CS538 Lecture 15: ML Modules

April 1997

James R. Larus
Computer Sciences Department
University of Wisconsin–Madison
larus@cs.wisc.edu

Outline:

- Structures
- Signatures
- Functors

ML Modules

- ADT facility
  - Specification: *what* module defines and implements
  - Implementation: actual definitions and implementation
  - Parameterized: same code works for many types
- Similar to ADT's in other languages
  - e.g. ADA
- New: "functions" on modules to parameterize them
- Not object-oriented
  - ML-2000

Three Parts

- *Signature*
  - Specification of module interface
- *Structure*
  - Implementation
- *Functor*
  - Function invoked to create implementation
  - Signature $\rightarrow$ Structure

Syntax

- Signature

```
signature <NAME>
sig
    ... type and value declarations
end;
```

- Functor

```
functor <functor name> ( <parameters> )
  : <signature name> =
struct
  ... definition of types and values declared in sig ...
end;
```

- Structure

```
structure <structure name>
  = <functor name> ( <parameters> );
```

Aside on Equality

- Values (but not functions) can be compared
  - Atomic types
  - Lists: same length and corresponding elements equal
  - Records: correspond fields equal
  - User-defined types: tree match
- Values that include functions cannot be compared

```
type T = (int -> int) list
```

- Special type variable (“a) for types that can be compared

Structure Example

- Maintain mappings
  - Pairs (‘d, ‘r) from domain (‘d) to range (‘r)
- Operations
  - **`create`**
  - **`lookup`**
  - **`insert`**

Mapping Structure

```
structure Mapping = struct
  exception NotFound;

  val create = nil;

  fun lookup(d, nil) = raise NotFound
  |    lookup(d, (e,r)::es) =
        if d=e then r
        else lookup(d, es);

  fun insert(d, r, nil) = [(d, r)]
  |    insert(d, r, (e,s)::es) =
          if d=e then (d,r)::es
          else (e,s)::insert(d, r, es)
end;
```

Note 2 keywords

structure proceeds identifier being declared a structure

struct .. end defines elements of structure

```
structure Mapping :
sig
  exception NotFound
  val create : 'a list
  val insert : ''a * 'b * (''a * 'b) list -> (''a * 'b) list
  val lookup : ''a * (''a * 'b) list -> 'b
```


Signatures

- Structures (modules) have a type
  - Called a signature
  - Can be used to restrict a structure
- Contains some of:
  - type
  - eqtype (comparable)
  - exception
  - val (values or functions)

Signature Example

- Consider a string-int mapping

```
signature SIMAPPING = sig
  exception NotFound;
  val create : (string * int) list;
  val insert : string * int * (string * int) list ->
    (string * int) list;
  val lookup : string * (string * int) list -> int
end;
```

- Can use to restrict mapping

```
structure SiMapping : SIMAPPING = Mapping
```

  - Another form of structure declaration

Restrictions

- Signature restricts a structure two ways
  - By specifying a more restrictive type for an identifier
  - Omitting identifier (hiding them)

```
sig
  exception NotFound;
  val lookup: string * (string * int) list -> int
end
```

Accessing Names in Structures

- Use qualified names

```
val m = SiMapping.create;
val m = nil : (string * int) list
val m = SiMapping.insert("in", 6, m);
val m = [("in",6)] : (string * int) list
val m = SiMapping.insert("a", 1, m);
val m = [("a",1), ("in",6)] : (string * int) list
SiMapping.lookup("in", m);
val it = 6 : int
```

- Open structure
  - All names exported by structure added to current environment

```
open SiMapping;
val m = create;
val m = nil : (string * int) list
val m = insert("in", 6, m);
val m = [("in",6)] : (string * int) list
```


Importing Operations into Structures

- Example: binary search tree requires comparison function

```
fun lt(x:string, y) = ...

structure StringBST =
struct
  datatype 'label btree =Empty
  | Node of 'label * 'label btree * 'label btree;
  val create = Empty;
  fun insert(x, T) = ...
  fun lookup(x, T) = ...
  ...
end;
structure StringBST :
  sig
    datatype 'a btree
      con Empty : 'a btree
      con Node : 'a * 'a btree * 'a btree -> 'a btree
    val create 'a btree
  val insert : string * string btree -> string btree
```

Problems

- ML inferred types for structure from definition of `lt`
  - Typing of structure depends on inference from use of functions
- But, code is more general
  - Binary tree could work for any type with a less-than function

Functors

- Operate on structure to produce a new structure
  - Define a functor with a less-than predicate as its argument
  - Incorporate predicate into new structure
  - Binary tree structure constructor
- Define signature of restrictions on binary tree labels

```
signature TOTALORDER = sig
  eqtype element;
  val lt : element * element -> bool
end;
```

  - Labels must have a total order
    - Equality and a less-than predicate

Functors, cont'd

- Define a functor

```
functor MakeBST(Lt: TOTALORDER):
  sig
    open Lt;
    type `label btree;
    exception EmptyTree;
    val create : element btree;
    val insert : element * element btree ->
      element btree;
    ...
  end
=
  struct
    open Lt;
    datatype `label btree = Empty
    | Node of `label * `label btree * `label btree;
    val create = Empty;
    fun insert(x, T) = ...
```

Only btree (not constructors) exported

Functors, cont'd

- Define a label type

```
structure String : TOTALORDER =
  struct
    type element = string;
    fun lt(x: string, y) =
      ...
  end;
```

- Apply the functor

```
structure StringBST = MakeBST(String);
```

Functor Variants

- Define the functor argument explicitly

```
structure StringBST = MakeBST(
  structure String : TOTALORDER =
   struct
    type element = string;
    fun lt(x: string, y) =
      ...
   end);
```

- Can pass arguments of any type to functor
  - Including structures

Using Modules

- Information hiding
- Abstract types
- Local Definitions
- Substructures
- Sharing

Hiding Information

- Define a structure to be another one, with a given signature

```
structure S2 : Sig = S1;
```

  - **Sig** can omit elements from $\mathbf{S_1}$
  - Hide elements from users of $\mathbf{S_2}$
  - But still available for use in function definitions in $\mathbf{S_1/S_2}$

Modified: April 9, 1997

Abstract Types

- Datatype that hides constructors

```
abstype <datatype definition>
with
  <declarations with constructors>
end
```

  - Can only create instance through interface

```
abstype `label btree = Empty
  | Node of `label * `label btree * `label btree
with
  val create = Empty;
  fun insert(x, T) = ...
  func lookup(x, T) = ...
  ...
end;
```


Copyright © 1996–97 by James R. Larus. All rights reserved.
Modified: April 9, 1997

Local Definitions

- Hide elements in abstract type or structure

```
local<definitions>
in <definitions>
end
```

  - Only second definitions can access first group

```
abstype `label btree = Empty
  | Node of `label * `label btree * `label btree
with
  val create = Empty;
  fun insert(x, T) = ...
  func lookup(x, T) = ...
  local
    exception EmptyTree;
    fun deletemin = ...
  in
    fun delete = ...
  end
```


Sharing

- Ensure that two substructures refer to same item
  - Part of signature

```
sharing type <type> = <type> = ... = <type>
sharing <structure> = <structure> = ... = <structure>
```

  - Asserts that types/structures must be the same

Sharing Example

```
signature ELEMENT = sig
  type element;
  val similar : element * element -> bool;
end;

signature BTREE = sig
  structure Element : ELEMENT;
  eqtype elt;
  sharing type elt = Element.element;
  datatype btree = Empty | Node of elt * btree * btree;
  val leaf : elt -> btree;
  val build : elt * btree * btree -> btree;
  val lookup : elt * btree -> bool
end;
```

Local type

Sharing

Modified: April 9, 1997

Example, cont'd

```
signature TREE = sig
  structure Element : ELEMENT;
  eqtype elt;
  sharing type elt = Element.element;
  datatype tree = Tree of elt * tree list;
  val build : elt * tree * tree -> tree;
  val lookup : elt * tree -> bool
end;

signature ALLTREES = sig
  structure Btree: BTREE;
  structure Tree: TREE;
  sharing Btree.Element = Tree.Element;
  sharing type Btree.elt = Tree.elt
end;
```

Two subtrees

Same elements

Modified: April 9, 1997