Standard ML Mini-tutorial

(in particular SML/NJ)

Programming Languages CS442

David Toman

School of Computer Science
University of Waterloo

Introduction

- SML (Standard Meta Language)
  - ⇒ originally part of the LCF project (Gordon et al.)
- Industrial strength PL (SML'90, SML'97)
  - ⇒ based *formal semantics* (Milner et al.)
- SML "Basis Library" (all you ever wanted)
  - ⇒ based on *advanced* module system
- Quality compilers:
  - ⇒ SML/NJ (Bell Labs)
  - ⇒ Moscow ML

Features

- Everything is built from expressions
  - ⇒ functions are first class citizens
  - ⇒ pretty much extension of our simple functional PL
- Support for structured values: lists, trees, . . .
- Strong type system
  - ⇒ let-polymorphic functions
  - ⇒ type inference
- Powerful module system
  - ⇒ signatures, implementations, ADTs,. . .
- Imperative features (e.g., I/O)

Tutorial Goals

1. Make link from our functional language to SML
2. Provide enough SML syntax and examples for A2
   - How to use SML/NJ interactive environment
   - How to write simple functional programs
   - How to define new data types
   - How to understand compiler errors
   - Where to find more information
3. Show type inference in action (so we understand what's coming)

Getting started

- Starting it up: `sml` in UNIX (click somewhere in W/XP)

Example

```
Standard ML of New Jersey, Version 110.0.7 [CM&CM]
-
```


⇒ great support in Emacs

- Notation and simple examples

Example

```
- 1;
val it = 1 : int
- 2+3;
val it = 5 : int
-
```


⇒ I type in `blue`, SML replies in `black`

Simple Declarations

- We can create declarations (bindings):

Example

```
- val x = 2*3+4;
val x = 10 : int
```

$\Rightarrow$ now `x` stands for `10`

- and use them:

Example

```
- val y = x*2;
val y = 20 : int
```

$\Rightarrow$ analogue of an *environment* $\{x = 10, y = 20\}$

Types of Simple Things

- there is more than integers:

Example

```
- 1.0;
val it = 1.0 : real
- "abc";
val it = "abc" : string
- #"a";
val it = #"a" : char
```

- and these types come with additional operations

Example

```
- "abc"^"def";
val it = "abcdef" : string
```

Functions

- λ-abstractions:

Example

```
- fn x => x+1;
val it = fn : int -> int
```

- functions can be "declared" and "used"

Example

```
- val twice = (fn x => 2*x);
val twice = fn : int -> int
- twice y;
val it = 40 : int
```

$\Rightarrow$ what if we wanted a recursive function?

Functions

- there is a `rec` construction (which almost nobody uses)
- functions are defined "explicitly" using a `fun` declaration:

Example

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


- but more commonly using *match patterns*:

Example

```
- fun fac 0 = 1
=   | fac n = n*(fac (n-1));
val fac = fn : int -> int
- fac 10;
val it = 3628800 : int
```


$\Rightarrow$ match patterns better cover all possible parameter values!

Complex Types: Tuples

- Pairs and *k*-tuples:

Example

```
- val pair = (1,"abc");
val pair = (1,"abc") : int * string
- val triple = (1,true,1.0);
val triple = (1,true,1.0) : int * bool * real
```

- and projections:

Example

```
- #3(triple);
val it = 1.0 : real
- val (x,y) = pair;
val x = 1 : int
val y = "abc" : string
```

Complex Types: Lists

- List construction

Example

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

- and operations:

Example

```
- hd l;
val it = 1 : int
- tl l;
val it = [2,3] : int list
- tl(tl(tl l));
val it = [] : int list
```

Functions on Lists

- Function that appends two (arbitrary) lists:

Example

```
- fun app nil l = l
=   | app (h::t) l = h::(app t l);
val app = fn : 'a list -> 'a list -> 'a list
```

⇒ what are the 'a types? polymorphic type variables

- And what does it do:

Example

```
- app [1,2,3] [4,5,6];
val it = [1,2,3,4,5,6] : int list
- app ["a","b"] ["c"];
val it = ["a","b","c"] : string list
```

⇒ the arguments must be lists of the same type

Polymorphic Functions

- polymorphic = “universal” functions (for all types)

Example

```
- fun mklist x = [x];
val mklist = fn : 'a -> 'a list
- mklist 1;
val it = [1] : int list
- mklist (mklist 1);
val it = [[1]] : int list list
- fn x=> mklist (mklist x);
val it = fn : 'a -> 'a list list
- it "a";
val it = [["a"]] : string list list
```

Higher-order Functions

- functions as parameters? the `map` function:

Example

```
- fun map f [] = []
=   | map f (h::t) = (f h)::(map f t);
val map = fn : ('a -> 'b) -> 'a list -> 'b list
```

- what does it do?

Example

```
- map (fn x=> x+1) [1,2,3];
val it = [2,3,4] : int list
- map (fn x=> [x]) [1,2,3];
val it = [[1],[2],[3]] : int list list
- fn l=>map (fn x=> [x]) l;
val it = fn : 'a list -> 'a list list
```

Datatypes

- what if we need more than pairs and lists
- SML provides datatypes (disjoint unions)

Example (Binary trees)

```
- datatype 'a bintr = LEAF of 'a
=                  | NODE of 'a bintr*'a bintr;
datatype 'a bintr = LEAF of 'a
                  | NODE of 'a bintr * 'a bintr
```


$\Rightarrow$ this works for any number of variants

- creating a new tree:

Example

```
- val tree = NODE (NODE(LEAF 1,LEAF 4),LEAF 7);
val tree = NODE(NODE(LEAF 1,LEAF 4),LEAF 7) : int bint
```

Datatypes (cont.)

- functions on trees: use pattern matching again

Example

```
- fun addl (LEAF n) = n
=   | addl (NODE(n1,n2)) = (addl n1)+(addl n2);
val addl = fn : int bintr -> int
- addl tree;
val it = 12 : int
```

- we can do better (a polymorphic function):

Example

```
- fun mapt f g (LEAF l) = (g l)
=   | mapt f g (NODE(n1,n2)) =
=               f (mapt f g n1) (mapt f g n2);
val mapt = fn : ('a -> 'a -> 'a) ->
                ('b -> 'a) -> 'b bintr -> 'a
```

Local Declarations

- local declarations `let <decl> in <exp> end`

Example

```
- fun addl (LEAF i) = i
=   | addl (NODE(n1,n2)) =
=       let val a1 = (addl n1)
=           val a2 = (addl n2)
=       in
=           a1+a2
=       end;
val addl = fn : int tree -> int
```


- local (helper) function declarations:

```
local <helper-fun-decl> in <main-fun-decl> end
```

Exceptions

- what does `hd nil` do? `1 div 0`?

Example

```
- 1 div 0;
uncaught exception divide by zero
  raised at: <file stdIn>
```

- we can have our own exceptions:

Example

```
- exception myex of int;
exception myex of int
- fun cf n = if (n<0) then raise (myex ~1)
=                     else (fac n);
val cf = fn : int -> int
- cf ~1 handle (myex n) => n;
val it = ~1 : int
```

Modules

- Structures (essentially named declarations)

```
structure IntLT = struct
  type t = int  val lt = (op <)  val eq = (op =)
end
```

$\Rightarrow$ access to components: `IntLT.lt`

- Signatures (essentially types of declarations)

```
signature ORDERED = sig
  type t
  val lt : t * t -> bool  val eq : t * t -> bool
end
```

- Ascription (match of signature and structure)

  $\Rightarrow$ `structure strid : sigexp = strexp` (transparent)
  $\Rightarrow$ `structure strid :> sigexp = strexp` (opaque)

- Parametrized module: `functors`

Compiler Error Messages

- incorrect base syntax:

```
- let x=1 in x end;
stdIn:4.1-4.7 Error: syntax error: deleting LET ID EQUA
stdIn:4.9 Error: syntax error found at IN
```

- undeclared identifiers:

```
- foo;
stdIn:4.1 Error: unbound variable or constructor: foo
```

- type problems:

```
- [1,"foo"];
stdIn:4.1-4.10 Error: operator and operand don't agree
  operator domain: int * int list
  operand:         int * string list
  in expression:
    1 :: "foo" :: nil
```

Summary and Quick Hints

- This should get you started with SML (go and try)
- Several helpful hints:
  1. reading program text from file:
     ```
     use "file.sml";
     ```
  2. print a string on “stdout”:
     ```
     print "string-here\n";
     ```
  3. fix-up defaults for printing:
     ```
     Compiler.Control.Print.printDepth := 50;
     Compiler.Control.Print.printLength:= 1000;
     Compiler.Control.Print.stringDepth:= 200;
     ```
  4. these “`GC #6.42.43.47.144.8522: (0 ms)`” are harmless
     $\Rightarrow$ unless they’re coming and coming (infinite loop)
  5. more help: `http://www.smlnj.org//index.html`
     more complete tutorial:
     `http://www.cs.cmu.edu/People/rwh/introsml/`