F28PL1 Programming Languages

Lecture 13: Standard ML 3

Type variable

- all types so far have been *ground*
  - i.e. all details known
- SML provides *type variables* to express unknown types
- Greek names
  - α - alpha - written ‘a
  - β - beta - written ‘b
  - γ - gamma - written ‘c
  - etc

Type variable

- in a type expression, all occurrences of the same type variable must refer to the same type
- e.g. `‘a * ‘a * ‘a`
- tuple with 3 elements of same unknown type
- all `‘a` refer to same type
- satisfied by:

```
(1,2,3) : int * int * int
```

- i.e. `‘a` == `int`

Type variable

- e.g. `‘a * ‘a * ‘a`
- satisfied by:

```
((1,”one”),(2,”two”),(3,”three”)) :
(int * string)*(int*string)* int*string)
```

- i.e. `‘a == int * string`

Type variable

- e.g. `(‘a * ‘b) * (‘a * ‘b)`
- tuple with 2 elements of same type
- 1st sub-element of each sub-tuple have same unknown type
  - ‘a could be any type
- 2nd sub-element of each sub-tuple have same unknown type
  - ‘b could be any type
- ‘a and ‘b could refer to same type but need

Type variable

- e.g. `(‘a * ‘b) * (‘a * ‘b)`
- satisfied by:

`((“Francis”,1.7),(“Frances”,1.65))  :`

` (string * real) * (string * real)`

- i.e. `‘a == string`, `‘b == real`

`((1,(“even”,false)),(2,(”even”,true)))  :`

`(int*(string*bool))*int*(string*bool))`

- i.e. `‘a == int`, `‘b == string * bool`

`((1,1),(2,4)) : (int * int) * (int * int)`

- i.e. `‘a == ‘b == int`

Tuple pattern

- extend patterns to include *tuples of patterns*
- e.g. join two strings in tuple together with a space in between

```
- fun tJoin (s1,s2) = s1^” “^s2;
> val tJoin =
   fn : string * string -> string
```

- `^` takes 2 `strings` so `s1` and `s2` must be `string`

```
- tJoin (“hello”,”there”);
> “hello there” : string
```

Tuple pattern

- e.g. swap elements of pair tuple

```
- swap (e1,e2) = (e2,e1);
> val swap = fn : ‘a * ‘b -> ‘b * ‘a
```

- can’t deduce type for `e1`; call it `‘a`
- can’t deduce type for `e2`; call it `‘b`

```
- swap (1,”two”);
> (“two”,1) : string * int
```

- for `‘a * ‘b` to be consistent with
- `(1,”two”) : int * string`
- `‘a` must be `int` ; `‘b` must be `string`

Tuple pattern

```
- swap ((1,”two”),(”one”,2));
> ((“one”,2),(1,”two”) :
  (string * int) * (int * string)
```

- `swap : ‘a * ‘b -> ‘b * ‘a`
- for a `* ‘b` to be consistent with

```
((1,”two”),(”one”,2)) :
(int * string) * (string * int)
```

- ‘a must be `int * string`
- ‘b must be `string * int`

Tuple pattern

- e.g. select first element from 2 element tuple

```
- fun first (e1,_) = e1
> val first = fn : ‘a * ‘b -> ‘a
- first (“hello”,”there”);
> “hello” : string
```

- e.g. select second from 2 element tuple

```
- fun second (_,e2) = e2
> val second = fn : ‘a * ‘b -> ‘b
- second (true,42);
> 42 : int
```

Lists

- arbitrary length sequence of same type
- if `‘a` is a type then `‘a list` is a list of `‘a`
- lists are *polymorphic*
  - list of any type, including lists & functions
- empty list
  - `[]` or `nil`
- list constructor: `::` - infix binary
- if *h* is `‘a` and *t* is `‘a list` then *h*`::`*t* is `‘a list`
- *h* is *head* of list
- *t* is *tail* of list

Lists

- NB all lists must end with empty list
- system shows list in bracketed shorthand

elem1::(*elem2*::...(*elemN*::[])...) ==>

[*elem1*,*elem2*...*elemN*]

```
- 1::(2::(3::[]));
> [1,2,3] : int list
- “Ann”::(“Bill”::(“Cyd”::[]));
> [“Ann”,”Bill”,”Cyd”] : string list
```

Lists

- don't need (...) with `::`

```
- (1,"one")::(2,"two")::(3,"three")::[];
> [(1,"one"),(2,"two"),(3,"three")] :
  (int * string) list
```

- singleton list *elem*`::[]` ==> `[`*elem*`]`

```
- 42::[];
> [42] : int list
```

- `::` has lower precedence than function calls

Lists

- e.g. generate list of integers from `n` to `1`

```
- fun ints 0 = [] |
      ints n = n::ints (n-1);
> val ints = fn : int -> int list
```

- `0` is `int` so `n` must be `int`
- `n` is `int` so `::` must return `int list`

```
- ints 4;
> [4,3,2,1] : int list
ints 4 ==> 4::ints 3 ==> 4::3::ints 2 ==>
4::3::2::ints 1 ==> 4::3::2::1::ints 0 ==>
4::3::2::1::[] ==> [4,3,2,1]
```

Lists

- e.g. generate list with n copies of value s
- base case: n=0 ==> return empty list
- recursion case: n>0 ==> put s on front of n-1 copies of s

```
- fun nCopies 0 _ = [] |
      nCopies n s = s::nCopies (n-1) s;
> val nCopies =
   fn : int -> ‘a -> ‘a list
```

- 0 is int so n must be int
- don’t know s’s type; call it ‘a; :: returns ‘a list

Lists

```
- nCopies 3 “o”;
> [“o”,”o”,”o”] : string list
```

- `nCopies : int -> ‘a -> ‘a list`
- to be consistent when `s` is a `string`, `‘a == string`

```
nCopies 3 “o” ==>
“o”::nCopies 2 “o” ==>
“o”::”o”::nCopies 1 “o” ==>
“o”::”o”::”o”::nCopies 0 “o” ==>
“o”::”o”::”o”::[] ==>
[“o”,”o”,”o”]
```

Lists

```
- nCopies 3 9;
> [9,9,9] : int list
```

- `nCopies : int -> ‘a -> ‘a list`
- to be consistent when `s` is an `int`, `‘a == int`

```
nCopies 3 9 ==>
9::nCopies 2 9 ==>
9::9::nCopies 1 9 ==>
9::9::9::nCopies 0 9 ==>
9::9::9::[] ==>
[9,9,9]
```

Recursion 2: lists

- list is either
  - empty
  - non-empty with a *head* and a *tail*
- can use *list patterns* in function definitions
- use [] as a constant
- make patterns with other patterns and ::
- (h::t)
  - h is a pattern to match list head
  - t is a pattern to match list tail
- must be bracketed

Recursion 2: lists

- recursion on lists
- base case: `[]`
  - return final value
- recursion case: (h`::`t)
  - do something to h
  - recurse on t
- e.g. sum elements of integer list
- base case: `[] ==> 0`
- recursion case: (`h::t`) `==>` add `h` to summing `t`

Recursion 2: lists

```
- fun sum [] = 0 |
      sum (h::t) = h+sum t;
> int list -> int
```

- `0` is `int` so…
- `+` must be `int` addition so…
- `h` must be `int` so…
- `h::t` must be `int list`

Recursion 2: lists

```
- sum [2,4,6];
> 12 : int
sum [2,4,6] ==>
sum 2::[4,6] ==>
2+sum [4,6] ==>
2+sum 4::[6] ==>
2+4+sum [6] ==>
2+4+sum 6::[] ==>
2+4+6+sum [] ==>
2+4+6+0 ==> 12
```

Recursion 2: lists

e.g. join all strings in list

- base case: `[]` ==> “”
- recursion case: `(h::t)` ==> join h to joining up all in `t`

```
- fun sJoin [] = “” |
      sJoin (h::t) = h^sJoin t;
> val sJoin =fn : string list -> string
```

- `^` takes 2 `strings` so h must be `string` so `h::t` must be `string list`

Recursion 2: lists

```
- sJoin [“a”,”bc”,”def”];
> “abcdef” : string
sJoin [“a”,”bc”,”def”] ==>
sJoin “a”::[“bc”,”def”] ==>
“a”^sJoin [“bc”,”def”] ==>
“a”^sJoin “bc”::[“def] ==>
“a”^”bc”^sJoin [“def”] ==>
“a”^”bc”^sJoin “def”::[] ==>
“a”^”bc”^”def”^sJoin [] ==>
“a”^”bc”^”def”^”” ==>
“abcdef”
```

Recursion 2: lists

- e.g. double all elements of integer list
- base case: `[] ==> []`
- recursion case: `(h::t) ==>` put twice h onto list from doubling all `t`

```
- fun double [] = [] |
      double (h::t) = 2*h::double t;
> val double = fn : int list -> int list
```

- 2 is `int` so `*` must be `int` so h must be `int` so `h::t` must be `int list`
- `::` must be `int list` construction

Recursion 2: lists

```
- double [5,3,1];
> [10,6,2] : int list
double [5,3,1] ==>
2*5::double [3,1] ==>
2*5::2*3::double [1] ==>
2*5::2*3::2*1::double [] ==>
2*5::2*3::2*1::[] ==>
[10,6,2]
```

Recursion 2: lists

- e.g. count how often 0 appears in a list
- base case: `[] ==> 0`
- recursion case1: `(h::t)` - h=0 ==> 1 + count 0 in `t`
- recursion case 2: `(h::t)` - h<>0 ==> count 0 in `t`

```
- fun count0 [] = 0 |
      count0 (0::t) = 1+count0 t |
      count0 (_::t) = count0 t;
> val count0 = fn : int list -> int
```

- `0` is `int` so `0::t` must be `int list`
- `0` is `int` so + must be `int` addition

Recursion 2: lists

```
- count0 [1,0,2,0,3,0];
> 3 : int
count0 [1,0,2,0,3,0] ==>
count0 [0,2,0,3,0] ==>
1+count0 [2,0,3,0] ==>
1+count0 [0,3,0] ==>
1+1+count0 [3,0] ==>
1+1+count0 [0] ==>
1+1+1+count0 [] ==>
1+1+1+0 ==>
3
```

Equality type

- e.g. count how often value v appears in list
- base case: `[] ==> 0`
- recursion case1: `(h::t)` - h=v ==> 1 + count v in `t`
- recursion case2: `(h::t)` - h<>v ==> count v in `t`

```
- fun count _ [] = 0 |
      count v (h::t) =
   if v=h
   then 1+count v t
   else count v t;
> val count = fn : ‘’a -> ‘’a list -> int
```

Equality type

```
- fun count _ [] = 0 |
      count v (h::t) =
   if v=h
   then 1+count v t
   else count v t;
> val count = fn : ''a -> ''a list -> int
```

- `''a` - *equality type variable*
- don't know anything about `v`, `h` or `t`
- know that `v` and `h` are the same equality type, say `''a`, so `h::t` must be a `''a list`

Equality type

```
- count “a” [“a”,”b”,”a”];
> 2 : int
```

- `count :: ‘’a -> ‘’a list -> int`
- for consistency when v is “a” and `(h::t)` is `[“a”,”b”,”a”]`, `‘’a` must be `string`

```
count “a” [“a”,”b”,”a”] ==>
1+count “a” [“b”,”a”] ==>
1+count “a” [“a”] ==>
1+1+count”a” [] ==>
1+1+0 ==> 2
```

Accumulation variables

- used to pass information from stage to stage of recursion
- e.g. count how many integer list elements are negative, zero or positive
- use a tuple to record counts: (*negative*,*zero*,*positive*)
- pass tuple from call to call
- at end of list return tuple

Accumulation variables

- base case: `[]` ==> return counts tuple
- recursion case1: `(h::t)`
  - h=0 ==> find counts for `t` with *zero* count incremented
- recursion case2: `(h::t)`
  - h<0 ==> find counts for `t` with *negative* count incremented
- recursion case3: `(h::t)`
  - h>0 ==> find counts for `t` with *positive* count incremented

Accumulation variables

```
- fun counts (n,z,p) [] = (n,z,p) |
      counts (n,z,p) (0::t) =
         counts (n,z+1,p) t |
      counts (n,z,p) (h::t) =
       if h<0
       then counts (n+1,z,p) t
       else counts (n,z,p+1) t;
> val counts =
   fn : int * int * int ->
        int list -> int * int * int
```

Accumulation variables

```
- counts (0,0,0) [1,~2,0,3,~4];
> (2,1,2) : int * int * int
counts (0,0,0) [1,~2,0,3,~4] ==>
counts (0,0,1) [~2,0,3,~4] ==>
counts (1,0,1) [0,3,~4] ==>
counts (1,1,1) [3,~4] ==>
counts (1,1,2) [~4] ==>
counts (2,1,2) [] ==>
(2,1,2)
```

- NB can’t update individual fields of tuple
- must copy tuple with changes

Accumulation variables

- e.g. generate list of squares from `m` to `n` in ascending order
- base case: `m>n` ==> return `[]`
- recursion case `m<=n` ==> put `m` squared onto list of squares from `m+1` to `n`

```
- fun squares m n =

    if m>n

    then []

    else sq m::squares (m+1) n;

> val squares =

   fn : int -> int -> int list
```

Accumulation variables

```
- squares 1 4;
> [1,4,9,16] : int list
squares 1 4 ==>
sq 1::squares 2 4 ==>
sq 1::sq 2::squares 3 4 ==>
sq 1::sq 2::sq 3::squares 4 4 ==>
sq 1::sq 2::sq 3::sq 4::squares 5 4 ==>
sq 1::sq 2::sq 3::sq 4::[] ==> [1,4,9,16]
```

- `m` is accumulation variable to pass start of new range from call to call

Local definitions

`let` *definition*

`in` *expression*

`end`

- *definition* establishes name/value associations for use in *expression* only
- *scope* of *definition* is *expression*

```
- let val x = 12
  in x*x*x
  end;
> 1728 : int
```

Local definitions

- very useful for tuple matching and selection

```
- let val ((given,family),age) =
            ((“Clark”,”Kent”),29)
  in given
  end;
> “Clark” : string
```

- particularly useful when function returns tuple and only want some elements
- NB don’t forget:
  - `val` before variable
  - `end` at end of definition

Exceptions

- break flow of control
- typically after some error
- when exception is *raised*
  - control is transferred to *handler*

`exception` *identifier*

- defines an *exception* with type constructor *identifier*

Exceptions

`raise` *identifier*

- initiates the exception
- transfers control to immediately enclosing handler
- if no handler then control is transferred to the system and program stops
- e.g. divide by 0

```
- exception DIVIDE_BY_ZERO;
> exception DIVIDE_BY_ZERO
```

Exceptions

```
- fun divide x y =
   if y=0
   then raise DIVIDE_BY_ZERO
   else x div y;
> val divide = fn: int -> int
- divide 3 0;
> exception DIVIDE_BY_ZERO
uncaught exception DIVIDE_BY_ZERO
```

Type aliases

- type aliases

`type` *identifier* = *type expression*

- *identifier* is an *alias* for *type expression*
- i.e. both denote same type

Type aliases

```
- type family = string;
> type family = sting
- type given = string;
> type given = string
- type person = family * given;
> type person = family * given
- type people = person list;
> type people = person list
```

- `family` and `given` are both aliases for `string`

Type aliases

```
- type family = string;
> type family = sting
- type given = string;
> type given = string
- type person = family * given;
> type person = family * given
- type people = person list;
> type people = person list
```

- `person` is an alias for

`family * given` is an alias for

`string * string`

Type aliases

```
- type family = string;
> type family = sting
- type given = string;
> type given = string
- type person = family * given;
> type person = family * given
- type people = person list;
> type people = person list
```

- `people` is an alias for `person list` is an alias for

`(family * given ) list` is an alias for

`(string * string ) list`

NJSML print depth

- NJSML will only print data structures to fixed depth
- thereafter indicates unprinted structure with #
- to change print depth:

```
- Control.Print.printDepth := integer;
```