

ML datatypes and patterns can make programs much more concise and elegant

1. type abbreviations
2. ML datatypes
3. patterns and local variables

Simple Type Abbreviations

Datatypes

Amtoft
from Hatcliff

Syntax:

```
type ⟨identifier⟩ = ⟨type expression⟩
```

Semantics: `type` makes an **abbreviation**, not a new type.

```
- type signal = int list;  
type signal = int list  
- val v = [1,2,3] : signal;  
val v = [1,2,3] : signal  
- val w = [1,2,3];  
val w = [1,2,3] : int list  
- v = w;  
val it = true : bool  
- hd v;  
val it = 1 : int
```

Type Names

Datatypes

Patterns

Local Definitions

Parametric Type Abbreviations

Datatypes

Amtoft
from Hatcliff

Type Names

Datatypes

Patterns

Local Definitions

Syntax:

```
type (<type parameters>) <identifier> =  
    <type expression>
```

Semantics: a **family** of abbreviations

```
- type ('k, 'v) table = ('k * 'v) list;  
type ('a, 'b) table = ('a * 'b) list  
  
- [ ("bob", 1), ("jane", 2) ]  
  : (string, int) table;  
val it = [ ("bob", 1), ("jane", 2) ]  
      : (string, int) table
```

- ▶ the ML `datatype` facility allows one to create tree structured data
- ▶ very flexible abstract mechanism for representing structured data

Simple Sum Types

Datatypes

Amtoft
from Hatcliff

```
datatype suit =  
    Clubs | Diamonds  
  | Hearts | Spades
```

```
– Clubs;  
val it = Clubs: suit
```

```
datatype card =  
    Club of int | Diamond of int  
  | Heart of int | Spade of int  
  | WildCard
```

```
– Heart 7;  
val it = Heart 7 : card
```

Type Names

Datatypes

Patterns

Local Definitions

Parametrized Sum Types

Datatypes

Amtoft
from Hatcliff

Type Names

Datatypes

Patterns

Local Definitions

```
datatype ('a, 'b) pair_or_single =  
    Pair of 'a * 'b  
  | Single of 'a
```

```
– [Pair("ab",7), Single "cd"];  
val it = [Pair ("ab",7), Single "cd"]  
  : (string, int) pair_or_single list
```

Recursive Types

Datatypes

Amtoft
from Hatcliff

```
datatype int_tree =  
  Leaf of int  
  | Node of int * int_tree * int_tree
```

```
– Node (5, Node(5, Leaf 4, Leaf 7), Leaf 3);  
val it =  
  Node (5,Node (5,Leaf #,Leaf #),Leaf 3) :  
  int tree
```

We may generalize to other types than `int`

```
datatype 'a tree =  
  Lf of 'a  
  | Nd of 'a * 'a tree * 'a tree
```

```
– Lf;  
val it = fn : 'a -> 'a tree
```

Type Names

Datatypes

Patterns

Local Definitions

Parametrized Recursive Types

Datatypes

Amtoft
from Hatcliff

```
datatype 'a tree =  
  Lf of 'a  
  | Nd of 'a * 'a tree * 'a tree  
  
- Lf;  
val it = fn : 'a -> 'a tree  
  
- Nd (3.0, Lf(3.5), Lf(4.2));  
val it = Nd (3.0, Lf 3.5, Lf 4.2) : real tree  
  
- Nd (3, Lf(5), Lf(4));  
val it = Nd (3, Lf 5, Lf 4) : int tree  
  
- Nd (3, Lf(5.0), Lf(4.6));  
  
.. Error: operator and operand don't agree..  
operator domain: int * int tree * int tree  
operand:         int * real tree * real tree
```

Type Names

Datatypes

Patterns

Local Definitions

General Form of Data Types

```
datatype (<type parameters>) <identifier> =  
  <first constructor expression> |  
  <second constructor expression> |  
  ... |  
  <last constructor expression>
```

We have already seen the predefined `option` types:

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

`List` is the quintessential datatype; conceptually

```
datatype 'a list =  
  nil  
| :: of 'a * 'a list
```

```
fun many_tests x =  
  if x = nil  
  then 0  
  else if tl(x) = nil  
        then 1  
        else if hd(x) = 3 andalso tl(x) = [2]  
              then 6  
              else 12;
```

can be written more **concisely** using **case patterns**:

```
fun many_tests x =  
  case x of  
    nil                => 0 |  
    (_ :: nil)        => 1 |  
    (3 :: (2 :: nil)) => 6 |  
    _                  => 12;
```

- ▶ the **wildcard** `_` (“underscore”) matches anything.
- ▶ Patterns can be nested

[Type Names](#)[Datatypes](#)[Patterns](#)[Local Definitions](#)

Patterns Save Computation

Datatypes

Amtoft
from Hatcliff

```
fun accesses x =  
  if x = nil  
  then 0  
  else if hd(x) < 20  
        then hd(x)  
        else 30
```

involves taking `hd` of `x` twice. This is avoided by:

```
fun accesses nil      = 0  
  | accesses (n::ns) =  
    if n < 20 then n else 30
```

- ▶ Variables used in patterns are bound to values (p.73)
- ▶ `fun` has an **implicit** case statement as has `fn`:

```
val accesses =  
  fn nil      => 0  
  | (n::ns) => if n < 20 then n else 30
```

Type Names

Datatypes

Patterns

Local Definitions

Having it Both Ways

Merging two sorted lists:

```
– fun merge(nil ,M) = M    | merge(L , nil) = L
  | merge(L,M) = if hd(L) < hd(M)
    then hd(L)::merge(tl(L),M)
    else hd(M)::merge(L , tl(M));
```

Introduce case patterns:

```
– fun merge(nil ,M) = M    | merge(L , nil) = L
  | merge(x::xs , y::ys) = if x<y
    then x::merge(xs , y::ys)
    else y::merge(x::xs , ys);
```

but now $x :: xs$ has to be recomputed. Use “as”:

```
– fun merge(nil ,M) = M
  | merge(L , nil) = L
  | merge(L as x::xs , M as y::ys) = if x<y
    then x::merge(xs ,M)
    else y::merge(L , ys);
```

as names the entire data plus destructs it using a pattern.

Type Names

Datatypes

Patterns

Local Definitions

Patterns Should be Exhaustive

Datatypes

Amtoft
from Hatcliff

```
– fun non_exhaustive nil = 0;  
... Warning: match nonexhaustive  
    nil => ...
```

```
val non_exhaustive = fn : 'a list -> int
```

```
– non_exhaustive nil;  
val it = 0 : int
```

```
– non_exhaustive [1,2];  
uncaught exception Match ...
```

A **non**-exhaustive pattern

- ▶ causes a compile-time **warning**
- ▶ can cause a run-time **exception** for data corresponding to missing cases
- ▶ still might be appropriate if we know we will never encounter certain data

Type Names

Datatypes

Patterns

Local Definitions

Pattern Matching versus Unification

Datatypes

Amtoft
from Hatcliff

```
– fun test_pair (p : int * int) =  
    if #1(p) = #2(p) then "yes" else "no";  
  
val test_pair = fn : int * int -> string
```

We would like to rewrite to

```
– fun test_pair (n,n) = "yes"  
    | test_pair _ = "no";
```

```
.. Error: duplicate variable in pattern(s): n
```

- ▶ a variable **cannot** be repeated in the same pattern
- ▶ this would be equivalent to **unification**
- ▶ which is a more powerful technique
(used, e.g., in Prolog)

Type Names

Datatypes

Patterns

Local Definitions

Local Variables

Datatypes

Amtoft
from Hatcliff

Type Names

Datatypes

Patterns

Local Definitions

```
fun double x = x + x;
```

```
fun dupdouble x y =  
    [(double(x), double(y)),  
     (double(x), double(y))];
```

The `let` construct declares local variables

```
fun dupdouble x y =  
    let val a = double(x);  
        val b = double(y)  
    in  
        [(a, b), (a, b)]  
    end;
```

and thus avoids duplicating computation

Patterns for Return Values

Datatypes

Amtoft
from Hatcliff

Type Names

Datatypes

Patterns

Local Definitions

When a local variable is bound to the result of a function

```
fun splitem x =  
  let val z = dupdouble x x  
    in (#1(hd(tl(z))),#2(hd(tl(z))))  
  end;
```

we may decompose that result using patterns:

```
fun splitem x =  
  let val [q,(s,t)] = dupdouble x x  
    in (s,t)  
  end;
```