MiniOOL

Gabriel Cemaj
gc2728@nyu.edu

NYU — December 4, 2020

Introduction

MiniOOL is a limited, minimal object oriented programming language written for the CS 3110 (Fall 2020) final project. MiniOOL Is an interpreted language written in OCaml, primarily following the specifications indicated by the "Syntax and Semantics of MiniOOL" given to us in class. This report will give an overview of the language as well as motivate and explain any deviations taken from the above mentioned spec.

Compile MiniOOL

MiniOOL has a few external dependencies. These include menhir and dune. Menhir is a parser generator package, similar to ocamlyacc. Dune is a OCaml build system similar to Makefile in GNU. To compile MiniOOL just run

Command Line

```
$ dune exec -- src/miniool.exe
```

Using MiniOOL

MiniOOL also has a few modes, the default is to run inside the interpreter and passing no command will do that. You can also pass in the "AST" flag to have MiniOOL print the AST of the expression, or pass in a file to run the entire file. The default is to run in interpreter mode.

Command Line

```
$ dune exec src/miniool.exe
$ var x = 10; var y = 50;
$ debug(x+y)

$ IntValue -> 60
```

To run the AST mode, call MiniOOL with the following

Command Line

```
$ dune exec -- src/miniool.exe --ast
```

To run a file do so as follows

Command Line

```
$ dune exec -- src/miniool.exe --src /path/to/file
```

These are also stack-able, so you may call MiniOOL to print the AST of every line

Command Line

```
$ dune exec -- src/miniool.exe --ast --src /path/to/file
```

Warning: When using MiniOOL in file mode ensure to terminate every line with the ';' symbol.

Design Decisions

The following section will act as an overview showcasing some of the deviations that this implementation took from the provided spec. These decisions were taken to make the language more usable.

Expressions as commands

In order to make some of the bellow decisions work, we create a new command that evaluates an expression and ignores the results. This will allow us to have commands who only contain expressions.

Procedure Calls

The original specification of the semantics suggested that procedure calls be treated as commands. This makes the use of procedures cumbersome. As there is no easy way to extract the result of a procedure. This implementation instead views procedure calls as expressions. Making the expressions allows procedure calls to have a value they can return, doing this simplifies the use of the procedure call to not need pseudo global variables. In order to facilitate the procedure call the syntax declaration for the procedure was also altered to allow the inclusion of a return variable. We first must alter the domain of a closure to be

$$clo\langle x, y, C, \xi\rangle \in Clo \triangleq clo(\textbf{Var} \times \textbf{Var} \times \textbf{Cmd} \times \textbf{Stack})$$

Procedure declarations now use the following syntax

Command Line

```
$ var ident = proc (input, output): output = input
```

Or formally

$$e ::= \texttt{proc(x,y)}:C$$

Where x defines the input argument, and y defines a variable whose assignment inside the body of the

function will be the value returned to the caller. We can now define the evaluation of Call expressions as

$$
\begin{aligned}
eval[e(e')]\langle \xi, h\rangle \triangleq & \texttt{let e1 =}\ eval[e]\langle \xi, h\rangle in \\
& \texttt{let e2 =}\ eval[e']\langle \xi, h\rangle in \\
& \texttt{match e1 with} \\
& \ | \ clo\langle x, y, C, \xi'\rangle - > \\
& \quad \texttt{let}\ \xi'' = \xi' \cdot decl\langle \emptyset[\texttt{x} \mapsto \texttt{l1}], \xi\rangle \\
& \quad \texttt{and let}\ \xi'' = \xi'' \cdot decl\langle \emptyset[\texttt{y} \mapsto \texttt{l2}], \xi\rangle \\
& \quad \texttt{and let}\ h' = h[\langle \texttt{l1, val}\rangle \mapsto e1] \\
& \quad \texttt{and let}\ h' = h[\langle \texttt{l2, val}\rangle \mapsto null] \\
& \quad \texttt{and}\ \langle block(C), \langle \xi'', h'\rangle\rangle \\
& \quad \texttt{and}\ h'\langle \xi''(y), \texttt{val}\rangle \\
& \ | \ \_ > error
\end{aligned}
$$

Without this, something like a Fibonacci function would look something like

fib-old.mini

```
var result;
var fib = proc(n):
    if n <= 2 then result = 1
    else {
        fib(n-1);
        var f1 = result;
        fib(n-2)
        var f2 = result;
        result = f1 + f2
    }
fib(20)
debug(result) //prints 6765
```

And with the new system.

fib-new.mini

```
var fib = proc(n, result):
    if n <=2 then result = 1
    else result = fib(n-1) + fib(n-2)

debug(fib(20)) //prints 6765
```

This also allows us to return a procedure from a procedure

return-proc.mini

```
var adder = proc(x, r1):
    r1 = proc(y, r2):  r2=x+y
var add8 = adder(8)

debug(add8(2)) //prints 10
```

Declare and assign

For ease of use we add an additional command that declares and assigns a variable at the same time

Command Line

```
$ var x = 10;
```

Command Sequence

For ease of use we add an additional command that represents a sequence of commands, these are represented by explicit curly braces. This is particularly useful when dealing with nested functions pr when having long else statements

Malloc and Fields

In order to not have to add limitations on the format of fields (i.e. limit fields to start with a uppercase) we make the following modifications. We first modify the semantics of a field to be $e.field$ where field a identifier. This makes so a field expression must end with a identifier for the field. We extend the domain of values to have a ObjectValue. A object value is made up a tuple of a location and a value. We then modify malloc to only act on null values. Attempting to call malloc on a variable that is not null will result in a runtime exception. Calling malloc on a variable will modify its value to store a location and a value, this acts very much like a pointer. Doing so allows us to have the location on the heap to add or get fields. This removes any limitations on the format of the field. Evaluation of a field expression can now be seen as

$$
\begin{aligned}
eval[e.field]\langle \xi, h\rangle \triangleq\ & \texttt{let e1 = } eval[e]\langle \xi, h\rangle \texttt{in} \\
& \texttt{match e1 with} \\
& \mid obj\langle l, val\rangle -> h\langle l, \texttt{field}\rangle \\
& \mid \_ -> error
\end{aligned}
$$

And assigment follows similarly but we set the value of the field instead of retrieving it.

Debug

For ease of usability a "debug" command is added. This will print a evaluation of an expression to the screen.

Expression to Boolean

We also added the ability to evaluate expressions in the bool_expr clause. To do this we define a default truthiness of a integer. We arbitrarily decide to treat 0's as false and all other numbers as true. This resembles design decision take by other languages.

Examples

Here are some sample programs written in MiniOOL

fib-stream.mini

```
var fib = proc(n,r): {
    var x = 0;
    var y = 1;
    r = proc(a, r2): {
        r2 = x + y ; x = y; y = r2
    }
}
var fibber = fib(1)
var i = 20
while i > 0 do {debug(fibber(1)); i = i -1}
```

fact.mini

```
var fact = proc(n,r):
    if n <= 1 then r=1
    else r = n*fact(n-1)

debug(fact(5)) // prints 120
```