Introduction to SML

Basic Types, Tuples, Lists, Modelling

Michael R. Hansen

mrh@imm.dtu.dk

Informatics and Mathematical Modelling

Technical University of Denmark

Overview

- Basic types: Integers, Reals, Characters, Strings, Booleans
- Language elements (expressions, precedence, association, locally declared identifiers, etc.) are introduced "on the fly"
- Tuples and Patterns (Records: next week)
- Lists
- Modelling —- a tiny example

Basic Types: Integers

A data type comprises

- a set of values and
- a collection of operations

Basic Types: Integers

A data type comprises

- a set of values and
- a collection of operations

Integers

Type name : `int`

Values : `~27`, `0`, `1024`

Operations: (A few selected)

Operator	Type	Precedence	Association
`~`	`int -> int`	Highest	
`* div mod`	`int * int -> int`	7	Left
`+ -`	`int * int -> int`	6	Left
`= <> < <=`	`int * int -> bool`	4	Left

See also the library `Int`

Reals

Type name : `real`

Values : `~27.0`, `0.0`, `1024.71717`, `23.4E~11`

Operations: (A few selected)

Operator	Type	Precedence	Association
`abs`	`real -> real`	Highest	
`* /`	`real*real -> real`	7	Left
`+ -`	`real*real -> real`	6	Left
`= <> < <=`	`real*real -> bool`	4	Left

See also the libraries `Real` and `Math`

Reals

Type name : `real`

Values : `~27.0`, `0.0`, `1024.71717`, `23.4E~11`

Operations: (A few selected)

Operator	Type	Precedence	Association
`abs`	`real -> real`	Highest	
`* /`	`real*real -> real`	7	Left
`+ -`	`real*real -> real`	6	Left
`= <> < <=`	`real*real -> bool`	4	Left

See also the libraries `Real` and `Math`

Some built-in operators are *overloaded*. `*`: `real*real -> real`
`int * int -> int`

Default is `int`

Overloaded Operators and Type inference

A squaring function on integers:

Declaration	Type	
`fun square x = x * x`	`int -> int`	Default

Overloaded Operators and Type inference

A squaring function on integers:

Declaration	Type	
`fun square x = x * x`	`int -> int`	Default

A squaring function on reals: `square: real -> real`

Declaration	

Overloaded Operators and Type inference

A squaring function on integers:

Declaration	Type	
`fun square x = x * x`	`int -> int`	Default

A squaring function on reals: `square: real -> real`

Declaration	
`fun square(x:real) = x * x`	Type the argument

Overloaded Operators and Type inference

A squaring function on integers:

Declaration	Type	
`fun square x = x * x`	`int -> int`	Default

A squaring function on reals: `square: real -> real`

Declaration	
`fun square(x:real) = x * x`	Type the argument
`fun square x:real = x * x`	Type the result

Overloaded Operators and Type inference

A squaring function on integers:

Declaration	Type	
`fun square x = x * x`	`int -> int`	Default

A squaring function on reals: `square:  real -> real`

Declaration	
`fun square(x:real) = x * x`	Type the argument
`fun square x:real = x * x`	Type the result
`fun square x = x * x:  real`	Type expression for the result

Overloaded Operators and Type inference

A squaring function on integers:

Declaration	Type	
`fun square x = x * x`	`int -> int`	Default

A squaring function on reals: `square:  real -> real`

Declaration	
`fun square(x:real) = x * x`	Type the argument
`fun square x:real = x * x`	Type the result
`fun square x = x * x:  real`	Type expression for the result
`fun square x = x:real * x`	Type a variable

Overloaded Operators and Type inference

A squaring function on integers:

Declaration	Type	
`fun square x = x * x`	`int -> int`	Default

A squaring function on reals: `square:  real -> real`

Declaration	
`fun square(x:real) = x * x`	Type the argument
`fun square x:real = x * x`	Type the result
`fun square x = x * x:  real`	Type expression for the result
`fun square x = x:real * x`	Type a variable

Choose any mixture of these possibilities

Characters

Type name `char`

Values `#"a"`, `#" "`, `#"\""` (escape sequence for `"`)

Operator	Type	
`ord`	`char -> int`	ascii code of character
`chr`	`int -> char`	character for ascii code
`= < <= ...`	`char*char -> bool`	comparisons by ascii codes

Examples

```
- ord #"a";
> val it = 97 : int

- ord #"A";
> val it = 65 : int
```

```
-  #"a" < #"A";
> val it = false : bool;

- chr 88;
> val it = #"X" : char
```

Strings

Type name `string`

Values `"abcd"`, `" "`, `""`, `"123\"321"` (escape sequence for `"`)

Operator	Type	
`size`	`string -> int`	length of string
`^`	`string*string -> string`	concatenation
`= < <= ...`	`string*string -> bool`	comparisons
`Int.toString`	`int -> string`	conversions

Examples

```
- "auto" < "car";
> val it = true : bool

- "abc"^"de";
> val it = "abcde": string
```

```
- size("abc"^"def");
> val it = 6 : int

- Int.toString(6+18);
> val it = "24" : string
```

Booleans

Type name `bool`

Values `false`, `true`

Operator	Type	
`not`	`bool -> bool`	negation

```
not true = false
not false = true
```

Expressions

e_1 `andalso` e_2 "conjunction $e_1 \wedge e_2$"

e_1 `orelse` e_2 "disjunction $e_1 \vee e_2$"

— are lazily evaluated, e.g.

```
1<2 orelse 5/0 = 1
⇝ true
```

Precedence: `andalse` has higher than `orelse`

Tuples

An ordered collection of n values $(v_1, v_2, \ldots, v_n)$ is called an n-tuple

Examples

`- ();` `> val it = () : unit`	0-tuple
`- (3, false);` `> val it = (3, false) : int * bool`	2-tuples (pairs)
`- (1, 2, ("ab",true));` `> val it = (1, 2, ("ab", true)) :` ?	3-tuples (triples)

Selection Operation: $\#i(v_1, v_2, \ldots, v_n) = v_i$. `#2(1,2,3) = 2`

Equality defined componentwise

```
- (1, 2.0, true) = (2-1, 2.0*1.0, 1<2);
> val it = true : bool
```

provided = is defined on components

Tuple patterns

Extract components of tuples

```
- val ((x,_),(_,y,_)) = ((1,true),("a","b",false));
> val x = 1 : int
  val y = "b" : string
```

Pattern matching yields bindings

Restriction

```
- val (x,x) = (1,1);
! Toplevel input:
! val (x,x) = (1,1);
!         ^
! identifier is bound twice in a pattern
```

Infix functions

Directives: `infix` d f and `infixr` d f. d is the precedence of f

Example: exclusive-or

```
infix 0 xor      (* or just infix xor
                    -- lowest precedence  *)

fun false xor true  = true
  | true  xor false = true
  | _     xor _     = false
```

type ?

```
- 1 < 2+3 xor 2.0 / 3.0 > 1.0;
> val it = true : bool
```

Infix status can be removed by `nonfix xor`

```
- xor(1 < 2+3, 2.0 / 3.0 > 1.0);
> val it = true : bool
```

Let expressions — `let` *dec* `in` *e* `end`

Bindings obtained from *dec* are valid only in *e*

Example: Solve $ax^2 + bx + c = 0$

Declaration of types and exceptions

```
type equation = real * real * real
type solution = real * real

exception Solve;  (* declares an exception *)
```

```
fun solve(a,b,c) =
    let val d = b*b-4.0*a*c
    in if d < 0.0 orelse a = 0.0 then raise Solve
       else ((~b+Math.sqrt d)/(2.0*a)
            ,(~b-Math.sqrt d)/(2.0*a))
    end;
```

The type of `solve` is `equation -> solution`

Local declarations — local dec_2 in dec_2 end

Bindings obtained from dec_1 are valid only in dec_2

```
local
   fun disc(a,b,c) = b*b - 4.0*a*c
in
   exception Solve;

   fun hasTwoSolutions(a,b,c) = disc(a,b,c)>0.0
                                andalso a<>0.0;

   fun solve(a,b,c) =
      let val d = disc(a,b,c)
      in if d < 0.0 orelse a = 0.0 then raise Solve
         else ((~b+Math.sqrt d)/(2.0*a)
              ,(~b-Math.sqrt d)/(2.0*a))
      end
end;
```

Example: Rational Numbers

Specification	**Comment**
type qnum = int * int	rational numbers
exception QDiv	division by zero
mkQ: int * int -> qnum	construction of rational numbers
++: qnum * qnum -> qnum	addition of rational numbers
--: qnum * qnum -> qnum	subtraction of rational numbers
**: qnum * qnum -> qnum	multiplication of rational numbers
//: qnum * qnum -> qnum	division of rational numbers
==: qnum * qnum -> bool	equality of rational numbers
toString: qnum -> string	String representation of rational numbers

Intended use

```
val q1 = mkQ(2,3);

val q2 = mkQ(12, ~27);

val q3 = mkQ(~1, 4) ** q2 -- q1;

val q4 = q1 -- q2 // q3;

toString q4;
> val it = "~2/15" : string
```

Operators are infix with usual precedences

Representation: (a, b), $b > 0$ and $\text{gcd}(a, b) = 1$

Example $-\frac{12}{27}$ is represented by $(-4, 9)$

Greatest common divisor (Euclid's algorithm)

`fun gcd(0,n) = n` `  | gcd(m,n) = gcd(n mod m,m);`	`- gcd(12,27);` `> val it = 3 : int`

Function to cancel common divisors:

```
fun canc(p,q) =
  let val sign = if p*q < 0 then ~1 else 1
      val ap = abs p
      val aq = abs q
      val d  = gcd(ap,aq)
  in (sign * (ap div d), aq div d)
  end;
-  canc(12,~27);
> val it = (~4, 9) : int * int
```

Program for rational numbers

```
exception QDiv;

fun mkQ (_,0) = raise QDiv
  | mkQ pr    = canc pr;

infix 6 ++ --
infix 7 ** //
infix 4 ==

fun (a,b) ++ (c,d) = canc(a*d + b*c, b*d);
fun (a,b) -- (c,d) = canc(a*d - b*c, b*d);
fun (a,b) ** (c,d) = canc(a*c, b*d);
fun (a,b) // (c,d) = (a,b) ** mkQ(d,c);
fun (a,b) == (c,d) = (a,b) = (c,d);
fun toString(p:int,q:int) =
        (Int.toString p) ^ "/" ^ (Int.toString q);
```

Append

The infix operator @ (called ‘append’) joins two lists:

$$[x_1, x_2, \ldots, x_m] \mathbin{@} [y_1, y_2, \ldots, y_n] \\ = [x_1, x_2, \ldots, x_m, y_1, y_2, \ldots, y_n]$$

Properties

$$\begin{aligned} [\,] \mathbin{@} ys &= ys \\ [x_1, x_2, \ldots, x_m] \mathbin{@} ys &= x_1 :: ([x_2, \ldots, x_m] \mathbin{@} ys) \end{aligned}$$

Declaration

```
infixr 5 @                        (*  right associative *)

fun [] @ ys = ys
  | (x::xs) @ ys = x::(xs @ ys);
```

Append: evaluation

```
infixr 5 @                          (* right associative *)

fun        [] @ ys = ys
  | (x::xs) @ ys = x::(xs @ ys);
```

Evaluation

$$
\begin{array}{lll}
 & \texttt{[1,2] @ [3,4]} & \\
\rightsquigarrow & \texttt{1::([2] @ [3,4])} & (\texttt{x} \mapsto 1,\ \texttt{xs} \mapsto [2],\ \texttt{ys} \mapsto [3,4]) \\
\rightsquigarrow & \texttt{1::(2::([] @ [3,4]))} & (\texttt{x} \mapsto 2,\ \texttt{xs} \mapsto [\,],\ \texttt{ys} \mapsto [3,4]) \\
\rightsquigarrow & \texttt{1::(2::[3,4])} & (\texttt{ys} \mapsto [3,4]) \\
\rightsquigarrow & \texttt{1::[2,3,4]} & \\
\rightsquigarrow & \texttt{[1,2,3,4]} &
\end{array}
$$

- Execution time is linear in the size of the first list

Append: polymorphic type

```
> infixr 5 @
> val @ = fn : 'a list * 'a list -> 'a list
```

- ’a is a *type variable*
- The type of @ is *polymorphic* — it has many forms

’a = int: Appending integer lists

```
[1,2] @ [3,4]; val it = [1,2,3,4] : int list
```

’a = int list: Appending lists of integer list

```
[[1],[2,3]] @ [[4]]; val it = [[1],[2,3],[4]] :
```

@ is a built-in function

Reverse $\texttt{rev}\ [x_1, x_2, \ldots, x_n] = [x_n, \ldots, x_2, x_1]$

```
fun naive_rev []        = []
 |  naive_rev(x::xs) = naive_rev xs @ [x];
```
val naive_rev = fn : 'a list -> 'a list

```
    naive_rev[1,2,3]
⇝  naive_rev[2,3] @ [1]
⇝  (naive_rev[3] @ [2]) @ [1]
⇝  ((naive_rev[] @ [3]) @ [2]) @ [1]
⇝  (([] @ [3]) @ [2]) @ [1]
⇝  ([3] @ [2]) @ [1]
⇝  (3::([] @ [2])) @ [1]
⇝  ...
⇝  [3,2,1]
```

efficient version is built-in (we come back to that)

Membership — equality types

$$\begin{aligned} & x \;\texttt{member}\; [y_1, y_2, \ldots, y_n] \\ = \;& (x = y_1) \lor (x = y_2) \lor \cdots \lor (x = y_n) \\ = \;& (x = y_1) \lor (x \;\texttt{member}\; [y_2, \ldots, y_n]) \end{aligned}$$

Declaration

```
infix member

fun x member []       = false
  | x member (y::ys) = x=y orelse x member ys;
infix 0 member
val member = fn : ''a * ''a list -> bool
```

- `''a` is an equality type variable no functions
- `(1,true) member [(2,true), (1,false)]` ⇝ false
- `[1,2,3]  member [[1], [], [1,2,3]]` ⇝ ?

An exercise

From list of pairs to pair of lists:

$$\texttt{unzip}\,[(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)] = ([x_1, x_2, \ldots, x_n], [y_1, y_2, \ldots, y_n])$$

Many functions on lists are predefined, e.g. `@`, `rev`, `length`, and also the SML basis library contains functions on lists, e.g. `unzip`. See for example `List`, `ListPair`

Exercises: G-bar and ...

1. A first part where the purpose is to make you more acquainted with recursion, basic types, lists and the use of libraries. This is a collection of small exercises.
2. The second part concerns efficient algorithms. In particular you shall develop two versions of merge sort, which both have a $n \log n$ worst case execution time.