Modeling an Algebraic Stepper

John Clements, Matthew Flatt⋆, and Matthias Felleisen⋆⋆

Department of Computer Science
Rice University
6100 Main St.
Houston, TX 77005-1892

Abstract. Programmers rely on the correctness of the tools in their programming environments. In the past, semanticists have studied the correctness of compilers and compiler analyses, which are the most important tools. In this paper, we make the case that other tools, such as debuggers and steppers, deserve semantic models, too, and that using these models can help in developing these tools.

Our concrete starting point is the algebraic stepper in DrScheme, our Scheme programming environment. The algebraic stepper explains a Scheme computation in terms of an algebraic rewriting of the program text. A program is rewritten until it is in a canonical form (if it has one). The canonical form is the final result.

The stepper operates within the existing evaluator, by placing breakpoints and by reconstructing source expressions from source information placed on the stack. This approach raises two questions. First, do the run-time breakpoints correspond to the steps of the reduction semantics? Second, does the debugging mechanism insert enough information to reconstruct source expressions?

To answer these questions, 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. We expect the approach to apply to other semantics-based tools.

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, 21]

⋆ Current Address: School of Computing, 50 S. Central Campus Dr., Rm. 3190, University of Utah, SLC, UT 84112-9205

⋆⋆ Work partially supported by National Science Foundation grants CCR-9619756, CDA-9713032, and CCR-9708957, and a state of Texas ATP grant.

and more general algebraic laws for the rest of the language [6]. An algebraic stepper is particularly helpful for teaching; 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.

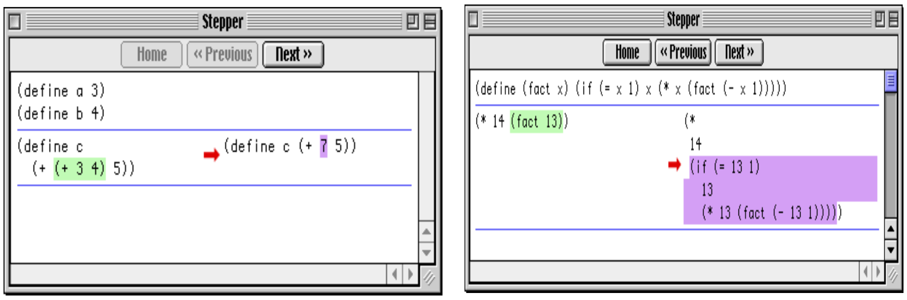
To prove that an algebraic stepper has these properties, we must model it at a reasonably high level so that the proof details do not become overwhelming.

In this paper, 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 section we briefly demonstrate our stepper. The third section 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 section 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 paper concludes with a brief discussion of related and future work.

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:

$$\begin{array}{l}
\boxed{\text{(fact 3)}} \\
= \text{(if } \boxed{\text{(= 3 1)}} \text{ 1 (}* \text{ (fact (}- \text{ 3 1)) 3))} \\
= \boxed{\text{(if false 1 (}* \text{ (fact (}- \text{ 3 1)) 3))}} \\
= \text{(}* \text{ (fact} \boxed{\text{(}- \text{ 3 1)}} \text{) 3)} \\
\ldots
\end{array}$$



An arithmetic reduction — A procedure application

Fig. 1. The stepper in action

Each step represents the entire program. The boxed subexpression is the standard redex. The sequence illustrates the reduction of function applications (β_v), primitive applications (δ_v), and **if** expressions.

DrScheme implements a more sophisticated version of the same idea. The two screen dumps of fig. 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. 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 paper and focus on the functional core.

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. Furthermore, it is possible to derive all kinds of low-level machines from a high-level semantics [5, 13]. 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 section 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 section to a model of a stepper for the core of a functional language.

Section 3.1 presents *continuation marks*, the central idea of our model. Section 3.2 formalizes this description as an extension of the functional reduction semantics of Scheme. Section 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

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-mark frames are transparent to the evaluation. When control returns to such a frame, the mark frame is removed. When a program adds a mark to a continuation that is already marked (that is, when two marks appear in succession), the new mark replaces the old one. This provision preserves tail-optimizations for all derived implementations. Not all machines are tail-optimizing, e.g., the original SECD machine [17], but due to this provision our framework works for both classes of machines.

In addition to the usual constructs of a functional language, our model contains two new continuation operations:

(**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.

The two programs in fig. 2 illustrate how an elaborator may instrument a factorial function with these constructs.[1] Both definitions implement a factorial

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

```
(define (fact n)                          (define (fact-tr n)
  (letrec                                   (letrec
    ([! (lambda (n)                           ([! (lambda (n a)
          (if (= n 0)                               (if (= n 0)
              (begin                                    (begin
                (display (c-c-m))                         (display (c-c-m))
                (newline)                                 (newline)
                1)                                        a)
              (w-c-m n                                  (w-c-m n
                (* n (! (- n 1))))))])                    (! (- n 1) (* a n)))))])
    (! n)))                                   (! n 1)))

> (fact 4)                                > (fact 4)

[(1 2 3 4)] ; displayed output            [(1)] ; displayed output

24                                        24
```

Fig. 2. The relationship between continuation-marks and tail-recursion

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.[2]

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 λ-calculus. The source represents the surface syntax, while the target is a representation of the intermediate compiler language. The source language of this section supports a primitive inspection facility in the form of a (**breakpoint**) expression. The target language has instead a continuation mark mechanism.

[2] This elision of continuation marks is the principal difference between our device and that of Moreau's dynamic bindings [19]. If we were to use dynamic bindings to preserve runtime information, the resulting programs would lose tail-call optimizations, which are critical in a functional world.

$$
\begin{aligned}
L, M, N &= n \mid \text{'}x \mid \textbf{true} \mid \textbf{false} \mid (\textbf{if}\ M\ M\ M) \mid (\textbf{cons}\ M\ M) \mid (\textbf{car}\ M) \mid \\
&\quad (\textbf{cdr}\ M) \mid \textbf{null} \mid (M\ M) \mid (P\ M\ M) \mid (\textbf{lambda}\ (x)\ M) \mid \\
&\quad x \mid (\textbf{breakpoint}) \\
P &= + \mid \mathit{eq?} \\
V, W &= n \mid \textbf{true} \mid \textbf{false} \mid \text{'}x \mid (\textbf{cons}\ V\ V) \mid x \mid \textbf{null} \mid (\textbf{lambda}\ (x)\ M) \\
E &= (\textbf{cons}\ E\ M) \mid (\textbf{cons}\ V\ E) \mid (\textbf{if}\ E\ M\ M) \mid (\textbf{car}\ E) \mid (\textbf{cdr}\ E) \mid \\
&\quad (E\ M) \mid (V\ E) \mid (P\ E\ M) \mid (P\ V\ E) \mid [] \\
\mathcal{L} &= E \cup \{\tau\} \\
\text{final states} &= V \cup \{\text{error}\} \qquad x \in \text{variables} \qquad n \in \text{numbers}
\end{aligned}
$$

$$
\begin{aligned}
E[(V\ W)] &\overset{\tau}{\mapsto} \begin{cases} E[[W/x]M] \text{ if } V = (\textbf{lambda}\ (x)\ M) \\ \text{error} \quad \text{otherwise} \end{cases} \\
E[(+\ V\ W)] &\overset{\tau}{\mapsto} \begin{cases} E[V + W] \text{ if } V \text{ and } W \text{ are numbers} \\ \text{error} \quad \text{otherwise} \end{cases} \\
E[(\mathit{eq?}\ V\ W)] &\overset{\tau}{\mapsto} \begin{cases} E[\textbf{true}] \text{ if } V \text{ and } W \text{ are the same symbol} \\ E[\textbf{false}] \text{ if } V \text{ and } W \text{ are different symbols} \\ \text{error} \quad \text{otherwise} \end{cases} \\
E[(\textbf{car}\ V)] &\overset{\tau}{\mapsto} \begin{cases} E[W] \text{ if } V = (\textbf{cons}\ W\ Z) \\ \text{error otherwise} \end{cases} \\
E[(\textbf{cdr}\ V)] &\overset{\tau}{\mapsto} \begin{cases} E[Z] \text{ if } V = (\textbf{cons}\ W\ Z) \\ \text{error otherwise} \end{cases} \\
E[(\textbf{if}\ V\ M\ N)] &\overset{\tau}{\mapsto} \begin{cases} E[M] \text{ if } V = \textbf{true} \\ E[N] \text{ if } V = \textbf{false} \\ \text{error otherwise} \end{cases} \\
E[(\textbf{breakpoint})] &\overset{E}{\mapsto} E[13]
\end{aligned}
$$

Fig. 3. Grammar and Reduction rules for the source language

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. 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

[3] Following Barendregt [2], we assume syntactic α-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.

$$
\begin{aligned}
M_t &= n \mid {}'x \mid \textbf{true} \mid \textbf{false} \mid (\textbf{if}\ M_t\ M_t\ M_t) \mid \textbf{cons}\ M_t\ M_t) \mid (\textbf{car}\ M_t) \mid \\
&\quad (\textbf{cdr}\ M_t) \mid \textbf{null} \mid (M_t\ M_t) \mid (P\ M_t\ M_t) \mid (\textbf{lambda}\ (x)\ M_t) \mid \\
&\quad x \mid (\textbf{w-c-m}\ M_t\ M_t) \mid (\textbf{c-c-m}) \mid (\textbf{output}\ M_t) \\
E_t &= (\textbf{w-c-m}\ V\ F_t) \mid F_t \\
F_t &= (\textbf{cons}\ E_t\ M_t) \mid (\textbf{cons}\ V\ E_t) \mid (\textbf{if}\ E_t\ M_t\ M_t) \mid (\textbf{car}\ E_t) \mid (\textbf{cdr}\ E_t) \mid \\
&\quad (E_t\ M_t) \mid (V\ E_t) \mid (P\ E_t\ M_t) \mid (P\ V\ E_t) \mid (\textbf{output}\ E_t) \mid [] \\
\mathcal{L}_t &= V \cup \{\tau\} \\
\text{final states} &= V \cup \{\text{error}\}
\end{aligned}
$$

The function '$\mapsto_t$' extends '$\mapsto$' with the following rules:

$$
\begin{aligned}
E_t[(\textbf{w-c-m}\ V\ (\textbf{w-c-m}\ W\ M))] &\overset{\tau}{\mapsto}_t E_t[(\textbf{w-c-m}\ W\ M)]\ (\text{where } E_t \neq E'_t[(\textbf{w-c-m}\ Z\ [])]) \\
E_t[(\textbf{w-c-m}\ V\ W)] &\overset{\tau}{\mapsto}_t E_t[W]\ (\text{where } E_t \neq E'_t[(\textbf{w-c-m}\ V\ [])]) \\
E_t[(\textbf{c-c-m})] &\overset{\tau}{\mapsto}_t E_t[X[\![E_t]\!]] \\
E_t[(\textbf{output}\ V)] &\overset{V}{\mapsto}_t E_t[13]
\end{aligned}
$$

where

$$
\begin{aligned}
X[\![(\textbf{w-c-m}\ V\ E_t)]\!] &= (\textbf{cons}\ V\ X[\![E_t]\!]) \\
X[\![E_t]\!] &= X[\![E'_t]\!] \text{ where } E_t = \begin{cases} (\textbf{cons}\ E'_t\ M) \\ (\textbf{cons}\ V\ E'_t) \\ (\textbf{if}\ E'_t\ M\ M) \\ \ldots \end{cases} \\
X[\![[]]\!] &= \textbf{null}
\end{aligned}
$$

Fig. 4. Extension of the source language M to the target language M_t

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 [18], where $\mathcal{L}$ denotes the set of labels. $\mathcal{L}$ includes the evaluation contexts along with τ, which denotes the transition without output. The only expression that generates output is the (**breakpoint**) expression. It displays the current evaluation context. 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 $\mapsto\!\!\!\rightarrow$ is defined as the transitive, reflexive closure of the relation $\mapsto$. That is, we say that $M_0 \overset{O}{\mapsto\!\!\!\rightarrow} M_n$ if there exist $M_0, \ldots, M_n$ such that $M_i \overset{l_i}{\mapsto\!\!\!\rightarrow} M_{i+1}$ and $O \in \mathcal{L}^* = l_0 l_1 \ldots l_{n-1}$.

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

$$\text{eval}(M) = \begin{cases} V & \text{if } M \mapsto\!\!\!\rightarrow V \\ \text{error} & \text{if } M \mapsto\!\!\!\rightarrow \text{error} \end{cases}$$

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:

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

The target language of our model is similar to the source language, except that it contains **w-c-m** and **c-c-m**, and an **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. 4.

The evaluation of the target language is designed to concatenate neighboring **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 **w-c-m**-expressions. In particular, the set E_t may include any kind of continuation, but its **w-c-m** variant F_t requires a subexpression that is not a **w-c-m** expression.

Note also the restriction on the **w-c-m** reductions that the enclosing context must not end with a **w-c-m**. This avoids two ambiguities: one that arises when two nested **w-c-m** expressions occur with a value inside the second, another that occurs when three or more **w-c-m** expressions appear in sequence.

For the target language, the set of labels is the set of values plus τ. The **output** instruction is the only instruction that generates output.

The standard reduction relation $\mapsto_t$ 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 $\mapsto_t$, written as $\mapsto\!\!\!\rightarrow_t$. The target language's evaluation function eval_t and trace function trace_t are adapted mutatis mutandis from their source language counterparts, with $\mapsto$ and $\mapsto\!\!\!\rightarrow$ replaced by $\mapsto_t$ and $\mapsto\!\!\!\rightarrow_t$.

Roughly speaking, (**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

$$\mathcal{A}[\![V]\!] = \begin{cases} (\textbf{cons}\ \mathcal{A}[\![V]\!]\ \mathcal{A}[\![W]\!]) & \text{if } V = (\textbf{cons}\ V\ \ W) \\ (\textbf{lambda}\ (x)\ \mathcal{A}[\![M]\!]) & \text{if } V = (\textbf{lambda}\ (x)\ M) \\ V & \text{otherwise} \end{cases}$$

$$\begin{aligned}
\mathcal{A}[\![(\textbf{breakpoint})]\!] &= (\textbf{w-c-m}\ (\textbf{list}\ \text{'}\mathsf{break})\ (\textbf{output}\ (\textbf{c-c-m}))) \\
\mathcal{A}[\![(V\ \ M)]\!] &= (\textbf{w-c-m}\ (\textbf{list}\ \text{'}\mathsf{appB}\ \mathcal{A}[\![V]\!])\ (\mathcal{A}[\![V]\!]\ \mathcal{A}[\![M]\!])) \\
\mathcal{A}[\![(M\ \ N)]\!] &= (\textbf{w-c-m}\ (\textbf{list}\ \text{'}\mathsf{appA}\ \mathcal{Q}[\![N]\!]) \\
&\quad\ ((\textbf{lambda}\ (\mathcal{F})\ (\textbf{w-c-m}\ (\textbf{list}\ \text{'}\mathsf{appB}\ \mathcal{F})\ (\mathcal{F}\ \mathcal{A}[\![N]\!])))\ \mathcal{A}[\![M]\!])) \\
\mathcal{A}[\![(\textbf{if}\ L\ \ M\ \ N)]\!] &= (\textbf{w-c-m}\ (\textbf{list}\ \text{'}\mathsf{if}\ \mathcal{Q}[\![M]\!]\ \mathcal{Q}[\![N]\!])\ (\textbf{if}\ \mathcal{A}[\![L]\!]\ \mathcal{A}[\![M]\!]\ \mathcal{A}[\![N]\!])) \\
\mathcal{A}[\![(\textbf{cons}\ V\ \ M)]\!] &= (\textbf{w-c-m}\ (\textbf{list}\ \text{'}\mathsf{consB}\ \mathcal{A}[\![V]\!])\ (\textbf{cons}\ \mathcal{A}[\![V]\!]\ \mathcal{A}[\![M]\!])) \\
\mathcal{A}[\![(\textbf{cons}\ M\ \ N)]\!] &= (\textbf{w-c-m}\ (\textbf{list}\ \text{'}\mathsf{consA}\ \mathcal{Q}[\![N]\!]) \\
&\quad\ ((\textbf{lambda}\ (\mathcal{F})\ (\textbf{w-c-m}\ (\textbf{list}\ \text{'}\mathsf{consB}\ \mathcal{F})\ (\textbf{cons}\ \mathcal{F}\ \mathcal{A}[\![N]\!]))) \\
&\quad\ \ \mathcal{A}[\![M]\!])) \\
\mathcal{A}[\![(\textbf{car}\ M)]\!] &= (\textbf{w-c-m}\ (\textbf{list}\ \text{'}\mathsf{car})\ (\textbf{car}\ \mathcal{A}[\![M]\!])) \\
\mathcal{A}[\![(\textbf{cdr}\ M)]\!] &= (\textbf{w-c-m}\ (\textbf{list}\ \text{'}\mathsf{cdr})\ (\textbf{cdr}\ \mathcal{A}[\![M]\!])) \\
\mathcal{A}[\![(P\ \ V\ \ M)]\!] &= (\textbf{w-c-m}\ (\textbf{list}\ \text{'}\mathsf{primB}\ P\ \ V)\ (P\ \ V\ \mathcal{A}[\![M]\!])) \\
\mathcal{A}[\![(P\ \ M\ \ N)]\!] &= (\textbf{w-c-m}\ (\textbf{list}\ \text{'}\mathsf{primA}\ P\ \mathcal{Q}[\![N]\!]) \\
&\quad\ ((\textbf{lambda}\ (\mathcal{F})\ (\textbf{w-c-m}\ (\textbf{list}\ \text{'}\mathsf{primB}\ P\ \mathcal{F})\ (P\ \mathcal{F}\ \mathcal{A}[\![N]\!])))\ \mathcal{A}[\![M]\!]))
\end{aligned}$$

where

$$\begin{aligned}
\mathcal{Q}[\![x]\!] &= (\textbf{list}\ \text{'}\mathsf{val}\ x) \\
\mathcal{Q}[\![\text{'}\mathsf{x}]\!] &= (\textbf{list}\ \text{'}\mathsf{quote}\ \text{'}\mathsf{x}) \\
\mathcal{Q}[\![M]\!] &= \begin{cases} (\textbf{list}\ \text{'}\mathsf{app}\ \mathcal{Q}[\![M_1]\!]\ \mathcal{Q}[\![M_2]\!]) & \text{if } M = (M_1\ M_2) \\ (\textbf{list}\ \text{'}\mathsf{if}\ \mathcal{Q}[\![M_1]\!]\ \mathcal{Q}[\![M_2]\!]\ \mathcal{Q}[\![M_3]\!]) & \text{if } M = (\textbf{if}\ M_1\ M_2\ M_3) \\ \ldots \end{cases}
\end{aligned}$$

Fig. 5. The annotating function, $\mathcal{A} : M \to M_t$

target language that creates the same effect via a combination of continuation marks and a simple **output** expression.

The elaboration function is defined in fig. 5.[4] It assumes that the identifier $\mathcal{F}$ does not appear in the source program. It also relies upon a quoting function, $\mathcal{Q}$, 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

[4] The **list** constructor is used in the remainder of the paper as a syntactic abbreviation for a series of **cons**es.

$$
\begin{aligned}
\mathcal{T}[\![(\mathbf{cons}\ (\mathbf{list}\ \text{'}\mathsf{appA}\ N)\ M)]\!] &= (\mathcal{T}[\![M]\!]\ \overline{\mathcal{Q}}[\![N]\!])\\
\mathcal{T}[\![(\mathbf{cons}\ (\mathbf{list}\ \text{'}\mathsf{appB}\ V)\ M)]\!] &= (V\ \mathcal{T}[\![M]\!])\\
\mathcal{T}[\![(\mathbf{cons}\ (\mathbf{list}\ \text{'}\mathsf{if}\ N\ L)\ M)]\!] &= (\mathbf{if}\ \mathcal{T}[\![M]\!]\ \overline{\mathcal{Q}}[\![N]\!]\ \overline{\mathcal{Q}}[\![L]\!])\\
\mathcal{T}[\![(\mathbf{cons}\ (\mathbf{list}\ \text{'}\mathsf{consA}\ N)\ M)]\!] &= (\mathbf{cons}\ \mathcal{T}[\![M]\!]\ \overline{\mathcal{Q}}[\![N]\!])\\
\mathcal{T}[\![(\mathbf{cons}\ (\mathbf{list}\ \text{'}\mathsf{consB}\ V)\ M)]\!] &= (\mathbf{cons}\ V\ \mathcal{T}[\![M]\!])\\
\mathcal{T}[\![(\mathbf{cons}\ (\mathbf{list}\ \text{'}\mathsf{car})\ M)]\!] &= (\mathbf{car}\ \mathcal{T}[\![M]\!])\\
\mathcal{T}[\![(\mathbf{cons}\ (\mathbf{list}\ \text{'}\mathsf{cdr})\ M)]\!] &= (\mathbf{cdr}\ \mathcal{T}[\![M]\!])\\
\mathcal{T}[\![(\mathbf{cons}\ (\mathbf{list}\ \text{'}\mathsf{primA}\ P\ N)\ M)]\!] &= (P\ \mathcal{T}[\![M]\!]\ \overline{\mathcal{Q}}[\![N]\!])\\
\mathcal{T}[\![(\mathbf{cons}\ (\mathbf{list}\ \text{'}\mathsf{primB}\ P\ V)\ M)]\!] &= (P\ V\ \mathcal{T}[\![M]\!])\\
\mathcal{T}[\![(\mathbf{cons}\ \text{'}\mathsf{break}\ \mathbf{null})]\!] &= []
\end{aligned}
$$

where

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

Fig. 6. The Translation function, $\mathcal{T} : V \to E$

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 converge. Finally, there is a function $\mathcal{T}$, shown in fig. 6, mapping the values produced by **output** in the target program to the corresponding evaluation contexts produced by (**breakpoint**) expressions. We extend $\mathcal{T}$ 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. *$eval(M_0) = V$ iff $eval_t(N_0) = \mathcal{A}[\![V]\!]$.*
2. *$eval(M_0) = error$ iff $eval_t(N_0) = error$.*
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. $trace[\![S]\!] = \mathcal{T}(trace[\![S_t]\!])$*

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 upon the two program evaluations. It is complicated by the fact that one step in the source program corresponds to either one, two, or four steps in the elaborated program. The additional steps in the elaborated program are **w-c-m** reductions, which patch up the invariant that the source program and the elaborated program are related by $\mathcal{A}$. ∎

4 Stepping with Continuation Marks

The full stepper is built on top of the framework of section 3, and also comprises an elaborator and reconstructor. The elaborator transforms the user's program into one containing **breakpoint**s 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 section we develop the model of our stepper implementation and its correctness proof. Subsection 4.1 describes the elaborator and the reconstructor, and formalizes them. Subsection 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 section 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 $\mathcal{A}$ of section 3.

The front end, $\mathcal{B}$, translates a plain functional language into the source language of section 3. More specifically, it accepts expressions in M_s, 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. Fig. 7 shows the definition of $\mathcal{B}$. The translation is syntax-directed according to the expression language. Since some expressions have subexpressions in non-tail positions, $\mathcal{B}$ 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 $\mathcal{I}_0$, $\mathcal{I}_1$, and $\mathcal{I}_2$ as temporary variables that do not appear in the source program. In this and later figures we use the **let**$*$ expression as syntactic shorthand.[5]

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 section 3. The execution of the target program produces a stream of output values. The function $\mathcal{T}$ of fig. 6 maps these values back to evaluation contexts of the intermediate language, that is, the source language of section 3. Since the instruction filling these contexts must be **breakpoint**, the reconstruction function

[5] The **let**$*$ expression is roughly equivalent to the sequential **let** of ML. It is used as syntactic shorthand for a corresponding set of applications like those in fig. 5.

$$
\mathcal{B}\llbracket V \rrbracket = \begin{cases} (\textbf{lambda}\ (x)\ \mathcal{B}\llbracket M \rrbracket) \\ \quad \text{if } V = (\textbf{lambda}\ (x)\ M) \\ (\textbf{cons}\ \mathcal{B}\llbracket M \rrbracket\ \mathcal{B}\llbracket N \rrbracket) \\ \quad \text{if } V = (\textbf{cons}\ M\ N) \\ V \text{ otherwise} \end{cases}
$$

$$
\begin{aligned}
\mathcal{B}\llbracket (M\ N) \rrbracket &= (\textbf{let}*\ ([\mathcal{I}_0\ \mathcal{B}\llbracket M \rrbracket] \\
&\qquad\qquad [\mathcal{I}_1\ \mathcal{B}\llbracket N \rrbracket] \\
&\qquad\qquad [\mathcal{I}_2\ (\textbf{breakpoint})]) \\
&\qquad (\mathcal{I}_0\ \mathcal{I}_1)) \\
\mathcal{B}\llbracket (V\ N) \rrbracket &= (\textbf{let}*\ ([\mathcal{I}_1\ \mathcal{B}\llbracket N \rrbracket] \\
&\qquad\qquad [\mathcal{I}_2\ (\textbf{breakpoint})]) \\
&\qquad (V\ \mathcal{I}_1)) \\
\mathcal{B}\llbracket (V\ U) \rrbracket &= (\textbf{let}*\ ([\mathcal{I}_2\ (\textbf{breakpoint})]) \\
&\qquad (V\ U)) \\
\mathcal{B}\llbracket (\textbf{if}\ M\ N\ L) \rrbracket &= (\textbf{let}*\ ([\mathcal{I}_0\ \mathcal{B}\llbracket M \rrbracket] \\
&\qquad\qquad [\mathcal{I}_1\ (\textbf{breakpoint})]) \\
&\qquad (\textbf{if}\ \mathcal{I}_0\ \mathcal{B}\llbracket N \rrbracket\ \mathcal{B}\llbracket L \rrbracket)) \\
\mathcal{B}\llbracket (\textbf{if}\ V\ N\ L) \rrbracket &= (\textbf{let}*\ ([\mathcal{I}_0\ (\textbf{breakpoint})]) \\
&\qquad (\textbf{if}\ V\ \mathcal{B}\llbracket N \rrbracket\ \mathcal{B}\llbracket L \rrbracket))
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{B}\llbracket (\textbf{car}\ M) \rrbracket &= (\textbf{let}*\ ([\mathcal{I}_0\ \mathcal{B}\llbracket M \rrbracket] \\
&\qquad\qquad [\mathcal{I}_1\ (\textbf{breakpoint})]) \\
&\qquad (\textbf{car}\ \mathcal{I}_0)) \\
\mathcal{B}\llbracket (\textbf{car}\ V) \rrbracket &= (\textbf{let}*\ ([\mathcal{I}_2\ (\textbf{breakpoint})]) \\
&\qquad (\textbf{car}\ V)) \\
\mathcal{B}\llbracket (\textbf{cdr}\ M) \rrbracket &= (\textbf{let}*\ ([\mathcal{I}_0\ \mathcal{B}\llbracket M \rrbracket] \\
&\qquad\qquad [\mathcal{I}_1\ (\textbf{breakpoint})]) \\
&\qquad (\textbf{cdr}\ \mathcal{I}_0)) \\
\mathcal{B}\llbracket (\textbf{cdr}\ V) \rrbracket &= (\textbf{let}*\ ([\mathcal{I}_2\ (\textbf{breakpoint})]) \\
&\qquad (\textbf{cdr}\ V)) \\
\mathcal{B}\llbracket (P\ M\ N) \rrbracket &= (\textbf{let}*\ ([\mathcal{I}_0\ \mathcal{B}\llbracket M \rrbracket] \\
&\qquad\qquad [\mathcal{I}_1\ \mathcal{B}\llbracket N \rrbracket] \\
&\qquad\qquad [\mathcal{I}_2\ (\textbf{breakpoint})]) \\
&\qquad (P\ \mathcal{I}_0\ \mathcal{I}_1)) \\
\mathcal{B}\llbracket (P\ V\ N) \rrbracket &= (\textbf{let}*\ ([\mathcal{I}_1\ \mathcal{B}\llbracket N \rrbracket] \\
&\qquad\qquad [\mathcal{I}_2\ (\textbf{breakpoint})]) \\
&\qquad (P\ V\ \mathcal{I}_1)) \\
\mathcal{B}\llbracket (P\ V\ U) \rrbracket &= (\textbf{let}*\ ([\mathcal{I}_2\ (\textbf{breakpoint})]) \\
&\qquad (P\ V\ U))
\end{aligned}
$$

Fig. 7. The stepper's breakpoint-inserting function, $\mathcal{B} : M_s \to M$

$\mathcal{R}$ is defined simply as the inversion of the annotation applied to the context filled with **breakpoint**. In other words, $\mathcal{R}\llbracket E \rrbracket = \mathcal{B}^{-1}\llbracket E[(\textbf{breakpoint})] \rrbracket$.[6] 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* $S = (M_0 \mapsto \cdots \mapsto M_n)$*, there exists an evaluation sequence* $S_t = (\mathcal{A}\llbracket \mathcal{B}\llbracket M_0 \rrbracket \rrbracket \mapsto_t \cdots \mapsto_t N_k)$ *such that* $S = \mathcal{R}\llbracket \mathcal{T}\llbracket \mathit{trace}\llbracket S_t \rrbracket \rrbracket \rrbracket$.

Proof Sketch: By the Elaboration theorem, it suffices to prove that, given a sequence S as in the theorem statement, there exists $S_a = (\mathcal{B}\llbracket M_0 \rrbracket \mapsto \cdots \mapsto N_{k'})$ such that $S = \mathcal{R}\llbracket \mathrm{trace}_t\llbracket S_a \rrbracket \rrbracket$.

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

[6] Inspection of the definition of $\mathcal{B}$ demonstrates that it is invertible.

steps maintains the invariant that the source program and the target program are related by $\mathcal{B}$. ∎

4.3 From Model to Implementation

From an implementation perspective, 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.

5 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 [24], Kellomaki's PSD [14], and several projects in the lazy FP community [12, 20, 22, 23]. 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 [15] 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 [16] for our implementation of the stepper. Bertot's work does not verify a stepper but simply assumes that the language evaluator *is* a stepper.

6 Conclusion

Our paper 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 [17] 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 paper 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.

References

[1] Balzer, R. M. EXDAMS — EXtendable Debugging And Monitoring System. In *AFIPS 1969 Spring Joint Computer Conference*, volume 34, pages 567–580. AFIPS Press, May 1969.

[2] Barendregt, H. P. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, revised edition, 1984.

[3] Bernstein, K. L. and E. W. Stark. Operational semantics of a focusing debugger. In *Eleventh Conference on the Mathematical Foundations of Programming*

Semantics, Volume 1 of Electronic Notes in Computer Science. Elsevier, March 1995.

[4] Bertot, Y. Occurrences in debugger specifications. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1991.

[5] Felleisen, M. Programming languages and their calculi. Unpublished Manuscript. http://www.cs.rice.edu/˜matthias/411/mono.ps.

[6] Felleisen, M. and R. Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 102:235–271, 1992.

[7] Ferguson, H. E. and E. Berner. Debugging systems at the source language level. *Communications of the ACM*, 6(8):430–432, August 1963.

[8] Findler, R. B., J. Clements, C. Flanagan, M. Flatt, S. Krishnamurthi, P. Steckler and M. Felleisen. Drscheme: A programming environment for Scheme. *Journal of Functional Programming*, 2001.

[9] Findler, R. B., C. Flanagan, M. Flatt, S. Krishnamurthi and M. Felleisen. DrScheme: A pedagogic programming environment for Scheme. In *International Symposium on Programming Languages: Implementations, Logics, and Programs*, number 1292 in Lecture Notes in Computer Science, pages 369–388, 1997.

[10] Flatt, M. PLT MzScheme: Language manual. Technical Report TR97-280, Rice University, 1997.

[11] Flatt, M., S. Krishnamurthi and M. Felleisen. Classes and mixins. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 171–183, January 1998.

[12] Hall, C. and J. O'Donnell. Debugging in a side effect free programming environment. In *ACM SIGPLAN symposium on Language issues in programming environments*, 1985.

[13] Hannan, J. and D. Miller. From operational semantics to abstract machines. *Journal of Mathematical Structures in Computer Science*, 2(4):415–459, 1992.

[14] Kellomaki, P. Psd — a portable scheme debugger, Feburary 1995.

[15] Kishon, A., P. Hudak and C. Consel. Monitoring semantics: a formal framework for specifying, implementing and reasoning about execution monitors. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 338–352, June 1991.

[16] Krishnamurthi, S. PLT McMicMac: Elaborator manual. Technical Report 99-334, Rice University, Houston, TX, USA, 1999.

[17] Landin, P. J. The mechanical evaluation of expressions. *Comput. J.*, 6(4):308–320, 1964.

[18] Milner, R. *Communication and Concurrency*. Prentice Hall, 1989.

[19] Moreau, L. A syntactic theory of dynamic binding. *Higher-Order and Symbolic Computation*, 11(3):233–279, 1998.

[20] Naish, L. and T. Barbour. Towards a portable lazy functional declarative debugger. In *19th Australasian Computer Science Conference*, 1996.

[21] Plotkin, G. D. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, pages 125–159, 1975.

[22] Sansom, P. and S. Peyton-Jones. Formally-based profiling for higher-order functional languages. *ACM Transactions on Programming Languages and Systems*, 19(1), January 1997.

[23] Sparud, J. and C. Runciman. Tracing lazy functional computations using redex trails. In *Symposium on Programming Language Implementation and Logic Programming*, 1997.

[24] Tolmach, A. *Debugging Standard ML*. PhD thesis, Department of Computer Science, Princeton University, October 1992.