SML's Module System

Lars-Henrik Eriksson

Functional Programming 1

Original slides by Tjark Weber based on notes by Sven-Olof Nyström

Today

- Structures
- Signatures
- Functors

Structures

Programming in the Large

So far in the course, we have covered basic language constructs that one needs to write individual functions.

This allows us to write small programs.

But it doesn't scale. A program that consists of thousands of individual functions would be a maintenance nightmare.

To manage larger software projects, we need more structure in our code: modules with precisely-specified interactions.

SML: Structures

A **structure** is a module (namespace): it consists of a collection of types, exceptions, values and substructures packaged together into a logical unit.

Example (a structure for counters):

```
structure Counter =
struct
  type T = int
  fun make_counter () = 0
  fun inc c = c+1
  fun dec c = if c=0 then 0 else c-1
  fun is_zero c = c=0
end
```

Structures: Dot Notation

To use a structure, one can access its components using **dot notation**.

Examples:

```
42 : Counter.T
Counter.make_counter ()
Counter.inc (Counter.make_counter ())
Counter.is_zero 42
```

Structures: open

To access a structure's components without dot notation, one can **open** the structure. This incorporates all of its components into the current environment.

Examples:

```
open Counter;
42 : T;
make_counter ();
inc (make_counter ());
is_zero 42;
```

Structures: Local open

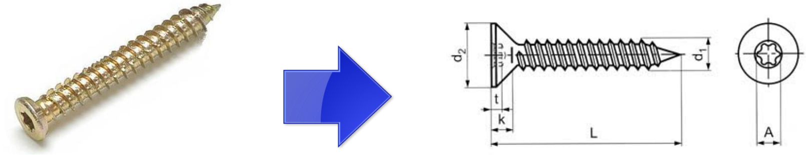
Opening structures globally pollutes the top-level environment and should be done sparingly. It is more common to open structures locally:

```
let
  open S
in
  <expression>
end
```

```
local
  open S
in
  <declaration>
end
```

Signatures

What is Abstraction (in Computer Science)?



What is Abstraction (in Computer Science)?

A way to introduce **new concepts** that are **meaningful** to humans.

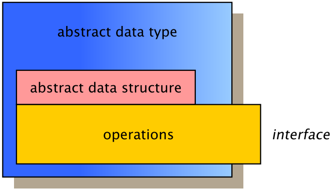
Abstraction tries to **reduce and factor out details** so that the programmer can focus on a few concepts at a time.

Examples: files, data structures, procedure calls, ...

(We think of files as something real, but files don't exist, they are just a bunch of bits on a hard drive. Come to think of it, bits don't exist either, they are just magnetic fluctuations on the surface of disk platters.)

Abstract Data Types

An abstract data type (ADT) is a model for data structures that have similar behavior.



An abstract data type is defined indirectly, by the **operations** that may be performed on it. It does *not* specify the actual implementation of the type.

Abstract data types are one of the most important concepts in all programming, because they allow **data encapsulation**: to separate implementation details from a well-defined interface.

A First Example: Integers

```
type int

val * = fn: int * int -> int
val + = fn: int * int -> int
val - = fn: int * int -> int
val < = fn: int * int -> bool
...
```

Do you know how Poly/ML actually represents integers in memory?

A First Example: Integers

```
type int

val * = fn: int * int -> int
val + = fn: int * int -> int
val - = fn: int * int -> int
val < = fn: int * int -> bool
...
```

Do you know how Poly/ML actually represents integers in memory?

You don't need to know! Poly/ML could use *any* implementation that supports the usual arithmetic operations on integers.

Another Example: A Type of Counters

Suppose we want to define a type of counters. Counters can be incremented and decremented (down to a fixed minimal value).

Earlier, we saw a concrete implementation of counters using integers:

```
structure Counter =
struct
  type T = int
  fun make_counter () = 0
  fun inc c = c+1
  fun dec c = if c=0 then 0 else c-1
  fun is_zero c = c=0
end
```

(This is *not* abstract: counters *are* integers.)

Another Example: A Type of Counters (cont.)

We could implement counters in a more imaginative way:

```
structure Counter =
struct
  type T = unit list
  fun make_counter () = ...
  fun inc c = ...
  fun dec c = ...
  fun is_zero c = ...
end
```

(This is *not* abstract: counters *are* lists.)

Another Example: A Type of Counters (cont.)

We could implement counters in a more imaginative way:

```
structure Counter =
struct
  type T = unit list
  fun make_counter () = []
  fun inc c = ...
  fun dec c = ...
  fun is_zero c = ...
end
```

(This is *not* abstract: counters *are* lists.)

Another Example: A Type of Counters (cont.)

We could implement counters in a more imaginative way:

```
structure Counter =
struct
  type T = unit list
  fun make_counter () = []
  fun inc c = () :: c
  fun dec c = ...
  fun is_zero c = ...
end
```

(This is *not* abstract: counters *are* lists.)

Another Example: A Type of Counters (cont.)

We could implement counters in a more imaginative way:

```
structure Counter =
struct
  type T = unit list
  fun make_counter () = []
  fun inc c = () :: c
  fun dec c = case c of [] => [] | _::cs => cs
  fun is_zero c = ...
end
```

(This is *not* abstract: counters *are* lists.)

Another Example: A Type of Counters (cont.)

We could implement counters in a more imaginative way:

```
structure Counter =
struct
  type T = unit list
  fun make_counter () = []
  fun inc c = () :: c
  fun dec c = case c of [] => [] | _::cs => cs
  fun is_zero c = null c
end
```

(This is *not* abstract: counters *are* lists.)

Another Example: A Type of Counters (cont.)

We could use a datatype:

```
structure Counter =
struct
  datatype T = EmptyCounter
             | UnitCounter of T
  fun make_counter () = ...
  fun inc c = ...
  fun dec (EmptyCounter) = ...
    | dec (UnitCounter c) = ...
  fun is_zero (EmptyCounter) = ...
    | is_zero (UnitCounter _) = ...
end
```

(This is *not* abstract: counters *are* elements of this datatype.)

Another Example: A Type of Counters (cont.)

We could use a datatype:

```
structure Counter =
struct
  datatype T = EmptyCounter
             | UnitCounter of T
  fun make_counter () = EmptyCounter
  fun inc c = ...
  fun dec (EmptyCounter) = ...
    | dec (UnitCounter c) = ...
  fun is_zero (EmptyCounter) = ...
    | is_zero (UnitCounter _) = ...
end
```

(This is *not* abstract: counters *are* elements of this datatype.)

Another Example: A Type of Counters (cont.)

We could use a datatype:

```
structure Counter =
struct
  datatype T = EmptyCounter
             | UnitCounter of T
  fun make_counter () = EmptyCounter
  fun inc c = UnitCounter c
  fun dec (EmptyCounter) = ...
    | dec (UnitCounter c) = ...
  fun is_zero (EmptyCounter) = ...
    | is_zero (UnitCounter _) = ...
end
```

(This is *not* abstract: counters *are* elements of this datatype.)

Another Example: A Type of Counters (cont.)

We could use a datatype:

```
structure Counter =
struct
  datatype T = EmptyCounter
             | UnitCounter of T
  fun make_counter () = EmptyCounter
  fun inc c = UnitCounter c
  fun dec (EmptyCounter) = EmptyCounter
    | dec (UnitCounter c) = c
  fun is_zero (EmptyCounter) = ...
    | is_zero (UnitCounter _) = ...
end
```

(This is *not* abstract: counters *are* elements of this datatype.)

Another Example: A Type of Counters (cont.)

We could use a datatype:

```
structure Counter =
struct
  datatype T = EmptyCounter
             | UnitCounter of T
  fun make_counter () = EmptyCounter
  fun inc c = UnitCounter c
  fun dec (EmptyCounter) = EmptyCounter
    | dec (UnitCounter c) = c
  fun is_zero (EmptyCounter) = true
    | is_zero (UnitCounter _) = false
end
```

(This is *not* abstract: counters *are* elements of this datatype.)

Another Example: A Type of Counters (cont.)

We could (just for the heck of it) use odd integers:

```
structure Counter =
struct
  type T = int
  fun make_counter () = ...
  fun inc c = ...
  fun dec c = ...
  fun is_zero c = ...
end
```

(This is *not* abstract: counters *are* integers.)

Another Example: A Type of Counters (cont.)

We could (just for the heck of it) use odd integers:

```
structure Counter =
struct
  type T = int
  fun make_counter () = 1
  fun inc c = ...
  fun dec c = ...
  fun is_zero c = ...
end
```

(This is *not* abstract: counters *are* integers.)

Another Example: A Type of Counters (cont.)

We could (just for the heck of it) use odd integers:

```
structure Counter =
struct
  type T = int
  fun make_counter () = 1
  fun inc c = c+2
  fun dec c = ...
  fun is_zero c = ...
end
```

(This is *not* abstract: counters *are* integers.)

Another Example: A Type of Counters (cont.)

We could (just for the heck of it) use odd integers:

```
structure Counter =
struct
  type T = int
  fun make_counter () = 1
  fun inc c = c+2
  fun dec c = if c=1 then 1 else c-2
  fun is_zero c = ...
end
```

(This is *not* abstract: counters *are* integers.)

Another Example: A Type of Counters (cont.)

We could (just for the heck of it) use odd integers:

```
structure Counter =
struct
  type T = int
  fun make_counter () = 1
  fun inc c = c+2
  fun dec c = if c=1 then 1 else c-2
  fun is_zero c = c=1
end
```

(This is *not* abstract: counters *are* integers.)

Structures Don't Provide Encapsulation

There are many different ways to implement counters.

Structures provide modularity, but not encapsulation. All of our structures

```
structure Counter =
struct
  type T = ...
  ...
struct
```

specify the actual implementation of the counter type.

Hence they do not protect counters, i.e., they do not enforce that counters are accessed *only* through the operations

make_counter, inc, dec, is_zero

SML: Signatures

A **signature** is an interface: it specifies the names of all the entities provided, the arities of type components, and the types of value components.

Example (a signature for counters):

```
signature COUNTER =
sig
  type T
  val make_counter: unit -> T
  val inc: T -> T
  val dec: T -> T
  val is_zero: T -> bool
end
```

SML: Opaque Ascription

Signatures can be **ascribed** to matching structures:

```
structure Counter :> COUNTER =
struct
  type T = int
  fun make_counter () = 0
  fun inc c = c+1
  fun dec c = if c=0 then 0 else c-1
  fun is_zero c = c=0
end
```

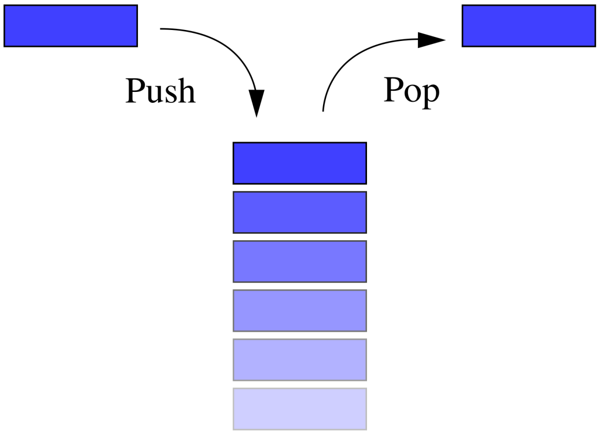
The structure must provide definitions for all of the signature's components. (It may contain additional and/or more general definitions.)

SML: Opaque Ascription (cont.)

Only those components of the structure that are declared in the signature remain visible.

- The structure implements an *abstract datatype* (ADT).
- The concrete data representation is *hidden*.
- The ADT can *only* be manipulated via the functions declared in the signature.
- It is *impossible* to access the data representation outside the structure.

Example: An ADT for Stacks



Example: An ADT for Stacks (cont.)

Stacks with elements of type 'a: 'a T

Interface:

- empty
  TYPE: 'a T
- push x s
  TYPE: 'a -> 'a T -> 'a T
  PRE: true
  POST: the stack s with x added as new top element
- pop s
  TYPE: 'a T -> 'a * 'a T
  PRE: s is non-empty
  POST: (the top element of s, s without its top element)
- Empty
  TYPE: exn

Example: An ADT for Stacks (cont.)

A corresponding signature:

```
signature STACK =
sig
  type 'a T
  val empty: 'a T
  val push: 'a -> 'a T -> 'a T
  val pop: 'a T -> 'a * 'a T
  exception Empty
end
```

Example: An ADT for Stacks (cont.)

A matching structure that implements stacks via lists:

```
structure Stack :> STACK =
struct
  type 'a T = 'a list
  val empty = ...
  fun push x s = ...
  fun pop [] = ...
    | pop (x::s) = ...
  exception Empty
end
```

Note the use of opaque ascription to hide all implementation details.

Example: An ADT for Stacks (cont.)

A matching structure that implements stacks via lists:

```
structure Stack :> STACK =
struct
  type 'a T = 'a list
  val empty = []
  fun push x s = ...
  fun pop [] = ...
    | pop (x::s) = ...
  exception Empty
end
```

Note the use of opaque ascription to hide all implementation details.

Example: An ADT for Stacks (cont.)

A matching structure that implements stacks via lists:

```
structure Stack :> STACK =
struct
  type 'a T = 'a list
  val empty = []
  fun push x s = x::s
  fun pop [] = ...
    | pop (x::s) = ...
  exception Empty
end
```

Note the use of opaque ascription to hide all implementation details.

Example: An ADT for Stacks (cont.)

A matching structure that implements stacks via lists:

```
structure Stack :> STACK =
struct
  type 'a T = 'a list
  val empty = []
  fun push x s = x::s
  fun pop [] = raise Empty
    | pop (x::s) = (x,s)
  exception Empty
end
```

Note the use of opaque ascription to hide all implementation details.

Example: An ADT for Stacks (cont.)

Using stacks:

```
Stack.empty;
Stack.push 42 Stack.empty;
Stack.pop (Stack.push 42 Stack.empty);
```

Because implementation details (like the use of lists) are hidden, all of the following expressions are ill-formed:

```
Stack.empty = [];
Stack.push 42 [];
Stack.pop [42];
```

Example: An ADT for Dictionaries

A dictionary (or associative array) maps unique keys to values.

John Smith	+1-555-8976
Lisa Smith	+1-555-1234
Sam Doe	+1-555-5030

Example: An ADT for Dictionaries (cont.)

Dictionaries with keys of type ''a and values of type 'b: ('' a, 'b) T

Interface:

- empty
  TYPE: ('' a, 'b) T
- insert k v d
  TYPE: '' a -> 'b -> (''a, 'b) T -> (''a, 'b) T
  PRE: true
  POST: the dictionary d updated (or extended) such that k maps to v
- lookup k d
  TYPE: '' a -> (''a, 'b) T -> 'b option
  PRE: true
  POST: SOME v if d maps k to some v, NONE otherwise

Example: An ADT for Dictionaries (cont.)

A corresponding signature:

```
signature DICTIONARY =
sig
  type (''a,'b) T
  val empty: (''a,'b) T
  val insert: ''a -> 'b -> (''a, 'b) T -> (''a, 'b)
  val lookup: ''a -> (''a, 'b) T -> 'b option
end
```

Example: An ADT for Dictionaries (cont.)

A matching structure that implements dictionaries via association lists:

```
structure Dictionary :> DICTIONARY =
struct
  type (''a,'b) T = (''a * 'b) list
  val empty = ...
  fun insert k v d = ...
  fun lookup k [] = ...
    | lookup k ((k',v) :: d) =
        ...
end
```

Note the use of opaque ascription to hide all implementation details.

Example: An ADT for Dictionaries (cont.)

A matching structure that implements dictionaries via association lists:

```
structure Dictionary :> DICTIONARY =
struct
  type (''a,'b) T = (''a * 'b) list
  val empty = []
  fun insert k v d = ...
  fun lookup k [] = ...
    | lookup k ((k',v) :: d) =
        ...
end
```

Note the use of opaque ascription to hide all implementation details.

Example: An ADT for Dictionaries (cont.)

A matching structure that implements dictionaries via association lists:

```
structure Dictionary :> DICTIONARY =
struct
  type (''a,'b) T = (''a * 'b) list
  val empty = []
  fun insert k v d = (k,v) :: d
  fun lookup k [] = ...
    | lookup k ((k',v) :: d) =
          ...
end
```

Note the use of opaque ascription to hide all implementation details.

Example: An ADT for Dictionaries (cont.)

A matching structure that implements dictionaries via association lists:

```
structure Dictionary :> DICTIONARY =
struct
  type (''a,'b) T = (''a * 'b) list
  val empty = []
  fun insert k v d = (k,v) :: d
  fun lookup k [] = NONE
    | lookup k ((k',v) :: d) =
        if k=k' then SOME v else lookup k d
end
```

Note the use of opaque ascription to hide all implementation details.

When to Use Abstract Datatypes?

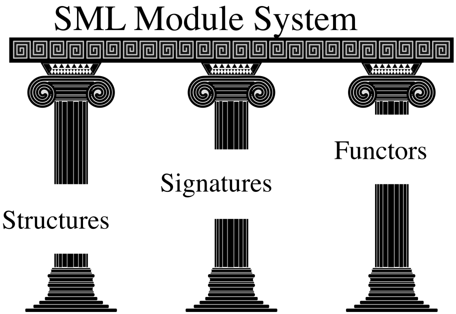
It is useful to make a datatype **abstract** when

- the implementation of the datatype is complex, or
- you want to separate implementation and interface, or
- you want to protect the underlying representation, or
- the datatype represents a natural abstraction.

Abstract datatypes are often a good way to split a program into parts that can be understood separately.

Functors

Structures, Signatures, Functors



SML's module system rests on three syntactic constructs: structures, signatures and functors.

We just covered structures (= modules, namespaces) and signatures (= interfaces). We'll now look at functors.

SML: Functors

A functor is a **function from structures to structures**.

That is, a functor accepts one or more arguments, which are usually structures of a given signature, and produces a structure as its result.

Functors are used to implement generic data structures and algorithms.

Functors: A First Example

```
signature INT = sig val x: int end;

functor Double(I: INT) =
struct
  val x = 2 * I.x
end;

structure Two = struct val x = 2 end;
structure Four = Double(Two);

Four.x;
```

Example: A Functor for Postfix Evaluation

Let's consider expressions given in postfix notation, i.e., every operator follows all of its operands.

For example: $3\ 4\ +\ 2\ *$

(One advantage of postfix notation is that such expressions are unambiguous, even without parentheses.)

Can you come up with an algorithm to compute the value of postfix expressions? (Hint: use a stack.)

Example: A Functor for Postfix Evaluation (cont.)

Note that your algorithm doesn't depend on how the stack is implemented. Any implementation of the stack interface will do!

In SML, we can define a functor that takes an arbitrary stack implementation and returns (a structure that contains) a function to evaluate postfix expressions.

Example: A Functor for Postfix Evaluation (cont.)

Let's say that an expression is given by a non-empty list of operators ($+$, $*$) and integer operands:

```
datatype atom = Int of int | Plus | Times
```

Recall our signature for stacks:

```
signature STACK =
sig
  type 'a T
  val empty: 'a T
  val push: 'a -> 'a T -> 'a T
  val pop: 'a T -> 'a * 'a T
  exception Empty
end
```

Example: A Functor for Postfix Evaluation (cont.)

```
functor POSTFIX(S: STACK) =
struct
  fun eval xs =
    let
      fun eval' (Int i, s) = S.push i s
        | eval' (Plus, s) =
            let
              val (b, s) = S.pop s
              val (a, s) = S.pop s
            in
              S.push (a+b) s
            end
        | eval' (Times, s) = ...
      val (v, _) = S.pop (foldl eval' S.empty xs)
    in
      v
    end
end
```

Example: A Functor for Postfix Evaluation (cont.)

Recall the Stack structure, our list-based implementation of stacks:

```
structure Stack :> STACK =
struct
  type 'a T = 'a list
  ...
end
```

Applying the POSTFIX functor to this structure will yield (a structure that contains) a function that uses list-based stacks to evaluate postfix expressions:

```
structure Postfix = POSTFIX(Stack);
```

```
Postfix.eval [Int 3, Int 4, Plus, Int 2, Times];
```