Topic VII

Data abstraction and modularity

References:


Largely based on an Introduction to SML Modules by Claudio Russo

<http://research.microsoft.com/~crusso>
A useful introduction to SML standard libraries, and a good example of modular programming.


http://www.standardml.org/
The Core and Modules languages consist of two sub-languages:

The Core language is for programming in the small, by expressing details of data structures and algorithms. It supports the definition of types and expressions denoting values of those types.

The Modules language is for programming in the large, by grouping related Core definitions of types and expressions into self-contained units with descriptive interfaces. The Modules language expresses software architecture. Both languages are largely independent.

SML consists of two sub-languages:

The Core and Modules languages
The Modules language lets one split large programs into separate units with descriptive interfaces.

```
fun even (n:nat) = iter true not n

fun iter b f i = if i = zero then b
  else f (iter b f (i-1))

else if i = zero then b
  then iter i f i

fun succ x = x + 1

val zero = 0

type nat = int
```

Writing a real program as an unstructured sequence of Core definitions quickly becomes unmanageable.

The Modules language
Abstract datatypes are types equipped with a set of operations, which are the only operations applicable to that type. Its representation can be changed without affecting the rest of the program.

Signatures let us specify what components a structure values, and functions.

Structures let us package up declarations of related types, values, and functions.

An abstract data type is a type equipped with a set of operations, which are the only operations applicable to that type.
In Modules, one can encapsulate a sequence of Core type definitions into a unit called a structure.

We enclose the definitions in between the keywords `struct` and `end`.

Example: A structure representing the natural numbers, as positive integers.

```
struct
  type nat = int
  val zero = 0
  type _nat = int

  fun succ x = x + 1
  fun iter b f i = if i = zero then b
                  else f (iter b f (i-1))
end
```
One can name a structure by binding it to an identifier.

The dot notation

Value `IntNat.iter` dynamically evaluates to a closure.

NB: Type `IntNat.nat` is statically equal to `int`.

Component of a structure are accessed with the dot notation.

```
fun even (n: IntNat.nat) = IntNat.iter true not n
```

```
end
```

```
... fun iter b f i = ...
...
```

```
structure IntNat = ...
structure IntNat = ...
```

```
... type nat = int
```

The dot notation
Nested and dot notation provides name-space control.

Sequencing dots provides deeper access (IntNatAdd.Nat.zero).

The dot notation (IntNatAdd.Nat) accesses a nested structure.

\[
\text{IntNatAdd.Nat.iter IntNatAdd.Nat.zero (IntNatAdd.add m) in}
\]
\[
\text{fun mult n m = IntNatAdd.add m in}
\]
\[
\ldots
\]
\[
\text{end}
\]
\[
\text{fun add n m = Nat.iter m Nat.succ in}
\]
\[
\text{structure Nat = IntNatAdd.Nat = IntNatAdd =}
\]
\[
\text{Nested structures can be nested inside other structures, in a hierarchy.}
\]
\[
\text{Nested structures}
\]
Concrete signatures

Signature expressions specify the types of structures by listing the specifications of their components.

As a signature expression consists of a sequence of component specifications, enclosed in between the keywords sig...
end.

```
sig
  val zero : nat
  val succ : nat -> nat
  val iter : 'a -> ('a -> 'a) -> nat -> 'a

  type nat = int
```

This signature fully describes the type IntNat.

Concrete signatures

The specification of type nat is \textit{concrete}: it must be \texttt{int}.

A signature expression consists of a sequence of component specifications of their components.
This specification of type \texttt{nat} is \textit{opaque}.

On the other hand, the following signature specifies structures that are free to use any implementation for type \texttt{nat} (perhaps \texttt{int}, or \texttt{word}, or some recursive datatype).

```
Opaque signatures

On the other hand, the following signatures
```
Example: Polymorphic functional stacks.

signature STACK =
  sig
    exception E
    type 'a reptype (*<-- INTERNAL REPRESENTATION *)
    val new: 'a reptype
    val push: 'a -> 'a reptype -> 'a reptype
    val pop: 'a reptype -> 'a reptype
    val top: 'a reptype -> 'a
  end

type 'a reptype (* -- INTERNAL REPRESENTATION *)

exception E

structure STACK =

Example: Polymorphic functional stacks.
structure MyStack:

STACK =

struct
exception E;
type 'a reptype = 'alist;
val new = [];
fun push x s = x::s;
fun split (h::t) = (h, t) |
    split _ = raise E;
fun pop s = #2(split s);
fun top s = #1(split s);
end;

fun push x s = x::s;
fun val new = []
  type 'a reptype = 'a list;
  exception E;
structure MyStack = STACK:
val MyEmptyStack = MyStack.new ;
val MyStack0 = MyStack.push 0 MyEmptyStack ;
val MyStack01 = MyStack.push 1 MyStack0 ;
val MyStack0' = MyStack.pop MyStack01 ;
MyStack.top MyStack0' ;

val MyEmptyStack = [] : 'a MyStack.reptype
val MyStack0 = [0] : int MyStack.reptype
val MyStack01 = [1,0] : int MyStack.reptype
val MyStack0 = [0] : int MyStack.reptype
val MyStack0' = [0] : int MyStack.reptype

val MyStack.top MyStack0' ;
val MyStack0 = MyStack.pop MyStack01 ;
val MyStack01 = MyStack.push 1 MyStack0 ;
val MyStack0 = MyStack.push 0 MyEmptyStack ;
val MyEmptyStack = MyStack.new ;
Named and nested signatures

Signatures may be named and referenced, to avoid repetition:

```
signature NAT =
  val zero : nat
  val succ : nat -> nat
  val 'a iter : 'a -> ('a->'a) -> nat -> 'a
end

signature Nat =
  (* references NAT *)
  structure Nat: NAT
  (* references NAT *)
  structure Add =
    val add : Nat.nat -> Nat.nat -> Nat.nat
    val a iter : a -> (a->a) -> a
    val succ : nat -> nat
    val zero : nat

signature Nats =
  structure Nat =
  structure Add =
    val add : Nat.nat -> Nat.nat -> Nat.nat
    val a iter : a -> (a->a) -> a
    val succ : nat -> nat
    val zero : nat
```

Nested signatures specify named sub-structures:

```
end
```
To avoid nesting, one can also directly include a signature

\textbf{Signature inclusion}

\textbf{NB:} This is equivalent to the following signature.

```
end

val \textit{add}: \textit{nat} \rightarrow \textit{nat} \rightarrow \textit{nat}
```

\textbf{NB:} To avoid nesting, one can also directly include a signature

\textbf{Signature inclusion}
Q: When does a structure satisfy a signature?

A: A structure satisfies a signature whenever:

- **Component-wise:** The structure must enrich this realized signature.
  - Type components in the signature:
    - The structure must realize (i.e., define) all of the opaque components at least the components of the signature.
  - The structure must realize a signature matches a structure whenever it implements at least the components of the signature.

- **Scheme:** Every specified type must be implemented equivalently.
- **Structure:** Every specified value must have a more general type scheme.
- **Component-wise:** Every specified type must be implemented equivalently.

Signature matching
Properties of signature matching

- The components of a structure can be defined in a different order than in the signature; names matter but ordering does not.
- A structure may contain more components, or signatures, than are specified in a matching signature; there is no need to pre-declare its many signatures and there is no need to pre-declare its matching signatures (unlike "interfaces" in Java and C#).
- Although similar to record types, signatures actually play a different number of different roles.
- Signature matching is structural. A structure can match a structure, matching signatures (unlike "interfaces" in Java and C#).
- A structure may contain more components, or signatures, than are specified in a matching signature, than in the signature; names matter but ordering does not.
Subtyping

Signature matching supports a form of subtyping not found in the Core language:

1. A structure with more type, value, and structure components may be used where fewer components are expected.
2. A value component may have a more general type than expected.
3. A structure then expected.
Using signatures to restrict access

The following structure uses a signature constraint to provide a restricted view of \texttt{IntNat}:

```ocaml
structure ResIntNat =
  IntNat : sig
  type nat
  val succ : nat \rightarrow nat
  val iter : nat \rightarrow (nat \rightarrow nat) \rightarrow nat
  end
```

\textbf{NB:} The constraint \texttt{str} prunes the structure \texttt{str}:

\begin{itemize}
  \item \texttt{ResIntNat.zero} is undefined;
  \item \texttt{ResIntNat.iter} is less polymorphic that \texttt{IntNat.iter}.
\end{itemize}
Although the :: operator can hide names, it does not conceal the definitions of opaque types. Thus, the fact that ResIntNat.nat = IntNat.nat = Int remains transparent.

For instance, the application ResIntNat.succ(~3) is still well-typed, because ~3 has type Int ... but ~3 is negative, so not a valid representation of a natural number!
The combination of these methods yields abstract structures. Of an abstract type declaration, further, we can hide the representation of a type by means opaque manners. A structure by constraining its signature in transparent or In SML, we can limit outside access to the components of

Information hiding

SML Modules
new, abstract type for each opaque type in $\texttt{str}$. The constraint $\texttt{str} : \texttt{str}$ prunes $\texttt{str}$ but also generates a new, abstract type for each opaque type in $\texttt{str}$.

```plaintext
val iter : \texttt{a} \rightarrow (\texttt{a} \rightarrow \texttt{a}) \rightarrow \texttt{nat} \rightarrow \texttt{a}
val succ : \texttt{nat} \rightarrow \texttt{nat}
val zero : \texttt{nat}
structure AbsNat =
  \texttt{IntNat} :> sig
type nat
val zero : nat
val succ : nat \rightarrow nat
val iter : \texttt{a} \rightarrow (\texttt{a} \rightarrow \texttt{a}) \rightarrow \texttt{nat} \rightarrow \texttt{a}

\texttt{structure} AbsNat =
  \texttt{IntNat} :> sig
type nat
val zero : nat
val succ : nat \rightarrow nat
val iter : \texttt{a} \rightarrow (\texttt{a} \rightarrow \texttt{a}) \rightarrow \texttt{nat} \rightarrow \texttt{a}
```

Listing signatures to hide the identity of types.
components.

In general, abstractions can also prune and specialize.

is not a natural number in our representation.

is what we want, since \( \sim 3 \) has type \( \text{int} \), not \( \text{AbsNat.nat} \). This is what we want, since \( \sim 3 \) is ill-typed: \( \sim 3 \) has type \( \text{int} \), not \( \text{AbsNat.nat} \) through the operations, zero, succ,

the only way to construct and use values of the abstract.

\text{AbsNat} defines an abstract dataype of natural numbers:

\text{AbsNat} is just \text{IntNat}, but with a hidden type

hidden, so that \text{AbsNat.nat} \neq \text{Int}.
val \texttt{it} = 0 : \texttt{int} \\
val \texttt{MyOpaqueStack}0 = - : \texttt{int} \texttt{MyOpaqueStack}.\texttt{reptype} \\
val \texttt{MyOpaqueStack}1 = - : \texttt{int} \texttt{MyOpaqueStack}.\texttt{reptype} \\
val \texttt{MyOpaqueStack}0' = - : \texttt{int} \texttt{MyOpaqueStack}.\texttt{reptype} \\
val \texttt{MyEmptyOpaqueStack} = - : 'a \texttt{MyOpaqueStack}.reptype \\
val \texttt{MyEmptyOpaqueStack} = - : \texttt{int} \texttt{MyOpaqueStack}.reptype \\
val \texttt{MyEmptyOpaqueStack} = - : \texttt{int} \texttt{MyOpaqueStack}.reptype \\
val \texttt{MyEmptyOpaqueStack} = - : \texttt{int} \texttt{MyOpaqueStack}.reptype \\
val \texttt{MyEmptyOpaqueStack} = - : \texttt{int} \texttt{MyOpaqueStack}.reptype \\

\texttt{structure MyOpaqueStack :> STACK = MyStack ;}

1. Opaque signature constraints
struct exception E
abstype 'a reptype = S of 'a list (* <--HIDDEN */

val new = S []
fun push x (S s) = S (x::s)
fun pop (S []) = raise E
| pop (S(_::t)) = S t
fun top (S []) = raise E
| top (S(h::_)) = h
end

(* REPRESENTATION *)
with
(* abstraction 'a reptype = S of 'a list --> HIDDEN *)

exception E

structure MyHiddenStack = STACK

val MyHiddenEmptyStack = MyHiddenStack.new
val MyHiddenStack0 = MyHiddenStack.push 0 MyHiddenEmptyStack
val MyHiddenStack01 = MyHiddenStack.push 1 MyHiddenStack0
val MyHiddenStack0' = MyHiddenStack.pop MyHiddenStack01
MyHiddenStack.top MyHiddenStack0'
val MyHiddenEmptyStack = - : 'a MyHiddenStack.reptype
val MyHiddenStack0 = - : int MyHiddenStack.reptype
val MyHiddenStack01 = - : int MyHiddenStack.reptype
val MyHiddenStack0' = - : int MyHiddenStack.reptype
val it = 0 : int
An SML functor is a structure that takes other structures as parameters. Functors let us write program units that can be combined in different ways. Functors can also express generic algorithms.
Functors also support parameterised structures, called functors. Example: The functor `AddFun` below takes any implementation, `N`, of naturals and re-exports its implementation, `N`, of naturals with an addition operation.

```
functor AddFun(N:NAT) =
  struct
    structure Nat = N
    fun add n m = Nat.iter n (Nat.succ) m
  end
```

Modules also support parameterised structures, called functors.
A functor is a function mapping a formal argument structure to a concrete result structure. The body of a functor may assume no more information about its formal argument than is specified in its signature. In particular, opaque types are treated as distinct type parameters. Each actual argument can supply its own, independent implementation of opaque types.
Functor application

A functor may be used to create a structure by applying it to

structure IntNatAdd = AddFun(IntNat)

The actual argument must match the signature of the formal parameter—so it can provide more components, or more general types.

above, AddFun is applied twice, but to arguments that differ in their implementation of type nat (AbsNat.nat \neq IntNat.nat).

Above, AddFun is applied twice, but to arguments that differ in their implementation of type nat (AbsNat.nat \neq IntNat.nat).
Example: Generic Imperative stacks.

```
signature STACK =
  sig
    type itemtype
    val push : itemtype -> unit
    val pop : unit -> unit
    val top : unit -> itemtype
  end;

structure STACK =
  EXEMPLE.
```
exception E;

functor Stack(T: sig type atype end) : STACK =
struct
  type itemtype = T.atype
  val stack : ref([]:itemtype list)
  fun push x = (stack := x :: !stack)
  fun pop () = case !stack of
                              [] => raise E
                          | _ :: s => !s
  fun top () = case !stack of
                [] => raise E
              | t :: _ => t
end

fun pop () = case !stack of
                  [] => raise E
               | _ :: s => !s
fun top () = case !stack of
               [] => raise E
              | t :: _ => t
fun push x = case !stack of
              [] => raise E
          | _ :: x :: !stack => !stack := x :: !stack

val stack = ref([]:itemtype list)
type itemtype = T.atype
struct
  functor Stack(T: sig type atype end) = STACK
exception E;
structure intStack = Stack(struct type = int end) ;
structure intStack : STACK
intStack.push(0) ;
intStack.top() ;
intStack.push(4) ;
intStack.pop() ;
intStack.top() ;
intStack.push(0) ;
val it = [4,3,2,1] : intStack.itemtype list
val it = [()] : unit list
val it = [1,2,3,4] : intStack.itemtype list

fn _ => let val top = intStack.top()
in intStack.pop() ; top end

map ( fn - => let val top = intStack.top() )
map ( push ( intStack ) )
Why functors?

Functors support:
- Code reuse.
  AddFun can be applied to different types \( \text{N.nat} \).
- Type abstraction.
  AddFun can be compiled before any argument is implemented.
- Code abstraction.
  AddFun can be compiled before any structure, reusing its body.
- Code reuse.
  AddFun may be applied many times to different types.

Why functors?