Some Advanced ML Features

Mooly Sagiv

Michael Clarkson, Cornell CS 3110 Data Structures and Functional Programming
University of Washington: Dan Grossman
Recap last week

Some benefits of no side-effects

Some advanced features
  – Exceptions
  – Module System
  – Imperative Features
<table>
<thead>
<tr>
<th>Class</th>
<th>Grammar/Meta-Variable</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identifiers</td>
<td>x, y</td>
<td>a, x, y, x_y, foo1000</td>
</tr>
<tr>
<td>Datatype Type constructors</td>
<td>X, Y</td>
<td>Nil, Const, List</td>
</tr>
<tr>
<td>Constants</td>
<td>c</td>
<td>2, 4.0, “ff”, []</td>
</tr>
<tr>
<td>Expressions e</td>
<td>e ::= x</td>
<td>c</td>
</tr>
<tr>
<td>Patterns</td>
<td>p ::= c</td>
<td>x [:t]</td>
</tr>
<tr>
<td>Declarations</td>
<td>d ::= p</td>
<td>e = e</td>
</tr>
<tr>
<td>Types</td>
<td>t ::= int</td>
<td>float</td>
</tr>
<tr>
<td>Values</td>
<td>v ::= c</td>
<td>(v₁, ..., vₙ)</td>
</tr>
</tbody>
</table>
Factorial in ML

let rec fac n = if n = 0 then 1 else n * fac (n - 1)

// val fac : int -> int = <fun>

let rec fac n : int = if n = 0 then 1 else n * fac (n - 1)

let rec fac = function
  | 0 -> 1
  | n -> n * fac(n - 1)

let fac n =
  let rec ifac n acc =
    if n=0 then acc else ifac n-1, n * acc
  in ifac n, 1
Plan

• Basic Programming in ML  [last week, this recitation]
• Advanced ML Programming [This week]
• Javascript
• Runtime Management for PL
• Type Inference for ML
• Scala
• ...
Benefits of Functional Programming

• No side-effects
• Referential Transparency
  – The value of expression e depends only on its arguments
• Conceptual
• Commutativity
• Easier to show that the code is correct
• Easier to generate efficient implementations
Let-expressions

3 questions:

• Syntax: \(\text{let } d_1 \text{ and } \ldots \text{ and } d_n \text{ in } e\)
  
  – Each \(d_i\) is any binding and \(e\) is any expression

• Type-checking: Type-check each \(d_i\) and \(e\) in a static environment that includes the previous bindings.
  Type of whole let-expression is the type of \(e\)

• Evaluation: Evaluate each \(d_i\) and \(e\) in a dynamic environment that includes the previous bindings.
  Result of whole let-expression is result of evaluating \(e\).
Silly Examples

let silly1 (z : int) =
  let x = if z > 0 then z else 34
  and y = x+z+9
  in
  if x > y then x*2 else y*y

let silly2 (z : int) =
  let x = 1
  in
  (let x = 2 in x+1) +
  (let y= x+2 in y+1)
// val silly2 : int -> int = <fun>

**silly2** is poor style but shows let-expressions are expressions

- Can also use them in function-call arguments, if branches, etc.
- Also notice shadowing
let rec append l1 l2 =
  match l1 with
  | []    -> l2
  | hd :: tl -> hd :: append tl l2
//val append : 'a list -> 'a list -> 'a list = <fun>

let x = [2;4] //val x : int list = [2; 4]
let y = [5;3;0] //val y : int list = [5; 3; 0]
let z = append x y
//val z : int list = [2; 4; 5; 3; 0]

x → [2] → [4] → \[can’t tell, but it’s the first one\]

y → [5] → [3] → [0] → 

or

z → [2] → [4] → 

x → [2] → [4] → 

y → [5] → [3] → [0] → 

z → [2] → [4] → [5] → [3] → [0] → 

List Example
Exceptions

• Captures abnormal behavior
  – Error handling
  – Integrated into the type system

exception Error // exception Error
let sqrt1 (x : float) : float =
  if x < 0.0 then raise Error
  else sqrt x
// val sqrt1 : float -> float = <fun>

exception FailWith of string

raise (FailWith "Some error message")
What’s the largest program you’ve ever worked on by yourself or as part of a team?

A. 10-100 LoC
B. 100-1,000 LoC
C. 1,000-10,000 LoC
D. 10,000-100,000 LoC
E. >= 100,000 LoC
Modularity

• **Modular programming**: code comprises independent *modules*
  – developed separately
  – understand behavior of module in isolation
  – reason locally, not globally
Java features for modularity

• classes, packages:
  – organize identifiers (classes, methods, fields, etc.) into namespaces

• Interfaces:
  – describe related classes

• public, protected, private:
  – control what is visible outside a namespace
Ocaml Features for Modularity

• **modules**: organize identifiers (functions, values, etc.) into namespaces

• **signatures**:
  – describe related modules

• **abstract types**:
  – control what is visible outside a namespace
Ocaml Modules

• Syntax:  
  module ModuleName = struct definitions end
• The name must be capitalized
• The definitions can be any top-level definitions  
  – let, type, exception
• Create a new namespace
• Every file myFile.ml with contents *D is essentially wrapped in a module definition*  
  module MyFile = struct D end
• Modules can be opened locally to save writing

```plaintext
module M = struct  let x = 42 end
//module M : sig val x : int end
let fourtytwo = M.x
// val fourtytwo : int = 42
```
module Stack = struct
  let empty = [ ]
  let is_empty s = s = [ ]
  let push x s = x :: s
  let pop s = match s with
    [ ] -> failwith "Empty"
    | x::xs -> (x, xs)
end

//module Stack :
// sig
//  val empty : 'a list
//  val is_empty : 'a list -> bool
//  val push : 'a -> 'a list -> 'a list
//  val pop : 'a list -> 'a * 'a list
// end

fst (Stack.pop (Stack.push 1 Stack.empty))
//- : int = 1
Might Seem Backwards...

Java:
```
    s = new Stack();
    s.push(1);
    s.pop()
```

OCaml:
```
let s = Stack.empty in
    let s’ = Stack.push 1 s in
        let (one, _) = Stack.pop s’
```
Abstraction

• Forgetting Information
• Treating different things as identical
Abstraction

• Programming language predefined abstractions
  – Data structures like list
  – Library functions like map and fold
• Programming languages enable to define new abstractions
  – Procedural abstractions
  – Data abstraction
  – Iterator abstraction
Procedural Abstraction

• Abstract implementation details
  – sqrt : float -> float

• List.sort : (‘a -> ‘a -> int) -> ‘a list -> ‘a list

• Abstracts how the functions are implemented
  – Both the implementation and the usage should obey the type contract
  – The implementation can assume the right type
  – Important for composing functions
Data Abstraction

• Abstract from details of organizing data
  – stacks, symbol tables, environments, bank accounts, polynomials, matrices, dictionaries, ...

• Abstract from implementation of organization:
  – Actual code used to add elements (e.g.) isn’t important
  – But types of operations and assumptions about what they do and what they require are important
Ocaml Advanced Modularity Features

• Functors and Signatures
• Functions from Modules to Modules
• Permit
  – Dependency injection
  – Swap implementations
  – Advanced testing
module type STACK = sig
  val empty : 'a list
  val is_empty : 'a list -> bool
  val push : 'a -> 'a list -> 'a list
  val pop : 'a list -> 'a * 'a list
end

module Stack : STACK = struct
  ...
  (* as before *)
end
module type STACK = sig
  type t
  val empty : t
  val is_empty : t -> bool
  val push : int -> t -> t
  val pop : t -> int * t
end

module Stack : STACK = struct
  type t = int list
  let empty = [ ]
  let is_empty s = s = [ ]
  let push x s = x :: s
  let pop s = match s with
    [ ] -> failwith "Empty"
  | x::xs -> (x,xs)
end
Summary Modularity

• ML provides flexible mechanisms for modularity

• Guarantees type safety
Side-Effects

• But sometimes side-effects are necessary
• The whole purpose of programming is to conduct side-effects
  – Input/Output
• Sometimes sharing is essential for functionality
• ML provides mechanisms to capture side-effects
  – Enable efficient handling of code with little side effects
let \( x = 3 \) in
let () = print_string ("Value of x is " ^ (string_of_int x)) in
\( x + 1 \)
;; value of x is 3- : int = 4

\( e ::= \ldots \mid (e_1; \ldots; e_n) \)

let \( x = 3 \) in
(print_string ("Value of x is " ^ (string_of_int x));
\( x + 1 \))
Refs and Arrays

• Two built-in data-structures for implementing shared objects

```ocaml
module type REF =
  sig
    type 'a ref
    (* ref(x) creates a new ref containing x *)
    val ref : 'a -> 'a ref
    (* !x is the contents of the ref cell x *)
    val (!) : 'a ref -> 'a
    (* Effects: x := y updates the contents of x
     * so it contains y. *)
    val (:=) : 'a ref -> 'a -> unit
  end
```
Simple Ref Examples

```ocaml
let x : int ref = ref 3
  in
  let y : int = !x
    in
    (x := !x + 1);
    y + !x
// - : int = 7
```
More Examples of Imperative Programming

• Create cell and change contents
  
  \[
  \text{val } x = \text{ref } \text{"Bob"};
  \]
  
  \[
  x := \text{"Bill"};
  \]

• Create cell and increment
  
  \[
  \text{val } y = \text{ref } 0;
  \]
  
  \[
  y := \lnot y + 1;
  \]

• While loop
  
  \[
  \text{val } i = \text{ref } 0;
  \]
  
  \[
  \text{while } \lnot i < 10 \text{ do } i := \lnot i + 1;
  \]
  
  \[
  i;
  \]
Summary References

• Provide an escape for imperative programming
• But insures type safety
  – No dangling references
  – No (double) free
  – No null dereferences
• Relies on automatic memory management
# Functional Programming Languages

<table>
<thead>
<tr>
<th>PL</th>
<th>types</th>
<th>evaluation</th>
<th>Side-effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>scheme</td>
<td>Weakly typed</td>
<td>Eager</td>
<td>yes</td>
</tr>
<tr>
<td>ML OCAML F#</td>
<td>Polymorphic strongly typed</td>
<td>Eager</td>
<td>References</td>
</tr>
<tr>
<td>Haskell</td>
<td>Polymorphic strongly typed</td>
<td>Lazy</td>
<td>None</td>
</tr>
</tbody>
</table>
Things to Notice

• Pure functions are easy to test

\[
\text{prop}_{\text{RevRev}} \ l \ = \ \text{reverse}(\text{reverse} \ l) \ == \ l
\]

• In an imperative or OO language, you have to
  – set up the state of the object and the external state it reads or writes
  – make the call
  – inspect the state of the object and the external state
  – perhaps copy part of the object or global state, so that you can use it in the post condition
Things to Notice

Types are everywhere.

reverse:: [w] -> [w]

• Usual static-typing panegyric omitted...

• In ML, types express high-level design, in the same way that UML diagrams do, with the advantage that the type signatures are machine-checked

• Types are (almost always) optional: type inference fills them in if you leave them out
Consider this function...

```plaintext
let reverse ls = match ls with
  [] -> []
| x :: xs -> reverse xs
```

... and its most general type:

```plaintext
:- reverse :: list `t_1 -> list `t_2
```

What does this type mean?

Reversing a list should not change its type, so there must be an error in the definition of reverse!
Recommended ML Textbooks

• L. C. PAULSON: ML for the Working Programmer
• J. Ullman: Elements of ML Programming
• R. Harper: Programming in Standard ML
Recommended Ocaml Textbooks

• Xavier Leroy: The OCaml system release 4.02
  – Part I: Introduction
• Jason Hickey: Introduction to Objective Caml
• Yaron Minsky, Anil Madhavapeddy, Jason Hickey: Real World Ocaml
Summary

• Functional programs provide concise coding
• Compiled code compares with C code
• Successfully used in some commercial applications
  – F#, ERLANG, Jane Street
• Ideas used in imperative programs
• Good conceptual tool
• Less popular than imperative programs