Programming Language Paradigms Final Exam

Abrar Habib Jay Noppone Pornpitaksuk

May 20, 2024

Abstract

1. Introduction

Interpreters play a crucial role in the execution of high-level programming languages by translating code into executable actions. The TLS interpreter originating from Friedman's *The Little Schemer*, was designed with lexical scoping and support for higher-order functions, illustrating use-cases and examples of these concepts [FF95]. Lexical scoping determines variable bindings based on their physical locations within the source code, creating a predictable and consistent environment for variable resolution. Technically speaking, if a variable were to be declared under a function, that variable is only accessible within the function declared and is not able to be called by outside functions outside of its scope. This essay provides a detailed argument demonstrating that the TLS interpreter correctly implements lexical scoping and higher-order functions.

2. Lexical Scoping and First-Class Functions

Lexical scope is a scope resolution technique used in programming languages where the scope of a variable is determined by its location in the source code. In lexical scoping, variables are resolved based on the structure of the program and the textual context in which they appear. To be more specific, Scheme's lexical scoping is limited to being statically scoped rather than dynamically scoped [HS85].

2.1. TLS Interpreter and Lexical Scoping

In the TLS interpreter, lexical scope is implemented through the use of environments represented as tables. The `lookup-in-table` function is responsible for looking up variables in an environment based on lexical scoping rules. This function traverses the environment table to find the variable and returns its value. The `extend-table` function is used to extend the environment with new variable bindings.

Below is an example function definition and application in TLS:

```
(define (combine-f f1 f2 combiner x y)
  (let ((result1 (f1 x))
        (result2 (f2 y)))
    (combiner result1 result2)))
```

In this example, `combine-f` is a higher-order function that accepts 3 higher-order function parameters and then arguments `x` and `y`. The function `combine-f` processes `(f1 x)` and `(f2 y)`, and then applies these to the let function that binds the results to result1 and result2, which are then both applied to the function `combiner`. The inner function captures the environment where all of the parameters are defined in order to evaluate the functions and non-function parameters. When this inner function is called, the lexical scope helps resolve the parameters given for this function.

3. Correctness of the TLS Interpreter

The TLS interpreter employs several key components to ensure correct implementation of lexical scoping and higher-order functions:

1. **Environment Representation**
   The environment is implemented as a table, which is extended and passed along during function application. This table maintains variable bindings for each lexical scope.

   ```
   (define extend-table cons)
   ```

2. **Function Definition and Application**
   When a `lambda` expression is encountered, the interpreter constructs a closure that includes the function body and the environment at the time of definition.

   ```
   (define *lambda
     (lambda (e table)
       (build (quote non-primitive)
              (cons table (cdr e)))))
   ```

   During function application, the interpreter extends the current environment with the function's parameters and their corresponding arguments.

   ```
   (define myapply-closure
     (lambda (closure vals)
       (meaning (body-of closure)
                (extend-table
                 (new-entry
                  (formals-of closure)
                  vals)
                 (table-of closure)))))
   ```

3. **Variable Resolution**
   The interpreter resolves variables by looking them up in the current environment, ensuring that the correct bindings are used based on the lexical scope.

```
(define lookup-in-table
  (lambda (name table table-f)
    (cond
      ((null? table) (table-f name))
      (else (lookup-in-entry name
                             (car table)
                             (lambda (name)
                               (lookup-in-table name
                                                (cdr table)
                                                table-f)))))))
```

   Example Demonstrating Lexical Scoping
   Consider the following example in TLS:

```
(define outer-fn
  (lambda (a)
    (lambda (b)
      (+ a b))))

(define inner-fn (outer-fn 5))

(inner-fn 10)
```

   In this example, `outer-fn` defines a function that returns another function. The inner function captures the environment where `a` is bound to `5`. When `inner-fn` is called with `10`, it correctly resolves `a` to `5` and `b` to `10`, resulting in the value `15`. The use of this example ensures that the value `5` will directly bound to the call for `b` no matter what parameter is called when calling the function `inner-fn`.

3.1. Structural Induction of the Interpreter

To prove the correctness of the TLS interpreter with lexical scoping, we use structural induction:

1. **Base Case:** For atomic expressions (constants and variables), the interpreter correctly resolves their values based on the current environment. At the time that the interpreter encounters a constant, it will simply return the constant itself. No values are changed and do not depend on environment rules. Overall, the interpreter correctly looks for the value in the environment; if the value is not found to pass the conditions, it will visit the else statement, calling the build function instead.

```
(define *const
  (lambda (e table)
    (cond
      ((number? e) e)
      ((eq? e #t) #t)
      ((eq? e #f) #f)
      (else (build (quote primitive) e)))))
```

2. **Inductive Case** For compound expressions (function applications and lambda expressions), the interpreter correctly constructs closures and extends environments, ensuring that variables are resolved based on their lexical scope.

```
(define meaning
  (lambda (e table)
    ((expression-to-action e) e table)))
```

   By induction, the interpreter maintains the invariant that variables are resolved based on their lexical scope at each step of the evaluation.

4. Conclusion

The interpreter written in the Little Schemer properly and correctly implements lexical scoping and higher-order functions by correctly working with and manipulating its environments. Variables are properly resolved when there are conflicts found, or when the base case is reached, showing consistent and efficient error handling. Through proof and induction, we've demonstrated the correctness of the interpreter's handling of these features within R5RS Scheme.

5. Bibliography

Citing the TLS book. [FF95] Citing the sicp book [HS85]

References

[FF95] D. P. Friedman and M. Felleisen. *The Little Schemer*. 4th. Cambridge, MA: MIT Press, 1995.

[HS85] G. J. S. Harold Abelson and J. Sussman. *Structure and Interpretations of Computer Programs (SICP)*. 4th. Cambridge, MA: MIT Press, 1985.