Mini Object Oriented language (MOO)

Frank Liu

December 5, 2024

1 Overview

MiniOO is a simplified object-oriented programming language designed to explore the foundational principles of programming languages, as part of the course CSCI-GA.3110-001: Honors Programming Languages at NYU. The language focuses on essential constructs such as variable declarations, assignments, procedures, field access, and parallel execution, allowing for a clear understanding of core programming concepts without the complexities of larger languages.

The language employs a small-step operational semantics model to demonstrate the execution of programs step-by-step. Its syntax is minimalistic, emphasizing concepts like lexical scoping, heap-based memory management, and error handling. MiniOO also includes features like closures to capture variable bindings and a simple mechanism for dynamic memory allocation using stacks and heaps.

Please refer to the Syntax and Semantics Document **MiniOO-SOS.pdf** for specifics about the specifications of MiniOO

2 How to Build and Use MiniOO

Building MiniOO

To build the MiniOO executable, use the provided `make` command:

```
make
```

This will compile the source files and create the `minioo` executable.

Running MiniOO

`minioo.ml` main file that reads input and runs command functions
(See line 29 `let run_commands commands = ...`)

The `minioo` executable offers two ways to run programs:

1. **Interactive Line-by-Line Interpreter**: Run `minioo` without any arguments to start an interactive session. Here you will be greeted by a friendly `#moo` prompt, and you can enter MiniOO commands line by line, and they will be executed immediately.

   ```
   ./minioo
   ```

   Ctrl + c or z to quit

2. **Script File Execution**: You can also run a pre-written MiniOO program saved in a file (e.g., `prog.moo`) by providing the filename as an argument:

   ```
   ./minioo < prog.moo
   ```

   This will execute the commands in the script file sequentially.

Verbose Options

MiniOO provides two verbosity options to help you understand the execution process:

- `-v`: Outputs the Abstract Syntax Tree (AST) at each step of execution and the final state of the program.
- `-vv`: Outputs both the AST and the program state (Stack and Heap) at **every line**.

To use these options, include them as arguments when running `minioo`. For example:

```
./minioo -v < prog.moo
./minioo -vv
```

These options are especially useful for debugging or for gaining deeper insights into how MiniOO interprets and executes commands.

3 Syntax and Semantics of my implementation of MiniOO

3.1 MiniOO Syntax

`miniooLEX.mll` Parser: `miniooMENHIR.mly`

Commands (`cmd`)

- `var X;` *Variable declaration*
- `f(e);` *Procedure call*
- `X = e;` or `x = e;` *Variable or field assignment*
- `malloc(X);` *Dynamic memory allocation*
- `skip;` *No operation*
- { `C1;C2` } *Sequential execution of commands*
- `while b C;` *While loop*
- `if b then C1 else C2;` *Conditional branching with* ***`else`***
- `if b C;` *Conditional branching without* ***`else`***
- { `C1 ||| C2` } *Parallel execution of commands*
- `atom(C);` *Atomic execution of commands*

Boolean Expressions (`bool_expr`)

- `true` or `false` *Boolean literals*
- `e1 == e2` *Equality comparison*
- `e1 < e2` *Less-than comparison*

Expressions (`expr`)

- `f` *Field name*
- `N` *Integer constant*
- `e1 - e2` *Arithmetic subtraction*
- `null` *The* ***`null`*** *value*
- `X` *Variable name*
- `e1.e2` *Field access*
- `proc Y:C` *Procedure definition*

Semantic Rules for Variables and Fields

One of the key semantic distinctions in MiniOO is how variables and fields are differentiated:

- **Variables** always start with an uppercase letter.
- **Fields** always start with a lowercase letter.

This convention is strictly enforced in MiniOO to ensure clarity in referencing and to avoid ambiguity between variable names and field names during interpretation. For example:

```
var X    // Declares a variable named X
X = 42  // Assigns the value 42 X
var x // Not Allowed!!! Syntax error
X.f = 123 // Assigning the value 12 to field f of X
```

3.2 Static Semantics

3.3 Transitional Semantics

`translationalSemantics.ml`
To implement the Heap I used an array of Hash Tables
`type heap = (string, tainted_value) Hashtbl.t array`
To implement the Stack I have used an Ocaml Hash Table
`type stack = (string, int) Hashtbl.t`

This way locations where simply indices of my heap, and my Stack just mapped Variable identifiers to indices on the heap where their value could be found. Allocating a new object location is simply just appending a new hash table to the heap and using that new index (size of heap -1).

Errors

In MiniOO, errors are treated as **blocking transitions**, meaning that whenever an error is encountered, it prevents the program from making further progress. Errors are thrown using exceptions rather than stored in values, ensuring that they are immediately raised and do not propagate invalid states in the execution.

General Error Handling

Most of MiniOO's execution model relies on throwing errors through the `TransitionError` exception when an invalid operation or semantic violation is encountered. For example:

- Attempting to access an undeclared variable.
- Assigning an incompatible value to a field or variable.

- Performing an arithmetic operation on non-integer values.

In these cases, MiniOO immediately raises a `TransitionError`, halting further execution and preserving the integrity of the program state.

Handling Errors in Parallel Execution

The only exception to the blocking behavior is in **parallel execution**, where MiniOO allows partial execution of commands even if one branch encounters an error. This is implemented using a `try-catch` approach, combined with randomized sequentialization.

The Parallel Semantics

When executing a parallel command, MiniOO:

1. Randomly picks one of the two command sequences to execute first.
2. Executes the chosen command sequence.
3. If the first command sequence succeeds, the second sequence is executed afterward.
4. If the first command sequence fails, the error is caught, and MiniOO attempts to execute the second sequence. Errors from both sequences are combined if both fail.

The implementation is as follows:

```
Parallel (cmds1, cmds2) -> (
    (* Randomly pick a number: 1 or 2 *)
    let first = if Random.int 2 = 0 then 1 else 2 in
    try
      (* Run the first chosen command *)
      if first = 1 then (
        eval_cmds stack heap cmds1;
        eval_cmds stack heap cmds2 (* Run the second command after the first *)
      ) else (
        eval_cmds stack heap cmds2;
        eval_cmds stack heap cmds1 (* Run the second command after the first *)
      )
    with
    | TransitionError msg ->
        (* Catch the error and continue running the second command *)
        (try
           if first = 1 then eval_cmds stack heap cmds2
           else eval_cmds stack heap cmds1
         with
         | TransitionError second_msg ->
```

```
            (* Raise a combined error if both commands fail *)
            raise (TransitionError (msg ^ " AND " ^ second_msg)));
        (* Re-raise the first error if the second succeeds *)
        raise (TransitionError msg)
)
```

Rationale for This Design

This approach ensures that:

- **Progress is attempted for both branches**: Even if one branch fails, the other branch is still executed.
- **Errors are not lost**: If both branches fail, the errors are combined and raised together.
- **Randomization introduces fairness**: By randomly selecting the execution order, MiniOO avoids biasing one branch over the other.

4 Examples

Example 1: Static Scoping (prog1.moo)

```
var R; var H; H=1; var P; P = proc Y: R = Y-H; var H; H = 2; P(4)
```

This program demonstrates the principle of **static scoping**, where variables declared in inner scopes do not affect variables in outer scopes with the same name.

The program is equivalent to the following, with unique variables introduced for clarity:

```
var R; var H1; H1 = 1; var P; P = proc Y: R = Y - H1; var H2; H2 = 2; P(4)
```

Here, the outer variable `H` is renamed to `H1`, and the inner variable `H` is renamed to `H2`. When the procedure `P` is called with `P(4)`, the expression `R = Y - H1` evaluates to `R = 4 - 1`, assigning `R = 3`.

This example highlights how static scoping ensures that variables in a procedure refer to their enclosing environment at the time of definition.

—

Example 2: Recursive Procedure (prog2.moo)

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

This program illustrates **recursion**. The procedure `P` recursively calls itself, decrementing `Y` until `Y < 1`, at which point it terminates. When `P(1)` is called, the program assigns `P = 1`, as the base case is reached after a single recursive call.

—

Example 3: Object Creation (prog3.moo)

```
var X; malloc(X)
X.c = 0
X.f = proc Y: if Y < 1 then X.r = X.c else X.f(Y - 1)
X.f(2)
```

This program showcases **dynamic object creation**. A new object `X` is allocated using `malloc`, and fields are initialized. The field `f` is assigned a recursive procedure that computes values based on the field `c`. Calling `X.f(2)` assigns `X.r = 0`, demonstrating how procedures can modify object fields dynamically.

—

Example 4: Weighted Coin Flip Sum Sampling (coinFlip.moo)

```
var X
var P; P = proc L : {atom(X = 0-L) ||| X = L}
var Face
var Weights; malloc(Weights); Weights.tails = 20; Weights.heads = 0-10
var Earnings; Earnings = 0
var I; I = 10; skip
while 0 < I {
  P(I);
  if X < 0 then Face = tails else Face = heads;
  Earnings = Earnings - Weights.Face;
  I = I - 1
}
Earnings = Earnings
```

The program simulates a series of coin flips in a `while` loop, updating the `Earnings` based on the flip outcome and the corresponding weight. This example combines parallelism, randomness, and sequential execution to compute a weighted sum based on simulated coin flips.

This program demonstrates **parallel execution, atomicity, object allocation**. The procedure `P` uses an `atom` block to ensure that `X = 0-L` executes atomically, preventing race conditions. The program initializes a `Weights` object with values for tails and heads. Using our implementation of parallelism, `P(L)` randomly sets `X` to either `0-L` (tails) or `L` (heads).