Implementation of MiniML Using OBJ3
---- CSE230 Project Report

Director : Joseph Gougen, Kai Lin
Author :
Huaxia Xia (hxia@cs.ucsd.edu)
Yuanfang Hu (yhu@cs.ucsd.edu)
Xianan Zhang (xzhang@cs.ucsd.edu)

Date : 03/16/2001

Index:

1. Introduction 1
2. Definition of MiniML language 1
   2.1 Syntax of MiniML 1
   2.2 Semantic of MiniML 2
3 . Implementation 3
   3.1 Description of objects 3
   3.2 The OBJ codes to implement ML semantics 3
   3.3 Discussion about the subsort-relations in the implement 3
   3.4 Function, Recursive Function, High-order Function 4
   3.5 Type decision system 4
   3.6 Precedence order 5
   3.7 Parameterized modules, module instantiation 5
   3.8 Efficiency 5
4. Verification on the Implementation 6
   4.1 Verification input codes 6
   4.2 Verification output 7
5. Verification on the ML semantics 10
   5.1 Verify the correctness of the recursive plus function: 10
   5.2 Verify the correctness of the factorial function: 12
   5.3 Verify the correctness of “exponent” function: 13
   5.4 Verify the correctness of “combination” function: 15
6. Conclusion: 17
7. Reference: 17
Appendix: The OBJ3 codes of the implementation 17

1. Introduction

In this project, we are expected to implement semantics for a fragment of ML using OBJ and use it to prove the correctness of some simple ML programs. After two weeks hard work, we get pretty good results. We implement basic data structure such as natural number, list, tuple and stack. Based on these basic data structures, we implement syntax and semantics of expression, assignment statement, function definition and call of ML in corresponding objects. With these objects, we can interpret some ML programs such as **recursive function call**, **high order function call** and we can use them to decide types of different variables including function variable (**type decision**). Because of the limited time, we don't implement list as function's formal arguments. But the basic list operations have been completed.

In this report, we describe the syntax and semantics of the fragment of ML language (we call it MiniML language) in section 2. Implementation is presented in section 3. Especial discussion on type system of this project is given, since it is one of the most important characteristics of this project and embodies some unique features of OBJ. In section 4, we show kinds of typical MiniML programs' interpretation results. In section 5, the partial correctness proof of several standard recursive functions are presented. In the end, we talk about our conclusions we got from this project in section 6.

2. Definition of MiniML language

2.1 Syntax of MiniML

Following is the BNF description of the syntax of MiniML language.

```
<ML> ::= ml(<Statement>)
<Statement> ::= <Exp> | <Assign> | <Fun> | {<Statement >;}
<Exp> ::= <Int> | <Bool> | <Func> |
      <Exp1> + <Exp2> | <Exp1> - <Exp2> | <Exp1> * <Exp2> |
      <Exp1> andalso <Exp2> | <Exp1> orelse <Exp2>|not <Exp>|
      <Exp1> <= <Exp2> | <Exp1> >= <Exp2> |
      <Exp1> < <Exp2> | <Exp1> > <Exp2> |
      <Exp1> is <Exp2> | <Exp1> <> <Exp2> |
      if <Exp1> then <Exp2> else <Exp3>
<Assign> ::= val <Ident> := <Exp>
<Fun> ::= fun <Ident> <List> is <Exp>
<Func> ::= <Ident> <List>
<Int>  ---  an integer
<Bool>  ---  a bool value
<List>  ---  a list
```

Notes:

(1) In ML, syntax of assignment statement is "val Var = Exp". We change "=" to ":=" because in OBJ "=" has been used by equation.

(2) In ML, syntax of justifying two integer expressions are equal is " IntExp = IntExp". We change "=" to "is" because in OBJ "=" has been used by equation.

(3) In ML, syntax of function definition is "fun Var ({Var : Type}) = Exp. We change "=" to "is" because in OBJ "=" has been used by equation. We also change "(" and ")" to "{" and "}", since we look formal arguments as a list in our implementation.

2.2 Semantic of MiniML

Following is the denotational description of the semantics of MiniML language. Value domains:

$\rho \in$ Env = Ident -> (Value + Loc + undefined) environments

$\sigma \in$ States = Ident -> (Value + unused) states

Semantic functions (S $\in$ Statement , E $\in$ Exp)

```
ml : ML -> Env -> States -> Value
   ml [[ S ]] ρσ =
       let
           ρ'σ' = stmt[[ S ]] ρσ
       in
           ρ'(Ident)
       end
stmt: Statement -> Env -> State -> State
   stmt [[ S1;S2 ]] ρσ =
       let
           ρ'σ' = stmt [[ S1 ]] ρσ
       in
           stmt [[ S2 ]] ρ'σ'
       end
exp: Exp -> Env -> States -> Value
   exp [[ E1 + E2 ]] ρσ = exp [[ E1 ]] ρσ + exp [[ E2 ]] ρσ
   exp [[ E1 – E2 ]] ρσ = exp [[ E1]] ρσ - exp [[ E2 ]] ρσ
   exp [[ E1 * E2 ]] ρσ = exp [[ E1 ]] ρσ * exp [[ E2 ]] ρσ
   exp [[ if E1 then E2 else E3 ]] ρσ =
        if exp [[ E1 ]] ρσ then exp [[ E2 ]] ρσ else exp [[ E3 ]]ρσ
   exp [[ E1 < E2 ]] ρσ = exp [[ E1 ]] ρσ < exp [[ E2 ]] ρσ
   exp [[ E1 > E2 ]] ρσ = exp [[ E1 ]] ρσ > exp [[ E2 ]] ρσ
   exp [[ E1 <= E2 ]] ρσ = exp [[ E1 ]] ρσ <= exp [[ E2 ]] ρσ
   exp [[ E1 >= E2 ]] ρσ = exp [[ E1 ]] ρσ >= exp [[ E2 ]] ρσ
   exp [[ E1 is E2 ]] ρσ = exp [[ E1 ]] ρσ is exp [[ E2 ]] ρσ
   exp [[ E1 andalso E2 ]] ρσ =  exp[[E1]] ρσ andalso exp[[E2]] ρσ
   exp [[ E1 orelse E2 ]] ρσ = exp [[ E1 ]] ρσ orelse exp [[ E2 ]]ρσ
   exp [[ not E1 ]] ρσ = not exp [[ E1 ]] ρσ
   exp [[ Bool ]] ρσ = Bool
   exp [[ Int ]] ρσ = Int
   exp [[ Ident ]] ρσ = ρ(Ident)
fun:   Fun -> Env -> States
   fun [[ fun Ident List is E ]] ρσ
     = ρ[Ident→fun Ident List is E] σ[Ident→fun Ident List is E]
func: Func -> Env -> States -> Value
   func [[ Ident List ]] ρσ = func [[ ρ(Ident) :List ]] ρσ
   func [[ fun Ident List1 is E :List2 ]] ρσ =
       let
           ρ'σ' = assign[[ List1 := List2 ]] ρσ
       in
           exp [[ E ]] ρ'σ'
       end
assign: Assign -> Env -> States -> States
   assign [[ Ident := E ]] ρσ = ρ[ Ident → ρ(E) ] σ[ Ident → ρ(E)]
```

3 . Implementation

In the part, we describe the structure of the implementation and some tricks.

3.1 Description of objects

a) ZZ implements syntax and semantics of integers.
b) LIST implements syntax and semantics of lists. The type of elements in list is not fixed when we define the object LIST. It is determined when list is instantiated.
c) VALUE is the super object of all sorts of values that can be stored in stack, including Int, Bool and definition of function.
d) TUPLE implements syntax and semantics of a couple of elements. The type of elements in tuple is determined when tuple is instantiated.
e) STACK implements syntax and semantics of a run-time stack, on which semantics of MiniML are implemented.
f) EXP implements syntax and semantics of expression construct. An expression can be an integer expression, a boolean expression, a function call or an identifier whose value is an integer or a boolean value.
g) FUN implements syntax and semantics of function definition in MiniML.
h) ASSIGN implements syntax and semantics of assignment statements in MiniML.
i) FUNC implements syntax and semantics of function call in MiniML.
j) STMT implements syntax and semantics of statements in MiniML.
k) TYPE implements type decision for the output. The types of an output can be "int", "bool", "list" and "primetype {*primetype}* ==> type". And we also define the "categories" of the output, i.e. if the output is a value or a function definition.
l) MLSEM interprets MiniML programs using the above objects.

3.2 The OBJ codes to implement ML semantics

The complete codes are in the appendix.

3.3 Discussion about the subsort-relations in the implement

The "subsort" of OBJ is a very powerful and flexible tool to define the relations between objects. We can let sort *A* be a subsort of another existed sort *B* so that *A* can use the well-defined operations in *B*. On the other hand, sort *A* can also be defined to be a supersort of sort B so that new operations defined in *A* can also be applied on *B*.

In the implementation of stack, we define "Value" as the sort of all kinks of data that can be put into stack. Then in the implementation of expression and function, we only need to define the following relations:

```
subsorts Int Bool IntList < Value .
subsorts Fun < Value .
```

Thus we can put data of sorts Int, Bool, IntList or Fun into the stack.

Another example is in the object of STMT. In ML, the statements may include assignments, expressions, function definition, etc. Without subsort, we have to define all kinds of possible combination. But now the following two sentences are enough:

```
subsorts Exp Fun Assign < Stmt .
op _;_ : Stmt Stmt -> Stmt .
```

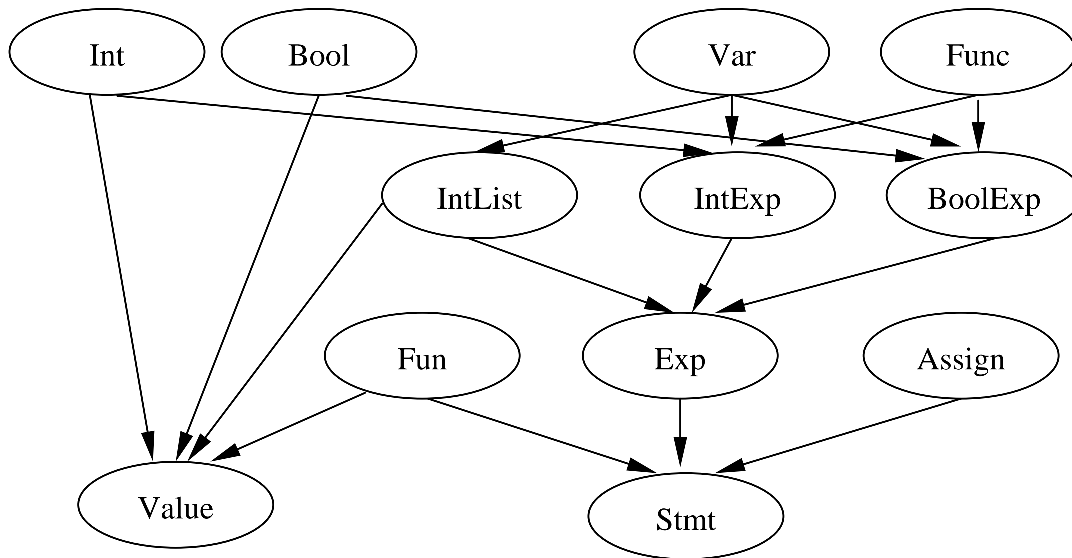
But the subsorts shouldn't be used arbitrarily. Sometimes, if the subsort-relations are not defined properly, there will be something wrong. For example, "Func" denotes for the function call. Following definition will make the reduction of "3 + f{x,y}" fail because "_+_" is an operation for IntExp:

```
subsorts IntExp BoolExp < Exp .
subsorts Func < Exp .
```

In stead, we should use:

```
subsorts IntExp BoolExp < Exp .
Subsorts Func < IntExp BoolExp .
```

The subsort-relations in our implementation are listed as below:



3.4 Function, Recursive Function, High-order Function

As described in the semantics, we implement function call as following:

```
    eq S[[ X Le ]] = [[ fetch(X,S) ]] [[ S ]] Le .
    eq [[ fun X Lv is E ]] [[ S ]] Le = [[ Lv equals Le ]]S [[E]] .
    cq [[Lv equals Le]]S = [[val hd(Lv) := hd(Le)]] ([[tl(Lv) equals
 tl(Le)]]S) if (| Lv | is | Le |) and (0 < | Lv |) .
    eq [[{} equals {} ]] S = S .
```

Hence recursive function call works very well because if *E* contains function call, it will be processed recursive.

High-order function call also works very well. If *Le* contains a function name, the function definition will be assigned to the corresponding formal variable in *Lv*. This looks like re-define the same function with another name.

3.5 Type decision system

Given an ML statement, it is expected to get a type as well as a value. OBJ supports the "overload" mechanism, which makes it easy to build a type decision system via overloading the operator "type" on different sorts.

In the object "TYPE", we define two operators "category" and "type", which respectively decide if the output is a value of a function definition and if the output is an integer, bool or list.

Thus "`red ml(fun f {x,y} is x <=y)`" will give the following output:

```
Result: fn f is fun f{x,y} is x <=y :: primetype*primetype ==> bool
```

More examples on type decision are given in Part 4.

3.6 Precedence order

OBJ3 provides a mechanism of "precedence number" to define the precedence order of the operators. It is a very useful mechanism. For an instance, in out earlier implementation, no explicit precedence order was defined on the operators. Then the "val x := 3 ; x + 5" will be regarded as "(val x := 3 ; x) + 5 ". So we define explicit precedence order as below:

```
Stmt : 120
    op _;_ : Stmt Stmt -> Stmt [assoc prec 120].
Assign : 110
    op val_:=_ : Var Exp -> Assign [prec 110].
Fun: 110
    op fun _ _ is _ : Var ArgList Exp -> Fun [prec 110].
Exp: 85-94
    ops (_andalso_)(_orelse_): BoolExp BoolExp -> BoolExp[prec 90].
    op not_ : BoolExp -> BoolExp [prec 86].
    op _is_ : IntExp IntExp -> BoolExp [prec 92].
    op _<_ : IntExp IntExp -> BoolExp [prec 86].
    op _<=_ : IntExp IntExp -> BoolExp [prec 86].
    op _>_ : IntExp IntExp -> BoolExp [prec 86].
    op _>=_ : IntExp IntExp -> BoolExp [prec 86].
    op _<>_ : IntExp IntExp -> BoolExp [prec 86].
    op _<>_ : BoolExp BoolExp -> BoolExp [prec 86].
    op if_then_else_ : BoolExp IntExp IntExp -> IntExp [prec 94].
    op if_then_else_ : BoolExp BoolExp BoolExp -> BoolExp [prec 94].
    ops (_+_) (_-_) (_*_) : IntExp IntExp -> IntExp [prec 85].
Func: 80
    op __ : Var ExpList -> Func [prec 80].
```

3.7 Parameterized modules, module instantiation

OBJ3 supports parameterized modules and flexible module instantiation. In our implementation, we define LIST as a parameterized module, then instantiate it to be IntList, BoolList, ArgList and ExpList. For example, ArgList is defined as:

```
    dfn ArgList is (LIST *(op [] to {}, op [_] to {_}))[Var] .
```

3.8 Efficiency

OBJ3 does reductions using substitution based on equations. So sometimes different kinds of definition may be greatly different in efficiency, although they may be equivalent in logic.

A good example comes from the definition of "ml". In the earlier implementation, we define "ml" as below:

```
  eq ml(P) = category(top([[P]]empty)) name*(top([[P]]empty)) is
             value* (top([[P]]empty))::(top([[P]]empty)) .
```

Then the number of rewriting for “ml(val x := 3 ; x * x)” is 91:

```
reduce in MLSEM : ml(val x := 3 ; x * x)
rewrites: 91
result Result: val (it).EXP is 9 :: int
```

If we define ml as:

```
eq result(< X,V >) = category(V) X is V :: type(V) .
eq ml(P) = result (top ([[P]]empty)) .
```

Then the number of rewriting is 25. (See section 4.2).

4. Verification on the Implementation

In the part, we give some examples to show the correctness of our implementation. The examples are in eight categories, which try to show the following things: (1) list, (2) assignment, (3) function definition, (4) basic function call, (5) recursive function call, (6) high-order function call, (7) type decision and (8) stack.

4.1 Verification input codes

The input file “test.obj” is as below:

```
open MLSEM .
ops i j : -> Int .
op t : -> Tuple .
op s : -> Stack .
op f : -> Fun .
op e : -> Exp .
op fc : -> Func .
op a : -> Assign .
op p : -> Stmt .
ops x y z it : -> Var .
ops f1 f2 : -> Var .
***> ******************Part 1. List ... ************************************
red ml(hd[2,3,4]) .
red ml(tl[2,3,4]) .
red ml(1 :: [2,3,4]) .
red ml([1]@[2,3,4]@[5]@[]) .
***> ******************Part 2. Assignment (Assign) ... *********************
red ml(val x := 3) .
red ml(val x := 3 ; x * x) .
***> Assignment can change the value & type of a variable ...
red ml(val x := 3 ; val y := 2 ; val x := x < 2 ; x ) .
***> ******************Part 3. Function definition (Fun) ... ***************
***> a function with one argument and the output is an integer ...
red ml(val x := 3 ; fun f1{y} is 3 * y ) .
***> function with two arguments and the output is an bool ...
red ml(val x := 3 ; fun f1{x,y} is x andalso y ) .
red ml(val x := 3 ; fun f1{x,y} is x <= y ) .
***> ******************Part 4. Basic Function calling (Func) ... ***********
red ml(val x := 3 ; fun f1{x,y} is x * y ; f1{5,3}) .
red ml(val x := 3 ; fun f1{x,y} is x * y ; f1{5,3}; it) .
red ml(val x := 3 ; fun f1{x,y} is x * y ; f1{5,x}; x) .
red ml(fun f1{x,y} is if y then (x + 1) else (x - 1) ; f1{3,false }) .
***> ******************Part 5. Recursive Function calling (Func) ... *******
red ml(fun f1{x} is (if x <= 0 then 1 else x * f1{x - 1}) ; f1{5}) .
red ml(fun f1{x} is (if x <= 0 then 1 else (x * f1{x - 1})) ; val y := 4 ;
f1{y}) .
***> ******************Part 6. High-order Function calling (Func) ... ******
red ml(fun f1{x} is x * x ; fun f2{x,y} is y{x} * 2 ; f2{3,f1}) .
```

```
***> ******************Part 7. Type Decision System (Type) ... *************
red ml(1) .
red ml(true) .
red ml([1,2]) .
red ml(val x := 3) .
red ml(val x := 3 ; x) .
red ml(fun f1{y} is 3 * y) .
red ml(fun f1{x,y} is x andalso y ) .
red ml(fun f1{x,y} is x andalso y ; f1{true, false}) .
***> ******************Part 8. State of the Stack (Stack)  ... *************
red [[val x := 3]]empty .
red [[val x := 3 ; x]]empty .
red [[val x := 3 ; fun f1{x,y} is x * y ]]empty .
red [[val x := 3 ; fun f1{x,y} is x * y ; f1{5,3}]]empty .
red [[val x := 3 ; fun f1{x,y} is x * y ; f1{5,3}; val x := true]]empty .
```

4.2 Verification output

```
                     \|||||||||||||||||/
                   --- Welcome to OBJ3 ---
                     /|||||||||||||||||\
         OBJ3 version 2.04oxford built: 1994 Feb 28 Mon 15:07:40
            Copyright 1988,1989,1991 SRI International
                   2001 Mar 17 Sat 0:05:17
OBJ> in project_xia_0316_final
==========================================
obj ZZ
==========================================
obj LIST
==========================================
obj VALUE
==========================================
obj TUPLE
==========================================
obj STACK
==========================================
obj EXP
==========================================
obj FUN
==========================================
obj ASSIGN
==========================================
obj FUNC
==========================================
obj STMT
==========================================
obj TYPE
==========================================
obj MLSEM
OBJ> in test
==========================================
open MLSEM
==========================================
ops i j : -> Int .
==========================================
op t : -> Tuple .
==========================================
op s : -> Stack .
==========================================
op f : -> Fun .
==========================================
op e : -> Exp .
==========================================
```

```
op fc : -> Func .
==========================================
op a : -> Assign .
==========================================
op p : -> Stmt .
==========================================
ops x y z it : -> Var .
==========================================
ops f1 f2 : -> Var .
==========================================
***> ******************Part 1. List ... ***********************************
==========================================
reduce in MLSEM : ml(hd [2,3,4])
rewrites: 8
result Result: val (it).EXP is 2 :: int
==========================================
reduce in MLSEM : ml(tl [2,3,4])
rewrites: 8
result Result: val (it).EXP is [3,4] :: int list
==========================================
reduce in MLSEM : ml(1 :: [2,3,4])
rewrites: 8
result Result: val (it).EXP is [1,2,3,4] :: int list
==========================================
reduce in MLSEM : ml([1] @ [2,3,4] @ [5] @ [])
rewrites: 10
result Result: val (it).EXP is [1,2,3,4,5] :: int list
==========================================
***> ******************Part 2. Assignment (Assign) ... *********************
==========================================
reduce in MLSEM : ml(val x := 3)
rewrites: 7
result Result: val x is 3 :: int
==========================================
reduce in MLSEM : ml(val x := 3 ; x * x)
rewrites: 25
result Result: val (it).EXP is 9 :: int
==========================================
***> Assignment can change the value & type of a variable ...
==========================================
reduce in MLSEM : ml(val x := 3 ; val y := 2 ; val x := x < 2 ; x)
rewrites: 34
result Result: val (it).EXP is false :: bool
==========================================
***> ******************Part 3. Function definition (Fun) ... ***************
==========================================
***> a function with one argument and the output is an integer ...
==========================================
reduce in MLSEM : ml(val x := 3 ; fun f1 {y} is 3 * y)
rewrites: 13
result Result: fn f1 is fun f1 {y} is 3 * y :: (primetype ==> int)
==========================================
***> function with two arguments and the output is an bool ...
==========================================
reduce in MLSEM : ml(val x := 3 ; fun f1 {x,y} is x andalso y)
rewrites: 19
result Result: fn f1 is fun f1 {x,y} is x andalso y :: ((primetype *
    primetype) ==> bool)
==========================================
reduce in MLSEM : ml(val x := 3 ; fun f1 {x,y} is x <= y)
rewrites: 19
result Result: fn f1 is fun f1 {x,y} is x <= y :: ((primetype * primetype) ==>
    bool)
```

```
==========================================
***> *****************Part 4. Basic Function calling (Func) ... **********
==========================================
reduce in MLSEM : ml(val x := 3 ; fun f1 {x,y} is x * y ; f1 {5,3})
rewrites: 78
result Result: val (it).EXP is 15 :: int
==========================================
Ambiguous term, two parses are:
ml(val x := (3).NzNat ; fun f1 {x,y} is x * y ; f1 {(5).NzNat,(3).NzNat} ;
    (it).MLSEM) -versus-
ml(val x := (3).NzNat ; fun f1 {x,y} is x * y ; f1 {(5).NzNat,(3).NzNat} ;
    (it).EXP)
differences are:

Arbitrarily taking the first as correct.
reduce in MLSEM : ml(val x := 3 ; fun f1 {x,y} is x * y ; f1 {5,3} ;
    (it).MLSEM)
rewrites: 87
result Result: val (it).EXP is 15 :: int
==========================================
reduce in MLSEM : ml(val x := 3 ; fun f1 {x,y} is x * y ; f1 {5,x} ;
    x)
rewrites: 102
result Result: val (it).EXP is 3 :: int
==========================================
reduce in MLSEM : ml(fun f1 {x,y} is if y then x + 1 else x - 1 ; f1
    {3,false})
rewrites: 80
result Result: val (it).EXP is 2 :: int
==========================================
***> *****************Part 5. Recursive Function calling (Func) ... *******
==========================================
reduce in MLSEM : ml(fun f1 {x} is if x <= 0 then 1 else x * f1 {x -
    1} ; f1 {5})
rewrites: 382
result Result: val (it).EXP is 120 :: int
==========================================
reduce in MLSEM : ml(fun f1 {x} is if x <= 0 then 1 else x * f1 {x -
    1} ; val y := 4 ; f1 {y})
rewrites: 333
result Result: val (it).EXP is 24 :: int
==========================================
***> *****************Part 6. High-order Function calling (Func) ... ******
==========================================
reduce in MLSEM : ml(fun f1 {x} is x * x ; fun f2 {x,y} is y {x} *
    2 ; f2 {3,f1})
rewrites: 121
result Result: val (it).EXP is 18 :: int
==========================================
***> *****************Part 7. Type Decision System (Type) ... ************
==========================================
reduce in MLSEM : ml(1)
rewrites: 7
result Result: val (it).EXP is 1 :: int
==========================================
reduce in MLSEM : ml(true)
rewrites: 7
result Result: val (it).EXP is true :: bool
==========================================
reduce in MLSEM : ml([1,2])
rewrites: 7
result Result: val (it).EXP is [1,2] :: int list
==========================================
```

```
reduce in MLSEM : ml(val x := 3)
rewrites: 7
result Result: val x is 3 :: int
==========================================
reduce in MLSEM : ml(val x := 3 ; x)
rewrites: 16
result Result: val (it).EXP is 3 :: int
==========================================
reduce in MLSEM : ml(fun f1 {y} is 3 * y)
rewrites: 10
result Result: fn f1 is fun f1 {y} is 3 * y :: (primetype ==> int)
==========================================
reduce in MLSEM : ml(fun f1 {x,y} is x andalso y)
rewrites: 16
result Result: fn f1 is fun f1 {x,y} is x andalso y :: ((primetype *
    primetype) ==> bool)
==========================================
reduce in MLSEM : ml(fun f1 {x,y} is x andalso y ; f1 {true,false})
rewrites: 75
result Result: val (it).EXP is false :: bool
==========================================
***> *****************Part 8. State of the Stack (Stack)  ... *************
==========================================
reduce in MLSEM : [[val x := 3]]empty
rewrites: 2
result Stack: push(< x,3 >,empty)
==========================================
reduce in MLSEM : [[val x := 3 ; x]]empty
rewrites: 11
result Stack: push(< (it).EXP,3 >,push(< x,3 >,empty))
==========================================
reduce in MLSEM : [[val x := 3 ; fun f1 {x,y} is x * y]]empty
rewrites: 5
result Stack: push(< f1,fun f1 {x,y} is x * y >,push(< x,3 >,empty))
==========================================
reduce in MLSEM : [[val x := 3 ; fun f1 {x,y} is x * y ; f1 {5,3}]]
    empty
rewrites: 73
result Stack: push(< (it).EXP,15 >,push(< f1,fun f1 {x,y} is x * y >,
    push(< x,3 >,empty)))
==========================================
reduce in MLSEM : [[val x := 3 ; fun f1 {x,y} is x * y ; f1 {5,3} ;
    val x := true]]empty
rewrites: 76
result Stack: push(< x,true >,push(< (it).EXP,15 >,push(< f1,fun f1
    {x,y} is x * y >,push(< x,3 >,empty))))
```

5. Verification on the ML semantics

5.1 Verify the correctness of the recursive plus function:

a) The definition of the function using ML:

$$fun\ double(N : \text{int}) = if\ \ N = 0\ then\ 0\ else\ 2 + double(N-1);$$

b) The explanation of the partial correctness:

Precondition is: $0 <= x$

Postcondition is: $s[[body]] = 2 * x$

Induction proof:

$$(1)\ x = 0 \Rightarrow double(x) = 2 * x$$
$$(2)\ x > 0 \Rightarrow \forall N,\ if\ \ 0 \le N < x,\ and\ double(N) = 2 * N,$$
$$then\ double(x) = 2 * x$$
$$If\ \ both\ (1)\ and\ (2)\ are\ true,\ we\ can\ know:$$
$$\forall x >= 0,\ double(x) = 2 * x$$

c) The OBJ code used for partial correctness verification:

```
th DOUBLE is pr ZZ .
  op db_ : Int -> Int [prec 1] .
  var N : Int .
  eq db(N) = (2 * N) .
endth

obj DOUBLEP is pr MLSEM .
  op dbp_ : Int -> IntExp .
endo

th PROOF is pr DOUBLE .
  pr DOUBLEP .
  op s : -> Stack .
  op x : -> Int .
  var Y : Int .
  eq 0 <= x = true .
  let body = if x is 0 then 0 else (2 + (dbp (x - 1))) .
  ***Induction hypothesis***
  cq dbp(Y) = db(Y) if (Y < x) and (0 <= Y) .
endth

open PROOF .    ***Prove the correctness of (1)***
  eq x = 0 .
  red s[[body]] is db(x) .
close

open PROOF .    ***Prove the correctness of (2)***
  eq 0 < x = true .
  red s[[body]] is db(x) .
close
```

d) The reduction result showing partial correctness:

```
OBJ> in verify_set
==========================================
th DOUBLE
==========================================
obj DOUBLEP
==========================================
th PROOF
==========================================
open PROOF
==========================================
eq x = 0 .
==========================================
reduce in PROOF : s[[body]] is db x
rewrites: 26
```

```
result Bool: true
==========================================
close
==========================================
open PROOF
==========================================
eq 0 < x = true .
==========================================
reduce in PROOF : s[[body]] is db x
rewrites: 22
result Bool: true
==========================================
close
OBJ>
```

5.2 Verify the correctness of the factorial function:

a) The definition of the function using ML:

$fun\ \ fact(N:\text{int}) = if\ \ N = 0\ then\ 1\ else\ N * fact(N-1);$

b) The explanation of the partial correctness:

Precondition is: $0 <= x$

Postcondition is: $s[[body]] = x!$

Induction proof:

$$(1)\ x = 0 \Rightarrow\ fact(x) = x!$$

$$(2)\ x > 0 \Rightarrow \forall N,\ if\ \ 0 \le N < x,\ and\ \ fact(N) = N!,$$

$$then\ \ fact(x) = x!$$

$$If\ \ both\ (1)\ and\ (2)\ are\ true,\ we\ can\ know:$$

$$\forall x >= 0,\ fact(x) = x!$$

c) The OBJ code used for verification:

```
th FAC is pr ZZ .
  op _! : Int -> Int [prec 1] .
  var N : Int .
  eq  0 ! = 1 .
  cq  N ! = N * ((N - 1) !) if 0 < N .
endth

obj FACP is pr MLSEM .
  op  fac_ : Int -> IntExp .
endo

th PROOF is pr FAC .
  pr FACP .
  op s : -> Stack .
  op x : -> Int .
  var Y : Int .
  eq 0 <= x = true .
  let body = if x is 0 then 1 else (x * ( fac (x - 1) )) .
  ***Induction hypothesis***
  cq fac(Y) = Y ! if (Y < x) and (0 <= Y) .
```

```
endth

open PROOF .    ***Prove the correctness of (1)***
  eq x = 0 .
  red s[[body]] is x ! .
close

open PROOF .    ***Prove the correctness of (2)***
  eq 0 < x = true .
  red s[[body]] is x ! .
close
```

d) The running result:

```
OBJ> in verify_fact
==========================================
th FAC
==========================================
obj FACP
==========================================
th PROOF
==========================================
open PROOF
==========================================
eq x = 0 .
==========================================
reduce in PROOF : s[[body]] is x !
rewrites: 25
result Bool: true
==========================================
close
==========================================
open PROOF
==========================================
eq 0 < x = true .
==========================================
reduce in PROOF : s[[body]] is x !
rewrites: 21
result Bool: true
==========================================
close
OBJ>
```

5.3 Verify the correctness of “exponent” function:

a) The definition of the function using ML:

$fun\ \exp(N, M) = if\ \ M = 0\ then\ 1\ else\ \ N * \exp(N, (M - 1));$

b) The explanation of the partial correctness:

Precondition is: $0 <= m$

Postcondition is: *s[[body]] = exponent n m*

Induction proof:

(1) m = 0, then exp(n,m) = exponent n m;

(2) m>0, then for any N, M,

if M<m, exp(N,M) = exponent N M;

If both (1) and (2) are true, we know:

For any n, m, if m>=0, then exp(n,m)=exponent n m

c) The OBJ code used for partial correctness verification:

```
th EXPONENT is pr ZZ .
  op exponent__ : Int Int -> Int .
  var N M : Int .
  cq  (exponent N M) = 1 if (M is 0) .
  cq  (exponent N M) = N * (exponent N (M - 1)) if 0 < M .
endth

obj EXPONENTP is pr MLSEM .
  op  exponentp__ : Int Int -> IntExp .
endo

th PROOF is pr EXPONENT .
  pr EXPONENTP .
  op s : -> Stack .
  ops m n : -> Int .
  var X Y : Int .
  eq 0 <= m = true .
  let body = if (m is 0) then 1 else n * (exponentp n (m -  1)) .
  ***Induction hypothesis***
   cq (exponentp X Y) = (exponent X Y) if (Y < m) .
endth

open PROOF .    ***Prove the correctness of (1)***
  eq m = 0 .
  red s[[body]] is (exponent n m) .
close

open PROOF .    ***Prove the correctness of (2)***
  eq 0 < m = true .
  red s[[body]] is (exponent n m) .
close
```

d) The reduction result showing partial correctness:

```
OBJ> in verify_exp
==========================================
th EXPONENT
==========================================
obj EXPONENTP
==========================================
th PROOF
==========================================
open PROOF
==========================================
eq m = 0 .
==========================================
reduce in PROOF : s[[body]] is exponent n m
rewrites: 31
result Bool: true
==========================================
close
```

```
==========================================
open PROOF
==========================================
eq 0 < m = true .
==========================================
reduce in PROOF : s[[body]] is exponent n m
rewrites: 26
result Bool: true
==========================================
close
OBJ>
```

5.4 Verify the correctness of "combination" function:

a) The definition of the function using ML:

$$fun\ comb(n,m) = if\ \ (m=0)\ orelse\ (m=n)\ then\ 1$$
$$else\ comb(n-1,m)+comb(n-1,m-1);$$

b) The explanation of the partial correctness:

Precondition is: $0 \le m \le n$

Postcondition is: $s[[body]] = comb\ n\ m$

Induction proof:

$(1)\ m = 0 \Rightarrow combination(n,m) = comb\ n\ m$

$(2)\ m = n \Rightarrow combination(n,m) = comb\ n\ m$

$(3)\ 0 < m < n \Rightarrow \forall N, M,$

$if\ \ 0 \le N < n\ and\ 0 \le M\ \le m, combination(N,M) = comb\ N\ M,$

$then\ combination(n,m) = comb\ n\ m$

$If\ \ both\ (1)\ and\ \ (2)\ are\ true,\ we\ can\ know:$

$\forall x >= 0, combination(n,m) = comb\ n\ m$

c) The OBJ code used for partial correctness verification:

```
th COMB is pr ZZ .
  op comb__ : Int Int -> Int .
  var N M : Int .
  cq  (comb N M) = 1 if (M is 0) .
  cq  (comb N M) = 1 if (M is N) .
  cq  (comb N M) = (comb (N - 1) M) + (comb (N - 1) (M - 1)) if 0
< M and M <N .
endth

obj COMBP is pr MLSEM .
    op  combp__ : Int Int -> IntExp .
endo

th PROOF is pr COMB .
  pr COMBP .
  op s : -> Stack .
  ops m n : -> Int .
  var X Y : Int .
```

```
  eq 0 <= m = true .
  eq 0 <= n = true .
  let body = if (m is 0) orelse (m is n) then 1 else (combp (n -
1) m) + (combp (n - 1) (m - 1)) .
  ***Induction hypothesis***
  cq (combp X Y) = (comb X Y) if (X < n) and (Y <= m) .
endth

open PROOF .    ***Prove the correctness of (1)***
  eq m = 0 .
  red s[[body]] is (comb n m) .
close

open PROOF .    ***Prove the correctness of (1)***
  eq m = n .
  red s[[body]] is (comb n m) .
close

open PROOF .    ***Prove the correctness of (2)***
  eq 0 < m = true .
  eq m < n = true .
  red s[[body]] is (comb n m) .
close
```

d) The reduction result showing partial correctness:

```
OBJ> in verify_comb
==========================================
th COMB
==========================================
obj COMBP
==========================================
th PROOF
==========================================
open PROOF
==========================================
eq m = 0 .
==========================================
reduce in PROOF : s[[body]] is comb n m
rewrites: 56
result Bool: true
==========================================
close
==========================================
open PROOF
==========================================
eq m = n .
==========================================
reduce in PROOF : s[[body]] is comb n m
rewrites: 60
result Bool: true
==========================================
close
==========================================
open PROOF
==========================================
eq 0 < m = true .
```

```
============================================
eq m < n = true .
============================================
reduce in PROOF : s[[body]] is comb n m
rewrites: 96
result Bool: true
============================================
close
OBJ>
```

6. Conclusion:

We have used OBJ3 implemented many interesting characteristics of ML, especially the recursive function call, the high-order function call and type decision.

OBJ3 is a powerful tool to specify the syntax and semantics of programming languages. It provides mechanisms for user definable sub-types, multiple inheritance, overloading, coercions, and more. The mechanism of *parameterized programming* is also very flexible and powerful. With parameterized modules, module instantiation and module expressions, it is very easy to define flexible program structuring and reuse.

7. Reference:

[1] Algebraic Semantics of Imperative Programs, by Joseph Goguen and Grant Malcolm
[2] Elements of ML Programming, ML97 Edition, by Jeffrey Ullman
[3] The Study of Programming Languages, by Ryan Stansifer
[4] OBJ3 Manual: http://www-cse.ucsd.edu/users/goguen/ps/iobj.ps.gz, by Joseph Goguen, Timothy Winkler, Jose Meseguer, Kokichi Futatsugi, and Jean-Pierre Jouannaud

Appendix: The OBJ3 codes of the implementation

```
*** =========== Object ZZ      =====================***
*** purpose: define some operations on Int .        ***
obj ZZ is pr INT .

  op  _is_ : Int Int -> Bool .
  op  _<>_ : Int Int -> Bool .
  op  _<>_ : Bool Bool -> Bool .

  var I J K L : Int .
  var B1 B2 : Bool .
  eq  I is I  =  true .
  eq  (I + J) is (K + J)  =  I is K .
  eq  (I - J) is (K - J)  =  I is K .
  cq  I is J  =  false   if  (I < J)  or  (J < I) .
  eq  I + - I  =  0 .
  eq  -(I + J)  =  - I + - J .
  eq  0 * I  =  0 .
  eq  - I * J  =  -(I * J) .
```

```
   eq  I - J  =  I + - J .
   eq  I * (J + K)  =  (I * J) + (I * K) .
   *** cq  I * J  =  I + (I * (J - 1))   if  0 < J .
   eq  (I + J) * K  =  (I * K) + (J * K) .

   eq  not(I <= J)  =  J < I .
   eq  not(I < J)  =  J <= I .
   eq  I + 1 <= J  =  I < J .
   eq  I < J + 1  =  I <= J .
   eq  I <= J + -1  =  I < J .
   eq  I <= J + - K  =  I + K <= J .
   eq  I < J + - K  =  I + K < J .
   eq  I + -1 < J  =  I <= J .
   eq  I <= I  =  true .
   eq  I < I  =  false .
   cq  I < I + J  =  true   if  0 < J .
   eq  I + -1 < I  =  true .
   cq  I + J < I  =  true   if  J < 0 .
   cq  I <= J  =  true   if  I < J .
   cq  I <= J + 1  =  true  if  I <= J .
   cq  I <= J + K  =  true  if  (I <= J)  and  (I <= K) .
   cq  I + J <= K + L  =  true  if  (I <= K)  and  (J <= L) .
   eq  I <> J = (I < J) or (J < I) .
   eq  I <> J = (I < J) or (J < I) .
   eq  B1 <> B2 = (B1 and not B2) or (not B1 and B2) .
endo

*** =========== Object LIST    ====================***
*** purpose: define a sort "List".                   ***
obj LIST[X :: TRIV] is
   sort List .
   sort NeList .
   sort ListElt .
   op [] : -> List .
   subsorts NeList < List .
   subsorts Elt < ListElt .
   op _@_ : List List -> List [assoc] .
   op _@_ : NeList List -> NeList [assoc].
   op _@_ : NeList NeList -> NeList [assoc].
   op _::_ : Elt List -> NeList .
   op tl_ : NeList -> List .
   op hd_ : NeList -> Elt .
   op _,_ : ListElt ListElt -> ListElt [assoc].
   op [_] : Elt -> NeList .
   op [_] : ListElt -> NeList .
   protecting NAT .
   op |_| : List -> Nat [prec 1].
   var E1 E2 : Elt .
   var EL1 EL2 : ListElt .
   var L : List .
   eq | [] | = 0 .
   eq | [ E1 ] @ L | = 1 + | L | .
   eq | [ E1 ] | = 1 .
   eq | [ E1 , EL1 ] | = 1 + | [ EL1 ] | .
   eq | [ EL1 , EL2 ] | = | [ EL1 ] | + | [ EL2] | .
   eq [ E1 ] @ [] = [ E1 ] .
   eq [] @ [ E1 ] = [ E1 ] .
```

```
   eq [ E1 ] @ [ E2 ] = [ E1 , E2 ] .
   eq [ E1 ] @ [ EL1 ] = [ E1 , EL1 ] .
   eq [ EL1 ] @ [ EL2 ] = [ EL1 , EL2 ] .
   eq E1 :: [] = [ E1 ] .
   eq E1 :: [ EL2 ] = [ E1, EL2 ] .
   eq hd ([ E1,EL1 ]) = E1 .
   eq tl ([ E1 ,EL1 ]) = [EL1] .
   eq hd ([ E1 ]) = E1 .
   eq tl ([ E1 ]) = [] .
   eq hd ([ E1 ] @ L) = E1 .
   eq tl ([ E1 ] @ L) = L .
endo

*** ============ Object Value   =====================***
*** purpose: define a sort "Value", which is the type***
***          of all the data which can be put into   ***
***          the stack.                              ***
obj VALUE is
   sort Value .
endo

*** ============ Object TUPLE   =====================***
*** purpose: define a sort "Tuple", which contains   ***
***          a name and a value to put into stack.   ***
obj TUPLE is
   pr VALUE .
   dfn Var is QID .
   sort Tuple .
   op <_,_> : Var Value -> Tuple .
   op nilTuple : -> Tuple .
   op name* _ : Tuple -> Var .
   op value* _ : Tuple -> Value .

   var X : Var .
   var V : Value .
   var T  : Tuple .
   eq name*  < X , V > = X .
   eq value* < X , V > = V .
   eq < name* T , value* T > = T .
endo

*** ============ Object STACK   =====================***
*** purpose: define a sort "Stack",                  ***
obj STACK is
   pr ZZ .
   pr TUPLE .
   sort Stack .
   op empty : -> Stack .
   op nilvalue : -> Value .
   op push(_,_) : Tuple Stack -> Stack .
   op top_ : Stack -> Tuple .
   op pop_ : Stack -> Stack .
   op fetch(_,_) : Var Stack -> Value .
   op fetch(_, empty) : Var -> Var .
   vars I J : Int .
   var  X : Var .
   var  T : Tuple .
```

```
  var  S : Stack .
  eq top push(T, S) = T .
  eq top empty = nilTuple .
  eq pop push(T, S) = S .
  eq pop empty = empty .
  cq fetch(X,push(T,S)) = value* T if X == name* T .
  cq fetch(X,push(T,S)) = fetch(X,S) if X =/= name* T .
  eq fetch(X, empty) = X .
endo

*** =========== Object EXP     =====================***
*** purpose: define a sort "Exp", which combines the ***
***          sorts of Int, List, Bool, Func.         ***
obj EXP is
  pr ZZ .
  pr STACK .
  dfn Var is QID .
  sorts IntExp BoolExp Exp .
  dfn IntList is LIST[Int] .

  subsorts Var < IntExp BoolExp IntList . *** ??? should be < Int Bool?
  subsorts Int < IntExp .
  subsorts Bool < BoolExp .
  subsorts IntExp BoolExp IntList < Exp .
  subsorts Int Bool IntList < Value .

  ops (_andalso_) (_orelse_) : BoolExp BoolExp -> BoolExp [prec 90].
  op not_ : BoolExp -> BoolExp [prec 86].
  op _is_ : IntExp IntExp -> BoolExp [prec 92].
  op _<_ : IntExp IntExp -> BoolExp [prec 86].
  op _<=_ : IntExp IntExp -> BoolExp [prec 86].
  op _>_ : IntExp IntExp -> BoolExp [prec 86].
  op _>=_ : IntExp IntExp -> BoolExp [prec 86].
  op _<>_ : IntExp IntExp -> BoolExp [prec 86].
  op _<>_ : BoolExp BoolExp -> BoolExp [prec 86].
  op if_then_else_ : BoolExp IntExp IntExp -> IntExp [prec 94].
  op if_then_else_ : BoolExp BoolExp BoolExp -> BoolExp [prec 94].
  ops (_+_) (_-_) (_*_) : IntExp IntExp -> IntExp [prec 85].

  op it : -> Var .
  op [[_]]_ : Exp Stack -> Stack .
  op _[[_]] : Stack Exp -> Value .

  vars I I' : Int .
  vars B B' : Bool .
  vars Li : IntList .
  var E : Exp .
  vars Ei Ei' : IntExp .
  vars Eb Eb1 Eb2 : BoolExp .
  vars X Y Z : Var .
  var S : Stack .

  eq [[ E ]] S = push(< it, S[[E]] >, S) .
  eq S[[ X ]] = fetch(X, S) .
  eq S[[ I ]] = I .
  eq S[[ Ei + Ei' ]] = S[[ Ei ]] + S[[ Ei' ]] .
  eq S[[ Ei - Ei' ]] = S[[ Ei ]] - S[[ Ei' ]] .
```

```
   eq S[[ Ei * Ei' ]] = S[[ Ei ]] * S[[ Ei' ]] .
   eq S[[ B ]] = B .
   eq S[[ Ei is Ei' ]] = S[[ Ei ]] is S[[ Ei' ]] .
   eq S[[ Ei < Ei' ]] = S[[ Ei ]] < S[[ Ei' ]] .
   eq S[[ Ei <= Ei' ]] = S[[ Ei ]] <= S[[ Ei' ]] .
   eq S[[ Ei > Ei' ]] = S[[ Ei' ]] <= S[[ Ei ]] .
   eq S[[ Ei >= Ei' ]] = S[[ Ei' ]] < S[[ Ei ]] .
   eq S[[ Ei <> Ei' ]] = S[[ Ei' ]] < S[[ Ei ]] .
   eq S[[ Eb andalso Eb1 ]] = S[[ Eb ]] and S[[ Eb1 ]] .
   eq S[[ Eb orelse  Eb1 ]] = S[[ Eb ]] or  S[[ Eb1 ]] .
   eq S[[ not Eb ]] = not (S[[ Eb ]]) .
   eq S[[if Eb then Ei else Ei']] = if S[[Eb]] then S[[Ei]] else
   S[[Ei']] fi .
   eq S[[if Eb then Eb1 else Eb2]] = if S[[Eb]] then S[[Eb1]] else
   S[[Eb2]] fi .
   eq S[[ Li ]] = Li .
endo

*** ============ Object FUN     =====================***
*** purpose: define a sort "Fun", which is the       ***
***          definition of a funciton.               ***
obj FUN is
   pr EXP .
   dfn ArgList is (LIST *(op [] to {}, op [_] to {_}))[Var] .
   sorts Fun .
   subsorts Fun < Value .
   op fun _ _ is _ : Var ArgList Exp -> Fun [prec 110].
   op [[_]]_ : Fun Stack -> Stack .
   op fname(_) : Fun -> Var .

   var X  : Var .
   var Lv : ArgList .
   var E  : Exp .
   var S  : Stack .
   var F  : Fun .
   eq  fname(fun X Lv is E) = X .
   eq [[F]]S = push((< fname(F),F >).TUPLE , S) .
endo

*** ============ Object ASSIGN =====================***
*** purpose: define a sort "Assign"                ***
obj ASSIGN is
   pr FUN .
   sort Assign .
   op val_:=_ : Var Exp -> Assign [prec 110].
   op [[_]]_ : Assign Stack -> Stack .

   var X  : Var .
   var E  : Exp .
   var S  : Stack .
   eq [[ val X := E ]] S = push( < X, S[[ E ]] > , S) .
endo

*** ============ Object FUNC    =====================***
*** purpose: define a sort "Func", which is for a    ***
***          function call, like f[x, y].            ***
obj FUNC is
```

```
  pr ASSIGN .
  dfn ExpList is (LIST *(op [] to {}, op [_] to {_}))[Exp] .
  sort Func .
  subsorts Func < IntExp BoolExp .
  dfn Var is QID .
  op __ : Var ExpList -> Func [prec 80].

  op wrong#args : -> Stack .
  op [[_equals_]]_ : ArgList ExpList Stack -> Stack .
  op [[_]][[_]]_ : Fun Stack ExpList -> Value .

  var E : Exp .
  vars Lv : ArgList .
  vars Le : ExpList .
  var X : Var .
  var S : Stack .

  eq S[[ X Le ]] = [[ fetch(X,S) ]] [[ S ]] Le .
  eq [[ fun X Lv is E ]] [[ S ]] Le = [[ Lv equals Le ]]S [[E]] .
  cq [[Lv equals Le]]S = [[val (hd(Lv)).Var := (hd(Le)).Exp]] ([[tl(Lv)
  equals tl(Le)]]S) if (| Lv | is | Le |) and (0 < | Lv |) .
  eq [[({}).ArgList equals ({}).ExpList ]] S = S .
endo

*** ============ Object STMT   =====================***
*** purpose: define a sort "Stmt", which includes   ***
***          assignments, expression and function-  ***
***          definition of ML.                      ***
obj STMT is
  pr FUNC .
  sort Stmt .
  subsorts Exp Fun Assign < Stmt .
  op _;_ : Stmt Stmt -> Stmt [assoc prec 120].
  op [[_]]_ : Stmt Stack -> Stack .

  vars P P' : Stmt .
  var E E' : Exp .
  var F : Fun .
  var Fc : Func .
  var A A' : Assign .
  var S : Stack .
  eq [[P ; P']] S = [[ P' ]] [[ P ]] S .
endo

*** ============ Object TYPE   =====================***
*** purpose: define a sort "Type", which defines    ***
***          the type of a results.                 ***
obj TYPE is
  pr FUNC .
  pr STACK .
  pr EXP .
  pr FUN .
  sort Category .     *** "val" of "fn"
  sort Type .          *** "int", "bool", or "list"
  sort PType .         *** Prime Type -- undecided type
  ops val fn : -> Category .
  ops int bool : -> Type .
```

```
  op  _list : Type -> Type [prec 15] .
  op  _==>_ : PType Type -> Type .
  op  primetype : -> PType .
  op  _*_ : PType PType -> PType .
  op  ptype(_) : ArgList -> PType .
  op  category(_) : Value -> Category .
  op  type(_) : Value -> Type .

  var X  : Var .
  var E  : Exp .
  var I : Int .
  var B : Bool .
  var Li : IntList .
  var Ei : IntExp .
  var Eb : BoolExp .
  var Lv : ArgList .
  var F  : Fun .
  eq category(I) = val .
  eq category(B) = val .
  eq category(Li) = val .
  eq category(F) = fn .
  eq type(I) = int .
  eq type(B) = bool .
  eq type(Li) = int list .
  eq type(fun X Lv is Ei) = ptype(Lv) ==> int .
  eq type(fun X Lv is Eb) = ptype(Lv) ==> bool .
  cq ptype(Lv) = primetype * ptype(tl(Lv)) if | Lv | > 1 .
  eq ptype({X}) = primetype .
endo

*** ============ Object MLSEM  ======================***
*** purpose: define a sort "MlSem", which is the    ***
***           semantics for the above "Mini-ML".     ***
obj MLSEM is
  pr STMT .
  pr TYPE .

  sort Result .
  op ml(_) : Stmt -> Result .
  op result_ : Tuple -> Result .
  op __is_::_ : Category Var Value Type -> Result .

  var P : Stmt .
  var X : Var .
  var V : Value .
  var F : Fun .
  var I : Int .
  var B : Bool .
  var Ei : IntExp .
  var Eb : BoolExp .
  eq result(< X,V >) = category(V) X is V :: type(V) .
  eq ml(P) = result (top ([[P]]empty)) .
endo
```