# Appendix to Lecture VII

## An introduction to SML Modules by Claudio Russo[a]

**References:**

- *ML for the Working Programmer* by Larry Paulson, Cambridge University Press. [A textbook for undergraduates and postgraduates.]

[a] ⟨http://research.microsoft.com/~crusso⟩



- *The Standard ML Basis Library* by Reppy *et al.*, Cambridge University Press. [A useful introduction to ML standard libraries, and a good example of Modular programming.]
- *The Definition of Standard ML* by Milner *et al.*, MIT Press. [A formal definition of SML, using structured operational semantics. Useful for language implementors and researchers.]
- *Purely Functional Data Structures* by Chris Okasaki, Cambridge University Press. [Contains clever functional data structures, implemented in Haskell and SML Modules.]
- ⟨http://www.standardml.org⟩



## Outline

*Aim:* To provide a gentle introduction to SML Modules.

- Review Core features related to Modules.
- Introduce the Modules Language, using small examples.
- Briefly relate Modules constructs to the Core language.
- Highlight some limitations of Modules.

**NB:** Only the important features of Modules are covered.



## 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.

# The Core language

The SML Core is a strongly-typed call-by-value functional language with impure features (state and exceptions).

Types are mostly implicit and inferred by the compiler.

SML programs must be statically well-typed before being evaluated.

The Core is *type sound*: evaluation of a well-typed expression is guaranteed to be free of run-time type errors.



# Core features

The SML Core has a number of other features:

- a rich collection of primitive types (e.g. `int`, `real`, `Int16.int`, `Word32.word`);
- mutually recursive polymorphic functions and datatypes;
- dynamically allocated, mutable references (type `'a ref`);
- exceptions;
- pattern matching on values.

Most of these features have little or no interaction with Modules.



# The Modules language

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

```
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))
...
(* 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.



# 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.

```
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
```

# The dot notation

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

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

Components of a structure are accessed with the *dot notation*.

```
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.



# Nested structures

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

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

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

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

Nesting and dot notation provides *name-space* control.



# Structure inclusion

To avoid nesting structures and dot notation, one can also directly `open` a structure identifier, importing its components:

```
struct   open Nat
         fun add n m = iter m succ n   end
```

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

```
struct   type nat = Nat.nat
         val zero = Nat.zero
         val succ = Nat.succ
         val iter = Nat.iter
         fun add n m = iter m succ n   end
```

Though convenient, it's bad style: the origin of an identifier is no longer clear and bindings are silently re-exported.



# 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`.

```
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

```
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*.



# 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 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
```



# 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 . . .



# Using signatures to restrict access

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

```
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!



# 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`.

Now, the actual implementation of `AbsNat.nat` by `int` is *hidden*, so that `AbsNat.nat` $\neq$ `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.



# Datatype and exception specifications

Signatures can also specify datatypes and exceptions:

```
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.



# 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` $\neq$ `IntNat.nat`).



# 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`.



# 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)` $\surd$
- `IntNatAdd.Nat.succ(IntNat.Nat.zero)` $\surd$
- `AbsNatAdd.Nat.succ(AbsNat.Nat.zero)` $\surd$
- `AbsNatAdd.Nat.succ(0)` $\times$
- `AbsNatAdd.Nat.succ(IntNat.Nat.zero)` $\times$



# 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 needs to evaluate the application `AddFun(IntNat)` at *compile-time* to simplify `IntNatAdd.Nat.nat` to `int`?

**No!** Its sufficient to know the *compile-time* types of `AddFun` and `IntNat`, ensuring a *phase distinction* between compile-time and run-time.

# 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.

```
functor OrdSet(O:sig type elem
                     val compare: (elem * elem) -> bool
                 end) = struct
  type set = O.elem list (* 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

```
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.

# 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.



# 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.



# 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) ✗
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
                         end
  sharing type MultNat.Nat.nat = AddNat.Nat.nat) =
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.