INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

ProQuest Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

UMI®
Modeling an Algebraic Stepper

by

John Clements

A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE
Master of Science

APPROVED, THESIS COMMITTEE:

Matthias Felleisen, Professor, Chair
Department of Computer Science

Keith Cooper, Professor
Department of Computer Science

William Wilson, Professor
Department of Electrical and Computer Engineering

Houston, Texas

April, 2001
Modeling an Algebraic Stepper

John Clements

Abstract

Programmers rely on the correctness of their tools. Semanticists have long studied the correctness of compilers, but we make the case that other tools deserve semantic models, too, and that using these models can help in developing these tools.

We examine these ideas in the context of DrScheme's stepper. The stepper operates within the existing evaluator, placing breakpoints and reconstructing source expressions from information placed on the stack. We must ask whether we can prove the correspondence between the source expressions emitted by the stepper and the steps in the formal reduction semantics.

To answer this question, we develop a high-level semantic model of the extended compiler and run-time machinery. Rather than modeling the evaluation as a low-level machine, we model the relevant low-level features of the stepper's implementation in a high-level reduction semantics. The higher-level model greatly simplifies the correctness proof. We expect the approach to apply to other semantics-based tools.
Contents

Abstract ii
List of Illustrations v

1 The Correctness of Programming Environment Tools 1

2 An Algebraic Stepper for Scheme 3

3 Marking Continuations 5
   3.1 Introduction to Continuation Marks ...................... 6
   3.2 Breakpoints and Continuation Marks .................... 8
   3.3 Properties of the Model ............................... 15

4 Stepping with Continuation Marks 17
   4.1 Elaboration and Reconstruction .......................... 17
   4.2 Properties of the Stepper .............................. 18

5 The Pragmatics of a Stepper 22
   5.1 One-Step Annotation ..................................... 22
   5.2 The Content of Marks .................................... 22
   5.3 Opaque Values ........................................... 23
   5.4 Unannotated Code ........................................ 24
   5.5 Scheme Macros and Source Correlation .................. 25
   5.6 Post-reduction Breakpoints .............................. 26
6 Related Work

7 Conclusion

Bibliography

A Detailed Proof Sketches

A.1 The Debugger Theorem ........................................ 36
  A.1.1 Well-Definition of $M$ .................................. 37
  A.1.2 Well-definition of $M_t$ ................................. 37
  A.1.3 The correspondence between $M$ and $A[M]$ .......... 38
A.2 The Stepper Theorem ........................................ 42
Illustrations

2.1 The stepper in action ........................................ 4
3.1 The relationship between continuation-marks and tail-recursion ... 6
3.2 Grammar and Reduction rules for the source language ................ 9
3.3 Extension of the source language $M$ to the target language $M_t$ ... 10
3.4 The annotating function, $A: M \rightarrow M_t$ ........................... 13
3.5 The Translation function, $T: V \rightarrow E$ ........................... 14
4.1 The stepper's breakpoint-inserting function, $B: M_s \rightarrow M$ (part 1) . 19
4.2 The stepper's breakpoint-inserting function, $B: M_s \rightarrow M$ (part 2) . 20
5.1 Unannotated Code: evaluation of (car (map square (list 1 2 3 4))) . 25
A.1 "Chasing the Arrows" .......................................... 38
Chapter 1

The Correctness of Programming Environment Tools

Programming environments provide many tools that process programs semantically. The most common ones are compilers, program analysis tools, debuggers, and profilers. Our DrScheme programming environment [9, 8] also provides an algebraic stepper for Scheme. It explains a program’s execution as a sequence of reduction steps based on the ordinary laws of algebra for the functional core [2, 22] and more general algebraic laws for the rest of the language [6]. An algebraic stepper is particularly helpful for teaching: students often have trouble with the key concepts of recursion, and seeing the evaluation of the program in a step-by-step fashion can enlighten them. Selective uses can also provide excellent information for complex debugging situations.

Traditionally researchers have used semantic models to verify and to develop compilation processes, analyses, and compiler optimizations. Other semantics-based programming environment tools, especially debuggers, profilers, or steppers, have received much less attention. Based on our development of DrScheme, however, we believe that these tools deserve the same attention as compilers or analyses. For example, a debugging compiler and run-time environment should have the same extensional semantics as the standard compiler and run-time system. Otherwise a programmer cannot hope to find bugs with these tools.

The implementation of an algebraic stepper as part of the compiler and run-time environment is even more complex than that of a debugger. A stepper must be able to display all atomic reduction steps as rewriting actions on program text. More
specifically, an embedded stepper must be guaranteed

1. to stop for every reduction step in the algebraic semantics; and

2. to have enough data to reconstruct the execution state in textual form.

To prove that an algebraic stepper has these properties, we must model it at a reasonably high level so that the proof details do not become overwhelming.

In this thesis, we present a semantic model of our stepper's basic operations at the level of a reduction semantics. Then we show in two stages that the stepper satisfies the two criteria. More precisely, in the following chapter we briefly demonstrate our stepper. The third chapter introduces the reduction model of the stepper's run-time infrastructure and presents the elaboration theorem, which proves that the stepper infrastructure can keep track of the necessary information. The fourth chapter presents the theory behind the algebraic stepper and the stepper theorem, which proves that the inserted breakpoints stop execution once per reduction step in the source language. Together, the two theorems prove that our stepper is correct modulo elaboration into a low-level implementation. The fifth chapter discusses some of the obstacles we overcame in the translation of the model to a concrete implementation. The thesis concludes with a brief discussion of related and future work. Appendix A fleshes out the theorems stated in the text.
Chapter 2

An Algebraic Stepper for Scheme

Most functional language programmers are familiar with the characterization of an evaluation as a series of reduction steps. As a toy example, consider the first few steps in the evaluation of a simple factorial function in Scheme:

\[
(fact\ 3)
\]

\[
= (if\ (=\ 3\ 1)\ 1\ (\ast\ (fact\ (-\ 3\ 1))\ 3))
\]

\[
= (if\ false\ 1\ (\ast\ (fact\ (-\ 3\ 1))\ 3))
\]

\[
= (\ast\ (fact\ (-\ 3\ 1))\ 3)
\]

\[
\ldots
\]

Each step represents the entire program. The boxed subexpression is the standard redex. The sequence illustrates the reduction of function applications ($\beta_v$), primitive applications ($\delta_v$), and if expressions.

DrScheme implements a more sophisticated version of the same idea. The two screen dumps of fig. 2.1 show the visual layout of the stepper window. The window is separated into three parts. The top shows evaluated forms. Each expression or definition moves to the upper pane when it has been reduced to a canonical form. The second pane shows the redex and the contractum of the current reduction step. The redex is highlighted in green, while the contractum—the result of the reduction—is highlighted in purple. The fourth pane is reserved for expressions that are yet to be evaluated; it is needed to deal with lexical scope and effects.

The two screen dumps of fig. 2.1 illustrate the reduction of an arithmetic expression
and a procedure call. The call to the procedure is replaced by the body of the procedure, with the argument values substituted for the formal parameters. Other reduction steps are modeled in a similar manner. Those of the imperative part of Scheme are based on Felleisen and Hieb's work on a reduction semantics for Scheme [6]; they require no changes to the infrastructure itself. For that reason, we ignore the imperative parts of Scheme in this thesis and focus on the functional core.
Chapter 3

Marking Continuations

The implementation of our stepper requires the extension of the existing compiler and its run-time machinery. The compiler must be enriched so that it emits instructions that maintain connections between the machine state and the original program text. The run-time system includes code for decoding this additional information into textual form.

To model this situation, we can either design a semantics that reflects the details of a low-level machine, or we can enrich an algebraic reduction framework with constructs that reflect how the compiler and the run-time system keep track of the state of the evaluation. We choose the latter strategy for two reasons.

1. The reduction model is smaller and easier to manage than a machine model that contains explicit environments, stacks, heaps, etc. The research community understands how compilers manage associations between variables and values. Modeling this particular aspect would only pollute the theorems and proofs, without making any contribution.

2. It is possible to derive a variety of low-level machines from a high-level semantics [5, 14]. These derivations work for our extended framework, which means the proof carries over to several implementations of the low-level mechanism.

The goal of this chapter is to introduce the source-level constructs that model the necessary continuation-based information management and to show that they can keep track of the necessary information. The model is extended in the next chapter to a model of a stepper for the core of a functional language.
(define (fact n)
  (letrec
    ([! (lambda (n)
         (if (= n 0)
           (begin
             (display (c-c-m))
             (newline)
             1)
           (w-c-m n
             (* n (! (- n 1))))))]
     (! n)))

> (fact 4)

(1 2 3 4) ; displayed output

24

> (fact 4)

(1) ; displayed output

24

Figure 3.1: The relationship between continuation-marks and tail-recursion

Chapter 3.1 presents continuation marks, the central idea of our model. Chapter 3.2 formalizes this description as an extension of the functional reduction semantics of Scheme. Chapter 3.3 introduces an elaboration function that inserts these forms into programs and states a theorem concerning the equivalence of a program, its elaborated form, and the annotations in the reduction semantics.

3.1 Introduction to Continuation Marks

In a reduction semantics, every reducible program is divided into a "redex," or "reducible expression," and a context — that is, everything outside the redex. In some functional languages, it is possible to capture this context as a first-class value; a procedure of one argument. These are called continuations. It is natural to regard a program’s continuation as a series of frames. In this context, a continuation mark is a distinct frame that contains a single value.
Continuation marks provide a means to "label" these contexts, or continuations, with program values, thus permitting a run-time observation of the dynamic program state. Our model provides two operations to manipulate these marks:

(with-continuation-mark mark-expr expr) : mark-expr and expr are arbitrary expressions. The first evaluates to a mark-value, which is then placed in a marked frame on top of the continuation. If the current top frame is already marked, the new frame replaces it. Finally, expr is evaluated. Its value becomes the result of the entire with-continuation-mark expression.

(current-continuation-marks) : The result of this expression is the list of values in the mark frames of the current continuation, from outermost to innermost.

The elimination of the outer mark when two occur consecutively is provided to allow the preservation of tail-optimizations for all derived implementations. In other words, our model ensures that an infinite series of tail calls will not consume an infinite amount of memory in the form of stale marks. Not all machines are tail-optimizing, e.g., the original SECD machine [18], but due to this provision our framework works for both classes of machines.

The two programs in fig. 3.1 illustrate how an elaborator may instrument a factorial function with these constructs. Both definitions implement a factorial function that marks its continuation at the recursive call site and reports the continuation-mark list before returning. The one in the left column is properly recursive, the one on the right is tail-recursive. The boxed texts are the outputs that applications of their respective functions produce. For the properly recursive program on the left, the box shows that the continuation contains four mark frames. For the tail-recursive variant, only one continuation mark remains; the others have been overwritten during the evaluation.

---

1For space reasons, with-continuation-mark and current-continuation-marks are abbreviated as w-c-m and c-c-m from now on.

2This elision of continuation marks is the principal difference between our device and that of
3.2 Breakpoints and Continuation Marks

To formulate the semantics of our new language constructs and to illustrate their use in the implementation of a stepper, we present a small model and study its properties. The model consists of a source language, a target language, and a mapping from the former to the latter.

The source and target language share a high-level core syntax, based on the $\lambda$-calculus. The source represents the surface syntax, while the target is a representation of the compiler's intermediate language. The source language of this chapter supports a primitive inspection facility in the form of a (breakpoint) expression. The target language has instead a continuation mark mechanism. The translation from the source to the target demonstrates how the continuation mark mechanism can explain the desired breakpoint mechanism.

The syntax and semantics of the source language are shown in fig. 3.2. The set of program expressions is the closed subset of $M$. The primitives are the set $P$. The set of values is described by $V$. The semantics of the language is defined using a rewriting semantics [6].$^3$ $E$ denotes the set of evaluation contexts. Briefly, a program is reduced by separating it into an evaluation context and an instruction—the set of instructions is defined implicitly by the left-hand-sides of the reductions—then applying one of the reduction rules. This is repeated until the process halts with a value or an error.

To model output, our reduction semantics uses a Labeled Transition System [19], where $L$ denotes the set of labels. $L$ includes the evaluation contexts along with $\tau$, which denotes the transition without output. The only expression that generates output is the (breakpoint) expression. It displays the current evaluation context.

---

$^3$Following Barendregt [2], we assume syntactic $\alpha$-equivalence to sidestep the problem of capture in substitution. We further use this equivalence to guarantee that no two lexically bound identifiers share a name.
\[
\begin{align*}
L, M, N &= n \mid 'x \mid \text{true} \mid \text{false} \mid (\text{if } M M M) \mid (\text{cons } M M) \mid (\text{car } M) \\
&\quad \mid (\text{cdr } M) \mid \text{null} \mid (M M) \mid (P M M) \mid (\text{lambda } (x) M) \\
&\quad \mid x \mid (\text{breakpoint}) \\
P &= + \mid \text{eq}?
\end{align*}
\]

\[
\begin{align*}
V, W &= n \mid \text{true} \mid \text{false} \mid 'x \mid (\text{cons } V V) \mid x \mid \text{null} \mid (\text{lambda } (x) M) \\
E &= (\text{cons } E M) \mid (\text{cons } V E) \mid (\text{if } E M M) \mid (\text{car } E) \mid (\text{cdr } E) \\
&\quad \mid (E M) \mid (V E) \mid (P E M) \mid (P V E) \mid []
\end{align*}
\]

\[
\begin{align*}
\mathcal{L} &= E \cup \{\tau\} \\
\text{final states} &= V \cup \{\text{error}\} \\
&\quad x \in \text{variables} \quad n \in \text{numbers}
\end{align*}
\]

\[
\begin{align*}
E[(V W)] &\xrightarrow{\tau} \begin{cases} 
E[[W/x]M] & \text{if } V = (\text{lambda } (x) M) \\
\text{error} & \text{otherwise}
\end{cases} \\
E[(+ V W)] &\xrightarrow{\tau} \begin{cases} 
E[V + W] & \text{if } V \text{ and } W \text{ are numbers} \\
\text{error} & \text{otherwise}
\end{cases} \\
E[(\text{eq}? V W)] &\xrightarrow{\tau} \begin{cases} 
E[\text{true}] & \text{if } V \text{ and } W \text{ are the same symbol} \\
E[\text{false}] & \text{if } V \text{ and } W \text{ are different symbols} \\
\text{error} & \text{otherwise}
\end{cases} \\
E[(\text{car } V)] &\xrightarrow{\tau} \begin{cases} 
E[W] & \text{if } V = (\text{cons } W Z) \\
\text{error} & \text{otherwise}
\end{cases} \\
E[(\text{cdr } V)] &\xrightarrow{\tau} \begin{cases} 
E[Z] & \text{if } V = (\text{cons } W Z) \\
\text{error} & \text{otherwise}
\end{cases} \\
E[(\text{if } V M N)] &\xrightarrow{\tau} \begin{cases} 
E[M] & \text{if } V = \text{true} \\
E[N] & \text{if } V = \text{false} \\
\text{error} & \text{otherwise}
\end{cases}
\]

\[
E[(\text{breakpoint})] \xrightarrow{E} E[13]
\]

Figure 3.2 : Grammar and Reduction rules for the source language
\[ M_t = n \mid 'x \mid \text{true} \mid \text{false} \mid (\text{if} \ M_t \ M_t \ M_t) \mid \text{cons} \ M_t \ M_t \mid (\text{car} \ M_t) \mid (\text{cdr} \ M_t) \mid \text{null} \mid (M_t \ M_t) \mid (\text{P} \ M_t \ M_t) \mid (\text{lambda} \ (x) \ M_t) \mid x \mid (\text{w-c-m} \ V \ M_t) \mid (\text{c-c-m}) \mid (\text{output} \ M_t) \]

\[ E_t = (\text{w-c-m} \ V \ F_t) \mid F_t \]

\[ F_t = (\text{cons} \ E_t \ M_t) \mid (\text{cons} \ V \ E_t) \mid (\text{if} \ E_t \ M_t \ M_t) \mid (\text{car} \ E_t) \mid (\text{cdr} \ E_t) \mid (E_t \ M_t) \mid (V \ E_t) \mid (P \ E_t \ M_t) \mid (P \ V \ E_t) \mid (\text{output} \ E_t) \mid [] \]

\[ \mathcal{L}_t = V \cup \{\tau\} \]

The function \( \rightarrow_t \) extends \( \rightarrow \) with the following rules:

\[ E_t[(\text{w-c-m} \ V \ (\text{w-c-m} \ W \ M))] \rightarrow_t E_t[(\text{w-c-m} \ W \ M)] \quad (\text{where} \ E_t \neq E_t'[\text{w-c-m} \ Z \ []]) \]

\[ E_t[(\text{w-c-m} \ V \ W)] \rightarrow_t E_t[W] \quad (\text{where} \ E_t \neq E_t'[\text{w-c-m} \ V \ []]) \]

\[ E_t[(\text{c-c-m})] \rightarrow_t E_t[X[E_t]] \]

\[ E_t[(\text{output} \ V)] \rightarrow_t E_t[13] \]

where

\[ X[(\text{w-c-m} \ V \ E_t)] = (\text{cons} \ V \ X[E_t]) \]

\[ X[E_t] = X[E_t'] \quad \text{where} \ E_t = \begin{cases} (\text{cons} \ E_t' \ M) \\ (\text{cons} \ V \ E_t') \\ (\text{if} \ E_t' \ M \ M) \\ \ldots \end{cases} \]

\[ X[\[] = \text{null} \]

Figure 3.3: Extension of the source language \( M \) to the target language \( M_t \)
Since the instruction at the breakpoint must in fact be (breakpoint), this is equivalent to displaying the current program expression. The expression reduces to 13, an arbitrarily chosen value. When we write $\mapsto$ with no superscript, it indicates not that there is no output, but rather that the output is not pertinent.

The relation $\mapsto$ is a function. That is, an expression reduces to at most one other expression. This follows from the chain of observations that:

1. the set of values and the set of instructions are disjoint,

2. the set of values and the set of reducible expressions are therefore disjoint,

3. the instructions may not be decomposed except into the empty context and the instruction itself, and therefore that

4. an expression has at most one decomposition.

Multi-step evaluation $\mapsto^*$ is defined as the transitive, reflexive closure of the relation $\mapsto$. That is, we say that $M_0 \mapsto^* M_n$ if there exist $M_0, \ldots, M_n$ such that $M_i \mapsto M_{i+1}$ and $O \in L^* = l_0 l_1 \ldots l_{n-1}$.

The evaluation function $\text{eval}(M)$ is defined in the standard way:

$$\text{eval}(M) = \begin{cases} V & \text{if } M \mapsto V \\ \text{error} & \text{if } M \not\mapsto \text{ error} \end{cases}$$

For a reduction sequence $S = (M_0 \mapsto M_1 \mapsto \cdots \mapsto M_n)$, we define $\text{trace}(S)$ to be the sequence of non-empty outputs:

$$\text{trace}(S) = \begin{cases} () & \text{if } S = (M) \\ \text{trace}(M_1 \mapsto \cdots \mapsto M_n) & \text{if } S = (M_0 \overset{L}{\mapsto} M_1 \mapsto \cdots \mapsto M_n) \\ (E \cdot \text{trace}(M_1 \mapsto \cdots \mapsto M_n)) & \text{if } S = (M_0 \overset{E}{\mapsto} M_1 \mapsto \cdots \mapsto M_n) \end{cases}$$

The target language of our model is similar to the source language, except that it contains \textbf{w-c-m} and \textbf{c-c-m}, and an \textbf{output} instruction that simply displays a given
value. The grammar and reduction rules for this language are an adaptation of that of the source language. They appear in fig. 3.3.

The evaluation of the target language is designed to concatenate neighboring \textit{w-c-m}'s, which is critical for the preservation of tail-call optimizations in the source semantics. Frame overwriting is enforced by defining the set of evaluation contexts to prohibit immediately nested occurrences of \textit{w-c-m}-expressions. In particular, the set $E_t$ may include any kind of continuation, but its \textit{w-c-m} variant $F_t$ requires a subexpression that is not a \textit{w-c-m} expression.

Note also the restriction on the \textit{w-c-m} reductions that the enclosing context must not end with a \textit{w-c-m}. This avoids two ambiguities. The first occurs when two nested \textit{w-c-m} expressions occur with a value inside the second; should the outer mark be eliminated (since two occur consecutively), or should the inner mark be eliminated (because it contains a value directly)? In the second, three or more \textit{w-c-m} expressions occur consecutively; in an unrestricted semantics, we could choose either the outermost pair or the next-to-outermost pair to reduce.

For the target language, the set of labels is the set of values plus $\tau$. The \textbf{output} instruction is the only instruction that generates output.

The standard reduction relation $\rightarrow_t$ is a function. This follows from an argument similar to that for the source language. Multiple-step reduction is defined as in the source language by the transitive, reflexive closure of $\rightarrow_t$, written as $\leftrightarrow_t$. The target language’s evaluation function $\text{eval}_t$ and trace function $\text{trace}_t$ are adapted mutatis mutandis from their source language counterparts, with $\rightarrow$ and $\leftrightarrow$ replaced by $\rightarrow_t$ and $\leftrightarrow_t$.

Roughly speaking, \textbf{(breakpoint)} is a primitive breakpoint facility that displays the program’s execution state. The purpose of our model is to show that we can construct an elaboration function $\mathcal{A}$ from the source language to the target language that creates the same effect via a combination of continuation marks and a simple \textbf{output} expression.
\[ A[V] = \begin{cases} 
\text{(cons } A[V] A[W]) & \text{if } V = \text{(cons } V W) \\
\text{(lambda } (x) A[M]) & \text{if } V = \text{(lambda } (x) M) \\
V & \text{otherwise}
\end{cases} \]

\[ A[(\text{breakpoint})] = \text{(w-c-m (list 'break) (output (c-c-m)))} \]
\[ A[(V M)] = \text{(w-c-m (list 'appB A[V]) (A[V] A[M]))} \]
\[ A[(M N)] = \text{(w-c-m (list 'appA Q[N])} \]
\[ \quad (\text{(lambda } (F) \text{(w-c-m (list 'appB } F) (F A[N]))) A[M]) \]
\[ A[(\text{cons } V M)] = \text{(w-c-m (list 'consB A[V]) (cons } A[V] A[M])} \]
\[ A[(\text{cons } M N)] = \text{(w-c-m (list 'consA Q[N])} \]
\[ \quad (\text{(lambda } (F) \text{(w-c-m (list 'consB } F) (cons } F A[N]))) \]
\[ \quad A[M]) \]
\[ A[(\text{car } M)] = \text{(w-c-m (list 'car) (car } A[M])} \]
\[ A[(\text{cdr } M)] = \text{(w-c-m (list 'cdr) (cdr } A[M])} \]
\[ A[(P V M)] = \text{(w-c-m (list 'primB } P A[V]) (P A[V] A[M])} \]
\[ A[(P M N)] = \text{(w-c-m (list 'primA } P Q[N])} \]
\[ \quad (\text{(lambda } (F) \text{(w-c-m (list 'primB } P } F) (P F A[N]))) A[M]) \]

where
\[ Q[x] = \text{(list 'val } x) \]
\[ Q[\text{'x}] = \text{(list 'quote } \text{'x}) \]
\[ Q[M] = \begin{cases} 
\text{(list 'app } Q[M_1] Q[M_2]) & \text{if } M = (M_1 M_2) \\
\text{(list 'if } Q[M_1] Q[M_2] Q[M_3]) & \text{if } M = (\text{if } M_1 M_2 M_3) \\
& \ldots
\end{cases} \]

Figure 3.4: The annotating function, \( A : M \rightarrow M_t \)
\[ T[(\text{cons (list 'appA N) M)\} = (T[M] \overline{Q}[N]) \]
\[ T[(\text{cons (list 'appB V) M)\} = (V T[M]) \]
\[ T[(\text{cons (list 'if N L) M)\} = (\text{if } T[M] \overline{Q}[N] \overline{Q}[L]) \]
\[ T[(\text{cons (list 'consA N) M)\} = (\text{cons } T[M] \overline{Q}[N]) \]
\[ T[(\text{cons (list 'consB V) M)\} = (\text{cons } V T[M]) \]
\[ T[(\text{cons (list 'car) M)\} = (\text{car } T[M]) \]
\[ T[(\text{cons (list 'cdr) M)\} = (\text{cdr } T[M]) \]
\[ T[(\text{cons (list 'primA P N) M)\} = (P T[M] \overline{Q}[N]) \]
\[ T[(\text{cons (list 'primB P V) M)\} = (P V T[M]) \]
\[ T[(\text{cons 'break null})\} = [] \]

where

\[ \overline{Q}[(\text{list 'val A[V])\} = V \]
\[ \overline{Q}[(\text{list 'quote 'x})\} = 'x \]
\[ \overline{Q}[M] = \begin{cases} 
(M_1 \ M_2) & \text{if } M = (\text{list 'app } \overline{Q}[M_1] \overline{Q}[M_2]) \\
(\text{if } M_1 \ M_2 \ M_3) & \text{if } M = (\text{list 'if } \overline{Q}[M_1] \overline{Q}[M_2] \overline{Q}[M_3]) \\
& \ldots 
\end{cases} \]

Figure 3.5: The Translation function, \( T : V \rightarrow E \)

The elaboration function is defined in fig. 3.4. It assumes that the identifier \( F \) does not appear in the source program. It also relies upon a quoting function, \( Q \), which translates source terms to values representing them, except for the unusual treatment of variable names. These are not quoted, so that substitution occurs even within marks.
3.3 Properties of the Model

The translation from the breakpoint language to the language with continuation
marks preserves the behavior of all programs. In particular, terminating programs in
the source model are elaborated into terminating programs in the target language.
Programs that fail to converge are elaborated into programs that also fail to con-
verge. Finally, there is a function $T$, shown in fig. 3.5, mapping the values produced
by output in the target program to the corresponding evaluation contexts produced
by (breakpoint) expressions. We extend $T$ to sequences of values in a pointwise
fashion.

Theorem 1 (Elaboration Theorem) For any program in the source language $M$,
the following statements hold for the program $M_0$ and the elaborated program $N_0 = \mathcal{A}[M]_0$:

1. $\text{eval}(M_0) = V$ iff $\text{eval}_t(N_0) = \mathcal{A}[V]$.

2. $\text{eval}(M_0) = \text{error}$ iff $\text{eval}_t(N_0) = \text{error}$.

3. If $S = (M_0 \mapsto \cdots \mapsto M_n)$, there exists $S_t = (N_0 \mapsto_t \cdots \mapsto_t N_k)$ s.t. $\text{trace}[S] = T(\text{trace}[S_t])$

Proof Sketch: The relevant invariant of the elaboration is that every non-value
is wrapped in exactly one w-c-m, and values are not wrapped at all. The w-c-m
wrapping of an expression indicates what kind of expression it is, what stage
of evaluation it is in, and all subexpressions and values needed to reconstruct the
program expression.

The proof of the theorem is basically a simulation argument for the two pro-
gram evaluations. It is complicated by the fact that one step in the source program

\footnote{The list constructor is used in the remainder of the thesis as a syntactic abbreviation for a series of conses.}
corresponds to many steps in the elaborated program. The additional steps in the elaborated program are principally \texttt{w-c-m} reductions that patch up the invariant that the source program and the elaborated program are related by \( \mathcal{A} \). For details, see appendix A. \footnote{1}
Chapter 4

Stepping with Continuation Marks

The full stepper is built on top of the framework of chapter 3, and also comprises an elaborator and reconstructor. The elaborator transforms the user’s program into one containing breakpoints that correspond to the reduction steps of the source program. At runtime, the reconstructor translates the state of the evaluation into an expression from the information in the continuation marks.

In this chapter we develop the model of our stepper implementation and its correctness proof. Section 4.1 describes the elaborator and the reconstructor, and formalizes them. Section 4.2 presents the stepper theorem, which shows that the elaborator and reconstructor simulate algebraic reduction.

4.1 Elaboration and Reconstruction

The stepper’s elaborator extends the elaborator from chapter 3.2. Specifically, the full elaborator is the composition of a “front end” and a “back end.” In fact, the back end is simply the function $A$ of chapter 3.

The front end, $B$, translates a plain functional language into the source language of chapter 3. More specifically, it accepts expressions in $M_3$, which is the language $M$ without the (breakpoint) expression. Its purpose is to insert as many breakpoints as necessary so that the target program stops once for each reduction step according to the language’s semantics. Figs. 4.1 and 4.2 show the definition of $B$. The translation is syntax-directed according to the expression language. Since some expressions have subexpressions in non-tail positions, $B$ must elaborate these expressions so that a breakpoint is inserted to stop the execution after the evaluation of the subexpressions.
and before the evaluation of the expression itself. We use $I_0$, $I_1$, and $I_2$ as temporary variables that do not appear in the source program. In this and later figures we use the $\texttt{let\*}$ expression as syntactic shorthand.\footnote{The $\texttt{let\*}$ expression is roughly equivalent to the sequential $\texttt{let}$ of ML. It is used as syntactic shorthand for a corresponding set of applications like those in fig. 3.4.}

The full elaborator is the composition of $\mathcal{B}$ and $\mathcal{A}$. It takes terms in $M_s$ to terms in $M_t$, via a detour through $M$.

Like the elaborator, the reconstructor is based on the infrastructure of chapter 3. The execution of the target program produces a stream of output values. The function $\mathcal{T}$ of fig. 3.5 maps these values back to evaluation contexts of the intermediate language, that is, the source language of chapter 3. Since the instruction filling these contexts must be $\texttt{breakpoint}$, the reconstruction function $\mathcal{R}$ is defined simply as the inversion of the annotation applied to the context filled with $\texttt{breakpoint}$. In other words, $\mathcal{R}[E] = B^{-1}[E[\texttt{breakpoint}]]$.\footnote{Inspection of the definition of $\mathcal{B}$ demonstrates that it is invertible.} Like $\mathcal{T}$, $\mathcal{R}$ is extended pointwise to sequences of expressions.

The full reconstructor is the composition of $\mathcal{R}$ and $\mathcal{T}$. It takes terms in $E_t$ to terms in $M_s$.

### 4.2 Properties of the Stepper

To prove that the stepper works correctly, we must show that the elaborated program produces one piece of output per reduction step in the source semantics and that the output represents the entire program.

**Theorem 2 (Stepping Theorem)** For an evaluation sequence $S = (M_0 \rightarrow \cdots \rightarrow M_n)$, there exists an evaluation sequence $S_t = (\mathcal{A}[\mathcal{B}[M_0]] \rightarrow_t \cdots \rightarrow_t N_k)$ such that $S = \mathcal{R}[\mathcal{T}[\text{trace}[S_t]]]$. 
\[ B[V] = \begin{cases} 
\text{\texttt{(lambda (x) B[M])}} & \text{if } V = \text{\texttt{(lambda (x) M)}} \\
\text{\texttt{(cons B[M] B[N])}} & \text{if } V = \text{\texttt{(cons M N)}} \\
V & \text{otherwise}
\end{cases} \]

\[ B[(M \ N)] = (\text{\texttt{let*}} ([I_0 \ B[M]]) \\
[I_1 \ B[N]] \\
[I_2 \ \text{\texttt{(breakpoint)}]}) \\
(I_0 \ I_1)) \]

\[ B[(V \ N)] = (\text{\texttt{let*}} ([I_1 \ B[N]]) \\
[I_2 \ \text{\texttt{(breakpoint)}]}) \\
(V \ I_1)) \]

\[ B[(V \ U)] = (\text{\texttt{let*}} ([I_2 \ \text{\texttt{(breakpoint)}]}) \\
(V \ U)) \]

\[ B[(\text{if } M \ N \ L)] = (\text{\texttt{let*}} ([I_0 \ B[M]]) \\
[I_1 \ \text{\texttt{(breakpoint)}]}) \\
(\text{if } I_0 \ B[N] \ B[L])) \]

\[ B[(\text{if } V \ N \ L)] = (\text{\texttt{let*}} ([I_0 \ \text{\texttt{(breakpoint)}]}) \\
(\text{if } V \ B[N] \ B[L])) \]

Figure 4.1: The stepper's breakpoint-inserting function, \( B : M_s \to M \) (part 1)
\[ B[\text{car } M] = \text{let\* (} [I_0 \ B[M]] \ \\
\quad [I_1 \ (\text{breakpoint})] \ \\
\quad (\text{car } I_0)] \]

\[ B[\text{car } V] = \text{let\* (} [I_2 \ (\text{breakpoint})] \ \\
\quad (\text{car } V)] \]

\[ B[\text{cdr } M] = \text{let\* (} [I_0 \ B[M]] \ \\
\quad [I_1 \ (\text{breakpoint})] \ \\
\quad (\text{cdr } I_0)] \]

\[ B[\text{cdr } V] = \text{let\* (} [I_2 \ (\text{breakpoint})] \ \\
\quad (\text{cdr } V)] \]

\[ B[(P \ M \ N)] = \text{let\* (} [I_0 \ B[M]] \ \\
\quad [I_1 \ B[N]] \ \\
\quad [I_2 \ (\text{breakpoint})] \ \\
\quad (P \ I_0 \ I_1)] \]

\[ B[(P \ V \ N)] = \text{let\* (} [I_1 \ B[N]] \ \\
\quad [I_2 \ (\text{breakpoint})] \ \\
\quad (P \ V \ I_1)] \]

\[ B[(P \ V \ U)] = \text{let\* (} [I_2 \ (\text{breakpoint})] \ \\
\quad (P \ V \ U)] \]

Figure 4.2: The stepper’s breakpoint-inserting function, \( B : M_s \rightarrow M \) (part 2)
Proof Sketch: By the Elaboration theorem, it suffices to prove that, given a sequence $S$ as in the theorem statement, there exists $S_a = (B[M_0] \leftrightarrow \cdots \leftrightarrow N_{\nu})$ such that $S = R[\text{trace}_S[S_a]]$.

The proof again uses a simulation argument. Evaluation of the source program for one step and evaluation of the target program for either one or two steps maintains the invariant that the source program and the target program are related by $B$. For more details, see appendix A.

The key idea in our theorems is that the stepper's operation is independent of the intermediate state in the evaluation of the elaborated program. Instead, the elaborated program contains information in the marked continuations that suffices to reconstruct the source program from the output. The correctness theorem holds for any evaluator that properly implements the continuation-mark framework. That is, the stepper's correct operation is entirely orthogonal to the implementation strategy and optimizations of the evaluator; as long as that evaluator correctly implements the language with continuation marks, the stepper will work properly.
Chapter 5

The Pragmatics of a Stepper

The model of chapters 3 and 4 represents the kernel of our stepper. Making this model workable, however, requires several adjustments. For example, continuation marks should contain source pointers, rather than raw text. Additionally, the stepper must deal with unannotated library code and Scheme macros. The stepper as implemented also captures the result of each reduction, in addition to the reduction’s antecedent. In other words, it provides both “before” and “after” snapshots for every step. In the following sections, we explain the major differences between the model and our implementation.

5.1 One-Step Annotation

In our model, the annotator and the reconstructor are both defined as the composition of two functions. This allows the stepper theorem’s proof to be built upon the debugger theorem’s proof, but it makes for cumbersome code. In the stepper’s implementation, annotation is performed in one pass. Furthermore, the need for multiple marks wrapping a single expression (as in the if expression or application) is eliminated by using assignments to update both the mark and the program state simultaneously.

5.2 The Content of Marks

The DrScheme annotator does not directly operate upon the source program. Instead, the initial parsing is performed by M3, DrScheme’s source-correlating preprocessor [17]. The input to the annotator is therefore a structured parse tree with
source pointers, and the annotator stores references to this source code in the inserted marks. This method ensures that all marks share references to the same source tree, decreasing memory consumption.

Unfortunately, the use of source pointers induces another significant difference between model and implementation. The model relies on variable substitution within quoted expressions to track variable bindings. With static source pointers, this is not possible. These source trees therefore contain free variable references. Storing the values of all free variables would be wasteful. The annotator therefore stores only the values of "needed" free variables in the annotated expression.

The calculation of "needed" variables proceeds as follows: an expression's mark needs the value of a variable if and only if it appears free in the expression, and it is in tail position with respect to that expression [12]. So, in the top-level program \texttt{(lambda (x y z) (if x y z))}, the mark wrapping the \texttt{if} expression would contain the bindings for \texttt{x}, \texttt{y}, and \texttt{z}. The mark wrapping the \texttt{x}, however, would not need to capture the binding for \texttt{x}, because this expression can only occur during the evaluation of the \texttt{if} expression. The references \texttt{y} and \texttt{z}, however, must contain the values of their respective bindings; these references are in tail position with respect to the enclosing \texttt{if}, and continuation marks wrapping the variable references will therefore replace ones that wrap the \texttt{if} expression.

5.3 Opaque Values

Many Scheme values are opaque. Since the stepper is an ordinary program and has no privileged access to compiler or run-time data, it has no way to inspect or mutate these values. Procedure closures, structures, and MzScheme's classes, objects, and units are all opaque. In order to display these values, the stepper must have access to their source text. Rather than compromise the division between stepper and compiler or disallow access entirely, the stepper elaborates all language forms that create opaque values so that these items' source texts appear in a (weak) hash table whose keys
are the values themselves. At reconstruction time, the stored records may then be retrieved for display.

5.4 Unannotated Code

Unannotated source code presents a problem for the reconstructor. By definition, this code places no (relevant) marks upon its continuations, and thus the reconstructor cannot determine how it operates. In fact, its very existence is difficult to detect, since unmarked frames are not represented in the continuation mark sets.

Nevertheless, unannotated procedures may be detected in the call chain. When a normally annotated procedure is applied, the continuation mark wrapping the body of that procedure immediately replaces the mark that wrapped the application, since the body of that procedure is in tail position with respect to the application. Unannotated procedures, however, have no such mark. Therefore, when the reconstructor encounters a mark for an application whose subexpressions have been completely evaluated, it must conclude that this application called an unannotated procedure (which is still being evaluated). This procedure's context is shown in the reconstruction as the unknown context, displayed as (...) [], where the hole is filled with the previously reconstructed text. Figure 5.1 shows an example that occurs in the evaluation of

\[(\text{car} (\text{map square (list 2 3 4)))}\]

where map refers to an unannotated library procedure.

When an unannotated procedure is called, the mark attached to the procedure's application is not replaced. The persistence of such a mark beyond its tail call seems to violate proper tail-call behavior. A simple argument shows this is not the case. In order to construct an example where the number of such continuation marks grows without bound, there must be at least one non-tail call between the call to unannotated code and a subsequent call into unannotated code. If there were not, then the latter's mark would replace the former's, and the number of marks would not
grow without bound. But if this is true, then Scheme's tail-call mandate places no restriction upon the implementation of the stepper, for if this example contains an unbounded number of non-tail calls, an implementation is under no requirement to operate in bounded space.

### 5.5 Scheme Macros and Source Correlation

DrScheme compiles user code in several stages. The first of these is a reader and parser. Next comes a program elaborator, which maps programs from a rich source language to a simpler core language. Thirdly, these core forms are annotated with continuation-marks. Finally, these core forms are reduced to a tree-structured intermediate code that is directly evaluated.

Since the annotation occurs after the elaboration, it is required to handle only a small set of language forms. This simplifies the design of the annotator. For instance, Scheme's cond form is reduced to a nested series of ifs.

The penalty for this arrangement arises in the reconstructor. The reconstructed
source text should appear similar to the original source, and should therefore present this series of if expressions as a cond. The reconstructor must therefore determine which if expressions came originally from a cond, and translate them back into their original forms.

This is possible in DrScheme because of the source correlation of M3. Each elaborated term contains a reference to source text. The annotator insures that this reference appears in the expression’s runtime mark, and will therefore be delivered to the reconstructor. The reconstructor can then group adjacent ifs that come from the same cond into a single displayed cond expression.

Note that this mechanism handles partially evaluated cond expressions automatically; as the branches of the cond are discarded, the reconstructed cond loses these branches as well. The reconstructor’s back-translation of cond expressions therefore constitutes an implicit semantics for cond, given the semantics for if. In fact, this result applies to any macro; an elaboration, a core semantics, and a reconstruction taken together constitute a reduction semantics for the source form.

5.6 Post-reduction Breakpoints

In DrScheme’s stepper, each reduction step is displayed in two pieces: the redex and the contractum. The model shows how to add breakpoints that display the redex. In order to highlight the contractum, the annotator must insert additional breakpoints.

To add post-reduction breakpoints, we must categorize the reductions that can occur and where their resulting expressions or values can appear. There are two kinds of reductions: those that replace the redex with a source-text subexpression, and those that replace the redex with a newly generated value. The first case comprises applications of user-defined procedures and if expressions. The second case includes the application of non-higher-order primitives such as +. There are also expressions that do not fall nicely into either category. These include the applications of higher-order primitives—most notably map and fold—and non-primitive unannotated code.
To understand the first case, consider an if expression:

\[(\text{if} \ V \ M \ N)\]

The contractum for this redex is either \(M\) or \(N\), and conversely \(M\) and \(N\) are evaluated only as a result of this reduction. The annotator may therefore place a “result” breakpoint before these two subexpressions. The reconstructed state at these breakpoints forms the “after” step for the if expression. In fact, Scheme has the property that the set of expressions that \(may\) occur in such an “after” position is exactly the set of expressions that \(only\) occur in an “after” position. The “tail position” predicate identifies these positions. Annotation of an expression therefore depends not only on the expression being annotated but also on its parent.

The second case is simpler. The annotator may capture the value returned by a primitive application by wrapping the application in a binding construct that captures its return value and displays it in a breakpoint. This may transform a tail call into a non-tail call. For “leaf” procedures such as \(+\) and \(eq?\), however, the added memory overhead is bounded by a constant, namely the size of a single stack frame.

Some higher-order primitives present a problem, because they do not fit neatly into either category: the result of such a reduction is not a subexpression of the program, and these primitives do not act atomically. In practice, the simplest course of action is to replace these primitives systematically with a library of annotated procedures. The additional overhead is minimal.

Unannotated code presents a larger problem. The stepper has no way to infer a reduction semantics for an application of an unannotated procedure, or even whether such a procedure is higher-order. In these cases, the whole notion of an “after” step is muddy at best. On the one hand, if the unannotated code makes a tail call to a user procedure, then wrapping the call to the unannotated procedure with a return-value capture (as for simple primitives) is a mistake, because it destroys tail-call optimization. On the other hand, if the unannotated procedure does not make such a tail call, it should be so wrapped. The best solution to this problem is simply
not to attempt "after" steps for unannotated code. The secondary consequence of this decision is that if an "after" breakpoint occurs immediately inside unannotated code—i.e., unannotated code calls back into annotated code—it should simply be ignored.
Chapter 6

Related Work

The idea of elaborating a program in order to observe its behavior is a familiar one. Early systems included BUGTRAN [7] and EXDAMS [1] for FORTRAN. More recent applications of this technique to higher-order languages include Tolmach’s smld [25], Kellomaki’s PSD [15], and several projects in the lazy FP community [13, 21, 23, 24]. None of these, however, addressed the correctness of the tool — not only that the transformation preserves the meaning of the program, but also that the information divulged by the elaborated program matches the intended purpose.

Indeed, work on modeling the action of programming environment tools is sparse. Bernstein and Stark [3] put forward the idea of specifying the semantics of a debugger. That is, they specify the actions of the debugger with respect to a low-level machine. We extend this work to show that the tool preserves the semantics and also performs the expected computation.

Kishon, Hudak, and Consel [16] study a more general idea than Bernstein and Stark. They describe a theoretical framework for extending the semantics of a language to include execution monitors. Their work guarantees the preservation of the source language’s semantics. Our work extends this (albeit with a loss of generality) with a proof that the information output by the tool is sufficient to reconstruct a source expression.

Bertot [4] describes a semantic framework for relating an intermediate state in a reduction sequence to the original program. Put differently, he describes the semantic foundation for source tracking. In contrast, we exploit a practical implementation of source tracking by Shriram Krishnamurthi [17] for our implementation of the step-
per. Bertot's work does not verify a stepper but simply assumes that the language evaluator is a stepper.
Chapter 7

Conclusion

Our thesis presents a high-level model of an algebraic stepper for a functional language. Roughly speaking, the model extends a conventional reduction semantics with a high-level form of weak continuation manipulations. The new constructs represent the essence of the stepper's compiler and run-time actions. They allow programs to mark continuations with values and to observe the mark values, without any observable effect on the evaluation. Using the model, we can prove that the stepper adds enough information to the program so that it can stop for every reduction step. At each stop, furthermore, the source information in the continuation suffices for a translation of the execution state into source syntax—no matter how the back end represents code and continuations.

Because the model is formulated at a high level of abstraction, the model and the proofs are robust. First, the model should accommodate programming environment tools such as debuggers and profilers that need to associate information about the program with the continuation. After all, marking continuations and observing marks are two actions that are used in the run-time environment of monitoring tools; otherwise, these tools are simply aware of the representations of values, environments, heaps, and other run-time structures. Indeed, we are experimenting at this moment with an implementation of a conventional debugger directly based on the continuation mark mechanism. Performance penalties for the debugger prototype run to a factor of about four.

Second, the proof applies to all implementations of steppers. Using conventional machine derivation techniques from the literature [5, 6], one can translate the model
to stack and heap machines, conventional machines (such as Landin's SECD [18] machine) or tail-optimizing machines (such as Felleisen's CE(S)K machine). In each case, minor modifications of the adequacy proofs for the transformations show that the refined stepper is still correct.

The model of this thesis covers only the functional kernel of Scheme. Using the extended reduction semantics of Felleisen and Hieb [6], the model scales to full Scheme without much ado. We also believe that we could build an algebraic stepper for Java-like languages, using the model of Flatt et al. [11]. In contrast, it is an open question how to accommodate the GUI (callback) and concurrency facilities of our Scheme implementation [10], both in practice and in theory. We leave this topic for future research.
Bibliography


Appendix A

Detailed Proof Sketches

To prove the thesis's theorems, we need a number of lemmas and intermediate results. In this appendix we outline the required lemmas, the order in which they arise, and the ways in which they depend on each other.

A note about the proofs: the metavariable used to refer to a program term carries implicit information. That is, if a term is described as $I$, this indicates that the term is a member of the set of instructions. Likewise, if we say that $M$ is of the form $E[I]$, we mean that $M$ may be decomposed into a valid evaluation context $E$ and a valid instruction $I$. Also, position subscripts are occasionally used to identify terms implicitly. That is, if a term has the form $(M_1M_2)$, and $M_1$ is shown to be a value, the label $V_1$ will denote the same term that $M_1$ does. Finally, the subscript $t$ that is used to distinguish terms of the source language and terms of the target language is frequently omitted. It should be clear which metavariables refer to terms of the source language, and which refer to those of the target language.

A.1 The Debugger Theorem

If the debugger theorem is to make any sense, the languages it uses must be well-defined. That is, for any legal program expression that is not a value, there must be a unique decomposition into an evaluation context and an instruction. We examine each of the languages in turn, and then turn our attention to the debugger theorem itself.
A.1.1 Well-Definition of $M$

The language $M$ is a standard formulation of a simple functional language, and it is reasonably clear that it is well-defined. In particular, a simple structural induction upon the program $M$ demonstrates all needed properties. The proof considers each kind of expression in turn, and concludes that it is either a value, and can only be a value, or that it is part of an evaluation context, and again that there is only one possible choice.

Our first lemma, the disjoint V/I lemma, concludes that the sets of values and instructions are disjoint. The sets are syntactically distinct, so this is trivial.

The second lemma, the disjoint V/E lemma, shows that values and expressions of the form $E[I]$ are disjoint. The base case of this lemma refers to the disjoint V/I lemma. It proceeds by a relatively trivial structural induction.

The proof of the existence and uniqueness of the decomposition of non-value expressions follows from these lemmas. Induction yields a decomposition of the appropriate subexpression, and the final context is the composition of the one-layer evaluation context associated with the top level of the expression with the context obtained via the induction.

A.1.2 Well-definition of $M_t$

This proof is similar to the proof for $M$, and features two parallel lemmas that establish that the set of values and the set of reducible expressions are disjoint. The proof is complicated because $M_t$ features some unusual restrictions on the set of evaluation contexts and the choice of instructions. In particular, an evaluation context may not contain two consecutive \texttt{w-c-m} forms (to force these into the instruction position), and the instructions are restricted to avoid ambiguity in terms that contain three or more consecutive \texttt{w-c-m} forms.
A.1.3 The correspondence between \( M \) and \( A[M] \)

The goal of the proof is to show that the evaluation of a program in the source language and a program in the target language (obtained by annotating the source term) proceed in sync. This consists of two lemmas. First, the result of annotating the one-step evaluation of a source term is the same as the result of taking some finite number of steps in the target language, using the annotation of the original source term. Figure A.1 illustrates this relation, and the way in which it may be used as the basis of an induction that "chases the arrows." We use a prefix notation for the reduction and annotation relations; \( R[M] \) describes the result of taking one step in the reduction semantics of the source language. \( S \) is used for reductions in the target language, and \( A \) refers to the annotation (as it does in the text of the thesis).

The principal task, then, is to prove that \( A[R[M]] = S^n[A[M]] \), for each program term \( M \). The exponent \( n \) is positive, and finite (though determined by \( M \)). Program errors crop up periodically during this proof, and in particular the prior statement needs amendment; we leave out all cases involving errors, and simply prove as a side condition that a program \( M \) in the source language that produces an error has an annotation that also does so in one step in the target language. This proceeds by a straightforward structural induction on the contexts, and frees us from the
consideration of program errors in the remainder of the proof.

Our first lemma, the A/D lemma, concerns the annotation of a decomposition. First, we wish to extend $A$ to accept evaluation contexts. To accomplish this, we define $A[]$ to be $[]$, and holes in a term are considered to match the "$M$" patterns in the definition of $A$.

The lemma, then, states that $A[E[M]] = A[E][A[M]]$, as long as $M$ is not a value. Also, the resulting context $A[E]$ is a valid context in the target language. As a special case of this, since the set of instructions is disjoint from the set of values, $A[E[I]] = A[E][A[I]]$. Note, however, that $A[I]$ is never a valid instruction.

This proof follows from a structural induction on $E$, inspecting the action of $A$ on each element of context. The only interesting thing about this proof is that annotation behaves differently depending on which subexpressions are values, so we must be somewhat careful.

Next, it turns out that (cons . . .) expressions are the source of some difficulty. In particular, any cons that contains a non-value, directly or indirectly, must be annotated with a w-c-m. However, when some expression deeply nested within a large cons tree is reduced to a value, these w-c-m forms must disappear. And they do, but there is a reduction for the removal of each one. For this reason, we must isolate the cons-form evaluation contexts that lie immediately outside the instruction.

To capture this notion, we introduce two new kinds of evaluation context: $G$ and $H$. They are defined as follows:

$$G = (\text{if } H \ M \ M)(\text{car } H)(\text{cdr } H)(H \ M)((V \ H)((P \ H \ M)((P \ V \ H)$$

$$H = (\text{cons } H \ M)(\text{cons } V \ H)[]$$

The set $H$ comprises the evaluation contexts consisting entirely of cons forms; the set $G$ includes all contexts in $H$ enclosed in a single non-cons context. Since both $G$ and $H$ are subsets of $E$, any property that holds for all members of $E$ also holds for members of $G$ and $H$. 
Once we have this notion, we demonstrate that any evaluation context $E$ is either of the form $H$ (consists entirely of cons or is empty), or can be decomposed uniquely into $E_1[G]$. A simple structural recursion on $E$ is all that is needed to show this.

For the rest of the proof, we show successively stronger proofs, each building upon the last. We begin with instructions, pass through $G$ and $H$, and wind up at a general $M$.

First, we show (the instruction lemma) that for any instruction $I$, $A[R[I]] = S^n[A[I]]$. Remember that we've already taken care of the error conditions, so we may assume that an error does not occur in the evaluation of $I$. This proof proceeds by case analysis of $I$. No induction is required.

The proof relies on two small lemmas. The first states that values are annotated into values. The second states that non-values are wrapped in w-c-m expressions. The rest of the proof follows, with a modicum of algebraic manipulation.

Second, we extend this result to expressions of the form $H[I]$. That is, $A[R[M]] = S^n[A[M]]$. This proof proceeds by induction on the structure of $H$. For the flavor of the proof as a whole, one case is shown here.

Suppose $H$ is of the form (cons $H_1$ $M_2$), and further suppose that both $R[[H[I]]]$ and $M_2$ are values. Then:

\[
A[R[M]] = A[R[(\text{cons } H_1[I] M_2)]] \\
= A[(\text{cons } R[H_1[I]] M_2)] \\
= (\text{cons } A[R[H_1[I]]] A[M_2]) \text{ (since both are values)} \\
= S^2[(\text{w-c-m (list 'consB } A[R[H_1[I]]]) (\text{w-c-m (list 'consB } A[R[H_1[I]]]) (\text{w-c-m (list 'consB } A[R[H_1[I]]]) A[M_2]))] \\
= S^3[(\text{w-c-m (list 'consA } Q[M_2]) ((\text{lambda } (F) (\text{w-c-m (list 'consB } F) (\text{cons } F A[M_2]))) A[R[H_1[I]]]])) \\
= S^3[(\text{w-c-m (list 'consA } Q[M_2]) ((\text{lambda } (F) (\text{w-c-m (list 'consB } F) (\text{cons } F A[M_2]))) (\text{cons } A[R[H_1[I]]]) A[M_2])]]] \text{ (by induction)}
\]
\[ S^{3+n}[ (\text{w-c-m} (\text{list 'consA Q[M_2]})) \\
    ((\lambda (F) (\text{w-c-m} (\text{list 'consB F}) (\text{cons F A[M_2]}))) \\
    A[H_1[I]])]) \\
= S^{3+n}[A[(\text{cons H_1[I]} M_2)]]) \\
= S^{3+n}[A[M]] \]

Other cases look similar.

In the third step, we must extend the proof to contexts in \( G \). Since a \( G \) is formed from an \( H \) by the addition of a single context frame, no induction is necessary. The algebraic manipulations required to complete this stage of the proof are similar to those required in the prior one. However, we also wish to show that the result of evaluating \( G[I] \) — that is, \( R[G[I]] \) — cannot be a value. This is easy to see, but necessary for the final step.

Finally, we can put all the pieces together; because \( R[G[I]] \) is not a value, the annotation of any context outside of \( G \) is unaffected by the step. A simple algebraic shuffle shows that the action of the original program and the annotated program match in the way stated earlier.

This is nearly enough, but we must also show that when a breakpoint occurs in the original program, the annotated program produces a value on the output that goes by \( T \) to the original evaluation context. This proof is complicated slightly by the fact that the information must be in the form of a program value when output by the annotated program, and the most straightforward induction plan fails. However, by building the induction in two steps, and drawing the correspondence first between
the evaluation context of the original program and the list output by the annotated program, and then between that list and the context again, we achieve the desired result.

A.2 The Stepper Theorem

The stepper theorem is similar to the debugger theorem in its proof. Again, our basic task is to show that annotation commutes with reduction. In this case, our annotation is written as $B$. The source language and target language both use the semantic function $R$.

The peculiarity of $\textbf{cons}$ crops up again, albeit in a slightly different way. In the stepper theorem, the annotation of $\textbf{cons}$ expressions forced the evaluation of a set of extraneous reductions; in the debugger theorem, no such reductions occur, but we can only obtain a guarantee of "non-value"-hood for expressions when we move outward to a non-$\textbf{cons}$ context.

Also, the mechanical drudgery of the proof increases slightly because forms with non-tail subexpressions have one more possible annotation. In the debugger theorem, expressions of the form $(V M)$ were annotated in the same way as those of the form $(V V)$. In the stepper theorem, this is not the case.

Our first step is to show that things work for the instructions; that is, that $B[R[I]] = R^n[B[I]]$. Furthermore, we show that in its first step, the annotated instruction $A[I]$ evaluates a breakpoint.
At this point, we need the invertible-B lemma, which states simply that $B$ is invertible. This proof employs structural induction on the language $M$, but it is also fairly clear; the annotation $B$ adds information that can easily be stripped away.

We may extend this result to expressions of the form $H[I]$ by a simple recursion on $H$; the reasoning is simpler in this case, because no annotations are performed directly upon the cons frames.

Next, we must consider the one-layer evaluation contexts $G$, and show that if an expression is of the form $G[I]$, again annotation commutes with evaluation, and again (like the debugger theorem) that the result $\mathcal{R}[G[I]]$ of reduction is not a value, and hence will not affect the annotation of enclosing contexts.

Once we have this in hand, all enclosing contexts may be folded in for free; since the evaluation of the inner expression takes a non-value to a non-value, the outer layers remain unchanged.