Report on MiniOOL Interpreter
Project for Honors Programming Languages
CSCI-GA.3110-001

Ekanshdeep Gupta

1 Introduction

MiniOOL is a language created for the course Honors Programming Langauges CSCI-GA.3110-001 at New York University. This document is a manual accompanying the interpreter created by the author. It describes how to use the interpreter along with some examples to demonstrate its use. It also describes some design decisions and changes made to the original specification which is described in the document "MiniOO-Syntax-And-Semantics.pdf". The interpreter is written in Ocaml. It uses `ocamllex` as the lexer and `menhir` as the parser.

2 Code Base

The source code for the interpreter consists of the following files:

1. `lexer.mll`: This file consists of the lexer specifications. It reads the input stream and generates a stream of tokens to be used in the parser.
2. `parser.mly`: This file consists of the parser specifications. It reads the token stream generated by the lexer and constructs an abstract syntax tree to be used by the interpreter as the internal representation of the MiniOOL code.
3. `abstractSyntax.ml`: This file defines all the types used for generating and storing the abstract syntax tree.
4. `staticSemantics.ml`: This file takes the abstract syntax tree and runs a static scoping check, ensuring that all variables are well-defined in scope.
5. `operationalTypes.ml`: This file defines all the types for storing the internal representation of the state of the interpreter while executing instructions.

6. `operationalSemantics.ml`: This file contains the functions which take the abstract syntax tree and actually execute all the instructions defined, by manipulating the internal state of the interpreter.

7. `printStuff.ml`: This file contains functions which generate pretty string representations of all the user-defined types used in the code. These `string_of_x` functions are used everywhere to generate printable strings.

3 Usage

To use the interpreter (on a *nix based system), open a terminal in the directory of the code and run `make`. This should compile the interpreter, provided all dependencies are available. The interpreter can be used in two ways:

1. To run code from a file, use the command `./main /location/of/source/file`, for example, `./main ./examples/factorial.oo`

2. To read code from `stdin`, invoke the inline mode, using the `-i` flag with the `./main` file. For example `echo "var x;" | ./main -i`. The command `./main -i` can also be called by itself to type MiniOOL code directly into the interpreter. It will all be executed together once EOF is read (`Ctrl+D` on *nix based systems, `Ctrl+Z` on Windows).

Upon execution, the interpreter will print first the abstract syntax tree of the code received, followed by the output of any `print()` statements in the during execution, followed by the final internal state of the interpreter once the execution is complete, ie a string representation of the stack, the heap, and the address counter.

3.1 Dependencies

The code requires standard Ocaml tools, `ocamlc` and `ocamllex`. The only external dependency is `menhir`, which can be installed via `opam` with the command `opam install menhir`.

4 Design Decisions

Internally, heaps and environments are implemented as lists with lookup and adding new value taking time $O(n)$. Stacks are implemented using the Ocaml `Stack` module.

4.1 Fields and Variables

Any string of letters, numbers and underscores, which begins with a letter is a valid identifier. To distinguish the identifiers for fields and variables, all variables must start with a small letter, and all fields must start with a capital letter. For

instance `v.F` refers to the field `F` of the variable `v`. This restriction is imposed in the lexer itself, and also checked at runtime.

4.2 Malloc

For a variable in the scope, all valid fields can be set and used freely. If for instance `v.F` is not defined before use, then an exception will be raised. Example: `var x; print(x.F)` returns `Fatal error: exception Failure("Location (0, F) not found.")`.

This is different from the behaviour defined in the specification, where fields like `x.F` are only available once `malloc(x)` has been called. Since fields are always available, that renders the `malloc()` command useless. It is still available, but is pointless to call.

Usage of variables and fields can be seen in `examples/variables_fields.oo`.

4.3 Scoping

The interpreter implements static scoping. A static scope check is run after parsing to ensure all variables are valid and in the scope.

4.4 Instruction disambiguation

To interpret complex control instructions unambiguously (for example to prevent dangling else problem), all commands need to be terminated with a `;`. Note that `while b C`, `if b C else C` and `proc y: C` are also commands and thus, need to be terminated with a (extra) `;`. This leads to the requirement of multiple `;` to terminate complex instructions, for example:
`p = proc y: if (y < 1) p = 1; else p(y - 1);;;`. This perhaps makes more sense if indented properly as follows:

```
p = proc y:
    if (y < 1)
        p = 1;
    else
        p(y - 1);
    ;
;
```

A `while` loop should be coded as follows:

```
while x < 0
    x = x + 1;
    x = x - 3;
;
```

Braces are essentially ignored by the parser. They can however be used for styling as follows:

```
while x < 0 {
    x = x + 1;
    x = x - 3;
};
```

Notice the `;` at the end of the `{}` block, which still needs to be there to terminate the `while` loop. Please refer to the code in `parser.mly` to see the exact parser specifications.

The `if` or the `else` part of the conditional can be left blank to skip it, or use the `skip` command. However, the semicolons still need to match up. Look at `examples/if_else_missing.oo` to see an example of how to do that.

4.5 Parallelism

While parallel code is syntactically supported, multi-threading is *not* implemented. Parallel code $\{ C_1 ||| C_2 \}$ is essentially executed as sequential code: first C_1 is executed, followed by C_2. Accordingly, the `atomic` statement is not required, but is still available. An example is given in `examples/parallel.oo`.

4.6 `print` statement

A `print` statement has been added to the language to query the state of variables during execution. It can be used as `print(expression)`, for example `print(x)` or `print(1+5)`.

5 Examples

5.1 Recursive Factorial (`examples/factorial.oo`)

The following example computes the factorial of 6:

```
var r;
r = 1;
var fact;
fact = proc n:
    print(n);
    if (n == 0)
        skip;
    else
        r = r * n;
        print(r);
        fact(n-1);
    ;
;
fact(6);
print(r);
```

The program prints `Iden: (Var x) : Val Int 720`, which indeed is the correct value for 6!.

5.2 Fibonacci Sequence (`example/fibonacci.oo`)

The following example computes the 10th term of the Fibonacci sequence.

```
var r;
var fibonacci;
fibonacci = proc n:
    if (n <= 1)
        r = n;
    else
        var temp1;
        fibonacci(n-1);
        temp1 = r;

        var temp2;
        fibonacci(n-2);
        temp2 = r;
        r = temp1 + temp2;
    ;
;

var n;
n = 10;
fibonacci(n);
print(r);
```

The program prints `Iden: (Var r) : Val Int 55`, which indeed is the 10th term of the Fibonacci sequence.

5.3 Iterated Fibonacci Sequence

The following code implements a much more efficient, iterated computation of the Fibonacci sequence. It computes the 50th term of the Fibonacci sequence.

```
var r;
var fibonacci;
fibonacci = proc n:
    if (n <= 1)
        r = n;
    else
        var fib;
        fib = 1;
        var prev_fib;
        prev_fib = 1;
        var i;
```

```
        i = 2;
        var tmp;
        while (i < n)
            tmp = fib;
            fib = prev_fib + fib;
            prev_fib = tmp;
            i = i + 1;
        ;
        r = fib;
    ;
;

var n;
n = 50;
fibonacci(n);
print(r);
```

The program prints `Iden: (Var r) : Val Int 12586269025`, which indeed is the 50th term of the fibonacci sequence.

5.4 (False) Parallelization (`/examples/parallel.oo`)

```
var x;
var y;

x = 1;
{
    x = 0;

    |||

    if x == 0
        y = 0;
    else
        y = 1;
    ;
    print(y);
};
```

The above code will always print `Iden: (Var y) : Val Int 0` since the parallel code is always executed in the same manner.