refines assumptions about type variables

Unify	With	Substitutes
`bool`	`'a`	`'a:bool`
`int * bool`	`'a * bool`	`'a:int`
`'a->('b*int)`	`int->(bool*'a)`	`'a:int,'b:bool`
`'a->('b*bool)`	`int->'a`	`FAIL`
`'a->('b*int)`	`'b->'a`	`FAIL`
`'a`	`'b`	`'a:'b or 'b:'a`

A Program Fails to Typecheck

- Unification fails at some point
- Encounter a variable not found in the type environment

More Subtle Points

- Why does the following not typecheck?

```
- fun f x = x x ;
```

- assume f: 'a->'b, x:'a, also x:'c->'d
- also 'a='c
- also 'c='c->'d ARRRGH!!

Modified: April 7, 1997

More, cont'd

- Should the following typecheck?

```
- let fun f x = x in ((f 3), (f "a")) end;
val it = (3,"a") : int * string

- fun f g = ((g 3), (g "a"));
  operator domain: int
  operand:         string
  in expression:
    g "a"
```

- g could be bound to **succ**, so this one should not typecheck
  - Our intuitive typechecking procedure fails on the let example, because at (f 3) we will conclude that type of **f** is **int->int**
  - However, with some modifications, our typechecking procedure could typecheck the let correctly, but (correctly) fail on the second example

Modified: April 7, 1997