15-150 Fall 2013

Stephen Brookes

LECTURE 2
Thursday, August 29

Announcements

- HOMEWORK 1 is out...
- Due Tuesday 3 September 11:59pm

Must be your OWN work

See course policy online

Today

- Expressions and types
- Declarations and scope
- Patterns and matching
- Equality in ML
- Specifications

Types

- basic types **int**, **real**, **bool**
- tuples **int * int**, **int * int * real**
- functions **int -> int**, **real -> int * int**
- lists **int list**, **(int -> int) list**

Values

- For each type `t` there is a set of *values*
- An expression of type `t` *evaluates to a value of type* `t` (or fails to terminate)

TYPE SAFETY

Values

- **int** ... *integers*
- **real** ... *real numbers*
- **int list** ... *lists of integers*
- **int -> int** ... *functions* from *integers* to *integers*

functions are values

Examples

expression	value : type
(3 + 4) * 6	42 : int
(3.0 + 4.0) * 6.0	42.0 : real
(42, 5)	(42, 5) : int * int

Standard ML of New Jersey [...]

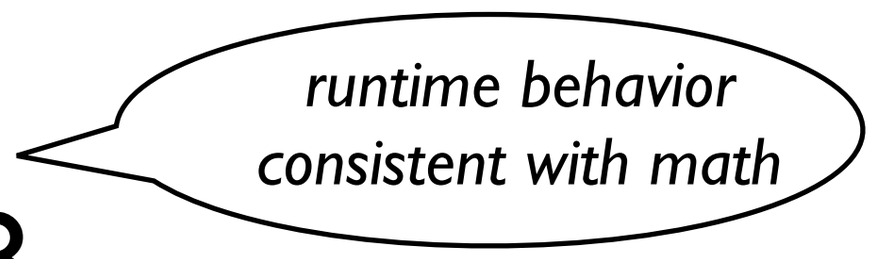
```
- 225 + 193;
val it = 418 : int
```

Don’t forget the semi-colon.

ML reports the type and value.

$$225 + 193 = 418$$

$$225 + 193 \Rightarrow^* 418$$



Examples

expression	value : type
fn (x:int):int => x+1	fn - : int -> int
fn (x:real):real => x+1.0	fn - : real -> real
Math.sin	fn - : real -> real

Examples

fn (x:int, y:int) : int*int => (x div y, x mod y)

type **int*int -> int*int**

already a value

(**fn** (x:int, y:int) : int*int => (x div y, x mod y)) (42, 5)

type **int*int**

evaluates to the value (8, 2)

Declarations

fun divmod(x:int, y:int) : int*int = (x div y, x mod y)

binds divmod

to a function of type int*int -> int*int

val (q:int, r:int) = divmod(42, 5)

binds q to 8, r to 2

Summary

- An expression of type t can be *evaluated*
- If it terminates, we get a *value* of type t
- ML reports type and value
  - val it = 3 : int
  - val it = fn - : int -> int
- Declarations produce *bindings*
- Bindings are *statically scoped*

Using declarations

```
fun check(x:int, y:int):bool =
  let
     val (q:int, r:int)= divmod(x, y)
  in
    (x = q*y + r)
  end
```

introduces check : int * int -> bool

Using declarations

```
val pi : real = 3.14;

fun square(r:real) : real = r * r;

fun area(r:real) : real = pi * square(r);
```

```
val pi : real = 3.14159;

fun area(r:real) : real = pi * square(r);
```

Using declarations

```
fun circ(r:real):real = 2.0 * pi * r;
```

```
fun circ(r:real):real =
 let
   val pi2:real = 2.0 * pi
 in
   pi2 * r
 end
```

```
local
   val pi2:real = 2.0 * pi
 in
   fun circ(r:real):real = pi2 * r
 end
```

Lists

- [1, 3, 2, 1, 21+21] : int list
- [true, false, true] : bool list
- [[1],[2, 3]] : (int list) list
- [ ] : int list, [ ] : bool list, ...
- 1::[2, 3] = [1, 2, 3]
- [1, 2]@[3, 4] = [1, 2, 3, 4]
- nil = [ ]

Patterns

- Wildcard: _
- Variable: x
- Constant: 42, true, ~3
- Tuple: $(p_1, ..., p_k)$
- List: nil, $p_1::p_2$, $[p_1, ..., p_k]$

no constant patterns for reals or functions

where

$p_1, ..., p_k$

are patterns

Syntactic constraint:
no variable appears *twice*
in the same pattern

ML =

- ML only permits use of = on types with an exact equality test
- Such types are called *equality types*
- Equality types include all types built from

int, **bool**

using tuple and list constructors,

e.g. **int list**

int * bool * int

(int * bool * int) list

Constant patterns only for equality types

Matching

- A *pattern* can be *matched* against a *value*
- If the match *succeeds*, it produces *bindings*

matching d::L against the value [2,4]
succeeds with bindings ⟦d:2, L:[4]⟧

matching d::L against the value [ ]
fails

Matching

- Matching 42 against the value 42 *succeeds*
- Matching 42 against the value 0 *fails*
- Matching x against *any* value v *succeeds* with the binding x:v
- Matching _ against any value *succeeds*

Matching

- Matching $p_1::p_2$ against [ ] *fails*
- Matching $p_1::p_2$ against $v_1::v_2$ *fails*
  if matching p_1 against v_1 *fails*,
  or matching p_2 against v_2 *fails*
- Matching $p_1::p_2$ against $v_1::v_2$ *succeeds*
  with bindings $L_1@L_2$
  if matching p_1 against v_1 *succeeds with* L_1
  and matching p_2 against v_2 *succeeds with* L_2

Scope

- Bindings have syntactically fixed *scope*

val x = 3.14;
... *scope of* x : 3.14

let
  val x = 3.14
in
  ... *scope of* x : 3.14
end;
... *not in scope*

eval

```
fun eval ([ ]:int list):int = 0
  |    eval (d::L) = d + 10 * (eval L);
```

- eval : int list -> int
  - This function definition uses *list patterns*
    - [ ] only matches empty list
    - d::L only matches non-empty list, binds d to head of list, L to tail
  - eval [2,4] = 42

eval

```
fun eval ([ ]:int list) : int = 0
  |    eval (d::L) = d + 10 * (eval L);
```

eval [2,4] =>* ⟦d:2, L:[4]⟧ (d + 10 * (eval L))

=>* 2 + 10 * (eval [4])

=>* 2 + 10 * (4 + 10 * (eval [ ]))

=>* 2 + 10 * (4 + 10 * 0)

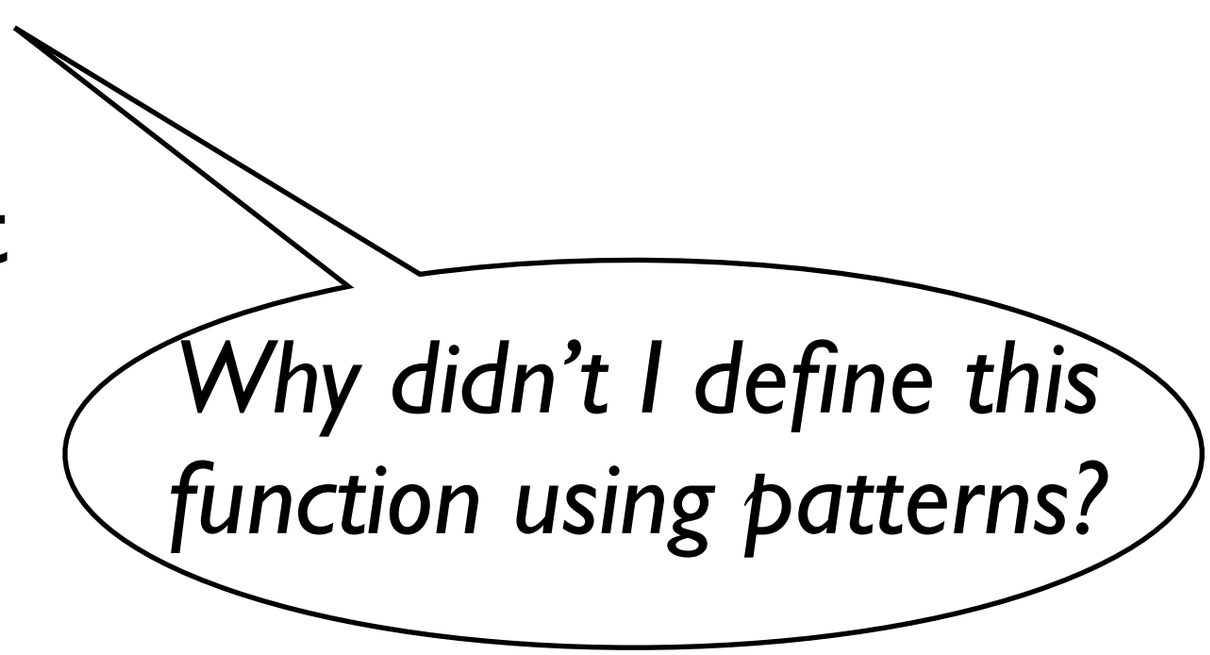
=>* 2 + 10 * 4

=>* 42

decimal

```
fun decimal (n:int) : int list =
    if n<10 then [n]
            else (n mod 10) :: decimal (n div 10);
```

- decimal : int -> int list
- decimal 42 = [2,4]
- decimal 0 = [0]



decimal

decimal 42

=>* if 42 < 10 then [42]

else (42 mod 10) :: decimal(42 div 10)

=>* (42 mod 10) :: decimal (42 div 10)

=>* 2 :: decimal (42 div 10)

=>* 2 :: decimal 4

=>* 2 :: (if 4 < 10 then [4] else ...)

=>* 2 :: [4]

=>* [2,4]