

ICS/CSE 141: Programming Languages

ML

School of Information and Computer Science
University of California, Irvine

Chris Stork
cstork@ics.uci.edu

Overview

1. ML's Background
2. Basic Building Blocks
3. Basic Types
4. Type Constructors
5. Patterns & Functions

ML's Background

Algol

Algol Family of Programming Languages

- **Algol 60**: “an improvement over many of its **successors**” — Dijkstra
 - **Impressive**: formal syntax, concise and abstract specification, recursion
 - **Weird**: pass-by-name, type loopholes (array and procedure parameters)
- **Algol 68**: over-designed and under-implemented. . .
 - **Algol W**: Wirth’s alternative, improves Algol 60 without going overboard
- **Pascal**: further refinement of Algol W, quite successful
 - **Impressive**: Wirth & Hoare specified most of Pascal using axiomatic semantics
- **C**: cross of Algol W and BCPL, **huge** success (partly due to Unix)
 - “C is quirky, flawed, and an enormous success.” — Dennis Ritchie

History leading up to ML

Logic for Computable Functions (LCF) Project

- Goal: System to **prove** interesting properties of functional programs.
- ML: **Meta Language** for that system, think “scripting language.”
- Write **programs** that attempt to construct **mathematical proofs**.

Tactic: One particular approach to prove a formula, may need. . .

- a **type system** to ensure **correct** proofs (at least structurally)
- **exceptions** since `tactic: formula -> proof` can **fail**
- **higher-order functions** for building complex tactics

ML's contributions

Common Themes of Algol Family

- instructions and imperative programming
- higher-order functions are “discouraged”
- sound type systems and (formal) semantics

ML breaks with the first two, inspired by LISP

- expressions and functional programming
- higher-order functions are “encouraged”
- sound type systems and (formal) semantics

For a complete list of ML's cool features see

<http://www.smlnj.org/sml.html>

Our Implementation & ML Intros

There are different kinds of ML. We use Standard ML as it was standardized in 1997 (SML97).

Our implementation

Moscow ML available at

<http://www.dina.kvl.dk/~sestoft/mosml.html>

(RPMs for SuSE & Redhat at

[http://nil.ics.uci.edu/~ics141ta/mosml/.](http://nil.ics.uci.edu/~ics141ta/mosml/))

(Use manual only to get everything set up, besides that it's about Moscow ML's special features which are of no interest to us.)

Good Intro

Robert Harper's online draft of "Programming in Standard ML"

(<http://www-2.cs.cmu.edu/~rwh/smlbook/online.pdf>)

Don't be shocked by overview example, but go straight to chapter 2!

Standard ML

Further information on SML is available at

- smlnj.org: Go to bottom of page for further background and documentation
- sml.sf.net: More implementations

Basic Building Blocks

Expressions

Interpreter determines **value** and **type**

- `(5+3)-2; ⇒ val it = 6 : int`
- `it+3; ⇒ val it = 9 : int`
- `if true then 1 else 5; ⇒ val it = 1 : int`
- `5 = 4; ⇒ val it = false : bool`

Interpreter performs **type checking** as well

- The input `if true then 1 else false;` is trouble...
- An expression can't have type `int` **or** type `bool`!

⇒ Type clash: expression of type `bool`
cannot have type `int`

Value Declarations

Declarations are **constants** not **variables**!

- `val x = 7+2; ⇒ val x = 9 : int`
- `val y = x+3; ⇒ val y = 12 : int`
- `val z = x*y - x div y; ⇒ val z = 108 : int`
- `val a = if x > y then z else 42; ⇒ val a = 42 : int`

Declarations can be **hidden** by “newer” ones

- `x; ⇒ val x = 9 : int`
- `val x = 1; ⇒ val x = 1 : int`
- `y; ⇒ val x = 12 : int`
- `val y = x+3; ⇒ val y = 4 : int`

Function Declarations I

Functions can be **named** or **anonymous**

- `fun f(x) = x + 5; ⇒ val f = fn : int -> int`
- `f(37); ⇒ val it = 42 : int`
- `val f = fn x => x+5; ⇒ val f = fn : int -> int`
- `f(37); ⇒ val it = 42 : int`
- `(fn x => x+5) 37; ⇒ val it = 42 : int`

Functions Declarations II

Functions can be **curried** or **uncurried**

- `fun f(x,y) = x+y; ⇒ val f = fn : int * int -> int`
- `f(10,20); ⇒ val it = 30 : int`
- `fun f x y = x+y; ⇒ val f = fn : int -> int -> int`
- `f 10 20; ⇒ val it = 30 : int`
- `f 10; ⇒ val it = fn : int -> int`
- `it 20; ⇒ val it = 30 : int`

Basic Types

Integer & Real

The type `int` is just what you would expect,

- except that negative numbers are written using **tilde** `~1`, `~2`, ...
- operations include `+`, `-`, `*`, `div`, ...
- `12 div 13` \Rightarrow `val it = 0 : int`

The type `real` is just what you would expect,

- except that negative numbers are written using **tilde** `~1.2`, ...
- operations include `+`, `-`, `*`, `/`, ...
- `12.0 / 13.0` \Rightarrow `val it = 0.923076923077 : real`

You can **not** mix `int` and `real`!

- explicit conversion functions `trunc`, `round`, `real`, etc.

Unit & Boolean

The type `unit` has only one element, `()`

- used like `void` in C to indicate the “absence” of a value
- useful for imperative **instructions** that change state without a value

The type `bool` has two values, `true` and `false`

- the obvious application are `if` expressions, see earlier
- there are operators `not`, `andalso`, `orelse`, etc.

- ```
fun equiv (x,y) =
 (x andalso y) orelse ((not x) andalso (not y));
```

⇒

```
val equiv = fn : bool * bool -> bool
```

# Type Constructors

# Tuples

A **tuple** is an **ordered** sequence of values, different types allowed

- `(1,2); ⇒ val it = (1, 2) : int * int`
- `(true,"b"); ⇒ val it = (true, "b") : bool * string`
- `(1,3,7); ⇒ val it = (1, 3, 7) : int * int * int`

Elements can be **accessed** using the # notation

- `#1(true,"b"); ⇒ val it = true : bool`
- `#2(true,"b"); ⇒ val it = "b" : string`
- `#3(true,"b"); ⇒ error`

## Records

A **record** is a **labeled** sequence of values, different types allowed

- $\{x=1, y=2\}; \Rightarrow \text{val it} = \{x=1, y=2\} : \{x: \text{int}, y: \text{int}\}$
- $\{\text{name}="Peter", \text{job}="Lecturer", \text{age}=32, \text{crazy}=\text{true}\};$   
 $\Rightarrow \text{val it} = \{\text{age} = 32, \text{crazy} = \text{true}, \text{job} =$   
 $"Lecturer",$   
 $\text{name} = "Peter"\} : \{\text{age} : \text{int}, \text{crazy} : \text{bool},$   
 $\text{job} : \text{string}, \text{name} : \text{string}\}$

Elements can be **accessed** using the # notation again

- $\#x\{x=1, y=2\}; \Rightarrow \text{val it} = 1 : \text{int}$
- $\#y\{x=1, y=2\}; \Rightarrow \text{val it} = 2 : \text{int}$
- $\#z\{x=1, y=2\}; \Rightarrow \text{error}$

## Lists

A **list** is a sequence of values, different types **not** allowed

- `[1,2,3,4]; ⇒ val it = [1, 2, 3, 4] : int list`
- `[fn x => x+1, fn x => x+2];  
⇒ val it = [fn, fn] : (int -> int) list`
- `[1,true]; ⇒ error`

A number of familiar operations on lists...

- `3 :: []; ⇒ val it = [3] : int list` “cons”
- `hd [1,2,3]; ⇒ val it = 1 : int` “car”
- `tl [1,2,3]; ⇒ val it = [2, 3] : int list` “cdr”

# Patterns & Functions

# Patterns I

We can use **more** than just identifiers in declarations

- `val t = (1,2,3);`  $\Rightarrow$  `val t = (1, 2, 3) : int * int * int`
- `val (x,y,z) = t;`  $\Rightarrow$   
`val x = 1 : int`  
`val y = 2 : int`  
`val z = 3 : int`
- the **pattern** `(x,y,z)` is **matched** against the value `t`
- the “components” of the pattern get the “component” values of `t`
- `val (x,y) = t;`  $\Rightarrow$  **error**
- type of pattern and type of value must agree

You know already how this is implemented!!

Is this like our `match` or `mmatch`??

## Patterns II

Patterns work for other type constructors as well...

- `val t = {x=1,y=2}; ⇒`  
`val t = {x=1,y=2} : {x: int, y: int}`
- `val {x,y} = t; ⇒`  
`val x = 1 : int`  
`val y = 2 : int`
- `val t = [1,2,3]; ⇒`  
`val t = [1, 2, 3] : int list`
- `val [a,b,c] = t; ⇒`  
`val a = 1 : int`  
`val b = 2 : int`  
`val c = 3 : int`
- `val H::T = t; ⇒`  
`val H = 1 : int`  
`val T = [2, 3] : int list`

# Functions I

Let's look at a basic **length** function, first in “Scheme”-style

```
fun len l =
 if null l then 0 else 1 + len (tl l);
```

The `null` function checks for empty lists, the `tl` function returns the `cdr`

ML yields the type `val 'a len = fn : 'a list -> int` for this<sup>a</sup>

Examples:

- `len []; ⇒ val it = 0 : int`
- `len [1,2,3]; ⇒ val it = 3 : int`

---

<sup>a</sup>For now, read `'a` as “any type,” we'll get to the details later...

## Functions II

Using **patterns**, we can rewrite the **length** function as follows

```
fun length [] = 0
 | length (H::T) = 1 + length T;
```

We can define functions “by cases,” closely mirrors mathematics...

again the type is `val 'a length = fn : 'a list -> int`  
for this

Examples:

- `length []; ⇒ val it = 0 : int`
- `length [1,2,3]; ⇒ val it = 3 : int`

## Functions & Patterns

Now we can explain why functions **always** have **one** argument in ML

- `fun f(x,y) = x + y; ⇒ val f = fn : int * int -> int`
- the **single** argument is a **tuple pattern** of length 2
- `fun f x y = x + y; ⇒ val f = fn : int -> int -> int`
- this is **shorthand** for **two** nested  $\lambda$  expressions

Three more remarks about patterns and functions

- functions with multiple “cases” are matched in **textual order**
- ML will warn you if you “forget a case,” if it can anyway!!
- identifiers within a pattern must be **unique**, `(x, x)` does not work (no `match`)

**Thank You!**