Chapter 9
Conclusion

(Version of 4 January 2005)

9.1. Functional programming in SML

Covered and fundamental elements

- Evaluation by reduction of expressions
- Recursion
- Functions as basic objects
- Higher-order functions
- Polymorphism via type variables
- Strong typing
- Type inference
- Pattern matching
- Definition of new types
- Type and value constructors
- Abstract datatypes
- Modules
- Exceptions and error recovery

Non-covered elements

- Imperative programming aspects,
  such as variables and references, control structures, ...
- Input/output
- Inference techniques

Interest of functional programming in SML

- Fast program development
- Easy representation of new types
- Easy realisation of abstract datatypes
- Power of the functional paradigm
- Power of the SML language itself
- Conciseness of the developed programs

Warning

The apparent ease of program development in SML
does *not* imply that one need not think nor be creative!

9.2. Beyond functional programming

Functional programming

The evaluation of $f(a)$ gives *at most one* result,
and always gives the *same* result

Multifunctional programming

The evaluation of $f(a)$ gives *several* (0, 1, or more) results,
either all-at-once or one-by-one
Example:

multifunction split L
TYPE: α list $\longrightarrow$ (α list $*$ α list)
PRE: (none)
POST: (xs,ys) such that xs @ ys = L

```
fun  split [ ] = ([ ],[ ])
  |  split (x::xs)  =  ([ ],x::xs)
                    || let val (L1,L2) = split xs
                       in  (x::L1,L2) end
```

```
-  split [4,5,2] ;
   val it = ( [] , [4,5,2] ) ;
   val it = ( [4] ,  [5,2] ) ;
   val it = ( [4,5] ,  [2] ) ;
   val it = ( [4,5,2] , [] ) ;
   no other solutions
```

- This feature does *not* exist in SML
- There are very few multifunctional languages

Relational programming (aka logic programming)

Example:

relation append (X,Y,Z)
TYPE: int list $*$ int list $*$ int list
PRE: (none)
POST: Z is the concatenation of X and Y

For which triples does the append relation hold?

```
append ([ ], [ ], [ ])
append ([3], [1,2], [3,1,2])
append ([4,8], [ ], [4,8])
append ([5,0,2,1], [2,3,0], [5,0,2,1,2,3,0])
...
```

- *No* differentiation between arguments and results!
- *Several* possible usages of the *same* program for append:

```
-   append ([1,2], [0,3], [1,2,0,3]).
  Yes

-   append ([1,2], [0,3], [1,5,3]).
  No

-   append ([1,2], [0,3], L).
  L=[1,2,0,3] ;
  No
```

- append (L1, L2, [1,5,3]).

```
L1=[], L2=[1,5,3] ;
L1=[1],  L2=[5,3] ;
L1=[1,5],  L2=[3] ;
L1=[1,5,3], L2=[] ;
No
```

- append (L1, [5,3], [1,5,3]).

```
L1=[1] ;
No
```

- append ([1,5], L2, L3).

```
L3=[1,5|L2] ;
No
```

- append (L1, L2, L3).

```
L1=[], L3=L2 ;
L1=[X], L3=[X|L2] ;
L1=[X,Y], L3=[X,Y|L2] ;
...
```

- append ([1,X,4], [Y|Ys], [1,2,4,3]).

```
X=2, Y=3, Ys=[] ;
No
```

- append ([1,2], [0,3], L), append (L, [4,2], R).

```
L=[1,2,0,3] , R=[1,2,0,3,4,2] ;
No
```

- append (L1, L2, [1,5,3]), L2=[X,Y].

```
L1=[1], L2=[5,3], X=5, Y=3 ;
No
```

- *Backtracking* mechanism to enumerate all the possibilities

How to "program" the append relation?
With relational programming languages: Prolog, Mercury, ...

Example:

```
append ([ ], Ys, Ys) ←
append ([X|Xs], Ys, [X|Zs]) ← append (Xs, Ys, Zs)
```

- Two clauses
- *Unification* mechanism,
  as a generalisation of pattern matching

Interest of relational programming

- Power of the logic paradigm
- Power of the relational framework