Introduction to SML
Sets (chapter 10)

Anne E. Haxthausen

ah@imm.dtu.dk

Computer Science and Engineering

Informatics and Mathematical Modelling

Technical University of Denmark

Learning Objectives

- The set concept
- An SML implementation of sets
- Modelling/programming with sets

The mathematical concept of sets

A *set* is a collection of elements where the order of elements and repetitions are of no concern.

Example (in standard math notation):

$$\{1,5,2\} = \{1,1,5,2\} = \{1,2,5\}$$

We are looking for an SML representations for sets of elements of the same type.

An SML set type constructor: `set`

If τ is a type, so is τ `set`

$\boxed{\tau\ \texttt{set}}$ is the type of all values representing finite sets $\{e_1, \ldots, e_n\}$ having elements e_i of type τ.

Examples:

- `int set`
- `(string * int list) set`

An implementation of the `set` type constructor

A set $\{e_1, \ldots, e_n\}$ can be represented by a list of its elements:
`[`e_1`, ..., `e_n`]`
satisfying the invariant that $e_i \neq e_j$ for $i \neq j$.

Note: a set can be represented by more than one list.

Declaration of a set type constructor `set`:

```
type 'a set = 'a list
```

An SML set module `Set` in appendix E provides this type constructor and associated polymorphic set operations.

Set operations, classification

basic constructors: `empty, singleton, fromList`

modifiers: `insert, delete, filter, map`

set combinators: `union, inter, diff`

single set observers:

- non Boolean: `toList, card`
- Boolean: `member, exists, all`

double set observers: `equal, subset`

others: `fold, find, split`

Below we show types and meaning.
For their implementation: see appendix E.

Set operations: types

```
val empty     : ''a set
val singleton : ''a                -> ''a set
val fromList  : ''a list -> ''a set
val insert    : ''a * ''a set      -> ''a set
val delete    : ''a set * ''a      -> ''a set
val filter    : (''a -> bool) -> ''a set -> ''a set
val map       : (''a -> ''b) -> ''a set -> ''b set
val union     : ''a set * ''a set -> ''a set
val inter     : ''a set * ''a set -> ''a set
val diff      : ''a set * ''a set -> ''a set
val toList    : ''a set -> ''a list
val card      : ''a set -> int
val member    : ''a * ''a set      -> bool
val exists    : (''a -> bool) -> ''a set -> bool
val all       : (''a -> bool) -> ''a set -> bool
val equal     : ''a set * ''a set -> bool
val subset    : ''a set * ''a set -> bool
val fold      : (''a * 'b -> 'b) -> 'b -> ''a set -> 'b
val find      : (''a -> bool) -> ''a set -> ''a option
val split     : ''a set -> (''a * ''a set) option
```


Set operations: meaning

SML expr	represents (when e, e_i, s, s_i represents e, e_i, s, s_i)
`empty`	$\{\}$
`singleton e`	$\{e\}$
`fromlist [`e_1`, ..., `e_n`]`	$\{e_1, ..., e_n\}$
`insert(e, s)`	$\{e\} \cup s$
`delete(s, e)`	$s \setminus \{e\}$
`filter p s`	$\{e \mid e \in s : p(e)\}$
`map f s`	$\{f(e) \mid e \in s\}$
s_1 `union` s_2	$s_1 \cup s_2$
s_1 `inter` s_2	$s_1 \cap s_2$
s_1 `diff` s_2	$s_1 \setminus s_2$
`card [`e_1`, ..., `e_n`]`	n
`e member s`	$e \in s$
`exists p s`	$\exists e \in s : p(e)$
`all p s`	$\forall e \in s : p(e)$
s_1 `equal` s_2	$s_1 = s_2$
s_1 `subset` s_2	$s_1 \subseteq s_2$
`fold f b s`	$f(e_1, ..., f(e_{n-1}, f(e_n, b))...)$ for $s = \{e_1, ..., e_n\}$

Modelling using sets: conference example

```
datatype country   = DK | UK | ES | SF
                   | PRC | C | D | IS | S | N;
type name          = string;
type title         = string;
type author        = name * country;
type contribution = title * author list;
type conference    = contribution Set.set;

val c1 = ("Processors", [("Lu",PRC), ("Olsen",DK)]);
val c2 = ("Scheduling", [("Suonio",SF),("Weber",D)]);
val c3 = ("RLucid", [("Plaice",C)]);
val c4 = ("Timed Observations", [("Ortega",ES)]);

val ourConf =  Set.fromList [c1, c2, c3, c4];
```


Modelling using sets: conference example

What are the nationalities of authors of a given contribution?

```
(* nationalitiesOf : contribution -> country set *)
fun nationalitiesOf(_,al) =
   Set.fromList (List.map (fn (_,ct)=> ct) al);
```

Test:

```
- Set.toList(nationalitiesOf c1);
> val it = [DK, PRC] : country list
```


Modelling using sets: conference example

For a given conference, how many contributions are co-authered by Nordic authors?

```
val nordic: country Set.set = Set.fromList[IS,DK,S,N,SF]

(* haveNordic : contribution -> bool*)
fun haveNordic c
   = not ( Set.equal
             (Set.inter(nationalitiesOf c, nordic),
              Set.empty)
         );
(* numNordic : conference -> int *)
fun numNordic conf =
   Set.card (Set.filter haveNordic conf);
```

Test:

```
- numNordic ourConf;
> val it = 2 : int
```


Modelling using sets: conference example

For a given conference, what is the title and the name of the first author of every conference contribution?

```
(* firstAuthors : conference -> (title * name) set *)
fun firstAuthors conf =
  Set.map (fn (t,(n,_)::_) => (t,n)) conf;
```

Test:

```
- Set.toList( firstAuthors ourConf);
> val it =
    [("Processors", "Lu"),
     ("Scheduling", "Suonio"),
     ("RLucid", "Plaice"),
     ("Timed Observations", "Ortega")]
    : (string * string) list
```

Modelling using sets: conference example

For a given conference, what are the countries of the conference contributions?

```
(* countries : conference -> country set *)
fun countries conf =
  Set.fold (fn (c,s) => Set.union(nationalitiesOf c, s))
           Set.empty
           conf;
```

Test:

```
- Set.toList( countries ourConf );
> val it = [PRC, DK, SF, D, C, ES] : country list
```


Set equality and sets of sets

Fact 1: Types τ `Set.set` are abstract and hence not equality types. I.e. τ `Set.set` is not an instance of `''a` .

Therefore you can't use `= : ''a * ''a -> bool` to compare sets. Use `equal : ''a set * ''a set -> bool` instead.

Fact 2: All `Set` functions require that the set element types are equality types.

So you can't apply them to sets of sets.

They require equality types as otherwise the functions may give wrong results, when elements that are sets are compared in function bodies with $=$ instead of `equal`.

`equal` takes into account that two lists that are permuations of each other represent the same set, but list equality ($=$) does not.

Another implementation of `set`

Declaration of another set type constructor `set`:

```
type 'a set = 'a list * ('a * 'a -> bool)
```

A set with equality:
`([`e_1`, ..., `e_n`], eq)`
must satisfy the invariant that `eq(`e_i, e_j`) = false` for $i \neq j$

All set operations are changed to take the equality predicate into account.