A Mini Object-Oriented Language (MiniOOL)

Zachary Ferguson

Fall 2018

1 Introduction

MiniOOL is a mini object-oriented language created for Honors Programming Language's (Fall 2018). This report provides instructions on how to build and use MiniOOL as well as a discussion of the design decisions, implementation details, and some examples covering the features of MiniOOL. This report greatly relies on the original specifications given in the "Syntax and Semantics of MiniOOL," so a through understanding of the syntax and semantics is recommended.

2 Building MiniOOL

To build MiniOOL, use the included make file to compile to source code and build the binary for the interpreter. Do this by simply running the command `make` while in the root of the MiniOOL directory.[1] Optionally, the documentation for the source code can be generated using `ocamldoc` by running the command `make docs`. See Section 5 for more instruction on how to run the example programs.

2.1 Dependencies

The source code for MiniOOL is almost self contained requiring no other dependencies besides the standard libraries and tool for OCaml including `ocamllex` and `ocamlopt`. The only external dependency of MiniOOL is Menhir which is used to build the parser. Menhir can be installed using opam using the command `opam install menhir`.

3 Using MiniOOL

After building MiniOOL, an executable file names `MiniOOL` is created.[2] To start an interactive session of MiniOOL type `./MiniOOL` from inside the MiniOOL project directory. This will produce a prompt where a single line of input can be entered at a time. The default output of MiniOOL is to echo the input renamed with unique names and print the values on the stack at the end of execution. Optionally one can run `./MiniOOL --verbose` to display the abstract syntax tree of the input.

The commands and expressions usable in MiniOOL are defined in the "Syntax and Semantics of MiniOOL." with the varients to the syntax explained in Section 4.

[1] In some instance, **if the source code is edited** and a `make` command is run the compilation will run into an error do to "inconsistent assumptions over [an] interface." This occurs because the compiled header files have not been updated with the new interface information. If this occurs it is recommended to remove binary files and run `make` again. This can be done with the command `make clean; make`.

[2] The `MiniOOL` executable is in fact a bash script that wraps the actual executable binary file created by the OCaml compiler, `bin/MiniOOL_raw`. This wrapper uses `rlwrap` (readline wrapper) utility, if it is available, to facilitate traversing the input with the left and right arrow keys and a history with the up and down arrow keys. If `rlwrap` is not installed then only `MiniOOL` is called.

3.1 Interpreter Directives

Separate from the standard commands in MiniOOL, interpreter directives single the interpreter to execute special operations. Directive cannot be use in conjunction with commands, they must be alone on a line with only a single directive allowed. The special character `\` starts indicates a directive and is not part of any other syntax of MiniOOL. The available directives are:

- `\exit` or `\quit`: exit MiniOOL
- `\help`: display helpful information about and how to use MiniOOL
- `\clear`: clear the stack and heap of all values (this will also reset the unique naming of variables)
- `\verbose`: verbosely display the steps of interpretation including the abstract syntax tree (same as the command line option `--verbose`)
- `\quite`: opposite of `\verbose`, turn off verbose information

4 Design Decisions

4.1 Fields

To distinguish between the variables and fields, Fields must start with a capital letter [A-Z] and variables must start with a lower case letter [a-z]. This is that `x.F` implies that `x` is a variable and `F` is a field of the variable `x`.

4.2 Implicit Global Variables

In order to implement undeclared variables being implicitly declared as a global variable with dynamic scope, two changes where made to the semantics. One change was made to the static semantics and another change to the operational semantics.

In the static semantics, no longer will an undeclared variable result in an error. When a variable is encountered in the static semantics check, it is either in scope or not. If the variable is in scope the original static semantics definition follows. Otherwise, if the variable is out of scope then it is declared as a global variable with a unique global name.[3] Therefore, the line in the static semantic definition $|x \to e.E = (x \notin e.V)$ is changed to $|x \to e.E = false$.

In the operational semantics, the values for implicit global variables are stored at a special location in the heap, l^*. The value of a implicit global variable, `x`, is located at $h(\langle l^*, \texttt{x} \rangle)$. This can be thought of as the implicit global variable as being a field of an implicit object that is always allocated at location l^*. The

[3] During the static semantic check all variable names are uniquified by appending a unique subscript string to the identifier string. This subscript string is the number of times the identifier has been previously declared. For example the variables in `var x; x = 1; var x; x = 2` would be renamed to `var` $\texttt{x}_0$`;` $\texttt{x}_0$ `= 1; var` $\texttt{x}_1$`;` $\texttt{x}_1$ `= 2`. This works even if the identifier contains a trailing number (e.g. `var x0` is renamed to `var` $\texttt{x0}_0$). For implicit global variables the special subscript "$_{\text{-}1}$" is used (e.g. `x = 1; var x; x = 2` is renamed to $\texttt{x}_{\text{-}1}$ `= 1; var` $\texttt{x}_0$`;` $\texttt{x}_0$ `= 2`). Renamed variable are guaranteed to be unique by the lexer because the regular expression for identifiers disallows subscripts.

modifications to the transitional semantics are:

$$\begin{aligned}
\text{eval}[\![\texttt{x}]\!] \triangleq\ & \text{if } \texttt{x} \in \text{dom}(\xi) \text{ then } h(\langle \xi(\texttt{x}), \texttt{val}\rangle) \\
& \text{else if } \langle l^*, \texttt{x}\rangle \in \text{dom}(h) \text{ then } h(\langle l^*, \texttt{x}\rangle) \\
& \text{else } h' = h[\langle l^*, \texttt{x}\rangle \mapsto \text{null}]; \text{null}
\end{aligned}$$

$$\begin{aligned}
\langle \texttt{x = } e, \langle \xi, h\rangle\rangle \Rightarrow\ & \text{match eval}[\![e]\!]\langle \xi, h\rangle \text{ with} \\
& \mid \text{error} \rightarrow \text{error} \\
& \mid v \rightarrow \langle \xi, h[\langle \text{if } \texttt{x} \in \text{dom}(\xi) \text{ then } (\xi(\texttt{x}), \texttt{val}) \text{ else } (l^*, \texttt{x})\rangle \mapsto v]\rangle
\end{aligned}$$

$$\begin{aligned}
\langle \texttt{malloc(x)}, \langle \xi, h\rangle\rangle \Rightarrow\ & \text{let } l \notin \text{loc}(h) \cup \text{null} \\
& \text{and } h' = h[\langle \text{if } \texttt{x} \in \text{dom}(\xi) \text{ then } (\xi(\texttt{x}), \texttt{val}) \text{ else } (l^*, \texttt{x})\rangle \mapsto l] \cup I(l) \text{ in} \\
& \langle \xi, h'\rangle \\
& \text{where } I(l) \triangleq \{\langle\langle l, \texttt{f}\rangle, \text{null}\rangle | \texttt{f} \in \texttt{Field}\}
\end{aligned}$$

4.3 Persistent Stack and Heap

To make the MiniOOL interpreter easier to use the stack and heap are persistent throughout the entire session. This allows the user to declare a variable on one line and use it on the next. For example the following is a example MiniOOL session:

```
# var x
Input:  var x0
Values:
x0:  null
# x = 1
Input:  x0 = 1
Values:
x0:  1
```

Where `x` is the same variable on both lines. To clear the stack and heap use the interpreter directive `\clear` (see Section 3.1 for more information about interpreter directives). Implementing this requires a modification to the operational semantics where variables are not popped from the stack until the end of the session, as opposed to after the procedding command(s).

4.4 Commands After Any Command

To facilitate implicit braces around command sequences an additional syntactic identifier is used. The command sequence, C^*, is either a command followed by a command sequence, a single command, or nothing (i.e. $C^* := C; C^* | C |$). This allows any command to have more commands after it without the need for explicit braces.

4.5 Equality and Inequality of Closures

Closures are defined as a reference to a record that contains a parameter, body, and stack field. Because the closures are implement in OCaml as references, it is easy to pass around and check if two referred closures are equivalent. In OCaml, the `==` and != operators are use to check if two reference have the same location in the machine. Therefore, the design choice was made to make closure comparison shallow. So even if the user defines two procedures with the same parameter and body, these two closures are not equivalent. This is semantically the same as the specifications because the stacks for each closure is unique and therefore the closures are not equivalent.

5 Examples

The MiniOOL interpreter only accepts single line input. The code snippets in this section are presented as multi-line input for readability, but these programs must be flattened to a single line before being interpreted by MiniOOL. For a MiniOOL source code file "source.mini", the source code can be flattened and interpreted using the bash command

```
printf "$(cat source.mini | tr -d '\n')\n" | ./MiniOOL --verbose
```

where `tr -d '\n'` removes all newlines.

The following sections cover the features available in MiniOOL. The source code for all of these files can be found in the `examples` directory (files ending in `.mini`). To run all the examples at once use the command `make examples`. This will run the example programs one by one.

MiniOOL will display three things. First, the input will be echoed with variables uniquely renamed according to the static semantics. Second, each example will be run with the flag `--verbose` which tells MiniOOL to print the abstract syntax tree. Last, the global values of variable declared on the stack will be printed. To check if the program ran successfully compare the value of the variable specified with the expected output. The script will pause in between each example, simply press any key to continue.

5.1 Recursive Factorial (`factorial.mini`)

The following example computes the factorial of 17 by defining a recursive procedure to compute factorial.

```
var r;
var factorial;
factorial = proc i:
    if (i <= 0) {
        r = 1;
    } else {
        factorial(i - 1);
        r = i * r;
    };
factorial(17);
r = 355687428096000 - r
```

where $17! = 355687428096000$ is used to check the correctness of the results. The resulting value of `r` is 0.

5.2 Recursive Fibonacci Series (`fibonacci.mini`)

The following code compute the n^{th} number in the Fibonacci Series, F_n, recursively:

```
var r;
var fibonacci;
fibonacci = proc n: {
    if (n <= 0) {
        r = 0;
    } else {
        if (n == 1) {
            r = 1;
        } else {
            fibonacci(n - 1);
            var tmp;
            tmp = r;
            fibonacci(n - 2);
            r = tmp + r;
        }
```

```
    };
};
var n; n = 17;
fibonacci(n);
r = 1597 - r;
```

where $F_{17} = 1597$ is used to check the correctness of the results. The resulting value of `r` is 0.

5.3 Iterative Fibonacci Series (`fibonacci_iterative.mini`)

The following code compute the n^{th} number in the Fibonacci Series, F_n, iteratively:

```
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 = 90;
fibonacci(n);
r = 2880067194370816120 - r;
```

where $F_{90} = 2880067194370816120$ is used to check the correctness of the results. The resulting value of `r` is 0.

5.4 Currying Parameters (`curry.mini`)

In MiniOOL, procedures only have one formal parameter. Although this may seem a limitation to procedures, in fact a procedure can take multiple arguments by using "currying". While a procedure `sum(x, y)` is not directly possible it can be achieved by defining a procedure `sum(x)` that "returns" a procedure `foo(y) = x + y` that takes one argument and "returns" the sum of `x` and `y`. The following MiniOOL code shows how to define and use such a procedure:

```
var r;
var sum;
sum = proc x: {
    r = proc y: {
        r = x + y;
    };
```

```
};
var x; x = 100;
var y; y = -100;
sum(x);
r(y);
```

The resulting value of `r` is 0. The value of `x` is accessible in `r = proc y: {r = x + y;};` because it creates a closure containing the current stack, which stores a location for the value of `x`.

5.5 Currying Unlimited Parameters (`curry_inf.mini`)

Extending from the currying example in Section 5.4, a procedure can be defined that takes an unlimited number of parameters. This is done by defining a procedure that takes one argument and "returns" two values: the intermediate results and a procedure to continue computations. The following MiniOOL program shows how one could accumulate the sum of the first 100 integers.

```
var r;
malloc(r);
var sum;
sum = proc x: {
    r.Result = x;
    r.Sum_rest = proc y: {
        r.Result = r.Result + y;
    };
};

sum(0);
var i; i = 1;
while (i <= 100) {
    r.Sum_rest(i);
    i = i + 1
};
r = 5050 - r.Result
```

Where 5050, the sum of the first 100 integers, is used to check the correctness of the results. The resulting value of `r` is 0.

5.6 Parallelism and Race Conditions (`race.mini`)

The following example illustrates the use and possible uncertainty problems of using the parallel operator, `|||`.

```
var x; x = 10; var r; {if x == 10 then r = x * 2 else skip ||| x = 5}
```

Depending on the order of commands run by the parallel operator the resulting value of `r` can be:

- `r = null` if `x = 5` is run first resulting in `x == 10` being `false` and r not being set.
- `r = 10` if the check `x == 10` is evaluated first then the command `x = 5` is evaluated and finally the comand `r = x * 2`.
- `r = 20` if the check `x == 10` and command `r = x * 2` are evaluated before the command `x = 5` is evaluated.

5.6.1 Atomic Commands (atomic_race.mini)

In this slightly altered version of the above example an atomic operation is added to evaluate the left side all at once.

```
var x; x = 10; var r; {atom(if x == 10 then r = x * 2 else skip) ||| x = 5}
```

Depending on the order of commands run by the parallel operator the resulting value of `r` can be:

- `r = null` if `x = 5` is run first resulting in `x == 10` being `false` and r not being set.
- `r = 20` if the command `atom(if x == 10 then r = x * 2 else skip)` is evaluated before the command `x = 5` is evaluated.

Note that the value of `r` cannot be 10 because the atomic command ensures that the body of the `if` is evaluated immediately after the evaluation of the conditional.

5.7 Implicit Global Variables with Dynamic Scope (dynamic_globals.mini)

The following MiniOOL program uses the implicit declaration of global variables and shows that they have dynamic scope:

```
var r; var p; p = proc y: r = y+h; h=2; p(4);
```

The resulting value of `r` is 6 in this example. The variable `h` is never declared, and is therefore treated as a implicit global variable. Although, `h` is initialized after the procedure p is declared, the value of `h` is used inside of p dynamically. After renaming variable to be unique the program is

```
var r0; var p0; p0 = proc y0:  r0 = (y0 + h-1); h-1 = 2; p0(4)
```