A Substitution Model for the Kernel of Scheme

Grant I. Ho*
gumbee@mit.edu
MIT Laboratory for Computer Science

Abstract: This paper presents the formal specifications for the development of a substitution model for the kernel of Scheme, written in the kernel of Scheme. A substitution model defines a set of rewrite rules according to which expressions in the functional Scheme kernel are successively reduced to other such expressions until a value for the original expression is returned. Since regular Scheme expressions can be derived from the syntax of a functional Scheme kernel, the kernel of Scheme was selected as the language of choice because it would lead to a simpler, cleaner, and more efficient implementation of the substitution model. The purpose in developing such a model was pedagogical. It would supplement the teaching of the substitution model to students first learning Scheme, serve as an expression debugger, and help in understanding the reduction process in this language.

Keywords: kernel of Scheme, keyword, expression, value, variable, instantiation, garbage-collection, enforce, relax.

1 Formal Definitions

1.1 Kernel Syntax

In designing a substitution model for the kernel of Scheme, it is necessary to define the syntax for the functional Scheme kernel as described in Revised^4 Scheme Report [1]. The kernel syntax consists of 4 parts: *keywords, expressions, auxillaries* and *constants. Keywords* include: if, lambda, and letrec. Although there are only three keywords defined in the functional kernel, extended Scheme grammar such as and, or, begin, cond, and let, can be derived from these three essential keywords. From these keywords, Scheme *expressions* (denoted by **N, M,** and **B**) can be formed: (if **M N B**), (lambda (<formals>) **B**), and (letrec (<bindings>) **B**). Formals and bindings are known as *auxillaries.* Finally, *constants* consist of *numerals* (eg. 0, -1, 3.14159), *booleans* (eg. #t, #f), and *scheme-constants* (eg. <<+>>, <<equal?>>, and <<cons>> built-in procedures).

It is important to realize that the primitive procedure for addition, +, is *not* a scheme-constant. Instead, it is a *variable* that identifies the primitive addition procedure built into the Scheme interpreter. Its associated symbol, <<+>>, *is* a scheme-constant because it represents the actual addition operation. In the substitution model, variables that represent primitive procedures have associated scheme-constants that are defined by the programmer in the initial-global-environment and enclosed between pointed brackets, <<>>.

1.2 Definition of Values

Two types of *values* exist in the Scheme kernel: *basic values* and *letrec values.* A *basic value* is the set of numerals, booleans, scheme-constants, and lambda values: (lambda (<variable>) value). *Letrec values* have the form: (letrec ((x_1 basic-value$_1$) (x_2 basic-value$_2$) (x_3 basic-value$_3$)) value) [4]. Any expression that is not a value is reduced by the rules of substitution model.

*Grant Ho was supervised by Professor Albert R. Meyer in the Theory of Computation Group in the summer of 1994. The three other undergraduate students who contributed significantly to this project were Daniel O'Brien '95, Brian Semmes '96, and Yi-Hung Li '97. Address: 3 Ames Street, Room Bemis 202, Cambridge MA 02142. WWW: http://www.mit.edu:8001:/people/gumbee/homepage.html

2 Substitution Model Overview

The substitution model developed by the author and his colleagues follows a functional design and is recursively controlled by a top-level evaluation loop as follows (see Figure 1): 1) a kernel Scheme expression is passed to the model as an argument, 2) the model checks the expression for its type (ie: if, letrec, or lambda) 3) the model applies one rewrite rule according to expression-type, and 4) a new kernel expression, reduced by one step, is returned and displayed.

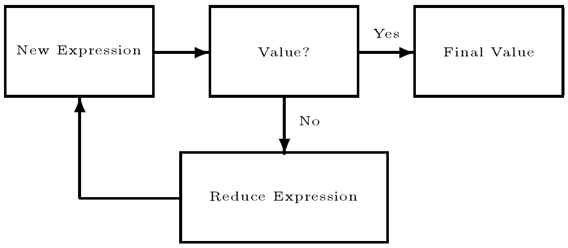

Figure 1: Block diagram of the substiution model loop.

2.1 Key Features of Substitution Model

One of the main innovations in creating a substitution model is the formation of letrec expressions in ways that correspond to the creation of *environment frames* (regions where variables are bound to values), thereby avoiding unnecessary copying. Instead of relying on the *beta-substitution* model in the evaluation of lambda expressions, the substitution model *desugars* lambda applications into letrec applications as follows:

((lambda (x) **M**) N) → (letrec ((x **N**)) **M**)

The result of this is a substitution model that more realistically and accurately reflects the behavior of time and space of the environment model [5].

There are 6 major sections in the substitution model code: 1) the initial-global-environment, 2) definitions of keywords and values, 3) main evaluation loop 4) rewrite rules, 5) garbage-collection, and 6) variable renaming.

2.1.1 Initial Global Environment

The initial-global-environment contains a list of primitive procedures and their associated kernel scheme-constants. Since primitive procedures are variables and not values, they must be *instantiated* with their corresponding scheme-constants in the reduction process. Instantiation occurs every time a variable is bound to a value in the initial-global-environment or another environment frame. The initial-global-environment also contains a lookup-table that maps a scheme-constant to its associated operator in the Scheme interpreter so that evaluation of the expression may take place.

2.1.2 Main Evaluation Loop

The main evaluation loop contains the kernel-eval and left-reduce procedures. Kernel-eval is the top-level evaluation loop that passes a Scheme expression entered by the user, performs a top-level-garbage-collect on the expression, and checks to see if the new expression is a value. If it is a value, then the expression is returned and displayed. Otherwise, the model returns the garbage-collected expression to the left-reduce procedure for a one-step reduction. Left-reduce takes the garbage-collected expression and the initial-global-environment as arguments, checks the expression for its type, and returns the expression, based on type, to a specific reduction procedure. An expression continues to be left-reduced by one step until a value is returned.

2.1.3 Rewrite Rules

There are five main reduction procedures in the substitution model: reduce-if, reduce-letrec, reduce-lambda, reduce-inits, and reduce-combination. Reduce-combination reduces expression combinations or user-defined procedures applied to their arguments (eg. (+ 1 2 3) or (factorial 5), where factorial is defined with a letrec expression). All reduction procedures in the substitution model employ the same idea in rewriting to a new expression. The car, or leftmost element of a given expression, is first left-reduced by one step, and continues to be reduced by only one step until it becomes a value. Then, left-reduce is called on the cdr of that expression such that the leftmost element of the new subexpression is left-reduced, again by one step, until *it* becomes a value. The process continues until all elements in the original expression have become values or until a single evaluation of the expression can take place according to expression-type [1].

To evaluate an expression or subexpression whose elements contain all values, left-reduce calls a procedure named apply-proc. Apply-proc finds the primitive operator, in the lookup-table, that corresponds to the car of the expression to be evaluated, and applies that built-in operator to the cdr of the expression. This step returns a value for the expression.

2.1.4 Garbage Collection

The addition of a top-level-garbage-collect procedure to the main evaluation loop maps a generic garbage-collect procedure to the cdr of a given expression, thereby forcing all subexpressions to be garbage-collected before reduction takes place. The garbage-collect procedure, which takes a letrec expression as an argument, ensures the removal of all bindings whose variables have already been instantiated into the body of the letrec expression, and hence have no more bound occurences [4]. To identify such variables, an auxillary procedure named free-variables was defined. The strategy behind a top-level garbage-collection was returning a simpler expression to left-reduce and hence, providing a faster and more efficient substitution model. Garbage-collection in general, also returns cleaner and neater expressions to the user.

2.1.5 Variable Renaming

Two complementary variable-renaming procedures are defined in the substitution model: enforce and relax. The enforce procedure takes a letrec expression and adds a fresh suffix number, starting from 1, to the ends of all variables of the same name found in that expression. For example, in the expression, (letrec ((x 1)) (+ x (letrec ((x 2)) x))), the binding-variable x has two occurences, one nested inside the other letrec. Thus, enforcing the expression returns (letrec ((x#1 1)) (+ x#1 (letrec ((x#2 2)) x#2))), where the variable x has been enforced to x#1 in the outer letrec and x#2 for the inner letrec. The purpose behind enforcing an expression is to avoid the possibility of variable-name clashing. Since a letrec expression can be defined to contain nested letrecs and the use of the same variable more than once, enforcing an expression at the top-level ensures that variables of the same name (with different bindings) have different suffixes . This avoids potential errors in variable instantiation.

Relax works opposite to the enforce procedure. After the end of the reduction process, relax takes the expression, which has already been enforced and garbage-collected, and returns an expression with free variables untouched and all bounded variables renamed with the minimum number of suffixes. The result is a cleaner expression that can be more easily understood by the user.

The following example illustrates the substitution model in action on a simple Scheme expression:

User enters:

```
(kernel-eval '(if (>2 1)
                  ((lambda (x y) (+ x y)) 3 4)
                  (+ 1 2))
```

Substitution model returns:

```
== (0) ==>
(if (> 2 1) ((lambda (x y) (+ x y)) 3 4) (+ 1 2))
== (1) ==>
(if (<<_>_>> 2 1) ((lambda (x y) (+ x y)) 3 4) (+ 1 2))
== (2) ==>
(if #t ((lambda (x y) (+ x y)) 3 4) (+ 1 2))
== (3) ==>
((lambda (x y) (+ x y)) 3 4)
== (4) ==>
(letrec ((x 3) (y 4)) (+ x y))
== (5) ==>
(letrec ((x 3) (y 4)) (<< + >> x y))
== (6) ==>
(letrec ((y 4)) (<< + >> 3 y))
== (7) ==>
(<< + >> 3 4)
== (8) ==>
7
;Value; done
```

In the zeroth step, top-level garbage-collection takes place on the entire if expression and all overlapping variable names are enforced with fresh suffixes. In the first step, the variable > becomes instantiated with its kernel scheme-constant, << _ > _ >>, as defined in the initial-global-environment. Since the *test* of the if expression contains all values, apply-proc is called on the *test* expression which evaluates to #t. Thus, the *consequent* of the if expression is returned as defined in Revised^4 Scheme [1]. In the fourth step, the lambda expression is desugared into a letrec expression by the rules of the reduce-lambda procedure and in the fifth step, the variable + becomes instantiated with its <<+>>. In the sixth step, the letrec init, (y 4), is garbage-collected since it is no longer has an occurence in the body of the expression. The body of letrec, (<< + >> 3 4) is returned, and apply -proc applies the primitive addition operator found in the lookup-table to the operands, 3 and 4. A final value of 7 is returned.

References

[1] Harold Abelson et al. Revised^4 Report on the Algorithmic Language Scheme. MIT Artificial Intelligence Laboratory, November 1991. A.I. Memo No. 848.

[2] Harold Abelson and Gerald J. Sussman with Julie Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, Cambridge, MA, 1985.

[3] Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science* 103:235-271, 1992.

[4] Albert R. Meyer. A Substitution Model Draft for Scheme: Formal Definitions. MIT Laboratory for Computer Science, February 1994.

[5] Albert R. Meyer. User's Guide for the Substitution Model. MIT Laboratory for Computer Science, April 1994.