Typing and ML

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 → bool
- (int → int) → 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 → 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 to get things to compile, and

Type Checking vs. Type 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.
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
  – 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)
• syntax not as arcane as LISP (nor as simple…)

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: Main Features

ML: Tutorial Review
(see your tutorial notes for complete details)

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 "Bob" else "Carol"
  >val it="Bob" : string

Patterns & Declarations

Patterns can be used in place of variables

Value declaration (general form): 
val <pat> = <exp>

E.g. (Declarations),
  - val myTuple = ("Jen", "Brad")
  - val x = "Jen" : string
  - val y = "Brad" : string
  - val myList = [1,2,3,4] : int list
  - val x :rest = myList;
  val x = 1 : int
  val rest = [2,3,4] : int list

Declarations

ML has let too!

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

- let
  val m=3
  val n=m*m
in
  m+n
end;
val it = 12 : int

Pattern Matching

Pattern matching is powerful:
• Allows programmers to see the arguments
• No more heads and tails (cars/cdr’s)

Tupple pattern matching
- val v=(2,"Test"),(3.2,#"A")
val v = (2,"Test"),(3.2,#"A") : (int * string) * (real * char)

- val ((i),(r))=v;
val i = 2 : int
val r = "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

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=*, 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,..}=st1;
  (* 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 “functions” not “procedures”

f : A → B means
for every x ∈ A, some element y=f(x) ∈ B
f(x) = run forever
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 Declarations

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);
  val foo = fn : int list -> int list
- fun quotrem (x,y) = (x div y), (x mod y);
  val quotrem = fn : int * int

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!*)
  val len= fn: 'a list -> int
  len ([5]);
  val it = 1 : int
  len ["Alice", "John"];
  val it = 2: int

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 = 2: 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);
  val foo = fn : int * int -> int list
- 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....
Important Issues

1. Function application is left-associative.
   Use brackets as necessary.
   \[ \text{abs square} \times y \text{ * (abs } z) \]
   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!
   \[ \text{Eg. length } 2::[1,3] \] is wrong: it means \((\text{length } 2) :: [1,3]\).
   Correct formulation?

Syntax becomes more complex when considering the following short notation:

- In ML:
  \[ \text{fun sum x y=} (\text{fn } y \Rightarrow x+y); \]

  So, its type is:
  \[ \text{fn : int -+ (int -+ int)} \]

- Similarly,
  \[ \text{fun sum3 x y z=} x+y+z \]

  So its type is:
  \[ \text{fn : int -+ int -+ int -+ int} \]

Exception Handling

Recall that many functions are partial – they are only defined for a subset of the function's domain type. Other values in the domain type may be treated as exceptions. (E.g., division by zero)

Exceptions are a control construct. They provide a structured form of jump to exit a construct such as a function invocation or a block.

Terminate part of computation:
- Jump out of construct
- Pass data as part of jump
- Return to most recent site set up to handle exception
- Unnecessary activation records may be deallocated
  (May need to free heap space, other resources)

Often used for unusual or exceptional condition, but not necessarily

ML Example

```
exception Determinant; (* declare exception name *)
fun invert (M) =     (* function to invert matrix *)
  ...
  if ...
  then raise Determinant (* exit if Det=0 *)
  else ...
end;
...
invert (myMatrix) handle Determinant => ...
```

Value for expression if determinant of myMatrix is 0

Built-in Exceptions

Classical example: division by 0.
- 5 div 3; (*) type of function div is int *)
- 5 div 0; (*) an exception is raised *)
uncaught exception divide by zero raised at: <file stdIn>

Other examples of built-in exceptions:
- hd([3,5,9]):
- val it = 3 : int
- uncaught exception Empty raised at: boot/list.sml:36.38-36.43

These both raise exceptions and stop, but exceptions can also be handled, a value assigned and computation continued.

User-Defined Exceptions

1. Declare an Exception to establish exception handler
   \[ \text{exception } \text{exception-nm} \text{ of } \text{type-expression} \]
   gives name of exception and optionally type of data passed when raised

2. Raise Exception
   \[ \text{raise } \text{exception-nm} \text{ (parameters)} \]
   raises an exception and passes data to handler

Example
- exception NegArg of int; (*) declare an exception *)
  exception NegArg of int
- fun fact N = if N=0 then 1
  else if N>0 then N*fact(N-1)
  else raise NegArg(N); (* raise excptn *)
- fact(5):
  val it = 120 : int
- fact(-5):
  uncaught exception NegArg raised at: stdIn:20.30-20.39

User-Defined Except’ns (cont.)

- 3. Exception Handling
  \[ <expression> handle <exception1> \Rightarrow <expression1> \]
  \[ \mid <exception2> \Rightarrow <expression2> \]
  \[ \mid <exceptionn> \Rightarrow <expressionn> \]
- If no exceptions are raised then return the value of <expression>
- If <exception> is raised, then return the value of <expression>

(See more general form + explanation in Mitchell)
Exception Handling

Example: \( \frac{N!}{(M! \cdot (N-M)!)} \)
- fun comb (N,M) =
  = if N < 0 then raise Negative(N) =
  = else if M < 0 then raise Negative(M) =
  = else if M > N then raise TooBig(M) =
  = else =
  = fact(N) div (fact(M) * fact(N-M));
val comb = fn : int * int -> int
- fun mycomb (N,M) =
  = comb (N,M)
  = handle Negative(X) => ~1
  = | TooBig(M) => 0;
val mycomb = fn : int * int -> int
- fun mycomb (12, 5); val it = 792 : int
- fun mycomb (~12, 5); val it = ~1 : int
- fun mycomb (5, 12); val it = 0 : int

Exceptions & Scoping

Exceptions are handled according to dynamic scoping. Otherwise ML uses static scoping. (More on this in a later unit of the course, but here is an illustrative example)

Example:
- fun h(1) =
  | h(2) = raise e2 =
  | h(3) = raise e3
  | h(_) = "ok";
- fun g(N) =
  = h(N)
  = handle e2 => "error g2"
  | e3 => "error g3";
- fun f(N) =
  = g(N)
  = handle e1 => "error f1"
  | e2 => "error f2";

Dynamic Scoping of Handlers

- exception e1 and e3 and e3;
- fun h(1) = raise e1
  | h(2) = raise e2
  | h(3) = raise e3
  | h(_) = "ok";
- fun g(N) =
  = h(N)
  = handle e2 => "error g2"
  | e3 => "error g3";
- fun f(N) =
  = g(N)
  = handle e1 => "error f1"
  | e2 => "error f2";

Typing of Exceptions

Typing of raise (exception)
- Recall definition of typing
  - Expression e has type t if normal termination of e produces value of type t
  - Raising exception is not normal termination
- Examples:
  - 1 + raise X

Typing of handle (exception) => (value)
- Converts exception to normal termination
- Need type agreement
- Examples:
  - 1 + ((raise X) handle X => e)
    *** Type of e must be int
  - 1 + (e1, handle X => e2)
    *** Type of e1, e2 must be int

ML