An Interpreted Scheme Dialect with a Reflective Tower

STEPHEN BARNES, Stanford University

This report discusses the implementation of an interpreter for a dialect of Scheme, written in Python. The language includes a simulated reflective tower, which is an abstraction allowing client code to pretend it is running on an infinitely deep stack of interpreters, where every interpreter interprets the interpreter above itself, ending with the client code on top of the tower. Every layer of the tower can modify the layer beneath it. This is accomplished finitely by dynamically instantiating new interpreter layers as they are needed. I will explain the operation of the system, its capabilities and limitations, and the reflection tools it provides. I will illustrate these with several examples of code written in the Scheme dialect.

ACM Reference Format:
Stephen Barnes. 2017. An Interpreted Scheme Dialect with a Reflective Tower. *Proc. ACM Hum.-Comput. Interact.* 1, 1, Article 1 (December 2017), 7 pages. https://doi.org/0000001.0000001

1 BACKGROUND

The reflective tower was first described by Brian Smith (1983) in his paper *Reflection and Semantics in Lisp*. He provided a formal definition of the notion of reflection in programming languages, and provided a definition for 3-LISP, a Lisp dialect with a reflective tower.

The reflective tower can be seen as a generalization of other reflection mechanisms in modern programming languages. For instance, in Python 2, instead of a straightforward variable assignment (“`x = 2`”) we can instead fetch the local symbol table and add the variable there directly (“`locals()['x'] = 2`”), though the interpreter recommends against this. Python also provides other forms of reflection, such as the definition of special member functions like `__get__` (used to index dictionary-like objects). These could be implemented in a language with a reflective tower by making the appropriate modifications to the interpreter. The same applies to Lua’s metatables or Lisp’s macros.

Despite the reflective tower being a generalization of these other metaprogramming capabilities, the latter are easier to use and probably more efficient. As such this project should be seen as an exercise in the field of programming languages, not as an attempt to construct a serious system intended for everyday use.

The next section will describe the syntax and semantics of a subset of Scheme implementing a reflective tower, and the ways in which an interpreter for this language, written in Python, resolves the problem of infinite regress. Finally, I will present some examples of the language’s capabilities, and a quick overview of my approach at writing the program.

2 LANGUAGE AND OPERATION

2.1 Basic syntax and semantics

This section defines the basic language constructs in the interpreted Scheme dialect.

Author’s address: Stephen Barnes, stbarnes@stanford.edu, Stanford University.

2573-0142/2017/12-ART1 $15.00
https://doi.org/0000001.0000001

First, the language has the same basic syntax as Scheme. Code consists of one or more expressions. All expressions are either atomic, or they are S-expressions. An atomic expression is either a symbol denoting some variable, or else it is a literal (a string, number, or boolean). S-expressions have the form `(operator arg1 arg2 ...)`, which is a list of expressions.

The semantics is likewise very simple. Literals evaluate to the denoted value. Symbols evaluate to the value of the variable with their name, in the appropriate context (see below). To evaluate an S-expression, we first recursively evaluate all the expressions contained in the list, then interpret the first object in the list as a function and apply the rest to it as its arguments.

There are some exceptions to the evaluation rule for S-expressions. If the operator is one of the symbols `if`, `lambda`, or `set`, special rules apply; these are the same as in Scheme. Additionally, there are four other keywords with special rules, namely `joots`, `set_global`, `set_universal`, and `floatinglambda`. These will be explained below.

2.2 Contexts

The language implements contexts as dictionaries mapping symbols (variable names) to values. Contexts are arranged in a "saguaro stack", meaning that every context stores a reference to its parent context; the name derives from the branching cactus-like shape of the resulting context tree. To find a symbol in a given context, we first search for the symbol in that context; if it is not present, we recursively try to find the symbol in the parent context. This terminates either when we find the symbol, or when we reach a context without any parent (the universal context), which will throw an exception. Contexts are automatically garbage-collected when no object holds a reference to them, which usually (but not always) happens when we leave the context.

A lambda-function captures the context in which it is defined, by storing a reference to it. When the lambda-function is called, it creates a subcontext of the captured context, adding in its own variable bindings. An extra `recurse` symbol binding is also added, pointing to the function itself; this allows anonymous lambda-functions to call themselves. This context-capture allows for the creation of closures, for instance in the curried function `(lambda (x) (lambda (y) (+ x y)))`.

I originally designed functions to store a frozen version of the context they were defined in, but I realized that contexts actually need to be modified even after capture, so a function should only store a reference to it. Consider for instance the following code for computing a triangular number $1 + 2 + \cdots + n$:

```
(set f (lambda (i accum)
    (if (= 0 i)
        accum
        (f (- x 1) (+ accum i)))))
(f 10 0)
```

The assignment by `set` of the function object to the symbol `f` happens after the definition of `f`, so if contexts could not change after capture then the recursive call to `f` would result in an undefined symbol error. In this case we can replace the recursive call with a call to `recurse`, but in more complicated situations such as mutual recursion this is either impossible or inconvenient. We could also solve the problem using a Python-like `def` operator which combines `set` and `lambda`, but this complicates things and leaves the problem unsolved for anonymous functions, especially in situations where we are procedurally defining multiple functions and would have to avoid naming collisions.

In developing several example programs using the reflection facilities described below, it became clear that it would be convenient to have a version of `lambda` that does not capture any local context but instead is executed with the context in which it is called, plus the additional local context containing the arguments as bindings and the `recurse` binding. The language allows creation of such functions with the `floatinglambda` keyword (since they float between contexts instead of being anchored to one context). Calling a floating function is roughly equivalent to using `set` to set its argument bindings and then executing the body of the function as if it were directly pasted into the code. Floating functions are also run at the interpreter layer at which they are called, rather than the one in which they were defined.

2.3 Towers and Interpreters

The Python program defines two classes called Tower and Interpreter. To run some client code, we first construct a Tower and then pass the code to it.

When a Tower is constructed, it creates a new "universal" context, which stores various functions that are common to all interpreters, such as basic arithmetic and list operations. The universal context also stores the default interpretation functions, which evaluate code by tokenizing and parsing it to produce an abstract syntax tree (AST), then evaluating the AST using structural recursion. A Tower also constructs a top interpreter and maintains a reference to it.

When an Interpreter is constructed, it creates a new "global" context, whose parent is the Tower's universal context. This global context contains only bindings that are specific to the individual interpreter; for everything else we defer to the universal context. It is important to note that an interpreter's stored global context is used for interpreting the layer *above* the interpreter; code executed by the interpreter itself must rely on the interpreter below. When code at some layer l defines a new variable, the interpreter at layer $d + 1$ updates its global context.

Note that variables defined in an interpreter's global context can override the definitions in the global context, for the layer it interprets. Therefore we can redefine an interpreter's `eval` function without affecting the behavior of other layers, and the universal context merely provides default values used when no special definition has been provided.

2.4 Avoiding Infinite Regress

When the Tower is asked to run code, it passes it to the top interpreter's `run` function. When an interpreter is asked to run code, the implicit infinite nature of the recursive tower starts to become problematic. Essentially an interpreter at depth d runs some expression like `(+ 1 2)` by asking its own interpreter (at depth $d+1$) to run the more complicated expression `(eval "(+ 1 2)")`. In other words, the interpreter at depth $d + 1$ must look up the definition of `eval` in the global context stored at depth $d + 1$, used for interpreting code at depth d. The problem, of course, is that this assumes an interpreter at depth $d + 1$ has been instantiated at all. Such a structure would imply that every interpreter has another interpreter already instantiated beneath it, making the infinite size of the tower literal rather than merely implicit.

The solution to this is to note that if an interpreter at depth d has no lower interpreter, the definition of `eval` at depth $d + 1$ would be the default definition, stored in the tower's universal context rather than in any global context. We only instantiate an interpreter at a given depth d when the layer above it, at $d - 1$, tries to run code in the layer d, using the joots operation described below. This system of falling back on the tower's universal context is the central mechanism which stops the infinite regress problem.

2.5 The joots operation

The term "joots" is an acronym for "jump out of the system", coined by Douglas Hofstadter to describe the act of stepping out of some system or framework and examining the system itself.

In the Scheme dialect described here, the joots operation allows a program to run code as if it were the code of the interpreter below it. All the language's other forms of reflection can be defined in terms of the joots operation. (A similar construct, called "exit", was used in Jefferson & Friedman (1996).)

Syntactically a joots expression is an S-expression whose first term is the symbol `joots`. When some interpreter at depth d evaluates a joots expression, it first accesses its own interpreter at depth $d+1$. If there is no such interpreter, it is instantiated; this is the only way interpreters past depth 1 are ever instantiated. Once we have the interpreter $d+1$, we fetch its own `ast_eval` function, which evaluates ASTs. (This is done by asking the interpreter at depth $d+2$ for the value of the `ast_expression` symbol in its global symbol table, or if no interpreter $d+2$ exists, we fall back on the universal context, as described above.) We then apply that `ast_eval` function to the rest of the terms in the joots expression.

As an example, every interpreter d's global context includes a `DEPTH` variable which stores the integer $d-1$ (since the variable is used by the layer above). Client code is designated as depth 0. Thus if we ask a tower to evaluate `DEPTH`, it will return 0. If we ask a tower to evaluate `(joots DEPTH)` it will return 1, and similarly `(joots (joots DEPTH))` will return 2.

Using a joots expression, a layer can thus rewrite its interpreter's bindings. For instance, consider the following code, which overrides the universal context's definition of `eval_literal`, called by `eval` to evaluate literals, replacing it with a function that always returns the string "Malkovich":

```
(joots (set eval_literal (lambda args "Malkovich")))
(print (+ 1 1))
(joots (print (+ 1 1)))
```

On the first line, the joots expression causes an interpreter at depth 2 to be created. Then the set expression is evaluated at the level of the depth-1 interpreter, which means that the depth-2 interpreter's global context (used to evaluate code at depth 1) will be given a value for the symbol `eval_literal`, which overrides the definition of that symbol in the universal context.

Therefore, when the line `(print (+ 1 1))` is evaluated (at depth 0, using the globals stored by depth 1), the two instances of "`1`" evaluate to the string "Malkovich". The `+` operator is overloaded to work on arbitrary-length lists of numbers or strings, so the middle line will print "MalkovichMalkovich". The third line, however, will print 2, because it is evaluated as depth-1 code, interpreted using the globals stored at depth 2, which have not been modified and thus default to the usual literal-evaluation function.

2.6 Python-Scheme Interface

The basic functionality of the interpreters, including the contents of the universal and global contexts, are written as Python code. Execution of code therefore involves calling both functions written in Scheme and functions written in Python. This necessitates some modifications to the form of the Python code.

First, where any of the Python functions would call some other Python function used for interpretation, we instead check whether that function has been redefined in the appropriate global context, and if so, call the new redefined value. This is accomplished by passing the interpreter object to functions such as `eval`.

For instance, when `eval` is used to evaluate expression like `(+ 1 2)` at depth d, we pass in both the expression and a reference to the interpreter; `eval` uses an if-elsif statement to determine whether the expression is atomic or an S-expression. Detecting that it is an S-expression, it passes the expression to a function `eval_s_expression`. However, instead of calling that function directly, it instead checks the definition of the symbol `eval_s_expression` for code running at depth d, by querying the globals of interpreter $d+1$ (or the universals if that interpreter doesn't exist). Then the retrieved value is called, with the expression and the reference to the interpreter d.

This implies that redefinitions only become effective the next time a function is called. This is unfortunately unavoidable because the alternative is incoherent. If a function like `eval` redefines itself at its own layer while it is executing, we cannot somehow "shift" execution to the new definition immediately. Such a shifting is often not even meaningfully possible; in the "Malkovich" example above, there is no clear way to translate an execution state to the new function since they are computing completely different things. Even if we could define a coherent execution-shifting function, we would have to check for modifications between every single line of Python code in order to reshift in time.

The Python code has been modified in some additional ways to make reflection easier:

(1) The universal and global contexts include some functions for fetching the current context, global context, and universal context. There are also functions for copying contexts and updating them with new values in bulk. This allows us to redefine several bindings in a context at once, by obtaining a reference to some context, copying it, changing values in the copy, and then updating the original with the copy's changed bindings.
(2) The code is written with a lot of functional decomposition, such as separating `eval` from `eval_s_expression`. This allows easier reflection, since we can modify a small sub-function like `eval_s_expression` without having to modify the whole `eval` function.
(3) In addition to the `set` keyword which creates or modifies local bindings, there is also a keyword `set_global` which works with the layer's global context, and a keyword `set_universal` which works with the tower's universal bindings.

3 EXAMPLES

Here are some examples illustrating the features of the language. The following defines a recursive function for computing factorials, using ordinary Scheme features with no reflection:

```
(begin
    (set factorial
        (lambda (x)
            (if (= x 0)
                1
                (* x (factorial (- x 1))))))
    (factorial 5))
```

This code uses the `set_global` function to keep track of a constant without having to pass it around, and the `infinite_loop` function to loop forever without exploding the stack:

```
(begin
    (set i 0)
    (infinite_loop
        (lambda ()
            (begin
                (set_global i (+ i 1))
                (print i)))))
```

This example creates a variable by directly adding it to its interpreter's context, instead of the simpler `(set x 10)`:

```
(begin
    (set_index (get_current_context) "x" 10)
    (print x))  ; prints 10
```

This code prints out the current depth and then runs itself at one level deeper. The function very quickly gets about 70 levels deep before we exceed Python's default recursion limit (which can be increased). Notice that we must use a `floatinglambda` here, or we will capture the starting layer's `DEPTH` (as well as always jootsing into layer 1 instead of going progressively deeper):

```
(set_universal descender
    (floatinglambda ()
        (begin
            (print "DESCENDER IS AT DEPTH" DEPTH)
            (joots (descender)))))
(descender)
```

Finally, this code modifies the tokenizer so that the rest of the document can be written using curly braces instead of curved parentheses. This makes use of some of the details of the default tokenizer's definition, including the `;FINISH-BEFORE` directive, used to stop tokenization at some point and continue after processing everything before it.

```
; example of how the FINISH-BEFORE directive works
(print "before redefining tokenization to use curly braces")
; define a tokenization function that uses curly braces
; this should return a list of tokens, then the remaining text (which we'll just make None)
(set_universal new_tokenize
    (lambda (s interpreter) (begin
        ; first remove comments
        (set s
            (remove_comments s INTERPRETER))
            ; then parse into tokens
        (set tokens
            (regex_findall (+ "\{|\}|[^\'{}\s]+|\'[^\']*\'") s))
        ; then replace the [] with (), to plug into parse_tokens properly
        (set tokens
            (map
                (lambda (x) (if
                    (= x "{")
                        "("
                        (if (= x "}")
                            ")"
                            x)))
                tokens))
        (‘ tokens #f))))
; now assign interpreter's tokenize to the new tokenization function
(joots (set tokenize new_tokenize))
;FINISH-BEFORE
{print "after redefining tokenization to use curly braces"}
{print {+ 1 {* 2 3}}}
```

Some more examples are included with the project submission in the `examples` folder.

4 DEVELOPMENT APPROACH

I began by writing a non-reflective Scheme interpreter in Python. This essentially functioned the same as the interpreter design described above, with input code passing through a pipeline of tokenization, parsing, and finally a structurally recursive evaluation function. I added doctests and test programs for faster iteration. Later I added the context object and support for closures. During this part I referenced a tutorial by Peter Norvig describing the implementation of a Scheme interpreter in Python (Norvig, 2010), though almost none of his code still exists in the final program due to rewriting things to work with the reflective tower.

Thereafter I diagrammed the design for the Tower and Interpreter objects, and planned out the desired control flow path for some example inputs. Implementing these objects required retooling most of the non-reflective interpreter code to properly slot into the Interpreter objects so as to allow them to be modified. Testing of the reflective abilities revealed the need for other language constructs, such as context-free functions.

Once this was done I added a few final features, such as REPL line history and editing shortcuts, command-line argument parsing, and Python doctests.

5 REFERENCES

(1) Jefferson, S. & Friedman, D. P. *A simple reflective interpreter*. LISP and Symbolic Computation, 1996, 9, 181-202.

(2) Norvig, P. *(How to Write a (Lisp) Interpreter (in Python))*. 2010. Web, http://norvig.com/lispy.html, retrieved December 2017.

(3) Smith, B. *Reflection and Semantics in Lisp*. Xerox Palo Alto Research Center, published in ACM 1983.