More on Types

what would be an example of something of type

(a) int list list list

(b) (int*string) list

(c) (string list * (int * (real * string)) * int)

(d) (((int * int) * (string list) * real) *
(real * string))

Type Checking

Type System for a language is a set of rules for associating a type with expressions in the language.

A type system *rejects* an expression if it does not associate a type with the expression.

Where possible, ML infers the type of an expression and reports errors when type of expression cannot be inferred.

```
2+2;
  val it = 4:int
```


How ML Deduces Types

1. Types of the operands and result of arithmetic operators must all agree.
Ex: `(a + b) * 2.0;`
`a` and `b` are real here and everywhere else.

2. When applying an arithmetic comparison, all operands should be of the same type.
Ex: `a <= 10;` means that `a` is an integer.

3. In conditional expression, the expression itself and its subexpressions following `then` and `else` should be of the same type.
Ex: `if 1 < b then a else 5.0;` means that `b` is integer and `a` is real.

Deducing Types (Cont'd)

4. If a variable or expression used as an argument of the function is of a known type, then the corresponding parameter of the function must be of that type. If the function parameter is of known type, then the variable or expression used as the corresponding argument must be of the same type.

5. If the expression defining the funciton is of a certain type, then the function returns a value of that type.

```
(1) fun comb(n,m) = (* assumes 0 <= m <= n *)
(2)   if m=0 orelse m=n then 1
(3)   else comb(n-1,m) + comb(n-1,m-1);
  val comb = fn:  int * int -> int
```

How did ML figure this out?

Problem: type inference is exponential time in the worst case!

When Are Types Equal?

ML uses *structural equivalence* for type equality.

Two types are structurally equivalent if and only if they are equivalent under the following rules:

SE1. A type name is structurally equivalent to itself

SE2. Two type expressions are structurally equivalent if they are formed by applying the same type constructor to structurally equivalent types

SE3. After a type declaration, `type n = T`, the type `n` is structurally equivalent to `T`.

`int` and `int` are structurally equivalent and so are

`int * int -> bool` and

`int * int -> bool`

Example of Structural Equivalence

Suppose we have the following basic types:

```
datatype texture = arcs | bands;
datatype direction = ne | se | sw | nw;
```

The type of the following expression can be inferred from its syntax:

```
[ [ (arcs,ne) ] ];
  val it = [[(arcs,ne)]]: (texture * direction) list list
```

The type of this expression is structurally equivalent to the type `quilt`:

```
type square = texture * direction;
type row = square list;
type quilt = row list;
```


Overloading

A symbol is *overloaded* if it has different meanings in different contexts. For example, operators like + and * are overloaded (can work to operate on integers - 2+2 and reals - 2.0+2.0)

Treatment of + in ML can be restated using the following rules:

If `E` has type `int` and `F` has type `int`
then `E + F` has type `int`.
If `E` has type `real` and `F` has type `real`
then `E + F` has type `real`.

When ML cannot resolve overloading, it complains, as in

```
fun add(x,y) = x+y;
  Error:  overloaded variable "+" cannot be resolved
```


Overloading (cont'd)

Can specify explicit types to resolve overloading. This helps ML infer the type of the function:

```
fun add(x,y):int = x+y;
  (* result is an integer *)
  val add = fn :  int * int -> int
fun add(x,y) = x + (y:int);
  (* y is an integer *)
  val add = fn :  int * int -> int
```

Coercion

Coercion is a conversion from one type to another, inserted automatically by a programming language.

Most programming languages treat the expression 2∗3.142 as if it were 2.0∗3.142; that is, they automatically convert the integer 2 into the real 2.0. ML does not do it:

```
2 * 3.142;
  Error:  operator and operand do not agree
  (tycon mismatch)
  operator domain:  int * int
  operand:  int * int
```

Have to specify type conversion explicitly:

```
real(2);
  val it = 2.0 :  real
real(2) * 3.142;
  val it = 6.284 :  real
```


Expression Evaluation

Types:
- Innermost (eager) - Scheme, ML
- Outermost (lazy)
- Selective - `if` statements - Scheme, ML

When they converge, innermost and outermost give the same result (Church-Rossler Theorem), but innermost might not converge and outermost take longer (because of selective evaluation).

Example: 91-function (McCarthy)

```
fun f(x) =
  if x > 100 then x-10 else f(f(x+11))
```


Expression Evaluation (Cont'd)

Innermost Evaluation:

```
f(100) = if 100 > 100 then 100-10 else f(f(100+11))
       = f(f(100+11))
       = f(f(111))
       = f( if 111>100 then 111-10 else f(f(111+11)) )
       = f( 111-10 )
       = f( 101 )
       = if 101 > 100 then 101 - 10 else f(f(101+11))
       = 101 - 10
       = 91
```

Outermost Evaluation:

```
f(100) = if 100 > 100 then 100 - 10
                      else f(f(100+11))
       = f(f(100+11))
       = if f(100+11) > 100 then f(100+11) - 10
                            else f(f(f(100+11)+11))

       f(100+11) = if 100+11>100 then 100+11
                                 else f(f(100+11+11))
                 = if 111>100 then 100+11-10
                              else f(f(100+11+11))
                 = 100+11-10
                 = 111-10
                 = 101
etc.
```

Try the rest at home.

Short Circuit Evaluation

short circuit evaluation for operators `orelse` (like OR), `andalso` (like AND) - do not evaluate the second parameter if not necessary.

Example:

```
E orelse F
```

if `E` is `true` and `F` does not converge, outermost computation will return `true` whereas innermost will not converge.

Similarly,

```
E andalso F
```

if `E` is `false` and `F` does not converge.

Expression Evaluation (Cont'd)

In general, expression evaluation is like tree-rewriting

Example 1: `7 * 7 - 4 * 2 * 3`

Example 2: `3 = 3 orelse 3 = 1`

Example 3: `1::2::3::[]`

Variable Bindings

Variable bindings are similar to Scheme. Keyword `let` (actually, this corresponds to Scheme's `let*`, i.e., sequential binding).

let val x=E1 in E2 end

`val` indicates that substitution is for a variable, not a function.

```
let val x = 3 in
   let val y = x+1 in
       y + y * y
   end
end;
```

```
let val x = 3
    val y = x+1 in
         x + y * y
end;
```

To do substitution for a function (define a temporary function), use keyword `fun`.

```
  let fun f(x) = E1 in E2 end
Ex: let fun even(x) =
        x = 0 orelse even(x-2)
      in even(8)
    end
```


Bindings (Cont'd)

`val`s and `fun`s can be mixed:

```
let val x = 3 in
    let fun square:int x = x * x in
        square x
    end
end;
```

These are sequential bindings. To do simultaneous binding, use `let and`:

```
let fun even(x) =
   if x=0 then true else
          if x=1 then false else odd(x-1)
and odd(x) =
   of x=0 then false else
          if x=1 then true else even(x-1)
in (even(24), odd(24))
end
```

This evaluates to the pair `(true,false)`.

Static and Dynamic Scoping

Static scoping - free variables in a procedure definition are bound at the point that the function is defined (*lexical* environment). Thus, can and should rename bound variables to avoid *variable capture*:

```
fun try1 (z, y) = z;
fun try2 (x)   = y;
compose (try1, try2) = ?
      fn (x, y) => y or
      fn (x, w) => y
```

Dynamic scoping: free variables are bound at the point that the function is evaluated (in *calling* or *activation* environment). To resolve the binding occurrence for a variable in `P`, necessary to reason about the execution history which leads to `P` being evaluated.

Which Scoping is Better?

- Static scoping leads to more efficient variable lookup during execution
- more readable code
- procedure will give the same results when called with the same parameters
- but turns out not to be so good in programming-in-large: in component reuse

- dynamic scoping allows for enhanced flexibility in configuring software components

Some languages (Ada and proposed for ML) will have *constrained genericity* - introducing a controlled form of dynamic scoping into statically-scoped languages.

String Manipulation

So far, we mostly saw list manipulation.
Main idea of string processing:
- explode the string to create a list
- manipulate letters as list elements
- implode the list to create a string.

Example - remove first letter of the string:

```
let val a = explode "winter"
  val a = ["w","i","n","t","e","r"] :  string list
let val b = tl[a]
  val b = ["i","n","t","e","r"] : string list
let val c = implode b
  val c = "inter" : string
```


String Manipulation (Cont'd)

Special functions for dealing with strings are:

- ^ - string concatenation
```
"cub" ^ "a"
  val it = "cuba" : string
```
- `ord` - gives ordinal value for a character
```
ord "a"
  val it = 97 : int
```
- `chr` - gives a character from ordinal value
```
chr 97
  val it = "a" : string
```
- `size` - gives string length
```
size "abc"
  val it = 3 : int
```
- `substring` - gives a portion of the string between position 1 and position 2, e.g.
```
substring ("winter", 1, size "winter"-1)
  val it = "inter" : string
```


User-defined String Functions

- Function `clip` to remove the last letter from the string:

```
fun clip s = substring(s,0,size s - 2);
clip "been" = "bee"
clip "raven" = "rave"
```

- Function `incFirst` to increment the first character:

```
incFirst "bad" = "cad"
incFirst "shin" = "thin"
fun incFirst a = chr(ord a+1)
   ^ substring(a,1,size s - 1)
```

- Function `sndhalf` takes a string and returns the second half of it.

```
sndhalf "abcd" = "cd"
fun sndhalf s =
  substring(s, size s div 2, size s-1);
```

Anonymous Functions

ML can have *anonymous functions*, i.e., functions without a name! These are useful for adapting existing functions so they can be used together with tools like `map`.

Syntax:
fn <*formal-parameter*> => <*body*>

Example (multiply its argument by 2):

```
fn x => x * 2
  val it = fn : int -> int
```

This function can be used by `map`:

```
map (fn x => x*2) [1,2,3,4,5];
  val it = [2,4,6,8,10] : int list
```


Functions - Cont'd

Most functions are assumed to be *curried*:

```
fun foobar s i = (size s) + i;
  val foobar = fn : string -> int -> int

fun baz = foobar "hello" ;
  val baz = fn : int -> int
```

Rewrite `foobar` as:

```
fun foobar s =
  fn i = size(s)+i;
  val foobar = fn : string -> int -> int
```

Now, try:

```
fun foobar (s i) = (size s)+i;
  val foobar = fn: string * int -> int
```


Function Composition

Everything in ML is expression evaluation. To achieve some nontrivial results, we often need to compose functions, like

```
chr(ord a+1) ^ substring(a,1,size s)
```

To ease up composing functions, ML provides a special operator `o`:
$(gof)x = g(f(x))$
So, the result of applying gof to x is equivalent to first applying f to x and then applying g to fx.

So, our previous expression can be written as

```
(chr o ord o (fn x => x+1)) a
  ^ substring(a,1,size s)
```


Examples of Function Composition

Suppose we define

```
fun comp F G = F o G;
fun add1 x = x+1;
```

Give type and value of the following:
(a) val compA1 = comp add1;

(b) val f = compA1 add1;

(c) f(2);

(d) val compCompA1 = comp compA1;

(e) val g = compCompA1 compA1;

(f) val h = g add1;

(e) h(2);