ML and Typechecking

ML - Meta Language. Began as the progrmaming language for a machine-assisted system for formal proofs.
Based explicitely on Lambda Calculus.

- Functions are first-class objects. But better than Scheme since has a more reasonable syntax (less parentheses) and pattern-matching (but less than in Prolog)
- Strongly typed (Scheme is untyped). Static typechecking is done at compile time. Has abstract data types

Reading - Sethi, Ch. 8-9.
Playground - Standard ML of New Jersey.

```
eddie% sml
Standard ML of New Jersey, Version 0.93, ...
val it = () : unit
-
```

Manual will be available in Annex library.

Quick Overview of ML

Types - like in Scheme but no `symbol`

boolean - true, false

string - "red", "white"

integer - 1, 10

real - 0.0, 3.14, ...

These are the built-in types.

Can also have user-defined types - using keyword `datatype`.

```
datatype direction = ne | se | sw | nw;
```

Here we listed constructors for this type. | stands for "OR", separating enumerated items.

Lists

In ML, all list elements have the same type. E.g.,

[1, 2, 3]

["red", "white", "blue"]

Type `A list` consists of all lists of elements, where each element belongs to type `A`.

`int list` - consists of all lists of integers

`::` - indicates a cons operator. So,

```
[1,2,3] = 1::[2,3] = 1::2::[3]
= 1::2::3::[]
```

Tuples

Then how do we represent non-homogeneous collections? Tuples!

Syntax: (<elem1>, <elem2>, ...).

Eg., `(1, "one")`

Projection functions - extract elements from pair.

```
fun first(x,y) = x;
fun second(x,y) = y;
```

Type of tuple - product of types of its elements. For example, type of `(1, "white")` is `int * string`.

Functions

Functions are from Domain to Range.

Type `A` $->$ `B` - set of all functions from `A` to `B`.

A function is *total* if it is defined for each element of `A`.

Otherwise, the function is *partial*

Strong influence by λ-calculus.

To apply a function - `f a`. Parentheses do not matter, so

```
f a = (f a) = f(a)
```

Application is left-associative:

```
f a b = (f a) b
```

`f a = b` means that `b` is the value of `f` at `a`

More on Types

structured types:

type	constructor	notation	example
function	$->$	infix	`int` $->$ `bool`
product	$*$	infix	`int` $*$ `int`
list	`list`	postfix	`string list`

$->$ has higher precedence than $*$ which has higher precedence than `list`.

To give a name to type - keyword `type`

```
type intpair = int * int
```

Declaration of Functions

`fun` *<name><formal-parameter>*=*<body>*;

Parentheses around the formal parameter are optional.

Examples:

```
fun successor n = n+1;
fun successor(n) = n + 1;
fun abs(n) = if n >= 0 then n else 0-n;
```

To call a function - use name of function in infix notation:

```
successor(2+3)
```

Some old functions:

```
fun len(x) =
    if null(x) then 0 else 1+len(tl(x));

fun fib(n) =
    if n=0 orelse n=1 then 1
    else fib(n-1) + fib(n-2);
```

Expression Evaluation

Innermost Evaluation(eager)

Suppose we want to evaluate <*name*><*actual parameter*>.

- Evaluate the expression represented by <*actual-parameter*>
- Substitute the result for the formal in the function body
- Evaluate the body
- Return value as result

This is *inside-out* evaluation of expressions, or, *call by value* (e.g., C, Pascal, Turing).

Example: `successor(plus(2,3))`
Scheme does the same thing.

Expression Evaluation(cont'd)

Selective evaluation - as in Project 1.

`if` $<$*condition*$>$ `then` $<$*expr*$_1$$>$ `else` $<$*expr*$_2$$>$

- evaluate $< expr_1 >$ only if $< condition >$ is true.

Outmost Evaluation from Left to Right ("lazy")

- Substitute the actuals for the formals in function body
- Evaluate the body
- Return its value as the result

This is "call by name" - evaluate parameters only when needed.

ML **does not** do that.

Expression Evaluation - Comparison

As we know from Church-Rossler theorem, all evaluations, if terminate, produce the same result.

However, in many cases "outermost" can terminate when "innermost" fails. See example of the "91-function" on p. 324 of Sethi.

Short-Circuit Evaluation - as in Scheme
- `andalso` is the same as `AND` in Scheme
- `orelse` is the same as `OR` in Scheme.

However, the following function does not do shortcircuit evaluation:

```
fun or(x,y) =
    if x = true then true
    else if y = true then true
    else false
```

Variable Bindings

These are the same as in Scheme. Keyword `let` (actually, this corresponds to Scheme's `let*`, i.e., sequential binding).

let val x=E1 in E2 end

(same as $\{E_1/x\}E_2\}$). `val` indicates that subsitution is for a variable, not a function.

Examples:

```
let val x = 2 in
   let val y = x+1 in
       y * y
   end
end
```

```
let val x = 2 in
   let val x = x+1 in
      x * x
   end
end
```

Variable Bindings, cont'd

To do substitution for a function, the keyword is `fun`, as in $\{E_1/\lambda x.x\ E_2\}$.

```
let fun f(x) = E1 in E2 end
```

```
Ex:fun even(x) =
      x = 0 orelse even(x-2)
   in even(8)
   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 evalues to the pair `(true,false)`.

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 inteferred.

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

How to Infer Types?

ML uses a simple rule to infer types:
"If f is a function of type A -> B, and a has type A, then f(a) has type B."

Instantiation of this rule for sum of integers:
"If + is a function of type int * int -> int, and the pair (E,F) has type int * int, then +(E,F) has type int."

To determine when type of argument a is in the domain of function f, have to be able to determine when two types are equal.

Type Equality

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;
```

Other Kinds of Type Equality

These can be obtained if we restrict rules **SE1-SE3**. Here are some possibilities:

- *Pure name equivalence* A type name is equivalent to itself, but no constructed type is equal to any other constructed type

- *Transitive name equivalence* A type name is equivalent to itself and can be declared equivalent to other type names. Then, the following types `S`, `T` and `U` are equivalent to each other and to `integer` because `integer` is a type name also:

```
type S = integer
  T = S
  U = integer
```

Other Kinds of Type Equality (cont'd)

- *Type expression equivalence*. A type name is equivalent only to itself. Two type expressions are equicalent if they are formed by apoplying the same constructor to equivalent expressions. In other words, the expressions have to be identical

C - uses structural equivalence for all types except records. For those, we have pure name equivalence.

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
```

(* *) indicate comments.

Coersion

Coersion 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
```

Polymorphism: Parameterized Types

ML is known for its support for *polymorphic* functions, i.e., functions that can be applied to parameters of more than one time. (More later)

For all lists, the function `hd` returns the head of a list:

```
hd [1,2,3];
  val it = 1 : int
hd ["a", "b", "c"];
  val it = "a" : string
```

So, what is type of `hd`?

```
hd;
  val it = fn : 'a list -> 'a
```

ML uses a leading quote to identify a type parameter. The type of `hd` can be read as "function from a list of any type `'a` to type 'a".

ML and Typechecking - Cont'd
Overview

- Functions
  - case analysis
  - unnamed functions
- Type inference
  - type variables
  - parametric polymorphism
  - ad-hoc polymorphism

Chapter 9 of Sethi.

Next time - datatypes. End of ML.

Functions

Syntax:

fun <*name*><*formal-parameter*> = <*body*>

Example:

fun succ n = n + 1

Some symbols:

```
:: = cons
@ = append
+ - * / <= < = <> >= >
^ = raising into power
div = integer division
mod = remainder from integer division
| = or (alternative)
(* *) = comments
```

Some familiar list operations

```
fun append(x, z) = if null(x) then z
                   else hd(x)::append(tl(x),z)
fun reverse(x, z) = if null(c) then z
                    else reverse(tl(x), hd(x)::z)
fun length(x) = if null(x) then 0
                else 1 + length(tl(x))
```

Case Analysis in ML

Matching:

_ matches with anything

$< var >$ matches with anything

[] matches with empty list

a::y matches with a list [a,...]

(x,y) matches with a tuple with two elements - a pair

Examples:

```
fun append([]. z) = z
|   append(a::y, z) = a :: append(y,z)
fun length([]) = 0
|   length(a::y) = 1 + length(y)
fun first(x,y) = x;
fun second(x,y) = y;
```

Case Analysis, Cont'd

ML gives errors if the cases in a function declaration are not exhaustive.

```
fun head(a::y) = a;
  Warning:  match not exhaustive
    a ::  y => ...
  val head = fn :  'a list -> 'a
```

It also gives errors if cases are redundant.

```
fun f(nul) = 0
| f(a::y) = a+f(y);
  Warning:  redundant patterns in match
    nul => ...
  --> a ::  y => ...
  val f = fn :  'a list -> int
```

Anonymous Functions in ML

Syntax: use **fn** instead of **fun**:

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

These work exactly like `lambda` functions in Scheme:

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

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

Dealing with Operators

Operators + and $*$ can be treated as the named functions `op +` and `op *`. These can be passed to other functions:

```
reduce op + [1,2,3,4,5] 0;
val it = 15 :  int

reduce op * [1,2,3,4,5] 1;
val it = 120 :  int
```

Functions - Cont'd

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
```

`val` is used to bind a variable to a value. Notice that function definitions and variable definitions are done differently. (They are exactly the same in Scheme)

Type Variables (new in ML)

We have already seen type constants (such as `string`, `int`) and type operators (such as `list`, `=>`). We refer to them generically as *type constructor*.

Type constructor takes a collection of type arguments and returns a valid type. For example, the following are all valid types:

```
string list
(int list) list
int * (int list)
```

Tuples are n-ary type constructors, while the arrow type is a binary type constructor.

Type Variables(Cont'd)

In addition to type constructors, ML has *type variables*, denoted by quoted identifiers:

```
'a, 'b, ...
```

Do not confuse ML type variables with Scheme symbols!

Program variables - "placeholders" for values at runtime
Type variables - placeholders for types.

Example:

```
fun Id x = x;
  Id = fn : 'a -> 'a
(Id 3, Id true, Id "hello")
  val it = (3, true, "hello") : int * bool * string
```

Type Variables (cont'd)

`Id` is the identity function. But not enough type information for the compiler to figure out the domain and range type, but it knows that both types should be the same. Compiler infers a *minimally constrained type* for `Id`, with `'a` as a placeholder.

In previous example, `Id` is used in three different places with different types (`int`, `bool` and `string`). Definition of `Id` makes no assumption about the structure of its argument, and so the type of its formal parameter could be filled in with any type whatsoever.

Parametric Polymorphism

The different occurrences of `Id` are filled in differently. The definition of `Id` is implicitly parameterized by the type of its argument, with this type information filled in at the points where `Id` is called.

This is an example of *parametric polymorphism*.

Implicit type parameters are not just placeholders. They also constrain the calling patterns of polymorphic procedures. For `Id`, its return type is the same as the type of its actual argument.

Parametric Polymorphism (Cont'd)

Another example:

```
fun apply P A = P A ;
  val apply = fn :  ('a -> 'b) -> 'a -> b
```

The type `'a -> 'b` of the procedure parameter is an example of a very general phenomenon - building type expressions involving type variables which give partial structure to the types of procedure's arguments and result.

Other examples of type expressions:

```
- 'a * 'b
- list
```

`apply` is defined over any procedure value.

Polymorphic Functions - Evaluation

Polymorphic functions may be defined over lists of any type, tuples, etc. This is the true power of polymorphism - the ability to define reusable library procedures which operate generically over particular data structures, but can be type-checked statically at compile-time!

For example, these are the types of our list operations:

```
fun length(x) = if null(x) then 0
    else 1 + length(tl(x))
```

val length = fn : ('a list) -> int

```
fun append(x, z) = if null(x) then z
    else hd(x)::append(tl(x),z)
```

val append = fn : ('a list) -> ('a list) -> ('a list)

Parametric Polymorphism in Other Languages

How would we do this in other programming languages?

Pascal, Turing - different versions of these operations for each list type in a program, even though each version would do exactly the same thing.

C - define `length` to operate over linked lists where the node element field has type `void*`, but this would weaken the type-checking.

C++ - has polymorphism (later in the semester) for arrays, records and pointers of any type, but user-defined functions cannot be polymorphic.

Ada - can define operations as generic procedures, but in instantiating their types we have to give their instances new names.

Parametric Polymorphism - Drawbacks

1. A polymorphic procedure cannot be passed as an argument to another procedure:

```
fun applyTwice P = (P true, P 3);
```

```
std_inL 0.0-0 Error: operator and operand don't
agree (tycon mismatch)
  operator domain: bool
  operand: int
  in expression: P 3
```

What is wrong? The compiler will not allow the formal parameter P to be polymorphic. From the first usage of `P` it determines that `P`'s argument should be `int`, but this conflicts with the second usage of `P`.

Another example:

```
let fun Id x in (Id true, Id 3)
```

val it = (true, 3) : bool * int

Here, ML "knows" what the polymorphic procedure `P` is and can check that the intended type of its usage should be, whereas in the former case we do not have enough information about `P` to type-check its usage.

This can be regarded as a weakness of the fact that ML is *implicitly typed*. More powerful languages (Pebble, Russell, Quest are proposed in research community) are based on explicit typing.

Parametric Polymorphism - Drawbacks (Cont'd)

2. ML does not really handle overloading.

Consider the following:

```
4 + 8;
```

val it = 12 : int

```
5.6 + 9.31 ;
```

val it = 14.91 : real

The + operation is overloaded to operate over both integers and reals. So, is + a polymorphic operation?

The difference between this and parametric polymorphism is that these are really two + operations, one for integers and the other for reals, with the compiler deciding on the basis of the type information which + operation to insert into the code. On the other hand, there is really only a single `length` operation, which executes the same code at each call site.

Ad-hoc Polymorphism

Overloading is sometimes referred to as *ad-hoc polymorphism*.

Ad-hoc and parametric polymorphisms do not interact well with each other:

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

std_in:10.18 Error: overloaded variable cannot be resolved: +

We want to say that `double` is defined over any type for which a + operation is defined. There is no way to say this in ML.

One exception - equality:

```
fun equal x y = ( x = y) ;
```

val equal = fn : ” a -> ”a -> bool

```
equal [1, 2, 3] [1, 2, 4] ;
```

val it = false : bool

Equality (cont'd)

```
equal (true, "hello") (true, "hello");
```

val it = true : bool

Note that domain type here is `''a` rather than `'a`. The double-quote annotation signifies that `equal` is only defined over data types for which equality is defined. Such types are referred to as equality types or *eqtypes*.

Equality is still a built-in operation, but the ML compiler can build equality procedures for any data type which does not include a function type (how would you check two procedures for equality?).

food for slide eater

food for slide eater