Introduction to ML, II

Lars Birkedal

Programming, Logic and Semantics Group
IT University of Copenhagen

Spring 2009
Plan

- More Standard ML: polymorphic types and type variables
- Expressions with let-bound variables
- Interpreters and one-stage execution; compilers and two-stage execution
- Dynamic semantics (run-time) versus static semantics (compile-time)
- Static checking of expressions: no unbound variables
- Postfix (‘reverse Polish’) notation
- Postscript, a stack-based language
- Abstract machines with stacks
fun len [] = 0
| len (x::xr) = 1 + len xr;

> val 'a len = fn : 'a list -> int

This means: For any type 'a, function len has type 'a list -> int.
The 'a is a type variable; it may be instantiated with any type. That is, len works for int list and string list and (string*int) list and so on.
fun member x [] = false  
  | member x (y::yr) = x=y orelse member x yr;

fun workday d = member d ["Mon", "Tue", "Wed", "Thu", "Fri"];

> val ''a member = fn : ''a -> ''a list -> bool

The "a is an equality type variable; it may be instantiated only with types that admit equality (=) comparison. For any type t that admits equality, function member has type t -> t list -> bool. That is, member works only on data that can be compared for equality. Almost all data can, but functions cannot.
SML: polymorphic datatypes

A tree is a tree is a tree ... regardless of the type of node values:

```
datatype 'a tree =
    Lf
  | Br of 'a * 'a tree * 'a tree;
fun sumtree Lf           = 0
  | sumtree (Br(v, t1, t2)) = v + sumtree t1 + sumtree t2;
```

Creating a list of tree nodes

```
fun preorder1 Lf = []
  | preorder1 (Br(v, t1, t2)) = v :: preorder1 t1 @ preorder1 t2
```

So `preorder1 (Br(1, Br(2, Lf, Lf), Br(3, Lf, Lf)))` is `[1, 2, 3]`.

And `preorder1 (Br("a", Br("b", Lf, Lf), Br("c", Lf, Lf)))` is `"a", "b", "c"`.

What is the type of `preorder1`?
Various kinds of type polymorphism, I

- Parametric polymorphism, as in Standard ML, Java 5.0 and C# 2.0:
  The type variable \( \alpha \) is a parameter that may range over arbitrary types.
  A parametric polymorphic function works the same way regardless how the type variables are instantiated.

- Bounded parametric polymorphism, as in Java 5.0, C# 2.0 (in Standard ML equality types):
  The type variable \( T \) is a parameter that may range over all types with certain properties.
  For instance, a List of \( T \) elements is printable if all its elements are:

```java
class List<T : IPrintable> : IPrintable { ... }
```
Various kinds of type polymorphism, II

- Ad hoc polymorphism, or overloading:
  The Java plus operator (+) may be used at certain types: `int`, `double`, `String`, ...

- Sometimes, ‘polymorphism’ is used to describe virtual method calls in object-oriented languages:
  The method call `o.m(...)` may call different methods depending on the class of the object bound to `o`.

A parametric polymorphic type is a strong assertion about a function. Examples:
There is only one terminating pure function of type `'a -> 'a`
There is only one terminating pure function of type `'a * 'b -> 'b * 'a`.
Object language expressions with variable bindings and nested scope

```
let z = 17 in z + z end
let z = 17 in (let z = 22 in 100 * z end) + z end
```

Abstract syntax

```
datatype expr =
  CstI of int
| Var of string
| Let of string * expr * expr
| Prim of string * expr * expr
```

```
val e1 = Let("z", CstI 17, Prim("+", Var "z", Var "z"));

val e2 = Let("z", CstI 17,
  Prim("+", Let("z", CstI 22, Prim("*", CstI 100, Var "z"))));
```
Evaluation of expressions with variables and nested scope

The environment $env$ binds variables to their values. A let-binding $let \ x = \ erhs \ in \ ebody$ evaluates $erhs$ and binds $x$ to the result during the evaluation of $ebody$.

```plaintext
fun lookup [] x = raise Fail (x ^ " not found")
  | lookup ((y, v)::r) x = if x=y then v else lookup r x;

fun eval (e : expr) (env : (string * int) list) : int =
  case e of
    CstI i => i
  | Var x => lookup env x
  | Let(x, erhs, ebody) =>
      let val xval = eval erhs env
      val env1 = (x, xval) :: env
      in eval ebody env1 end
  | Prim("+", e1, e2) => eval e1 env + eval e2 env
  | Prim("*", e1, e2) => eval e1 env * eval e2 env
  | Prim("-", e1, e2) => eval e1 env - eval e2 env
  | Prim _ => raise Fail "unknown primitive"

fun run e = eval e []
```

Lars Birkedal (PLS, ITU)
Introduction to ML, II
Spring 2009 9 / 31
Checking that every variable occurrence is bound (declared)

A variable occurrence bound by an enclosing let-binding is called *bound*; otherwise it is *free*.

In $let\ x = 1 + z\ in\ (let\ y = 2\ in\ x + y\ end) + y\ end$, variable $x$ occurs bound, $z$ free, and $y$ both bound and free.

A simple check: Is every variable occurrence bound by an enclosing let-binding?

If so, we say the expression is *closed*.

This is checked by e.g. Java and Pascal compilers. Classic C compilers check it for variables but not functions.
The call `closedin e env` returns true if every variable in `e` is bound or appears in the list `env` of variables:

```ml
fun closedin (e : expr) (env : string list) : bool =
  case e of
  | CstI i => true
  | Var x => member x env
  | Let(x, erhs, ebody) =>
    let val env1 = x :: env
    in closedin erhs env andalso closedin ebody env1 end
  | Prim(ope, e1, e2) => closedin e1 env andalso closedin e2 env;
```

An expression is closed if it is closed in the empty environment:

```ml
fun closed1 e = closedin e [];
```

A closed expression can be evaluated without causing the interpreter to fail.
Computing sets of free variables

Alternatively, we may compute the set of free variables and check that it is empty. We represent sets as lists. This is simple but inefficient; we could use binary trees or hashsets for efficiency.

union(xs, ys) is $xs \cup ys$, the set of all elements in $xs$ or $ys$, without duplicates:

\[
\begin{align*}
\text{fun union ([], ys) = ys} \\
\text{union (x::xr, ys) = if member x ys then union(xr, ys) else x :: union(xr, ys)}
\end{align*}
\]

minus(xs, ys) is $xs \setminus ys$, the set of all elements in $xs$ but not in $ys$:

\[
\begin{align*}
\text{fun minus ([], ys) = []} \\
\text{minus (x::xr, ys) = if member x ys then minus(xr, ys) else x :: minus(xr, ys)}
\end{align*}
\]
Find all variables that occur free in expression e

```ml
fun freevars e : string list =
  case e of
    CstI i => []
  | Var x => [x]
  | Let(x, erhs, ebody) =>
    union (freevars erhs, minus (freevars ebody, [x]))
  | Prim(ope, e1, e2) => union (freevars e1, freevars e2);
```

Alternative definition of the closedness check:

```ml
fun closed2 e = (freevars e = [])
```
Towards compilation of expressions

So far variables in expressions have been represented as names (strings).
In compiled programs, variables are usually represented as addresses or indexes or offsets: integers, simply.
To make expressions more machine-like, we replace symbolic variable names with numerical indexes.

```
datatype texpr = (* target expressions *)
                   TCstI of int
               | TVar of int  (* index into runtime environment)
               | TLet of texpr * texpr  (* erhs and ebody)
               | TPrim of string * texpr * texpr
```

Translating variable name to variable index at compile-time
A compile-time environment \texttt{cenv} is a list of variable names. Function \texttt{getindex} determines the run-time index of a variable as its position \((0, 1, \ldots)\) in \texttt{cenv}:

```
fun getindex [] x = raise Fail "Variable not found"
| getindex (y::yr) x = if x=y then 0 else 1 + getindex yr x
```
Compiling from expr to texpr

Function \texttt{tcomp} is a compiler from the source language \texttt{expr} to the target language \texttt{texpr}.

```
fun tcomp e (cenv : string list) : texpr =
  case e of
    CstI i => TCstI i
  | Var x => TVar (getindex cenv x)
  | Let(x, erhs, ebody) =>
    let val cenv1 = x :: cenv
    in TLet(tcomp erhs cenv, tcomp ebody cenv1) end
  | Prim(ope, e1, e2) =>
    TPrim(ope, tcomp e1 cenv, tcomp e2 cenv);
```

The order of variable names in \texttt{cenv} at compile-time must reflect the order of variable values at run-time.
The compiler works only for closed expressions — a free variable cannot be compiled to an index.

What is \texttt{Let("x", CstI 17, Prim("+", Var "x", CstI 33))} compiled to?

What about \texttt{Let("x", CstI 17, Prim("+", Var "x", Let("x", CstI 34, Var "x")))}?
Execution of target expressions

We define a new interpreter `teval` to evaluate target expressions with variable indexes.

The run-time environment `renv` is a list of variable values (integers for now).

```haskell
fun teval (e : texpr) (renv : int list) : int =
  case e of
    TCstI i => i
  | TVar x => List.nth(renv, x)
  | TLet(erhs, ebody) =>
    let val xval = teval erhs renv
    val renv1 = xval :: renv
    in teval ebody renv1 end
  | TPrim("+", e1, e2) => teval e1 renv + teval e2 renv
  | TPrim("*", e1, e2) => teval e1 renv * teval e2 renv
  | TPrim("-", e1, e2) => teval e1 renv - teval e2 renv
  | TPrim _ => raise Fail "unknown primitive"
```

At run-time there are no variable names, only variable indexes. The names were removed at compile-time.

Library function `List.nth(i, xs)` returns the `i`’th element of list `xs`, counting from 0.
A stack machine is an abstract machine with an evaluation stack for storing intermediate results.

A ‘constant’ instruction (such as 7) pushes the constant onto the stack top (shown to the right).

The ‘+’ instruction pops the two top-most values, adds them, and puts the result back on the stack.

An SML datatype for representing stack machine instructions:

```plaintext
datatype rinstr =
    RCstI of int | RAdd | RSub | RMul | RDup | RSwap
```

Lars Birkedal (PLS, ITU)
Introduction to ML, II
Spring 2009
The effect of stack machine instructions, schematically (stack-top is to the right):

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Stack before</th>
<th>Stack after</th>
</tr>
</thead>
<tbody>
<tr>
<td>RCst $i$</td>
<td>$s$</td>
<td>$s, i$</td>
</tr>
<tr>
<td>RAdd</td>
<td>$s, i_1, i_2$</td>
<td>$s, (i_1 + i_2)$</td>
</tr>
<tr>
<td>RSub</td>
<td>$s, i_1, i_2$</td>
<td>$s, (i_1 - i_2)$</td>
</tr>
<tr>
<td>RMul</td>
<td>$s, i_1, i_2$</td>
<td>$s, (i_1 \times i_2)$</td>
</tr>
<tr>
<td>RDup</td>
<td>$s, i$</td>
<td>$s, i, i$</td>
</tr>
<tr>
<td>RSwap</td>
<td>$s, i_1, i_2$</td>
<td>$s, i_2, i_1$</td>
</tr>
</tbody>
</table>
An implementation of a stack machine in SML

The stack is represented by a list, with the stack top at the head of the list (to the left):

```sml
fun reval ([] : rinstr list) (v :: _) = v
| reval ([] : rinstr list) [] = raise Fail "reval: no result"
| reval (inst :: rest) stk =
  case (inst, stk) of
    (RCstI i, stk) => reval rest (i::stk)
    (RAdd, i2 :: i1 :: stkr) => reval rest ((i1+i2)::stkr)
    (RSub, i2 :: i1 :: stkr) => reval rest ((i1-i2)::stkr)
    (RMul, i2 :: i1 :: stkr) => reval rest ((i1*i2)::stkr)
    (RDup, i1 :: stkr) => reval rest (i1 :: i1 :: stkr)
    (RSwap, i2 :: i1 :: stkr) => reval rest (i1 :: i2 :: stkr)
```

When there are no more instructions, the value on the stack top is returned.

Evaluating the expression \(10 + 17 \times 17\):

```sml
val rpn1 = reval [RCstI 10, RCstI 17, RDup, RMul, RAdd] []
```
Compilation of a variable-free expression to a sequence of stack machine instructions

This is the same as transforming the expression into postfix form.

```haskell
fun rcomp e : rinstr list =
  case e of
    CstI i => [RCstI i]
  | Prim("+", e1, e2) => rcomp e1 @ rcomp e2 @ [RAdd]
  | Prim("*", e1, e2) => rcomp e1 @ rcomp e2 @ [RMul]
  | Prim("-", e1, e2) => rcomp e1 @ rcomp e2 @ [RSub]
  | Prim _ => raise Fail "unknown primitive"
```

To compile e1 + e2, compile e1 then e2, concatenate their instruction sequences, and put ‘+’ at the end.

The expression 10 + 17 * 17 compiles to [RCstI 10, RCstI 17, RCstI 17, RMul, RAdd].

Net effect principle:
Executing a compiled expression puts the expression’s value on top of the stack, changing nothing below it.

Compiler correctness:
```
eval e [] equals reval (rcomp e) []
```
Often intermediate results and variables are stored in the same stack. This is possible when the language has static nested scopes.

```haskell
datatype sinstr =
    SCstI of int (* push integer *)
  | SVar of int (* push variable from env *)
  | SAdd (* pop args, push sum *)
  | SSub (* pop args, push diff. *)
  | SMul (* pop args, push product *)
  | SPop (* pop value/unbind var *)
  | SSwap (* exchange top and next *)
```

The expression `let z = 17 in z + z end` would be compiled to these instructions:

```
SCstI 17, SVar 0, SVar 1, SAdd, SSwap, SPop
```

The purpose of `SSwap, SPop` is to remove the variable binding (of `z`) below the intermediate result (34).
A unified-stack abstract machine

```
fun seval ([] : sinstr list) (v::_) = v
| seval ([] : sinstr list) [] = raise Fail "seval: no result"
| seval (inst :: rest) stk =
  (case (inst, stk) of
    (SCstI i, stk ) => seval rest (i :: stk)
    (SVar i, stk ) => seval rest (List.nth(stk, i)
    (SAdd, i2::i1::stkr) => seval rest (i1+i2 :: stkr)
    (SSub, i2::i1::stkr) => seval rest (i1-i2 :: stkr)
    (SMul, i2::i1::stkr) => seval rest (i1*i2 :: stkr)
    (SPop, _ :: stkr) => seval rest stkr
    (SSwap, i2::i1::stkr) => seval rest (i1::i2::stkr))
```
The evaluation of \( \text{let } z = 17 \text{ in } z + z \text{ end} \), that is \( \text{SCstI 17, SVar 0, SVar 1, SAdd, SSwap, SPop} \):

---

**Expression in postfix**

<table>
<thead>
<tr>
<th>SCstI 17, SVar 0, SVar 1, SAdd, SSwap, SPop</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \Rightarrow ) SVar 0, SVar 1, SAdd, SSwap, SPop</td>
</tr>
<tr>
<td>( \Rightarrow ) SVar 1, SAdd, SSwap, SPop</td>
</tr>
<tr>
<td>( \Rightarrow ) SAdd, SSwap, SPop</td>
</tr>
<tr>
<td>( \Rightarrow ) SSwap, SPop</td>
</tr>
<tr>
<td>( \Rightarrow ) SPop</td>
</tr>
<tr>
<td>( \Rightarrow ) (empty)</td>
</tr>
</tbody>
</table>

---
Compiling to a machine with only one stack

The compile-time variable environment contains variable names, and dummies \texttt{Intrm} for intermediate values. The corresponding run-time environment positions will hold variable values resp. intermediate results.

\begin{verbatim}
datatype rtvalue =
  Bound of string       (* A bound variable *)
  | Intrm                (* An intermediate result *)
\end{verbatim}
fun scomp e (cenv : rtvalue list) : sinstr list =
case e of
  CstI i => [SCstI i]
| Var x => [SVar (getindex cenv (Bound x))]
| Let(x, erhs, ebody) =>
  scomp erhs cenv @ scomp ebody (Bound x :: cenv) @
| Prim("+", e1, e2) =>
  scomp e1 cenv @ scomp e2 (Intrm :: cenv) @ [SAdd]
| Prim("-", e1, e2) =>
  scomp e1 cenv @ scomp e2 (Intrm :: cenv) @ [SSub]
| Prim("*", e1, e2) =>
  scomp e1 cenv @ scomp e2 (Intrm :: cenv) @ [SMul]
| Prim _ => raise Fail "scomp: unknown operator"
Compiling for the unified-stack machine

The expression \( \text{let } x = 17 \text{ in } x + x \text{ end} \) is compiled to:

\[
\text{SCst 17, SVar 0, SVar 1, SAdd, SSwap, SPop}
\]

The expression \( \text{let } z = 17 \text{ in let } z = 22 \text{ in } 100 \times z \text{ end} + z \text{ end} \) is compiled to:

\[
\text{SCstI 17, SCstI 22, SCstI 100, SVar 1, SMul, SSwap, SPop, SVar 1, SAdd, SSwap, SPop}
\]
Bytecodes for stack machines
Internally, each instruction for a stack machine is often represented by a few small integers. For example:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Bytecode</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCst i</td>
<td>0 i</td>
</tr>
<tr>
<td>SVar x</td>
<td>1 x</td>
</tr>
<tr>
<td>SAdd</td>
<td>2</td>
</tr>
<tr>
<td>SSub</td>
<td>3</td>
</tr>
<tr>
<td>SMul</td>
<td>4</td>
</tr>
<tr>
<td>SPop</td>
<td>5</td>
</tr>
<tr>
<td>SSwap</td>
<td>6</td>
</tr>
</tbody>
</table>
An abstract stack machine in Java (file Machine.java)

class Machine {
    final static int
        CST = 0, VAR = 1, ADD = 2, SUB = 3, MUL = 4, POP = 5, SWAP = 6;

    static int seval(int[] code) {
        int[] stack = new int[1000]; // evaluation and variable
        int sp = -1; // pointer to current stack
        int pc = 0; // program counter
        int instr; // current instruction

        while (pc < code.length)
            switch (instr = code[pc++]) {
                case CST:
                    stack[sp+1] = code[pc++]; sp++; break;
                case VAR:
                    stack[sp+1] = stack[sp-code[pc++]]; sp++; break;
                case ADD:
                    stack[sp-1] = stack[sp-1] + stack[sp]; sp--; break;
                case SUB:
                    stack[sp-1] = stack[sp-1] - stack[sp]; sp--; break;
                ...
            }
        return stack[sp];
    }
}
SML: higher-order functions

fun map f [] = []
  | map f (x::xr) = f x :: map f xr;

SML: anonymous functions

fun double x = 2 * x;

map double [4, 5, 89];

map (fn x => 2 * x) [4, 5, 89];
SML: the list iterator foldr

An application foldr f e xs replaces ‘::’ by f and [] by e in list xs:

fun foldr f e [] = e
  | foldr f e (x::xr) = f(x, foldr f e xr);

fun len xs = foldr (fn (_, res) => 1+res) 0 xs;

fun sum xs = foldr (fn (x, res) => x+res) 0 xs;

fun prod xs = foldr (fn (x, res) => x*res) 1 xs;

fun map g xs = foldr (fn (x, res) => g x :: res) [] xs;

Using foldr and map to compute freevars for arbitrary primitives

fun freevars e : string list =
  case e of
    CstI i => []
  | Var x => [x]
  | Let(x, erhs, ebody) =>
      union (freevars erhs, minus (freevars ebody, [x]))
  | Prim(ope, es) => foldr union [] (map freevars es)
fun tfold f e Lf = e
  | tfold f e (Br(v,t1,t2)) = f(v, tfold f e t1, tfold f e t2);

fun sumtree t = tfold (fn (v, r1, r2) => v + r1 + r2) 0 t;