MiniOO Project

Will Whitney
wwhitney@cs.nyu.edu

Tokenization

To simplify the work of parsing MiniOO programs, I have imposed a few restrictions on the form of the language:

Disambiguating variables and fields:

- Variables must start with an uppercase letter.
- Fields must start with a lowercase letter.

As suggested in the MiniOO Syntax and Semantics, this ensures that `Var ∩ Field = ∅`.

Preventing conflicts in variable renaming by scope:

- Variables may not contain numbers.

During the static scoping stage I rename all variables by appending an index. By ensuring that variable names do not include numbers, I prevent the second occurrence of `X1` (which would be renamed `X11`) from conflicting with the eleventh occurrence of `X` (also renamed to `X11`).

Parsing

To represent the syntax of MiniOO programs, I have defined the following mutually recursive set of types:

```
type cmdNode = Empty
  | VardecNode of varNode * cmdNode
  | CallNode of exprNode * exprNode
  | MallocNode of varNode
  | VarAssignNode of varNode * exprNode
  | FieldAssignNode of exprNode * exprNode * exprNode
  | SkipNode
  | SeqNode of cmdNode * cmdNode
  | WhileNode of boolNode * cmdNode
  | CondNode of boolNode * cmdNode * cmdNode
  | ParallelNode of cmdNode * cmdNode
  | AtomNode of cmdNode
and exprNode =
    NumNode of int
  | MinusNode of exprNode * exprNode
  | NullNode
  | VarAccessNode of varNode
  | FieldLiteralNode of fieldNode
  | FieldAccessNode of exprNode * exprNode
  | ProcNode of varNode * cmdNode
and varNode = VarNode of string
and fieldNode = FieldNode of string
and boolNode =
    TrueNode
  | FalseNode
  | LessNode of exprNode * exprNode
;;
```

A few simple examples of programs and their ASTs follow. Here I use "Scope" to denote a variable declaration, and all (more-) indented lines which follow a (less- or) un-indented line are child nodes in the AST of the less-indented line.

`var X; X=1`

```
Scope: variable X
    VarAssign
        Variable: X
        Num: 1
```

`var X; if true X=1 else X=1-1`

```
Scope: variable X
    Cond
        True
        VarAssign
            Variable: X
            Num: 1
```

```
        VarAssign
            Variable: X
            Minus
                Num: 1
                Num: 1
```

This program is (semantically) incorrect, as `X` is not assigned before it is used. However, it still parses into an AST.

```
var X; while 1<X X=X-1
Scope: variable X
    While
        Less
            Num: 1
            VarAccess
                Variable: X
        VarAssign
            Variable: X
            Minus
                VarAccess
                    Variable: X
                Num: 1
```

Static semantics

The static semantics in my implementation of MiniOO are the same as those defined in the project document.

I implement static scope checking with a set of mutually recursive functions: `scope_cmd`, `scope_expr`, `scope_bool`, and `scope_var`. To check the scope rules of a program, one can call `scope_cmd` on the command node which is the root of the program's AST. I use a global Hashtbl instance to keep track of the number of occurrences of each variable name (e.g., how many unique variables called X the scoping function has seen) to ensure that no two different variables are ever renamed to the same thing, even if they occur in different subtrees. I use an associative list to keep track of the number (e.g. X3) that each variable name (e.g. X) maps to in the current scope as I walk the AST.

The `scope_cmd` function will return a new AST in which every variable is renamed such that any two VarNode instances will have the same identifier string if and only if they refer to the same actual variable.

Some examples of scope checking and renaming:

```
var X; while 1<X X=Y-1
Fatal error: exception Failure("Variable not declared: Y")

var X; var X; X=1
Scope: variable X0
    Scope: variable X1
        VarAssign
            Variable: X1
            Num: 1

var X; var X; X=proc X: X=1
Scope: variable X0
    Scope: variable X1
        VarAssign
            Variable: X1
            Proc
                Variable: X2
                VarAssign
                    Variable: X2
                    Num: 1
```

Semantic domains

I define the following types for semantic domains, which represent runtime types. They are used along with the AST types defined above to match different cases in the step and iterator functions.

```
type boolType =
  | True
  | False
  | BoolError of string
and objType =
    Object of int
and locType =
  | ObjLoc of objType
  | NullLoc
and valType =
  | FieldVal of fieldNode
  | IntVal of int
  | LocVal of locType
  | Closure of varNode * controlType *
stackType
and tvalType =
  | Value of valType
  | ValueError of string
and envType =
    Environment of varNode * objType
```

```
and frameType =
  | DeclFrame of envType
  | CallFrame of envType * stackType
and stackType =
    Stack of frameType list
and heapType =
 Heap of ((objType * fieldNode),  tvalType)
Hashtbl.t
and stateType =
  | State of stackType * heapType
and controlType =
  | CmdCtrl of cmdNode
  | BlockCtrl of controlType
and configType =
  | Nonterminal of controlType * stateType
  | Terminal of stateType
  | ConfigError of string
```

Transition semantics

At the highest level of my interpreter, I have the function `iterator`:

```
iterator config =
  match config with
  | Nonterminal (ctrl, state) ->
     let (stack, heap) = unwrap_state state in
     iterator (step ctrl state)
  | Terminal (State (stack, heap)) ->
     State (stack, heap)
  | ConfigError s -> failwith ("Error propagated up to iterator:\n" ^ s)
```

It recursively calls `step` (which implements the ⇒ operator from the transition semantics) and itself until it reaches a terminal state; it thus implements the operator ⇒*.

The bulk of the transition semantics lies in the function `step`. In implementing these semantics, there are a few decisions to make with regard to how data is represented and how certain operations are implemented.

Stack

My stack is implemented as a list of Frames; where each frame is either a Declaration Frame or a Call Frame. The stack is passed recursively into each subtree from its parent. As only single-declaration is possible in MiniOO, I define Environment as a pair of (variable, object) instead of as a list.

Heap

My heap is implemented as a global Hashtbl mapping (location, field) pairs to heap values; since values are only ever added to the heap and the heap is global (though variable access is mediated by the local stack) this is a simple way to get correct behavior.

In parallel I maintain a Hashtbl which keeps track of the set of heap locations which have been allocated. When I interpret a `malloc(X)` call, instead of initializing any fields, I update this allocation table. At the time of a field access `e1.e2` I check to see whether the location defined by `e1` has been allocated; if it hasn't, I throw an error. If it has I return the value of the field `e1.e2`, if it has an entry in the heap, else `null`. This follows exactly the transition semantics of MiniOO but does not require spurious field allocations.

Transitions

The semantics of the instruction Sequential(C1, C2) (that is, `{C1;C2}`) are that C1 should be executed to completion, then C2 should be executed in the state returned by the last step of C1. The simplest way to implement this functionality is to call iterator on C1 instead of completing one step at a time:

```
step Nonterminal (Sequential(C1, C2), state) = Nonterminal (C2, iterator C1 state)
```

For the Parallel instruction `{C1 ||| C2}`, the execution of the two subtrees defined by commands C1 and C2 can occur in arbitrary order (except as limited by the Atom instruction). As such, my interpreter performs the transformation Parallel(C1, C2) → Sequential(C1, C2) at runtime.

The Atom instruction `atom(C)` requires that C be executed to completion without interleaving any other commands. As previously described, my function `iterator` is exactly this operator ⇒*. So I have implemented Atom as:

```
step Nonterminal (Atom C, state) = Terminal (iterator C state)
```

Examples

This program was provided as an example in the syntax and semantics. I provide at each step of execution the state of the stack and heap. Where it says "Completed execution!" indicates the termination of a subtree, e.g. at the end of the assignment of P to the closure value.

`var P; {P = proc Y: if Y<1 P=1 else P(Y-1); P(1)}`

```
Stack:
DeclFrame: P0 → 0

Heap:
(0, val) → LocVal NullLoc
----------------------------------
Stack:
DeclFrame: P0 → 0

Heap:
(0, val) → LocVal NullLoc
==================================
Completed execution!
Stack:
DeclFrame: P0 → 0

Heap:
(0, val) →
    Closure:
        Var: Variable: Y0,
        Ctrl: [...]
        Stack:
             DeclFrame: P0 → 0
----------------------------------
Stack:
DeclFrame: P0 → 0

Heap:
(0, val) →
    Closure:
        Var: Variable: Y0,
        Ctrl: [...]
        Stack:
             DeclFrame: P0 → 0
----------------------------------
Stack:
Call: [Y0 → 1] Calling stack:
    DeclFrame: P0 → 0

DeclFrame: P0 → 0

Heap:
(0, val) →
    Closure:
        Var: Variable: Y0,
        Ctrl: [...]
        Stack:
             DeclFrame: P0 → 0

(1, val) → IntVal 1
----------------------------------
Stack:
Call: [Y0 → 2] Calling stack:
    Call: [Y0 → 1] Calling stack:
        DeclFrame: P0 → 0

    DeclFrame: P0 → 0
DeclFrame: P0 → 0

Heap:
(2, val) → IntVal 0
(0, val) →
    Closure:
        Var: Variable: Y0,
        Ctrl: [...]
        Stack:
             DeclFrame: P0 → 0

(1, val) → IntVal 1
==================================
Completed execution!
Stack:
DeclFrame: P0 → 0

Heap:
(2, val) → IntVal 0
(0, val) → IntVal 1
(1, val) → IntVal 1
```

Here is an example of a program which leads to a runtime error:

```
var X; {X=proc Y: X=1; if X<1 X=1 else X=1-1}
Fatal error: exception Failure("Error propagated up to iterator:
Boolean in CondNode was error:
Number 1 in LessNode was not an IntVal")
```

The runtime error propagates upward through the execution trace until it becomes a ConfigError in the iterator, at which point execution terminates. We can modify this program slightly so that it runs:

```
var X; {X=proc Y: X=Y; {X(1); if X<1 X=1 else X=1-1}}
Stack:
DeclFrame: X0 → 0

Heap:
(0, val) → IntVal 0
(1, val) → IntVal 1
```

A simple example of a While command, where we construct the value Three, assign it to X, and then decrement X until it is no longer greater than zero. The trace is quite long for this program, so I include a few salient points in it instead of the entire thing.

```
var Three; {Three=1-1-1-1; {Three=1-Three; var X; {X=Three; while 1-1<X X=X-1}}}
(* After X=Three *)
Stack:
DeclFrame: X0 → 1
DeclFrame: Three0 → 0


Heap:
(0, val) → IntVal 3
(1, val) → IntVal 3
```

```
(* After one iteration of the loop *)
Stack:
DeclFrame: X0 → 1
DeclFrame: Three0 → 0


Heap:
(0, val) → IntVal 3
(1, val) → IntVal 2
```

```
==================================
Completed execution!
Stack:
DeclFrame: X0 → 1
DeclFrame: Three0 → 0


Heap:
(0, val) → IntVal 3
(1, val) → IntVal 0
```

I implement the While command by a transformation into a Conditional which contains another While inside of it:

```
| WhileNode (boolean, cmd) ->
    let true_cmd_step = cmd in
    let true_cmd_repeat = WhileNode (boolean, cmd) in
    let true_cmd = SeqNode (true_cmd_step, true_cmd_repeat) in
    let false_cmd = Empty in
```

```
let cond_cmd = CondNode (boolean, true_cmd, false_cmd) in
Nonterminal (CmdCtrl cond_cmd, state)
```

This means that the While loop from the previous example:

```
While
    Less
        Minus
            Num: 1
            Num: 1
        VarAccess
            Variable: X0
    VarAssign
        Variable: X0
        Minus
            VarAccess
                Variable: X0
            Num: 1
```

becomes this Conditional-Sequential-While control:

```
Cond
    Less
        Minus
            Num: 1
            Num: 1
        VarAccess
            Variable: X0
    Seq
        VarAssign
            Variable: X0
            Minus
                VarAccess
                    Variable: X0
                Num: 1
        While
            Less
                Minus
                    Num: 1
                    Num: 1
                VarAccess
                    Variable: X0
            VarAssign
                Variable: X0
                Minus
                    VarAccess
                        Variable: X0
                    Num: 1
    Empty
```

That is, if the condition is true, run the command one time and then return a new configuration with the new state and the same While as the control. If the condition is not true, terminate.

Field assignment to a variable which has not been allocated will result in a runtime error:

```
var X; X.f=1
Fatal error: exception Failure("Error propagated up to iterator:
location in FieldAssign was not a location type")
```

Once we Malloc(X), we can assign to its fields. Note the Allocated table, which reflects that X gets allocated at the end of the first subtree (the first "Completed execution!" log here):

```
var X; {malloc(X); X.f=1}
Stack:
DeclFrame: X0 → 0

Heap:
(0, val) → LocVal NullLoc

Allocated:
---------------------------------
Stack:
DeclFrame: X0 → 0

Heap:
(0, val) → LocVal NullLoc

Allocated:
=================================
Completed execution!
Stack:
DeclFrame: X0 → 0

Heap:
(0, val) → LocVal ObjLoc: Object: 1

Allocated: [Object: 1]
---------------------------------
Stack:
DeclFrame: X0 → 0

Heap:
(0, val) → LocVal ObjLoc: Object: 1

Allocated: [Object: 1]
=================================
Completed execution!
Stack:
DeclFrame: X0 → 0

Heap:
(1, f) → IntVal 1
(0, val) → LocVal ObjLoc: Object: 1

Allocated: [Object: 1]
```

Additionally, now that X has been allocated we can access fields which have not been assigned, and they read as null:

```
var X; var Y; {malloc(X); {X.f=1; Y=X.g}}
Stack:
DeclFrame: Y0 → 1
DeclFrame: X0 → 0

Heap:
(0, val) → LocVal ObjLoc: Object: 2
(2, f) → IntVal 1
(1, val) → LocVal NullLoc

Allocated: [Object: 2]
```