Typing and ML

CSC324
Fall 2004
Sheila McIlraith

Acknowledgement:

The material in these notes is derived from a variety of sources, including:

Elements of ML Programming (Ullman),

Concepts in Programming Languages (Mitchell)

and the notes of Wael Aboelsaddat, Tony Bonner, Eric Joanis, Gerald Penn, and Suzanne Stevenson.

Typing

A name for a set of values and some operations which can be performed on that set of values.

A collection of computational entities that share some common property.

E.g.,

reals

integers

strings

int $\rightarrow$ bool

(int $\rightarrow$ int) $\rightarrow$ bool

What constitutes a type is language dependent.

Uses/Merits

Program organization and documentation

- Separate types for separate concepts
- Indicate intended use of declared identifiers

Identify and prevent errors

- Compile-time or run-time checking can prevent meaningless computation such as

  5 + true - Charlotte

Support optimization

- Compiler can generate better code if it knows what s in each variable, e.g., short integers require fewer bits.
- Access record component by known offset

Type errors

Definition

A type error occurs when execution of program is not faithful to the intended semantics, i.e., the programmer's intended interpretation.

Hardware errors

function call y() where y is not a function

may cause jump to instruction that does not contain a legal op code

Unintended semantics

int_add(3, 4.5)

not a hardware error but the bits representing 4.5 will be interpreted as an integer

Type Safety & Type Checking

A programming language is *type* safe if no program is allowed to violate its type distinctions.

Scheme, ML and Java are type safe.

C and C++ are not.

The process of verifying and enforcing the constraints of types is called *type checking*.

Type checking can either occur at compile-time (static) or at run-time (dynamic).

Compile- vs. Run-time

Scheme: run-time (dynamic) type checking

(car x) checks first to make sure x is a list

ML and Java: compile-time (static) type checking

f(x) must have f: A $\rightarrow$ B and x:A

Trade-off:

Both prevent type errors

Run-time checking slows down execution

Compile-time checking restricts program flexibility

E.g., Scheme list elements can have diff. types,

ML lists elements must have the same type

Static typing can make programming more difficult, initially. It s harder t get things to compile, and

Type Checking- vs. Inference

Standard Type Checking:

```
int  f(int x) { return x+1;};
int  g(int y)  {return f(y+1)*2;};
```

Look at body of each function and use declared types to check for agreement.

Type Inference:

Looks at code without type info and figures out what types could have been declared.

ML is designed to make type inference tractable.

A cool algorithm!

Widely regarded as an important language innovation.

ML type inference gives you some idea of how other static analysis algorithms might work. It uses constraint satisfaction techniques.

Type Inference

This is type inference:

E.g. A3 := B4 + 1;

Q: What type is A3 and B4 ?

A: Must be integer

E.g. if test then

Q: What type is test ?

A: Must be Boolean

Sound type system: a type system in which all types can always be inferred in any valid program.

ML s Type Inference Algorithm (Mitchell):

1. Assign a type to the expression and each subexpression by using the known type of a symbol of a type variable.
2. Generate a set of constraints on types by using the parse tree of the expression.
3. Solve these constraints by using unification, which is a substitution-based algorithm for solving systems of equations.

ML

Developed at Edinburgh (early 80s) as Meta-Language for a program verification system

- Now a general purpose language
- There are two basic dialects of ML
  - Standard ML (1991) & ML 2000
  - Caml (including Objective Caml, or OCaml)

A pure functional language

- Based on *typed lambda calculus*
- Grew out of frustration with Lisp!
- Major programs can be written w/o variables

Widely accepted

- reasonable performance (claimed)
- can be compiled
- syntax not as arcane as LISP (nor as simple)

ML: Main Features

Functional Language
- HOFs, recursion strongly encouraged, etc.

Combination of Lisp and Algol features

Strong, static typing w/ type inference
- Quite a fancy type system!

Polymorphism
- a function can take arguments of various types

Abstract & recursive data types
- supported through an elegant type system,
- the ability to construct new types, and
- constructs that restrict access to objects of a given type through a fixed set of ops defined for that type.

Pattern matching
- Function as a template

Exception handling
- Allow you to handle errors/exception

Elaborate module system
- Most highly developed of any language

ML: Tutorial Review

SML environment basics

Each ML expression has a type associated w/ it.
- Interpreter builds the type expression
- Cannot mix types in expressions
- Must explicitly coerce/type-case
- e.g. real(2) + 3.0 : real

Data types (w/ operators):
- Basic: unit, bool, integer, real, string
- Constructors : list, tuple, array, record, function
- operators infix, can be overloaded.

Read-eval-print
- Compiler infers type before compiling & executing.

E.g.,

```
- (5+3)-2;
> val it = 6 : int
- If 5>3 then    Bobelse  Carol   ;
>val it= Bob  : string
- 5-4;
> val it=false : bool
```

Assignment

val <constant-name> = <expression>;

Patterns & Declarations

Patterns can be used in place of variables

<pat> ::= <id>|<tuple>|<cons>|<record>|

Value declaration (general form):

val <pat> = <exp>

E.g.,

- val myTuple = (Jen , Brad);

*val myTuple = (Jen","brad") : string * string*

- val(x,y) = myTuple;

Return value?:

- val myList = [1,2,3,4];

Return value?:

- val x::rest = myList;

Return value?:

Local declarations:

- let val x = 2+3 in x*4 end;

val it = 20 : int

Declarations

ML has let too!

Local declarations:

```
- let val x = 2+3 in x*4 end;
val it = 20 : int

- let
    val m=3            (* ; is optional *)
    val n=m*m
in
   m+n
end;
Return value?:
```

Pattern Matching

Pattern matching is powerful:

- Allows programmers to see the arguments
- No more heads and tails (cars/cdrs)

Tupple pattern matching

-val v=((2, "Test"),(3.2,#"A"));
Return value?

-val ((i,s),(r,c))=v;
val i = 2 : int
val s = "Test" : string
val r = 3.2 : real
val c = #"A" : char

-val (p1,p2)=v;val p1 = (2,"Test") : int * string
*val p2 = (3.2,#"A") : real * char*

-val (_,(r,_))=v; (*_ (don t care) matches anything!*)
val r = 3.2 : real

Pattern Matching

Record pattern matching

```
-type stInfo={name:string, id:int, gpa:real};
type stInfo = {gpa:real, id:int, name:string}

-val st1:stInfo={name=    jen", id=123, gpa=4.0};
val st1 = {gpa=4.0,id=123,name="jen"} : stInfo

-val {name=N, gpa=G, id=_}=st1;   (* order doesn't matter! *)
val G = 4.0 : real
val N =  jen" : string

-val {gpa,id, name}=st1;      (* this is an abbreviation in ML *)
val gpa = 4.0 : real
val id = 123 : int
val name =  jen" : string

-val {name,...}=st1l;    (* to specify subset of fields *)
val name =  jen" : string
```

Functions

Like Scheme there are:

- Defined functions
- Anonymous functions
- Recursive functions
- Higher-order functions
- And you can pass functions as parameters, and return them as values.

Unlike Scheme,

we call these things functionsnot procedures

---

f: A $\rightarrow$ B means

for every x $\in$ A,

f(x) = some element y=f(x) $\in$ B
f(x) = run forever
f(x) = terminate by raising an exception

A function maps a type to another one: accepts only one argument.

What if we need multiple arguments?

Function Declarations

Function Declaration

Single clause definition

fun <fname> (<pat>) =<exp>;

Function arguments (patterns) don t always need parentheses, but it doesn t hurt to use them

Examples:

- fun fahrToCelsius t = (t -32) * 5 div 9;

val fahrToCelsius = fn : int -> int

- fun foo L = (1 + hd L) :: (tl L);

Return value:?

- fun quotrem (x,y) = ((x div y), (x mod y));

Return value?:

Function Declarations

Multiple-clause definition

```
fun <fname> (<pat1>) = <exp1>
  | <fname> (<pat2>) = <exp2>
  |
  | <fname> (<patn>) = <expn>
```

Lazy: The first pattern that matches the actual parameter will be chosen.

Examples:

```
-fun sum (x,y)= x+y;
val sum = fn: int*int -> int

-sum (2,3);
val it = 5 : int

-fun len (nil) = 0          (*nil or [ ]  Also we can drop ()*)
  | len (h::rest) = 1+len(rest); (* () is necessary!*)
Result returned?:

-len ([5]);
val it = 1: int
-len ["Alice", "John"];
val it = 2: int
```

Function Declarations

Watch out!

```
- val z=4;
val z = 4 : int

-fun sumz (x,y)= x+y+z;
val sumz = fn: int*int -> int

-sumz (2,3);
val it = 9 : int

- val z=7;
val z = 7 : int

-sumz (2,3);
val it = 9 : int
```

No variable can occur twice in a pattern

```
- fun eq(x,x)=true
    | eq(x,y)=false;
Error: duplicate variable in pattern(s)
```

If the pattern doesn't exhaust all possible values, we get a warning.

Function Declarations

Example:

```
- fun listsum L = if (null L) then 0
                  else (hd L) + listsum (tl L);
val listsum = fn : int list -> int

- listsum [1,2,3];
val it = 6 : int
```

Better:

```
- fun listsum [] = 0
      | listsum L = (hd L) + listsum (tl L);
```

Best

```
- fun listsum [] = 0
  | listsum (h::t) = h + listsum t;
```

Anonymous Functions

fn <pat> => <expr>
This is just like a Scheme lambda expression
(lambda (<pat>) (exp))

Examples:

```
-(fn(x,y)=> x+y) (2,3);
val it = 5 : int

-val mysum=  fn (x,y)=> x+y;
val mysum = fn : int * int -> int

-mysum(2,3)
val it = 5 : int
```

The following declarations are identical:

```
- fun f(n) = 2*n;
val f = fn : int -> int

- val f = fn n => 2*n;
val f = fn : int -> int
```

Anonymous Functions

What is this doing?

```
- fun foo (m, n) =
        if m > n then [ ] else m :: foo(m+1, n);
```

Result returned?:

```
- foo(1,6);
```

Recursive Functions

Examples:

```
- fun append(nil, ys) = ys
    | append(x::xs,ys) = x :: append(xs,ys);
val append = fn : 'a list * 'a list -> 'a list
```

```
- fun reverse nil = nil
    | reverse(x::xs) = append((reverse xs),[x]);
val reverse = fn : 'a list -> 'a list
```

There is a more efficient reverse .we ll see this later.

Mutual Recursion

The following is wrong:

```
fun even 0 = true
    | even x = odd (x-1);  (*wrong: odd not defined*)
```

The following is correct, using mutual recursion:

```
fun even 0 = true
  | even x = odd (x-1)
and odd 0 = false
  | odd x = even (x-1);
```

Important Issues

1. Function application is left-associative. Use () if nec .

   abs square x + y * abs z *means*

   (abs square) x + (y * (abs z)

   Our error: operator and operand don't agree

2. The combination of tuples, functions, infix ops, type constructors can be syntactically tricky when defining/calling functions!

Eg. length 2::[1,3] *is wrong: it means*

(length 2) :: [1,3].

Correct formulation?:

Eg.

```
fun f1 nil=0 |
      f1 h::t= 1+f1 t;
```

Error: infix operator "::" used without "op" in fun dec

Error: clauses don't all have same no. of patterns

Correct formulations?:

Important Issues (cont.)

The syntax becomes more complex when considering the following short notation:

In ML

```
fun sum x y=x+y;
```

is short for

```
fun sum x= (fn y=>x+y);
```

So, its type is:

```
fn : int -> (int -> int)
```

Similarly,

```
fun sum3 x y z = x+y+z
```

is short for:

```
fun sum3 x =
        (fn y =>
          (fn z => x+y+z));
```

So it s type is:

```
fn : int -> int -> int -> int
```