Introduction to SML

15-814 Fall 2003

Aleksey Kliger

What is SML?

- Mostly-pure safe strongly-typed functional programming language
- Suitable both for programming in the small and programming in the large

Programming in the small

SML in the small

Core-SML: a program is a series of declarations and an expression to be evaluated

- The declarations provide definitions of types, and functions — “the environment”
- The expression is evaluated to get the answer
- Comments are written `(* like this *)`

Expressions

- Every expression in SML has a type.
  - But never have to write it down. SML will *infer* the type for you.
  - If you're get type-errors, sometimes helps to write out types of expressions.
- The most basic expressions are values
- All other expressions are evaluated down to values
  - Some expressions have side-effects (*e.g.,* print), but I won't talk about those today.

Values: basic types

Type	Values	Comments
unit	()	A trivial type with a single value
bool	`true`,`false`	
int	0, 1, 42, ~ 1	integers
real	0.0, 1.2, ~ 2.45	won't be used much in this course
char	`#"a"`, `#newline`	characters
string	"foo", "", "bar baz"	*not* the same as a list of characters

Expressions: basic types

Expression	Types	Result type	Comments
`if` b `then` e_1 `else` e_2	b of type `bool` e_1, e_2 of the same type	same as e_i	
e_1 `andalso` e_2 e_1 `orelse` e_2	e_1, e_2 : `bool`	`bool`	"`&&`", "`||`"
$e_1 + e_2$, $e_1 - e_2$, $e_1 * e_2$	e_1, e_2 either both `int` or both `real`	same as e_i	overloaded operators
$e_1 < e_2$	e_1, e_2 either both `int` or both `real`	`bool`	also overloaded
$e_1 = e_2$	e_1, e_2 : same equality type	`bool`	
e_1 / e_2	e_1, e_2 : `real`	`real`	real division
e_1 `div` e_2	e_1, e_2 : `int`	`int`	integer division
e_1 `^` e_2	e_1, e_2 : `string`	string	concatenation

For now, "equality types" are just `unit`, `int`, `char`, `string`, but not `real`.

- A wart in SML. As we'll see, you can avoid = most of the time

More types: tuples, lists, option

Type	Values
int * bool	(3,true)
int * string * string	(42, "foo", "J.Random Hacker")
int list	nil, 2::3::nil
(int * bool) list	[], [(3,true), (~ 2,false)]
(string * bool) option	NONE, SOME ("foo", 23)
int option list	[NONE, NONE, SOME 1, SOME 3, NONE]

- Tuples are of fixed but arbitrary arity
- Lists are homogeneous
- List and option are polymorphic: more on that later

More types: defining your own

Two mechanisms:

- Type abbreviations:
  - `type age = int`
  - `type person = string * age`
    - `("J. Random Hacker", 12)`
- New datatypes:
  - ```
    datatype employee =
      Grunt of person
    | Manager of person * employee list
    ```
    - `Grunt ("J. Random Hacker", 12)`
    - ```
      Manager (("PHB", 51),
                 [Grunt ("J.Random Hacker", 12)])
      ```
  - The datatype declares several *constructor* names that must be unique to the datatype
  - May be recursive (*e.g.*, `employee` above)

Datatypes: two built-in ones

Turns out that `list` and `option` are standard datatypes

- `datatype 'a option = NONE | SOME of 'a`
  - `'a` is a *type variable*: stands for any type
    - `SOME 42 : int option`
    - `SOME ("J. Random Hacker, 12) : person option`
    - `NONE : 'a option`
- ```
  datatype 'a list = nil
                   | :: of 'a * 'a list
  ```

  plus a *fixity* declaration to make `::` be right-associative
  ```

Analyzing values: Patterns

In SML, analysis of values is done using *pattern matching*:

- Informally speaking, a pattern describes the structure of a value, and binds some variables to the subcomponents of the value.

Pattern	Matches	Comments
`_`	anything	wildcard
`x`	anything	binds `x` to the value
42	the integer 42	
`false`	the boolean $false$	
(pat_1, pat_2)	a pair (v_1, v_2) if pat_i matches v_i	
$(x, \_)$	matches `(false, 42)`, binds x to `false`	
(pat_1, pat_2, pat_3)	a triple (v_1, v_2, v_3) ...	
`NONE`	matches `NONE` of any option type	
`SOME` pat	matches `SOME` v if pat matches v	
pat_1`::`pat_2		

val Declarations

Patterns may be used (*e.g.*, at the SML prompt) to define some variables:

- `val x = 42`
- `val (x,y) = (42, false)`
- ```
  val Manager (phb,lackeys) =
    Manager (("PHB",51),
                [Grunt ("J.Random Hacker", 12)])
  ```
- `val piApprox = 3.14159`
- `val SOME x = NONE`
  - Compiler comes back with a warning “Non-exhaustive match”, then runs the code anyway and comes back with a runtime error “binding failure” or “match error”
  ```

case Expressions

Analyze a value by cases: like a generalized if-expression.

```
case (42, false) of
   (x, true) => x
 | (23, false) => 17
 | _ => 0
```

- Tests each pattern in order, executes the branch that matches.
- Exhaustiveness-checks at compile-time: generates a warning.

fun Declarations

In addition to `val` declarations, can also define functions:

```
fun employeeName (Manager ((name, _), _)) = name
  | employeeName (Grunt (name,_))         = name
```

The compiler:

- infers the type of the function `employee -> string`
- checks that we covered all the cases for `employee`s

Recursion, polymorphism

Functions may be recursive, and indeed, polymorphic:

- `fun length nil = 0`
- `  | length (_::l') = 1 + (length l')`
- Since we don't case about the elements of the list, this function has type `'a list -> int`

let Expressions

We can have some local declarations within an expression.

```
fun countSubordinates (Grunt _) = 0
  | countSubordinates (Manager m) =
    let
      val (_,grunts) = m
    in
      length grunts
    end
```

- Each declaration in scope from its point of definition until the `end`.
- Useful for naming intermediate results, and helper functions

Mutual Recursion - Datatypes, functions

Sometimes, useful to have mutually recursive datatypes:

```
datatype 'a evenlist = Empty | EvenCons of 'a * 'a oddlist
and      'a oddlist = One of 'a | OddCons of 'a * 'a evenlist
```

Similarly, can have mutually recursive functions:

```
fun evenlength Empty              = 0
  | evenlength (EvenCons (_,ol)) = 1 + oddlength ol

and oddlength  (One _)            = 1
  | oddlength  (OddCons  (_,el)) = 1 + evenlength el
```

Anonymous and first class functions

In SML, as in other functional languages, functions may return functions, or take functions as arguments:

- `fun addN n = fn x => n + x`
- Take an `int`, return anonymous function that adds n to x
- Has type `int -> int -> int`
- `fun modifyAge f (name,age) = (name, f age)`
- Two patterns: match a thing and call it f, match a `person`
- Can be given the type: `(int -> int) -> person -> person`

Anonymous and first class functions, cont'd

- Example: `modifyAge (addN 1)` has type `person -> person`
- `fun map f nil = nil`
- `  | map f (x::xs) = (f x) :: (map f xs)`
- This function has type `('a -> 'b) -> 'a list -> 'b list`
- `map (modifyAge (addN 1)) somePeople`

Exceptions

- Functions must deal with unexpected input. Sometimes there is no sensible result type.
- Sometimes one can modify the function to return an option type, and return `NONE` on bad input.
- However sometimes need truly exceptional behavior: no sensible way to deal with bad data locally.

Exceptions: declaration

Exceptions are declared by an exception declaration:

- `exception NegativeAge of person`

Exceptions: raise

```
fun canRentCar (p as (_,age)) =
  if age <= 0 then raise (NegativeAge p)
    else age >= 25
```

Note I snuck in another pattern in here: y `as` pat matches if pat matches the entire value, and also binds y to that value

Exceptions: handle

Handling exceptions is done as follows:

```
canRentCar aPerson
  handle (NegativeAge p) => false
      |  Div => false (* raised by integer divide by zero *)
```

Handle has some patterns that matches some exceptions. No need to handle all exceptions: unhandled ones propagate up to top level. May reraise an exception:

```
foo handle e => raise e
```

Programming in the large

Programming in the large

Full SML: a program is a collection of structures (*i.e., modules*) and an expression to be evaluated

- Collaboration of multiple programmers
- Factoring of independent components
- Code reuse

Structures

- A structure declaration is a namespace for type, variable and function definitions.

```
structure BinaryTree = struct
  datatype 'a tree = ...
  val empty = ...
  fun addElem (t,x) = ...
  fun map f t = ...
  fun isEmpty t = ...
  fun toList t = ...
end
```

- Outside the structure, refer to `BinaryTree.empty`, etc.

Signatures as the type of structures

- Just as values have types, structures have a signature.
- Compiler will infer a *principal* signature for a structure that includes:
  - the definition of every type abbreviation and datatype
  - the declaration of every exception
  - the type of every value and function

```
sig
  datatype 'a tree = ...
  val empty : 'a tree
  val addElem : 'a * 'a tree -> 'a tree
  val mapTree : ('a -> 'b) -> 'a tree -> 'b tree
  ...
end
```

Abstraction: non-principal signatures

You can also write down less specific signatures:

- Hide the definition of types or datatypes
  - Abstract data types
- Hide helper functions

```
signature BINARY_TREE = sig
  type 'a tree
  val empty : 'a tree
  val addElem : 'a * 'a tree -> 'a tree
  val map : ('a -> 'b) -> 'a tree -> 'b tree
end
```

nature Ascription: hiding the implementat

```
structure BinaryTree :> BINARY_TREE = struct
  datatype 'a tree = ...
  val empty = ...
  fun addElem (t,x) = ...
  fun map f t = ...
  fun isEmpty t = ...
  fun toList t = ...
end
```

- Outside the implementation, the representation of trees and several functions are inaccessible.

Functors: parametrized implementation

To facilitate code reuse, possible to parametrize the implementation of a structure by zero or more other structures:

```
functor BalancedBinaryTreeFn (structure B : BINARY_TREE) =
struct
  fun balance t = ... (* mentions B.addElem, etc *)
end
```

Then use as:

```
structure MyBBT = BalancedBinaryTreeFn (structure B = BinaryTree)
structure BetterBBT = BalancedBinaryTreeFn (structure B = BetterBinaryTree
```

Conclusion

Conclusion: what I haven't told you

- Anything about side-effects: I/O, mutable store, concurrency, etc.
  - Turns out you can accomplish a lot (nearly everything for this course) without them.
- The truth about equality types.
  - But you can mostly pretend they don't exist.
- Record types: like tuples with named components
- Projection functions for tuples and records

What I haven't told you, cont'd

- Advanced modular programming: substructures, sharing specs, where-type
- Useful library functions.
  - Read the SML Basis documentation on the SML/NJ webpage www.smlnj.org
- Compilation management
  - aka `CM`, documented on the SML/NJ webpage. Will provide instructions in the programming assignments.

Further reading

- Ullman, “Elements of ML Programming”
- Paulson, “ML for the Working Programmer”
- Harper, “Programming in Standard ML”
- SML/NJ webpage
- Our course website