Department of Computer Science and Engineering
University of California at San Diego

CSE 230
Winter 1998

Problem Set 8 Solutions

1. The task here is to formalize the description of the semantics of objects and classes as higher-order functions. Imagine an object-oriented extension of ML with single inheritance, initialization of objects at the time they are declared, and dynamic binding of method names to method definitions. Define an abstract syntax for class definitions and object declarations in this language, and corresponding ML concrete datatypes. Implement two ML functions: one that given a class definition, produces a function that can generate objects of that class, and a second that processes object declarations.

   Write a brief description of your implementation strategy, and give an interesting small example of object-oriented programming using your implementation.

   [Solution due to Dan Clouse.] First some utilities.

```
exception abort;

fun error (str:string) =
    let val dummy = print str
    in raise abort end;

fun assoc [] oldassocs a = oldassocs a
  | assoc ((a1, b1) :: rest) oldassocs a =
        if a = a1 then b1
        else assoc rest oldassocs a;
```

   Now datatypes for class descriptions including for lambda expressions, commands, integer expressions, and variables.

```
datatype variable = Var of string;

datatype object = Other of (variable -> int list -> int)
                | Self;

datatype int_expression = Constant of int
                        | Contents of variable
                        | Plus of (int_expression * int_expression)
                        | Minus of (int_expression * int_expression)
                        | Times of (int_expression * int_expression)
                        | Send of (object * variable * (int_expression list));

datatype command =
     Assign_Stmt of (variable * int_expression)
   | If_Stmt of (int_expression * (command list) * (command list))
   | Return_Stmt of (int_expression);

datatype lambda_expr = Lambda_Expr of ((variable list) * (command list));

datatype class_descr =
  Class_Descr of (parent * (variable list) * (variable * lambda_expr) list)
and parent = No_Parent | Parent of class_descr;
```

Now some routines to create and manipulate stores. Updates can be made to existing values within an existing store without reassignment to the store.

```
datatype ''a binding = No_Binding | Binding of (variable * ''a ref)

fun make_string (Var name) = name;

val null_store = [];

fun find [] id = No_Binding
  | find ((name, value) :: tail) id =
    if name = id then Binding (name, value)
    else find tail id

fun get_value_ref No_Binding = error ("Internal error: No binding")
  | get_value_ref (Binding (id, value_ref)) = value_ref;

fun fetch store id =
    let val found = find store id in
        if found = No_Binding then
            error ((make_string id) ^ ": Uninitialized variable")
        else
            !(get_value_ref found)
    end
```

The function `update` changes the value in store of the given `id` to the given value. The updating of store takes place inside this routine, so if the variable is currently in store this change takes place even if store is not reassigned outside. If the variable is not currently in the store a new store containing the variable is returned. `update_list` updates the given store by changing the variables in the id list to the corresponding values in the value list. New variables are added to the returned store.

```
fun update store id value =
    let val found = find store id in
        if found = No_Binding then
            ((id, ref value) :: store)
        else
            let val ptr = (get_value_ref found)
                val dummy = ptr := value
            in
                store
            end
    end

fun update_list store (id :: id_tail) (value :: value_tail) =
        update_list (update store id value) id_tail value_tail
  | update_list store [] [] = store
  | update_list store (id :: id_tail) [] =
        error "Not enough actual params"
  | update_list store [] (value :: value_tail) =
        error "Too many actual params"

fun update_list_to_value store (id :: id_tail) value =
        update_list_to_value (update store id value) id_tail value
  | update_list_to_value store [] value = store

fun init_state_store No_Parent state_var_ids =
        update_list_to_value null_store state_var_ids 0
```

```
  | init_state_store (Parent (Class_Descr (grandma, parent_state, unused)))
        state_var_ids = update_list_to_value
            (init_state_store grandma parent_state) state_var_ids 0
```

The `Method_Store` data type allows storage of non-equality datatypes. This is not supported by the state store data type, so it was necessary to implement two different types of store.

```
datatype Method_Store = Store of (variable -> lambda_expr);

val m_null_store = let fun f (Var str) = error ("No such method: " ^ str)
                   in Store f end;

fun m_fetch (Store f) v = f v;

fun m_update (Store f) v a = Store (assoc[(v, a)] f);

fun m_update_list store ((id, value) :: tail) =
    m_update (m_update_list store tail) id value
  | m_update_list store [] = store;
```

The routine `combine_method_lists` combines the current methods with those of all ancestors. The result is a single method list with ancestors later in the list.

```
fun combine_method_lists No_Parent methods = methods
  | combine_method_lists
            (Parent (Class_Descr (grandma, unused, parent_methods)))
            methods =
        methods @ (combine_method_lists grandma parent_methods)

fun init_method_store parent methods =
    m_update_list m_null_store (combine_method_lists parent methods)
```

The routine `generate_class` returns a function that when called generates an object of the class described by the given class description.

```
fun send object msg_name param_values =
    object msg_name param_values

fun generate_class (Class_Descr (parent, state_var_ids, methods)) =
    fn init_actual_params =>
        let
            val return_var_id = (Var "$return$var$");
```

The local variable `perm_state_store` contains the values of the state of the object including any inherited state. These can be changed by calls to update. `perm_method_store` contains the methods for the object including any inherited methods.

```
            val perm_state_store = init_state_store parent state_var_ids

            val perm_method_store = init_method_store parent methods

            fun eval_int_expr store (Constant n)        = n
              | eval_int_expr store (Contents id)       = fetch store id
              | eval_int_expr store (Plus (e1, e2))     =
                    (eval_int_expr store e1) + (eval_int_expr store e2)
              | eval_int_expr store (Minus (e1, e2))    =
```

```
        (eval_int_expr store e1) - (eval_int_expr store e2)
  | eval_int_expr store (Times (e1, e2))    =
        (eval_int_expr store e1) * (eval_int_expr store e2)
  | eval_int_expr store (Send (Other obj, msg_id, expr_list)) =
        (send obj msg_id (eval_int_expr_list store expr_list))
  | eval_int_expr store (Send (Self, msg_id, expr_list)) =
        (send_self msg_id (eval_int_expr_list store expr_list))

and eval_int_expr_list store [] = []
  | eval_int_expr_list store (expr :: tail) =
        ((eval_int_expr store expr) ::
         (eval_int_expr_list store tail))

and execute_command store (Assign_Stmt (lvalue_id, expr)) =
        update store lvalue_id (eval_int_expr store expr)
  | execute_command store (If_Stmt (expr, true_commands,
                                      false_commands)) =
        if (eval_int_expr store expr) = 0 then
            (execute_command_list store false_commands)
        else
            (execute_command_list store true_commands)
  | execute_command store (Return_Stmt expr) =
        update store return_var_id (eval_int_expr store expr)

and execute_command_list store [] = store
  | execute_command_list store (command :: tail) =
        let val dummy = (execute_command store command) in
            execute_command_list store tail
        end
```

The helper function `execute_method` executes the method identified by the given lambda expression. Values for the arguments are found in `param_values`. This function returns the return value from any `return` command executed in the method. `send_self` searches the permanent method store for a method named `msg_name`, and executes the matching method using `param_values` as the value for a parameterized message. It is a fatal error for `msg_name` to not be found in the permanent method store.

```
and execute_method (Lambda_Expr (formal_params, commands))
                   param_values =
    fetch (execute_command_list
           (update (update_list perm_state_store formal_params
            param_values) return_var_id 0)
           commands)
          return_var_id

and send_self msg_name param_values =
    execute_method (m_fetch perm_method_store msg_name)
       param_values
```

An `init` message is sent to the object before returning, with parameters as passed by the user.

```
val dummy = send_self (Var "init") init_actual_params

in send_self end;
```

Here are some examples. First, bank account class 2 adds the maintenance of a count of all transactions while still inheriting the `get_balance` method from bank account class 1.

```
val init_method_1 = Lambda_Expr (
    [(Var "amount")],
    [Assign_Stmt ((Var "balance"), (Contents (Var "amount")))]);
val deposit_method_1 = Lambda_Expr (
    [(Var "amount")], [Assign_Stmt ((Var "balance"),
        (Plus (Contents (Var "balance"), Contents (Var "amount"))))]);
val withdraw_method_1 = Lambda_Expr (
    [(Var "amount")], [Assign_Stmt ((Var "balance"),
        (Minus (Contents (Var "balance"), Contents (Var "amount"))))]);
val get_balance_method_1 = Lambda_Expr (
    [], [Return_Stmt (Contents (Var "balance"))]);

val acct_method_list_1 =
    [((Var "init"), init_method_1),
     ((Var "deposit"), deposit_method_1),
     ((Var "withdraw"), withdraw_method_1),
     ((Var "get_balance"), get_balance_method_1)];

val bank_account_class_descr_1 =
    Class_Descr (No_Parent, [(Var "balance")], acct_method_list_1)

val init_method_2 = Lambda_Expr (
    [(Var "init_bal"), (Var "init_count")],
    [Assign_Stmt ((Var "balance"), (Contents (Var "init_bal"))),
     Assign_Stmt ((Var "xaction_count"), (Contents (Var "init_count")))]);
val deposit_method_2 = Lambda_Expr (
    [(Var "amount")],
    [Assign_Stmt ((Var "balance"),
        (Plus (Contents (Var "balance"), Contents (Var "amount")))),
     Assign_Stmt ((Var "xaction_count"),
        (Plus (Contents (Var "xaction_count"), Constant 1)))]);
val withdraw_method_2 = Lambda_Expr (
    [(Var "amount")],
    [Assign_Stmt ((Var "balance"),
        (Minus (Contents (Var "balance"), Contents (Var "amount")))),
     Assign_Stmt ((Var "xaction_count"),
        (Plus (Contents (Var "xaction_count"), Constant 1)))]);
val get_count_method_2 = Lambda_Expr (
    [], [Return_Stmt (Contents (Var "xaction_count"))]);

val acct_method_list_2 =
    [((Var "init"), init_method_2),
     ((Var "deposit"), deposit_method_2),
     ((Var "withdraw"), withdraw_method_2),
     ((Var "get_xactions"), get_count_method_2)];

val bank_account_class_descr_2 =
    Class_Descr ((Parent bank_account_class_descr_1),
    [(Var "xaction_count")], acct_method_list_2);
```

Now, a dynamic binding example. The return value from a `msg_B` sent to a child object should be 14, to a parent should be 13.

```
val mth_init = Lambda_Expr ([], []);
val mth_A_1 = Lambda_Expr ([], [Return_Stmt (Constant 13)]);
val mth_B_1 = Lambda_Expr ([], [Return_Stmt (Send (Self, (Var "msg_A"), []))]);
```

```
val method_list_1 =
    [((Var "init"), mth_init),
     ((Var "msg_A"), mth_A_1),
     ((Var "msg_B"), mth_B_1)];
val class_descr_1 = Class_Descr (No_Parent, [], method_list_1);

val mth_A_2 = Lambda_Expr ([], [Return_Stmt (Constant 14)]);
val method_list_2 = [((Var "msg_A"), mth_A_2)];
val class_descr_2 = Class_Descr (Parent class_descr_1, [], method_list_2);
```

2. By searching the web, you will find articles on XML that give parts of the answer to this question from different points of view. In a nutshell, XML does not directly allow for embedded executable code, so the wilder claims of Bolcer are not literally valid.