~ Topic VII ~

Data abstraction and modularity
SML Modules

References:
♦ Chapter 7 of *ML for the working programmer* (2ND EDITION) by L. C. Paulson. CUP, 1996.

---

The Core and Modules languages

SML consists of two sub-languages:
♦ The *Core* language is for *programming in the small*, by supporting 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 *Core* expresses details of *data structures* and *algorithms*. The *Modules* language expresses *software architecture*. Both languages are largely independent.

---

[A useful introduction to SML standard libraries, and a good example of modular programming.]

♦ (http://www.standardml.org/)

---

The Modules language

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

```
val zero = 0
fun succ x = x + 1
fun iter b f i = |
  if i = zero then b
  else f (iter b f (i-1))

... (* thousands of lines later *)
fun even (n:nat) = iter true not n
```

The SML Modules language lets one split large programs into separate units with descriptive interfaces.
SML Modules
Signatures and structures

- An abstract data type is a type 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.
- Structures let us package up declarations of related types, values, and functions.
- Signatures let us specify what components a structure must contain.

The dot notation

One can name a structure by binding it to an identifier.

```sml
structure IntNat =
  struct
    type nat = int
    ...
    fun iter b f i = ...
  end
```

Components of a structure are accessed with the dot notation.

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

NB: Type IntNat.nat is statically equal to int. Value IntNat.iter dynamically evaluates to a closure.

Structures

In Modules, one can encapsulate a sequence of Core type and value definitions into a unit called a structure. We enclose the definitions in between the keywords `struct ... end`.

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

```sml
struct
  type nat = int
  val zero = 0
  fun succ x = x + 1
  fun iter b f i = if i = zero then b
                    else f (iter b f (i-1))
end
```

Nested structures

Structures can be nested inside other structures, in a hierarchy.

```sml
structure IntNatAdd =
  struct
    structure Nat = IntNat
    fun add n m = Nat.iter m Nat.succ n
    end
    ...
    fun mult n m =
      IntNatAdd.Nat.iter IntNatAdd.Nat.zero (IntNatAdd.add m) n
  end
```

Concrete signatures

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

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

```plaintext
sig  
  type nat = int
  val zero : nat
  val succ : nat -> nat
  val 'a iter : 'a -> ('a->'a) -> nat -> 'a
end
```

This signature fully describes the *type* of `IntNat`.

The specification of type `nat` is *concrete*: it must be `int`.

Opaque signatures

On the other hand, the following signature

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

specifies structures that are free to use *any* implementation for type `nat` (perhaps `int`, or `word`, or some recursive datatype).

This specification of type `nat` is *opaque*.

**Example**: Polymorphic functional stacks.

```plaintext
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;

structure MyStack: STACK =
  struct
    exception E;
    type 'a reptype = 'a list;
    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;
```
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 it = 0 : int

**Signature inclusion**

To avoid nesting, one can also directly *include* a signature identifier:

```
sig include NAT
  val add: nat -> nat ->nat
end
```

**NB:** This is equivalent to the following signature.

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

**Named and nested signatures**

Signatures may be *named* and referenced, to avoid repetition:

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

**Nested** signatures specify named sub-structures:

```
signature Add =
  sig structure Nat: NAT (* references NAT *)
    val add: Nat.nat -> Nat.nat -> Nat.nat
  end
```

**Signature matching**

**Q:** When does a structure satisfy a signature?

**A:** The type of a structure *matches* a signature whenever it implements at least the components of the signature.

- The structure must *realise* (i.e. define) all of the opaque type components in the signature.
- The structure must *enrich* this realised signature, component-wise:
  - every concrete type must be implemented equivalently;
  - every specified value must have a more general type scheme;
  - every specified structure must be enriched by a substructure.
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 components of more general types, than are specified in a matching signature.

Signature matching is structural. A structure can match 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 number of different roles.

Subtyping

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

- A structure with more type, value, and structure components may be used where fewer components are expected.
- A value component may have a more general type scheme than expected.

Using signatures to restrict access

The following structure uses a signature constraint to provide a restricted view of IntNat:

```latex
structure ResIntNat =
  IntNat : sig type nat
    val succ : nat->nat
    val iter : nat->(nat->nat)->nat->nat
end
```

**NB:** The constraint `str:sig` prunes the structure `str` according to the signature `sig`:

- `ResIntNat.zero` is `undefined`;
- `ResIntNat.iter` is `less` polymorphic that `IntNat.iter`.

Transparency of _:_

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!
In SML, we can limit outside access to the components of a structure by constraining its signature in transparent or opaque manners.

Further, we can hide the representation of a type by means of an abstype declaration.

The combination of these methods yields abstract structures.

Now, the actual implementation of AbsNat.nat by int is hidden, so that AbsNat.nat ≠ int.

AbsNat is just IntNat, but with a hidden type representation.

AbsNat defines an abstract datatype of natural numbers: the only way to construct and use values of the abstract type AbsNat.nat is through the operations, zero, succ, and iter.

For example, the application AbsNat.succ(~3) is ill-typed: ~3 only has type int, not AbsNat.nat. This is what we want, since ~3 is not a natural number in our representation.

In general, abstractions can also prune and specialise components.

Using signatures to hide types identities

With different syntax, signature matching can also be used to enforce data abstraction:

```
structure AbsNat =
  IntNat :> sig type nat
  val zero: nat
  val succ: nat->nat
  val 'a iter: 'a->('a->'a)->nat->'a
end
```

The constraint str :> sig prunes str but also generates a new, abstract type for each opaque type in sig.

Opaque signature constraints

```
structure MyOpaqueStack :> STACK = MyStack ;
val MyEmptyOpaqueStack = MyOpaqueStack.new ;
val MyOpaqueStack0 = MyOpaqueStack.push 0 MyEmptyOpaqueStack ;
val MyOpaqueStack01 = MyOpaqueStack.push 1 MyOpaqueStack0 ;
val MyOpaqueStack0' = MyOpaqueStack.pop MyOpaqueStack01 ;
MyOpaqueStack.top MyOpaqueStack0' ;
val MyEmptyOpaqueStack = - : 'a MyOpaqueStack.reptype
val MyOpaqueStack0 = - : int MyOpaqueStack.reptype
val MyOpaqueStack01 = - : int MyOpaqueStack.reptype
val MyOpaqueStack0' = - : int MyOpaqueStack.reptype
val it = 0 : int
```
Datatype and exception specifications

Signatures can also specify datatypes and exceptions:

```sml
structure PredNat =
  struct
    datatype nat = zero | succ of nat
    fun iter b f i = ...
    exception Pred
    fun pred zero = raise Pred
      | pred (succ n) = n
  end

  sig
    datatype nat = zero | succ of nat
    val iter: 'a->('a->'a)->(nat->'a)
    exception Pred
    val pred: nat -> nat (* raises Pred *)
  end
```

This means that clients can still pattern match on datatype constructors, and handle exceptions.

SML Modules

Functors

♦ 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

Modules also supports parameterised structures, called **functors**.

**Example:** The functor `AddFun` below takes any implementation, `N`, of naturals and re-exports it with an addition operation.

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

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 an actual argument:

```
structure IntNatAdd = AddFun(IntNat)
structure AbsNatAdd = AddFun(AbsNat)
```

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

Above, `AddFun` is applied twice, but to arguments that differ in their implementation of type `nat (AbsNat.nat ≠ IntNat.nat)`.  

**Example:** Generic imperative stacks.

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

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 => ( stack := s )
    fun top()
        = case !stack of [] => raise E
         | t::_ => t
end;

structure intStack
    = Stack(struct type atype = int end) ;

structure intStack : STACK
    intStack.push(0) ;
    intStack.top() ;
    intStack.pop() ;
    intStack.push(4) ;
    val it = () : unit
    val it = 0 : intStack.itemtype
    val it = () : unit
    val it = () : unit

map ( intStack.push ) [3,2,1] ;
map ( fn _ => let val top = intStack.top() in intStack.pop(); top end )
    [(),(),(),()] ;
val it = [(),(),(),()] : unit list
val it = [1,2,3,4] : intStack.itemtype list

Why functors ?

Functors support:

  Code reuse.
  AddFun may be applied many times to different structures, reusing its body.

  Code abstraction.
  AddFun can be compiled before any argument is implemented.

  Type abstraction.
  AddFun can be applied to different types N.nat.
Are signatures types?

The syntax of Modules suggests that signatures are just the types of structures ... but signatures can contain opaque types.

In general, signatures describe families of structures, indexed by the realisation of any opaque types.

The interpretation of a signature really depends on how it is used!

In functor parameters, opaque types introduce polymorphism; in signature constraints, opaque types introduce abstract types.

Since type components may be type constructors, not just types, this is really higher-order polymorphism and abstraction.

Structures as records

Structures are like Core records, but can contain definitions of types as well as values.

What does it mean to project a type component from a structure, e.g. IntNatAdd.Nat.nat?

Does one need to evaluate the application AddFun(IntNat) at compile-time to simplify IntNatAdd.Nat.nat to int?

No! It's sufficient to know the compile-time types of AddFun and IntNat, ensuring a phase distinction between compile-time and run-time.

Type propagation through functors

Each functor application propagates the actual realisation of its argument's opaque type components.

Thus, for

structure IntNatAdd = AddFun(IntNat)
structure AbsNatAdd = AddFun(AbsNat)

the type IntNatAdd.Nat.nat is just another name for int, and AbsNatAdd.Nat.nat is just another name for AbsNat.nat.

Examples:

- IntNatAdd.Nat.succ(0) √
- IntNatAdd.Nat.succ(IntNat.Nat.zero) √
- AbsNatAdd.Nat.succ(AbsNat.Nat.zero) √
- AbsNatAdd.Nat.succ(0) ×
- AbsNatAdd.Nat.succ(IntNat.Nat.zero) ×

Generativity

The following functor almost defines an identity function, but re-abstracts its argument:

functor GenFun(N:NAT) = N :> NAT

Now, each application of GenFun generates a new abstract type: For instance, for

structure X = GenFun(IntNat)
structure Y = GenFun(IntNat)

the types X.nat and Y.nat are incompatible, even though GenFun was applied to the same argument.

Functor application is generative: abstract types from the body of a functor are replaced by fresh types at each application. This is consistent with inlining the body of a functor at applications.
**Why should functors be generative?**

It is really a design choice. Often, the invariants of the body of a functor depend on both the types *and values* imported from the argument.

```ocaml
functor OrdSet(O: sig type elem
          val compare: (elem * elem) -> bool
       end) = struct
  type set = O.elemlist (* ordered list of elements *)
  val empty = []
  fun insert e [] = [e]
   | insert e1 (e2::s) = if O.compare(e1,e2)
      then if O.compare(e2,e1) then e2::s else e1::e2::s
      else e2::insert e1 s
  end :> sig type set
    val empty: set
    val insert: O.elem -> set -> set
  end
```

For

```ocaml
structure S = OrdSet(struct type elem=int
       fun compare(i,j)= i <= j end)
structure R = OrdSet(struct type elem=int
       fun compare(i,j)= i >= j end)
```

we want \( S.set \neq R.set \) because their representation invariants depend on the compare function: the set \( \{1, 2, 3\} \) is \( \{1,2,3\} \) in \( S.set \), but \( \{3,2,1\} \) in \( R.set \).

**Why functors?**

- Functors let one decompose a large programming task into separate subtasks.
- The propagation of types through application lets one extend existing abstract data types with type-compatible operations.
- Generativity ensures that applications of the same functor to data types with the same representation, but different invariants, return distinct abstract types.

**Sharing specifications**

Functors are often used to combine different argument structures.

Sometimes, these structure arguments need to communicate values of a *shared* type.

For instance, we might want to implement a sum-of-squares function \((n, m \mapsto n^2 + m^2)\) using separate structures for naturals with addition and multiplication ...
Sharing violations

functor SQ(structure AddNat: sig
  structure Nat: sig type nat end
  val add: Nat.nat -> Nat.nat -> Nat.nat
end
structure MultNat: sig
  structure Nat: sig type nat end
  val mult: Nat.nat -> Nat.nat -> Nat.nat
end) =
  struct fun sumsquare n m
    = AddNat.add (MultNat.mult n n) (MultNat.mult m m) \times 
  end

The above piece of code is *ill-typed*: the types AddNat.Nat.nat and MultNat.Nat.nat are opaque, and thus different. The add function cannot consume the results of mult.

Sharing specifications

The fix is to declare the type sharing directly at the specification of MultNat.Nat.nat, using a concrete, not opaque, specification:

functor SQ(
  structure AddNat: sig
    structure Nat: sig type nat end
    val add: Nat.nat -> Nat.nat -> Nat.nat
  end
  structure MultNat: sig
    structure Nat: sig type nat = AddNat.Nat.nat end
    val mult: Nat.nat -> Nat.nat -> Nat.nat
  end
) =
  struct fun sumsquare n m
    = AddNat.add (MultNat.mult n n) (MultNat.mult m m) √
  end

Sharing constraints

Alternatively, one can use a post-hoc *sharing specification* to identify opaque types.

functor SQ(
  structure AddNat: sig structure Nat: sig type nat end
  val add: Nat.nat -> Nat.nat -> Nat.nat
end
structure MultNat: sig structure Nat: sig type nat end
  val mult: Nat.nat -> Nat.nat -> Nat.nat
  sharing type MultNat.Nat.nat = AddNat.Nat.nat
end
  struct fun sumsquare n m
    = AddNat.add (MultNat.mult n n) (MultNat.mult m m) √
  end

Limitations of modules

Modules is great for expressing programs with a complicated static architecture, but it’s not perfect:

- Functors are *first-order*: unlike Core functions, a functor cannot be applied to, nor return, another functor.

- Structure and functors are *second-class* values, with very limited forms of computation (dot notation and functor application): modules cannot be constructed by algorithms or stored in data structures.

- Module definitions are *too sequential*: splitting mutually recursive types and values into separate modules is awkward.