Advice about Debugger Construction

Arthur Nunes-Harwitt
Nassau Community College
One Education Drive
Garden City, NY 11530
nunesa@ncc.edu

Abstract
A debugger is a tool that allows the programmer to view some aspect of the running program. This paper will present a specification of the debugger commands step and next for the call-by-value λ-calculus with constants. The operationally-defined CEK-machine will be extended to implement these commands in a way that is proven to be faithful to the specification. This theory will then be used to develop actual debuggers written in Scheme.

1 Introduction
A debugger is a fundamental programming language tool. Debuggers have many features, such as allowing programmers to inspect variables. They also help programmers identify problematic code by allowing them to move through the code in small or large jumps as it executes. This feature will be the focus of the paper.

A short time ago I had the task of writing a debugger for a small interpreted language. When looking around for texts on programming language implementation, I found quite a variety[1, 12, 10, 6]. There were even books[1, 12] discussing the relatively esoteric topic of garbage collection. In contrast, there is little to be found on the topic of debuggers. What does exist seems to explain only how to implement the step command. The purpose of this paper is to address this gap by providing a theoretically justified model of a debugger for an interpreter that efficiently implements both step and next commands.

This paper is structured as follows. First, I will summarize some of the previous theoretical work on debuggers. After that, a debugger specification will be given that defines how movement is to occur through the code, given a particular debugger command. Then an operational semantics implementation of the specification will be developed and proved correct. Finally, since sometimes it is more natural to use denotational semantics, I will discuss the connection between the operational and the denotational approaches.

2 Debugger Overview

There are others who have studied the problem of constructing a correct debugger. While these approaches do formalize aspects of debuggers, they do not completely clarify how to efficiently implement multiple debugger commands.

Kishon, Hudak, and Consel describe a methodical way to extend a language defined by a denotational semantics[8]. They introduce a monitoring semantics that makes it possible to observe before and after the behavior of code that is statically annotated. Their approach allows for simple definitions of tracers and profilers. They hint that they can implement a stepper, but it is not clear that they can efficiently implement both step and next commands.

The formalism adopted by Bernstein and Stark[3] is similar to the formalism used in this paper in that a small-step operational semantics is used to define debugger commands. Instead of describing the traditional step and next debug commands, though, the authors describe commands for a ‘focusing’ debugger. And while they establish that their debugger preserves the meaning of the language, they offer no help in implementing their theory.

Clements, Flatt, and Felleisen develop a stepper for DrScheme[4]. To do this, they first introduce an interesting and powerful new primitive: with-continuation-mark. This primitive allows the programmer to attach arbitrary data to the current continuation. They then explain how to rewrite and instrument the source code by adding with-continuation-mark procedures to keep track of the labels and breakpoint procedures to stop execution. Their notion of step is
\[ c_{\text{op}} \in \text{Operators} \]
\[ c \in \text{Constants} \supset \text{Operators} \]
\[ x \in \text{Variables} \]
\[ V \in \text{Values} ::= c \mid x \mid (\lambda x.M) \]
\[ M, N \in \Lambda ::= V \mid (M M) \]

Evaluation contexts are defined equivalently either by
\[ \mathcal{E} ::= [] \mid \mathcal{E}[V] \mid \mathcal{E}[(\mathcal{E} \mathcal{M})] \]

\[ M \xrightarrow{\beta} N \]
\[ \mathcal{E}[M] \xrightarrow{\delta} \mathcal{E}[N] \]
\[ (c_{\text{op}} c) \xrightarrow{\delta} V \]

Figure 1: Call-by-value \( \lambda \)-calculus with constants

\[ E \in \text{Env} ::= [] \mid (x, (V, E)) :: E \]
\[ K \in \mathcal{K} ::= \emptyset \mid \text{fun}(V, E, K) \mid \text{arg}(M, E, K) \]
\[ \text{lookup}(x, (y, (V, E)) :: E') = \begin{cases} (V, E) & \text{if } x = y \\ \text{lookup}(x, E') & \text{otherwise} \end{cases} \]
\[ C \in \mathcal{C} ::= \langle M, E, K \rangle \]
\[ \langle M, N, E, K \rangle \implies \langle M, E, \text{arg}(N, E, K) \rangle \]
\[ \langle V, E, \text{arg}(N, E', K) \rangle \implies \langle N, E', \text{fun}(V, E, K) \rangle \quad \text{if } V \notin \text{Variables} \]
\[ \langle c, E, \text{fun}(c_{\text{op}}, E', K) \rangle \implies \langle \delta(c_{\text{op}}, c), [], K \rangle \]
\[ \langle V, E, \text{fun}((\lambda x.M), E', K) \rangle \implies \langle M, (x, (V, E)) :: E', K \rangle \quad \text{if } V \notin \text{Variables} \]
\[ \langle x, E, K \rangle \implies \langle \text{lookup}(x, E)_1, \text{lookup}(x, E)_2, K \rangle \]

Figure 2: CEK-machine

3 A Correct Debugger

In this section, I will define a specification for a debugger for the call-by-value \( \lambda \)-calculus with constants[9]. (See Figure 1 for the definition of the call-by-value \( \lambda \)-calculus.) The CEK-machine[5] will be used to interpret this language. (See Figure 2 for the definition of the CEK-machine.) A debugger extension of this machine will be proved correct. Before going into too much detail, I would like to distinguish between two approaches to debugging: watching and stepping. **Watching** identifies relevant sub-expressions of the executing program. **Stepping** identifies relevant sub-expressions of the original program (often by highlighting them in a text window). (See Figure 3 for a sample stepper interaction.) Clearly, the two are related. Section 3.1 focuses on the watching functionality. Section 3.2 shows how the watcher can be augmented to make it a stepper.

3.1 A Correct Watcher

The call-by-value \( \lambda \)-calculus with constants is at the core of many programming languages (including Scheme[7] and Common LISP[11]). This simplified language has enough structure so that it is possible to outline how to construct a watcher. This watcher accepts three commands: **step**, **next**, and **continue**. The **step** command moves over expressions at the finest level of granularity by moving over only values. The **next** command moves over entire expressions at a time. The **continue** command simply allows execution to proceed.
position:  ▷ (fact 5)
debug command> step
position:  ▷ fact
debug command> step
position:  fact ▷
debug command> step
position:  ▷ 5
debug command> step
position:  ▷ 5 ▷
debug command> step
position:  ▷ (if (= n 0) 1 (* n (fact (- n 1))))
debug command> step
position:  ▷ (= n 0)
debug command> next
position:  (= n 0) ▷
debug command> step
position:  ▷ (* n (fact (- n 1)))
debug command> step
position:  ▷ *
debug command> step
position:  * ▷
debug command> step
position:  ▷
debug command> continue
value:  120

Figure 3: Sample stepper interaction.

\[
\begin{align*}
M_d &\in \Lambda_d := \mathcal{E}[M] | \mathcal{E}[V] \\
M_e &\in \Lambda_e = \Lambda \cup \Lambda_d \\
K_d &\in \mathcal{K}_d ::= kd_\emptyset | \text{fun}(V, E, K_d) | \text{arg}(M, E, K_d) | \text{fun}_d(V, E, K) | \text{arg}_d(M, E, K) \\
K_e &\in \mathcal{K}_e = \mathcal{K} \cup \mathcal{K}_d \\
C &\in \mathcal{C} ::= \langle M, E, K_e \rangle \\
C_d &\in \mathcal{C}_d ::= \langle M, E, K \rangle | \langle V, E, K \rangle \\
C_e &\in \mathcal{C}_e = \mathcal{C} \cup \mathcal{C}_d
\end{align*}
\]

Figure 4: Debugger language extensions.
\begin{align*}
\text{[step 1]} & \quad \mathcal{E}[(\cdot M N)] \xrightarrow{\text{step}} \mathcal{E}[(\cdot M N)] \\
\text{[step 2]} & \quad \mathcal{E}[V] \xrightarrow{\text{step}} \mathcal{E}[V] \\
\text{[step 3]} & \quad \mathcal{E}[(V \cdot M)] \xrightarrow{\text{step}} \mathcal{E}[(V \cdot M)] \\
\text{[step 4]} & \quad E[(c_{\text{op}} e)] \xrightarrow{\text{step}} \mathcal{E}[V] \\
\text{[step 5]} & \quad (\lambda x. M) V \xrightarrow{\beta_v} N \\
& \quad \mathcal{E}[(\lambda x. M) V] \xrightarrow{\text{step}} \mathcal{E}[\cdot N]
\end{align*}

Figure 5: Step command specification.

\[
\begin{align*}
\epsilon(V) & = V \\
\epsilon(\cdot M) & = M \\
\epsilon(\cdot V) & = V \\
\epsilon((Me N_e)) & = (\epsilon(Me) \epsilon(N_e)) \\
\hline
M & \rightarrow N \\
M & \xrightarrow{0} M \\
M & \xrightarrow{0} N
\end{align*}
\]

Figure 6: Definitions of $\epsilon$ and $\xrightarrow{0}$.

The behavior of each command is specified using a ‘dotted’ extension of the call-by-value $\lambda$-calculus. The dot indicates either that an expression is about to be evaluated or that a value has been evaluated, and thereby models the output of the debugger. (See Figure 4 for the definition of this extension.) The specification is essentially a new language, but it behaves the same way as the original language. I will show that debugging steps do not change the meaning of the original computation, and that the abstract machine that implements the specification does so correctly. Because each command is modeled as a reduction relation, there is little interaction between the debugger commands, and the proof is essentially modular.

The construction of the continue command is actually trivial: simply let the original interpreter run to completion. The other two commands require more work. The next two subsections go through many of the details involved in implementing step and next.

### 3.1.1 The Correctness of the Step Command

The specification of the step command is given in Figure 5. Rule (1) says that applications get stepped into. Rule (2) says that values are stepped over. Rule (3) says that step moves from an evaluated expression to the next expression in an application. Rule (4) says that step moves from inside a constant application to outside the result of the application. Rule (5) says that step moves from inside an application of an abstraction to a value to the beginning of the body of the abstraction.

To show that these relations do not change the meaning of the computation, two definitions are needed. One is for an erasure function that maps ‘dotted’ terms to plain terms, and one is for a reduction that is the reflexive closure of the reduction relation of the plain call-by-value $\lambda$-calculus. Taking a step with a dotted term will then correspond to using this new reduction with the dot erased.

**Definition 1** The erasure function $\epsilon : \Lambda_e \rightarrow \Lambda$ is defined in Figure 6.

**Definition 2** The relation $\xrightarrow{0}$ is reflexive closure of $\rightarrow$. This definition is given formally in Figure 6.

**Definition 3** A term $Me \in \Lambda_e$ is closed iff \(fv(Me) = \emptyset\) (i.e. $Me$ has no free variables).

**Theorem 1** For any closed term $Md \in \Lambda_d$ if $Md \xrightarrow{\text{step}} N_d$ then $\epsilon(Md) \xrightarrow{0} \epsilon(N_d)$.

**Proof Sketch:**

Structural induction on terms in the domain of $\xrightarrow{\text{step}}$. 

\[ \]
The implementation of the \texttt{step} command as an extension to the CEK-machine is given in Figure 7. These reductions are intended to reflect the corresponding reductions for dotted \(\lambda\)-terms. To show that this machine faithfully implements the specification, some more definitions are in order.

\textbf{Definition 4}  The relation \(\xrightarrow{\text{step}}\) is defined in Figure 8.

\textbf{Definition 5}  The function \(\psi : \mathcal{C}_e \rightarrow \mathcal{L}_e\) maps machine configurations to terms; the details are given in Figure 8.

With these definitions, taking a step with a configuration will correspond to taking a step with the term that corresponds to the configuration.

\textbf{Definition 6}  A configuration \(C_e \in \mathcal{C}_e\) is closed iff \(\psi(C_e)\) is closed.

\textbf{Theorem 2}  For any closed configuration \(C_d \in \mathcal{C}_d\) if \(C_d \xrightarrow{\text{step}} C_d'\) then \(\psi(C_d) \xrightarrow{\text{step}} \psi(C_d')\).

\textbf{Proof Sketch:}  Structural induction on configurations in the domain of \(\xrightarrow{\text{step}}\).

\subsection{3.1.2 The Correctness of the \texttt{Next} Command}

The specification of the \texttt{next} command is given in Figure 9. Rule (1) says that any term gets stepped over. Rule (2) says that \texttt{next} moves from inside an application to outside the result of the application. Rule (3) says that \texttt{next} moves from an evaluated expression to the next expression in an application.

The stage is already set to show that these relations do not change the meaning of the computation. Taking a step with a dotted term will then correspond to using the reflexive transitive closure of the original reduction with the dot erased.

\textbf{Theorem 3}  For any closed term \(M_d \in \mathcal{L}_d\) if \(M_d \xrightarrow{\text{next}} N_d\) then \(\epsilon(M_d) \xrightarrow{\text{next}} \epsilon(N_d)\).

\textbf{Proof Sketch:}  Structural induction on terms in the domain of \(\xrightarrow{\text{next}}\).

The implementation of the \texttt{next} command as an extension to the CEK-machine is given in Figure 10. These reductions are intended to reflect the corresponding reductions for dotted \(\lambda\)-terms. While it may be possible to implement \texttt{next} using multiple \texttt{step} operations, a separate implementation is more realistic: here the machine can run at full speed until the expression has been evaluated. To show that this machine faithfully implements the specification, another definition is needed.

\textbf{Definition 7}  The relation \(\xrightarrow{\text{next}}\) is defined in Figure 10.

\textbf{Theorem 4}  For any closed configuration \(C_d \in \mathcal{C}_d\) if \(C_d \xrightarrow{\text{next}} C_d'\) then \(\psi(C_d) \xrightarrow{\text{next}} \psi(C_d')\).

\textbf{Proof Sketch:}  Structural induction on configurations in the domain of \(\xrightarrow{\text{next}}\). The correctness of the CEK-machine is relied upon in some of the cases.
\[
\frac{C_d \xrightarrow{\text{step}} C_e \xrightarrow{\cdot} C'_d}{C_d \xrightarrow{\text{next}} C'_d}
\]

\[\psi : C_e \rightarrow \Lambda_e\]

\[\psi((M_e, E, K_e)) = \psi'(K_e)[\nu(M_e, E)]\]

\[\psi': \mathcal{K}_e \rightarrow \mathcal{E}\]

\[\psi'(K_d) = \psi'(K)\]
\[\psi'(k_0) = []\]
\[\psi'(\text{fun}(V, E, K)) = \psi'(K)[\nu(V, E) []]\]
\[\psi'(\text{arg}(M, E, K)) = \psi'(K)[[] \nu(M, E)]\]

\[\nu : \Lambda_e \times \text{Env} \rightarrow \Lambda_e\]

\[\nu(\cdot, M, E) = \cdot \nu(M, E)\]
\[\nu(V \cdot, E) = \nu(V, E)\cdot\]
\[\nu(c, E) = c\]
\[\nu(x, []) = x\]
\[\nu(x, (y, (V, E)) :: E') = \begin{cases} 
\nu(V, E) & \text{if } x = y \\
\nu(x, E') & \text{otherwise}
\end{cases}\]
\[\nu((\lambda x. M), E) = (\lambda x. \nu(M, E \cdot x))\]
\[\nu((M_e N_e), E) = (\nu(M_e, E) \nu(N_e, E))\]

Figure 8: Definitions of \(\psi\) and \(\xrightarrow{\text{step}}\).

\[
\frac{M \xrightarrow{\cdot} V}{\mathcal{E}[-] \xrightarrow{\text{next}} \mathcal{E}[\cdot]} \quad [\text{next 1}]
\]

\[
\frac{(V_1 V_2) \xrightarrow{\cdot} V}{\mathcal{E}[-] \xrightarrow{\text{next}} \mathcal{E}[-]} \quad [\text{next 2}]
\]

\[
\mathcal{E}[-] \xrightarrow{\text{next}} \mathcal{E}[-] \quad [\text{next 3}]
\]

Figure 9: Next command specification.

\[
\begin{aligned}
\langle \cdot, M, E, K \rangle & \xrightarrow{\text{next}} \langle M, E, K_d \rangle \\
\langle V \cdot, E, \text{arg}(N, E', K) \rangle & \xrightarrow{\text{next}} \langle \cdot, N, E', \text{fun}(V, E, K) \rangle \\
\langle c, E, \text{fun}(c_{\text{op}}, E', K) \rangle & \xrightarrow{\text{next}} \langle \delta(c_{\text{op}}, c), [], K \rangle \\
\langle V \cdot, E, \text{fun}(\lambda x. M), E', K \rangle & \xrightarrow{\text{next}} \langle M, (x, (V, E)) :: E', K_d \rangle
\end{aligned}
\]

\[
\frac{C_d \xrightarrow{\text{next}} C_e \xrightarrow{\cdot} C'_d}{C_d \xrightarrow{\text{next}} C'_d}
\]

Figure 10: CEK implementation of \(\xrightarrow{\text{next}}\).
3.2 A Correct Stepper

Stepping extends watching by identifying the sub-expressions in the original program rather than the sub-expressions in the execution. Thus while the watcher might tell us that an expression has evaluated to \( V \), the stepper would indicate that the expression that evaluated to \( V \) was \( M \). Adding labels to the terms and the continuations is sufficient to allow the watcher to do stepping. Labels from the terms are transferred to continuations, and labels on either the term or the continuation are used to identify the sub-expression. In particular, when the debugger is sitting before a term, the label that is displayed is the label on the term; when the debugger is sitting after a value, the label to be displayed is not on the value, but on the continuation. Thus, the label of a term must be saved in the continuation. The label of an application is always saved since it may be needed later; otherwise, the label is saved if the debugger is sitting before the term and the rules indicate that the continuation is to be marked with a ‘d.’ The correctness results of the previous section are not affected by the addition of labels.

3.3 Adding Details

Although the call-by-value \( \lambda \)-calculus and the associated CEK-machine provide enough structure to see how a debugger should work, evaluating more realistic programs requires an extension. Figure 11 outlines an extended language. The RK-machine is a new abstract machine that comes closer to resembling a real interpreter. It evaluates expressions in the extended language. A Scheme implementation of this machine together with the debugger can be found at http://www.matcmp.sunynassau.edu/~nuenesa/comp_soft.html.

4 A Denotational Approach

The previous section indicates how one might go about constructing a debugger for an interpreter, assuming a particular kind of abstract machine. One feature of these machines is that they mix together evaluation and term de-structuring. The denotational approach separates the two and focuses specifically on what a term means. Interpreters are frequently written this way: the case analysis is on the structure of the term rather than the structure of a configuration. A sample denotational interpreter is given in Figure 12.
\[ T[c]_{\rho \kappa} = \kappa c \]
\[ T[x]_{\rho \kappa} = \kappa \rho(x) \]
\[ T[(\lambda x. M)]_{\rho \kappa} = \kappa(\lambda \kappa'. (\lambda v. T[M]_{\rho(x \rightarrow v)\kappa'})) \]
\[ T[(M \cdot N)]_{\rho \kappa} = T[M]_{\rho(\lambda m. T[N]_{\rho(\lambda n. ((m \cdot I) \kappa) n)})} \]

Figure 12: Simple denotational interpreter

\[ B[M]_{\rho \kappa} \]
\[ B[c]_{\rho \kappa} = \text{step} \quad T[M]_{\rho \kappa} \]
\[ B[x]_{\rho \kappa} = \text{step} \quad A \rho(x) \kappa \]
\[ B[(\lambda x. M)]_{\rho \kappa} = \text{step} \quad A(\lambda \kappa'. (\lambda v. T[M]_{\rho(x \rightarrow v)\kappa'})) \kappa \]
\[ B[(M \cdot N)]_{\rho \kappa} = \text{step} \quad B[M]_{\rho(\lambda m. T[N]_{\rho(\lambda n. ((m \cdot I) \kappa) n)})} \]
\[ B[M]_{\rho \kappa} = \text{next} \quad T[M]_{\rho(\lambda v. A \kappa)} \]

\[ A \kappa \]
\[ A v(\lambda n. ((m \cdot I) \kappa) n)) \]
\[ A v(\lambda m. T[M]_{\rho(\lambda n. ((m \cdot I) \kappa) n)}) \]
\[ A v(\lambda n. ((m \cdot I) \kappa) n)) \]
\[ A v(\lambda m. T[M]_{\rho(\lambda n. ((m \cdot I) \kappa) n)}) \]

Figure 13: Denotational style debugger
One might wonder whether the debugger outlined above can be expressed in a denotational style. Instead of writing \( \langle M, E, K \rangle \), one could write \( B[M]_{\rho K} \), where \( B \) would play the same role as the dot, indicating that the debugger is sitting before the term \( M \). Overwriting can still play the role of reading debug commands. Figure 13 outlines an implementation of \texttt{step} and \texttt{next}.

An unusual aspect of this implementation is its assumption that it is possible to look inside a continuation. One key feature that the operational semantics uses is the fact that continuations are transparent. In fact, it is necessary to look inside the continuation, because the specification has several rules of the form \( E[V] \). However, with denotational descriptions they are traditionally opaque.

In practice, this is not a problem, as it is possible to supply the interpreter with transparent continuations. One way to create them is to do closure-conversion\[2\] explicitly for the continuation functions. The code portion of the continuation function closure is unique and is therefore identifiable. Furthermore, if explicit closures are used to represent functions, then the code portion of a function closure could be replaced to step into a function rather than passing in an interpreter operator. A Scheme implementation of a denotational style interpreter/debugger with closure-converted continuations can be found at http://www.matcmp.sunynassau.edu/~nunesa/comp_soft.html.

5 Conclusion

While many debugger implementations exist today, it has been difficult to find a simple description of one. This paper presented a novel debugger specification, and a correct operational semantics implementation of multiple debugger commands which was both simple and efficient. An alternative denotational semantics showed how to make use of the debugger definition outside the context of an abstract machine. Pointers to actual Scheme implementations were also given.

In order to extend the debugger described in this paper, it is necessary to specify and implement a debug rule for each special form. Is it possible to automatically generate such rules for derived special forms? The debuggers in this paper work with interpreters. What does it take to extend these methods so that they work for a compiler? These questions remain for future research.

6 Acknowledgements

I would like to thank several people for their help. John Micco and Mike Karr first got me interested in the problem of designing interpreter debuggers. Mitch Wand and Matthias Felleisen provided encouragement for these ideas. John Clements took the time to discuss his paper with me. Finally, Melissa Nunes-Hawwitt and Harry Mairson carefully read previous drafts of this paper.

References


