Introduction to SML
Tagged Values I (Chapter 7)

Michael R. Hansen and Anne E. Haxthausen

mrh@imm.dtu.dk and ah@imm.dtu.dk

Informatics and Mathematical Modelling

Technical University of Denmark

Overview of chapters 7 and 8

Main topics:

- *`datatype` declarations* and their use
  - non recursive (chap. 7)
  - recursive (chap. 8)
- *`abstype` declarations* and their use (chap. 8)

Additional topics (chap. 7):

- *`case` expressions*
- *patterns*
- *exceptions*
- *partial functions*

This slide set covers topics from chapter 7.

Motivation for disjoint union types

Task: define a representation for a collection of shapes.

Representation of different shapes:

```
type Circle = {radius: real}
type Square = {side: real}
type Triangle = {side1:real,side2:real,side3:real}
```

How can we represent a collection of different shapes?

In a list all elements must have the same type, so we must make a common type `Shape` to use lists:

```
type shapeCollection = Shape list
```

How can we define the common `Shape` type?

Motivation for disjoint union types

A *bad* solution for a common shape type:

```
type Shape =
  { kind : string,
    radius: real,
    side : real,
    side1 : real, side2 : real, side3 : real}
```

A *good* solution: define `Shape` as a disjoint union of the `Circle`, `Square` and `Triangle` representations.

Union types can be defined by SML `datatype` declarations.

Disjoint Union: An Example

A *shape* is either a circle, a square, or a triangle

- the union of three disjoint sets

A *datatype* declaration for shapes:

```
datatype shape = Circle of real
               | Square of real
               | Triangle of real*real*real;
```

Answer from the SML system:

```
> datatype  shape
>   con Circle = fn : real -> shape
>   con Square = fn : real -> shape
>   con Triangle = fn : real * real * real -> shape
```


Constructors of a `datatype`

The *tags* `Circle`, `Square` and `Triangle` are *constructors* of values of type `shape`

```
- Circle 2.0;
> val it = Circle 2.0 : shape

- Triangle(1.0, 2.0, 3.0);
> val it = Triangle(1.0, 2.0, 3.0) : shape

- Square 4.0;
> val it = Square 4.0 : shape
```

Equality on `shape`s is defined provided ...

```
 - Triangle(1.0, 2.0, 3.0) = Square 2.0;
 > val it = false :  bool
```


Constructors in Patterns

```
fun area(Circle r)        = Math.pi * r * r
  | area(Square a)        = a * a
  | area(Triangle(a,b,c)) =
              let val d = (a + b + c)/2.0
              in Math.sqrt(d*(d-a)*(d-b)*(d-c))
              end;
> val area = fn : shape -> real
```

- a constructor only matches itself

```
    area (Circle 1.2)
⇝  (Math.pi * r * r, [r ↦ 1.2])
⇝  ...
```


The `case`-expression

Form:

```
case exp of
      pat1 => e1
    | pat2 => e2
      ...
    | patk => ek
```

Example:

```
fun area s =
   case s of
         (Circle r)        => Math.pi * r * r
       | (Square a)        => a*a
       | (Triangle(a,b,c)) =>
             let val d = (a + b + c)/2.0
             in Math.sqrt(d*(d-a)*(d-b)*(d-c))
             end;
```

Enumeration types – the predefined `order` type

```
datatype order = LESS | EQUAL | GREATER;
```

Predefined 'compare' functions, e.g.

$$\texttt{Int.compare}(x,y) = \begin{cases} \texttt{LESS} & \text{if } x < y \\ \texttt{EQUAL} & \text{if } x = y \\ \texttt{GREATER} & \text{if } x > y \end{cases}$$

Example:

```
fun countLEG []       = (0,0,0)
 |  countLEG(x::rest) =
       let val (y1,y2,y3) = countLEG rest in
          case Int.compare(x,0) of
              LESS    => (y1+1,y2  ,y3  )
            | EQUAL   => (y1  ,y2+1,y3  )
            | GREATER => (y1  ,y2  ,y3+1)
       end;
```


Polymorphic types, the predefined `option` type

Example: `datatype 'a option = NONE | SOME of 'a`

The type `'a option` is parameterized with the element type.
The value constructors are polymorphic:

```
con 'a NONE = NONE : 'a option
con 'a SOME = fn : 'a -> 'a option
```

`SOME 3` has type `int option` which is an *instance* of `'a option`.

Polymorphic types contain type variables: `'a list, 'a option, ...`

Monomorphic types do not contain type variables:
`int, int list, bool list, int option, ...`

Partial functions

A function $f : \tau_1 \rightarrow \tau_2$ is *partial*, if application $f(v)$ is undefined for some value $v : \tau_1$.

Three possibilities for treating $f(v)$ when declaring f in SML:

- let $f(v)$ be undefined (e.g. no match, no termination, ...)

```
fun fact 0 = 1 | fact n = n * fact(n-1);
```

- let $f(v)$ raise an user-defined exception

```
fun fact1 n = if n < 0 then raise BadArgument n
                       else fact n;
```

- let $f(v)$ return a special value (`NONE`) and
  let $f(x)$ return `SOME` y when $f(x)$ is defined to give y

```
fun fact2 n = if n < 0 then NONE
                       else SOME (fact n);
```


Partial functions, example

```
- fact1 3;
> val it = 6 : int

- fact2 3;
> val it = SOME 6 : int option

- fact1 ~1;
! Uncaught exception:
! BadArgument

-  fact2 ~1;
> val it = NONE : int option
```

Exceptions

Exceptions are used to terminate the evaluation of an expression with an “error signal”.

Issues:

- *declaration* of exception constructors $\mathtt{Eid}$:
  $\mathtt{exception}\ \mathtt{Eid}$ or $\mathtt{exception}\ \mathtt{Eid}\ \mathtt{of}\ ty$
- exception *raising*:
  $\mathtt{raise}\ \mathtt{Eid}$ or $\mathtt{raise}\ \mathtt{Eid}\ exp$
- exception *handling* (catching) :
  $exp\ \mathtt{handle}\ pat_1 \Rightarrow exp'_1 \mid \ldots \mid pat_n \Rightarrow exp'_n$
  where pat_i is an exception raising pattern of one of the forms:
  $\mathtt{Eid}_i$ or $\mathtt{Eid}_i\ exp_i$, and exp'_i has same type as exp

Exceptions: examples

Declaring an exception: `exception BadArgument of int;`

Specifying an exception to be *raised*:

```
fun fact1 n = if n < 0 then raise BadArgument n else fact n;
```

No handling of exception:

```
- fact1 ~2;
> ! Uncaught exception:
> ! BadArgument
```

Handling an exception:

```
- fun usefact1 n =
-   ... fact1 n ...
-   handle BadArgument n => "Bad argument "^Int.toString n;

- usefact1 ~2;
> val it = "Bad argument ~2" : string
```