Introduction to SML
Modules (chapter 11)

Anne E. Haxthausen

ah@imm.dtu.dk

Computer Science and Engineering
Informatics and Mathematical Modelling
Technical University of Denmark

Learning Objectives

- to create and use SML modules

Purpose of modules

Modules are used to organize programs in order to

- increase readability
- allow for reuse

SML module entities:

- *a structure* = a collection of declarations
- *a signature* = a collection of specifications
  used as interfaces for structures
- *a functor* = a parameterized structure
  (denotes a function from structures to structures)

Declaring structures

structure id = struct dec_1; ...; dec_n end

Example

```
structure Set =
struct
   type 'a set = 'a list;
   val empty = [];
   fun singleton a = [a];
   ...
end;
```

Using structures: the dot notation and open

After a declaration

`structure sid = struct ... id ... end`

entities declared with name `id` can be accessed by the name `sid.id`

`open sid` implies that `id` can be directly used without prefix "`sid.`"

Example

```
structure Set =
struct
   type 'a set = 'a list;
   val empty = [];
   ...
end;

Set.empty;  (* empty is not allowed *)

open Set;

empty;  (* now empty is allowed *)
```


Signatures

A *signature* is the users interface to a structure: it specifies the names and types of those entities that should be public.

`signature id = sig` $spec_1$`; ...;` $spec_n$ `end`

Example

```
signature Set =
sig
   type 'a set
   val empty     : ''a set
   val singleton : ''a -> ''a set
   ...
end
```

can be used as interface for the structure `Set`.

Ascribing a signature to a structure

Syntactically two ways:

- *Opaque*: `structure sid :> sigexp = strexp`
- *Transparent*: `structure sid : sigexp = strexp`

Consequences of ascribing a signature to a structure:

- At declaration time: checked that `strexp` *matches* `sigexp`.
- At application time: the access to the entities in `strexp` is restricted. Exactly how depends on the kind of ascription.

Rules for matching

`strexp` *matches* `sigexp` if:

- each component specified in the signature is declared with the same name in the structure (– there may be components that are declared in the structure, but not specified in the signature)
- the type of an identifier in the signature must be an instance of the type inferred for that identifier in the structure

Signature ascription can hide components

If the signature does not specify a component that is declared in the structure, then the identifier becomes private.

Example

```
signature T =
  sig val f : int -> int end

structure T :> T =
  struct
    fun aux x = x+1;
    fun f(x) = aux(x+7)
  end

T.f(3);  (* allowed *)

T.aux(10) (* not allowed as aux is not exported *)
```


Signature ascription can enforce less polymorpic types

Let ty be the specified type of id in sigexp, and ty′ be a type inferred for id in strexp.
If ty is a proper instance of ty′ this influences the type checking.

Example

```
structure intSet1 =
  struct
    fun singleton a = [a]
  end;

intSet1.singleton(true); (* ok, as intSet1.singleton : 'a -> 'a list *)

signature intSet =
  sig
    val singleton : int -> int list
  end;

structure intSet2 :> intSet =
  struct
    fun singleton a = [a]
  end;

intSet2.singleton(true); (* not ok, as intSet2.singleton : int -> int list *)
```


Opaque signature ascription can hide type representations

If the signature gives a specification `type id` or `type tvl id` and the structure the implementation `type id = ty` or `type tvl id = ty`, then *opaque* signature ascription hides the type representation `ty`, i.e. the type named `id` becomes *abstract*.

Example

```
signature SetSig =
  sig
    type 'a set
    val empty     : ''a set
    val singleton : ''a -> ''a set
  end;
structure Set =
  struct
    type 'a set = 'a list;
    val empty = [];
    fun singleton a = [a]
  end;
structure Set2 :> SetSig = Set;

Set2.singleton(2) = [2]; (* not ok, the list representation is hidden *)
```


Transparent signature ascription does *not* hide type representations

Example

```
signature SetSig =
  sig
    type 'a set
    val empty     : ''a set
    val singleton : ''a -> ''a set
  end;
structure Set =
  struct
    type 'a set = 'a list;
    val empty = [];
    fun singleton a = [a]
  end;
structure Set1 : SetSig = Set;

Set1.singleton(2) = [2]; (* ok *)
```

SML libraries

Put Set signature declaration in file "Set.sig"

Put Set structure declaration in file "Set.sml"

In an interactive SML session:

```
compile  ``Set.sig'';
compile  ``Set.sml'';
```

Produces the files "Set.ui" "Set.uo".

To use the library in an interactive session:

```
load ``Set'';
```


Functor declarations

```
functor fid (strid :  ArgSig) :> ResSig = strexp
```

Example

```
signature EQ =
  sig
    type element
    val eq: element * element -> bool
  end
```

```
signature SetEqPr =
  sig
    type element
    type set
    val empty : set
    val singleton: element -> set
    val member: element * set -> bool
    ...
  end
```

```
functor SetFctPr(s : EQ) :> SetEqPr where type element = s.element =
  struct
    type element = s.element
    type set = s.element list
    val empty = [];
    fun singleton x = [x];
    fun member(x, ys) = List.exists (fn y => s.eq(x,y)) ys
    ...
  end
```


Functor applications

```
functor fid (strid :  ArgSig) :> ResSig = strexp
```

Application of `fid`: `fid (`$\texttt{strexp}_{act}$`)`

Requirement: $\texttt{strexp}_{act}$ `must match ArgSig`

Example

```
structure intset = SetFctPr(struct type element = int val eq = op= end);

val is = intset.empty;

structure intlistset =
  SetFctPr( struct type element = int list val eq = op= end );

val ils = intlistset.empty;
```