

The Tail-Recursive SECD Machine

John D. Ramsdell*
The MITRE Corporation

January 1999

Abstract

One method for producing verified implementations of programming languages is to formally derive them from abstract machines. Tail-recursive abstract machines provide efficient support for iterative processes via the ordinary procedure call mechanism. This document argues that the use of tail-recursive abstract machines incurs only a small increase in theorem-proving burden when compared with what is required when using ordinary abstract machines. The position is supported by comparing correctness proofs performed using the Boyer-Moore theorem prover.

A by-product of this effort is a syntactic criterion based on tail contexts for identifying which procedure calls must be implemented as tail calls. The concept of tail contexts was used in the latest Scheme Report, the only language specification known to the author that defines the requirement that its implementations must be tail recursive.

Keywords: tail recursion, tail call, SECD machine, CEK machine, verified implementation, Boyer-Moore theorem prover.

*© 1999 Kluwer Academic Publishers. The definitive version of this paper appears in the *Journal of Automated Reasoning*, 23(1):43–62, July 1999. Author's address: The MITRE Corporation, 202 Burlington Road, Bedford, MA, 01730-1420. Email: ramsdell@mitre.org. This document reflects the views of the author and makes no statement about the views of The MITRE Corporation.

1 Introduction

Most programming languages provide special syntax for the specification of iterative processes. The space consumed by the execution of an iterative construct is easily understood. The space used is bounded and independent of the number of iterations executed.

Some programming languages rely on procedure calls to express iterative processes. In these languages, the space used by call chains built out of carefully selected procedure calls is bounded and independent of the number of calls in the chain. An implementation of a programming language that provides this kind of space bound is called tail recursive.

A tail-recursive implementation of a programming language differs from an implementation that performs tail call optimizations. With a tail-recursive implementation, a programmer can rely on the fact that procedure calls that occur in certain syntactic contexts will always be implemented as tail calls. Implementing these calls as tail calls is not an optional optimization, but a correctness requirement of all implementations.

New programming styles are available to programmers using tail-recursive implementations. For example, in continuation-passing style (CPS), nearly every procedure completes by calling another procedure that is required to be implemented as a tail call. When running programs written in continuation-passing style, implementations that fail to meet the requirement quickly run out of space.

Many programming languages require that their implementations be tail recursive; the Scheme programming language being a prime example. When this paper was written, the requirement for Scheme was expressed solely in the following paragraph on page 9 in the IEEE Scheme Standard [11]:

Implementations of Scheme are required to be properly tail-recursive. This allows the execution of an iterative computation in constant space, even if the iterative computation is described by a syntactically recursive procedure. Thus with a tail-recursive implementation, iteration can be expressed using the ordinary procedure-call mechanics, so that special iteration constructs are useful only as abbreviations.

Notice that the requirement describes properties of tail-recursive implementations, but never defines the notion. Yet Scheme implementors seem

to be able to identify a tail-recursive implementation when they see one, and experienced Scheme programmers seem to be able to identify procedure calls that can be used to construct iterative processes in any tail-recursive implementation.

By the time this document was revised, the Scheme community added text to the fifth revision of the Scheme Report [12]. Pages 7–8 clearly define the requirement that its implementations be tail recursive. A formal definition of the requirement can be found in [4].

This document argues that the use of tail-recursive abstract machines incurs only a small increase in theorem-proving burden when compared with what is required when using ordinary abstract machines. The position is supported by comparing correctness proofs performed using the Boyer-Moore theorem prover.

A by-product of this effort is a syntactic criterion based on tail contexts for identifying which procedure calls must be implemented as tail calls. The concept of tail contexts was used in the latest Scheme Report.

The syntactic criterion for identifying tail calls follows naturally from the study of tail-recursive abstract machines. Abstract machines for the lambda value calculus will be the objects studied.

More than twenty years ago, Gordon Plotkin published a famous paper that examines the relation between the lambda calculus and ISWIM, a programming language based on the lambda calculus [13]. ISWIM has an operational semantics given by the SECD machine. The paper includes a proof that the semantics defined by the SECD machine is equivalent to a recursively defined function that implements applicative order evaluation of lambda terms. In that order, the bodies of lambda terms are not evaluated, and the arguments of combinations are evaluated before they are substituted.

Plotkin found that a variation of the lambda calculus produces a theory that naturally defines the computations of the SECD machine. The calculus is enriched with constants and a delta rule that models the actions of primitives in ISWIM. The terms that may be substituted are restricted to being the terms that are not combinations. This restriction on substitution matches the requirement that the arguments of combinations are evaluated before they are substituted. The result is called the lambda value calculus because Plotkin called terms that are not combinations values.

The SECD machine is not tail recursive. Every application of a function to an argument causes the push of information onto the dump, the SECD machine's equivalent of a control stack. The SECD machine implements the

caller saves convention, which means that the callee is free to do as it pleases with the SEC (stack, environment, and code) part of an SECD's machine state.

This document presents a tail-recursive SECD machine (TR-SECD). There are several ways of defining a TR-SECD machine. The TR-SECD machine within implements a callee saves convention, which means that the callee must ensure that a return to the caller results in the restoration of the SEC part of the TR-SECD machine. This version of a TR-SECD machine is motivated by the Scheme 311 byte code [3].

Many efforts aimed at formally deriving efficient implementations of programming languages produce implementations that are not tail recursive. Some start with the SECD machine [8], and some start with the Categorical Abstract Machine [10]. The motivation for this work was to show that using tail-recursive abstract machines results in implementations that efficiently handle iterative processes with only a modest increase in theorem-proving burden.

Support for the thesis is provided by presenting two fully automated correctness proofs. The first proof shows that a recursively defined evaluation function for the lambda value calculus is equal to an evaluation function defined by the SECD machine [14]. The second proof shows that the same recursively defined evaluation function is equal to an evaluation function defined by the TR-SECD machine [15], a variation of the SECD machine that is tail recursive. A study of the two proofs shows that the correctness proof of the TR-SECD machine is only slightly more complex than the one for the SECD machine.

Each correctness proof was performed using the Boyer-Moore theorem prover NQTHM-1992 [2]. The logic of the Boyer-Moore theorem prover is first order, and the system provides extensive support for recursively defined total functions and inductive proofs.

The TR-SECD machine has two transition rules for combinations, one of which is called the tail combination rule and is used in iterative computations. The requirements imposed on proofs in the Boyer-Moore logic lead to a correctness proof of the TR-SECD machine that shows that a grammar naturally identifies tail combinations. This point will be explained in the body of the document.

This document gives a high-level description of the correctness proofs of the two SECD machines. It then describes an encoding of the descriptions in the logic of the Boyer-Moore theorem prover and compares the relative

difficultly of both proofs. The implications of the proof on the use of a grammar to identify tail combinations is next. The same grammar is shown to identify tail combinations in a related tail-recursive abstract machine called the CEK machine. Finally, the application of these ideas to identifying tail calls in Scheme is discussed.

2 The Lambda Value Calculus

The notation used in this document follows that of Plotkin. Some inductive definitions of sets have been replaced by context-free grammars. The set defined by a grammar is the language it generates. A metavariable decorated with a bar stands for both a syntactic category in a grammar and the set defined by it.

The notation used for λ -terms is due to de Bruijn [1]. For machine-generated proofs, the notation is preferable to the standard one because every element in the set of terms in standard notation that differ only because of a change in bound variables corresponds to a single term in de Bruijn notation.

The set of λ -terms \bar{M} is determined by a set of variables, a disjoint set of constants, and improper symbols, λ , (, and). In de Bruijn notation, a variable is a positive integer. Let \bar{K} and \bar{N} be the set (and syntactic category) of constants and variables respectively. The set of λ -terms is defined by

$$\begin{aligned}\bar{M} &\rightarrow \bar{V} \mid (\bar{M}\bar{M}) \\ \bar{V} &\rightarrow \bar{A} \mid \bar{N} \\ \bar{A} &\rightarrow \bar{K} \mid \lambda\bar{M}\end{aligned}$$

A term of the form λM is an abstraction, and one of the form (MM') is a combination. The set of values is \bar{V} and the set of answers is \bar{A} .

The lambda value calculus has two reduction rules. The β -rule corresponds to function application, and the δ -rule defines the actions of primitives. These actions are given by the partial function δ that maps a pair of constants to a closed λ -term.

The β -rule is $(\lambda MV) = \mathbf{subst}(M, V, 1)$, where the result of substituting value V for variable N in M is given by $\mathbf{subst}(M, V, N)$. The definition of substitution using de Bruijn notation is presented in Figures 1 and 2. The definition came directly from a mechanical proof of the Church-Rosser theorem using the Boyer-Moore theorem prover as described in Section 6.4 of [16].

$$\begin{aligned}
&\text{subst} : \text{terms} \times \text{values} \times \text{variables} \rightarrow \text{terms} \\
&\text{subst}(K, V, N) = K \\
&\text{subst}(\lambda M, V, N) \\
&\quad = \lambda \text{subst}(M, \text{bump}(V, 0), N + 1) \\
&\text{subst}(N, V, N') = V, \quad (N = N') \\
&\text{subst}(N, V, N') = N, \quad (N < N') \\
&\text{subst}(N, V, N') = N - 1, \quad (N > N') \\
&\text{subst}((MM'), V, N) \\
&\quad = (\text{subst}(M, V, N) \text{subst}(M', V, N))
\end{aligned}$$
Figure 1: de Bruijn Substitution—`subst`

This section of Shankar’s book contains a careful explanation of de Bruijn notation used here.

$$\begin{aligned}
&\text{bump} : \text{terms} \times \text{naturals} \rightarrow \text{terms} \\
&\text{bump}(K, n) = K \\
&\text{bump}(\lambda M, n) = \lambda \text{bump}(M, n + 1) \\
&\text{bump}(N, n) = N, \quad (N \leq n) \\
&\text{bump}(N, n) = N + 1, \quad (N > n) \\
&\text{bump}((MM'), n) = (\text{bump}(M, n) \text{bump}(M', n))
\end{aligned}$$
Figure 2: de Bruijn Substitution—`bump`

3 The SECD Machines

The first theorem in Plotkin’s paper states that the function `eval` given in Figure 3 defines a semantics for ISWIM programs that is identical to the one defined by the SECD machine. This section presents the definitions needed for the same theorem. The definitions differ from Plotkin’s for several reasons, one of which is the correctness proof is factored into a stage that verifies the correctness of environments and substitution, and a later stage that verifies the correctness of the control structures.

$$\begin{aligned}
\text{eval} &: \text{terms} \rightarrow \text{answers} \\
\text{eval } K &= K \\
\text{eval } \lambda M &= \lambda M \\
\text{eval}(MM') &= \text{eval}(\text{subst}(M'', A, 1)), \\
&\quad (\text{eval } M = \lambda M'' \text{ and } \text{eval } M' = A) \\
\text{eval}(MM') &= \delta(K, K'), \\
&\quad (\text{eval } M = K \text{ and } \text{eval } M' = K')
\end{aligned}$$

Figure 3: Evaluation Function

In addition to λ -terms, the objects manipulated by the SECD machine are given by the following grammar:

$$\begin{array}{ll}
\bar{L} & \rightarrow (\bar{A}, \bar{E}) & \text{Closures} \\
\bar{S} & \rightarrow \text{nil} \mid \bar{L} :: \bar{S} & \text{Stacks} \\
\bar{E} & \rightarrow \text{nil} \mid \bar{L} :: \bar{E} & \text{Environments} \\
\bar{C} & \rightarrow \text{ret} \mid \text{call} :: \bar{C} \mid \bar{M} :: \bar{C} & \text{Code} \\
\bar{D} & \rightarrow \text{halt} \mid (\bar{S}, \bar{E}, \bar{C}, \bar{D}) & \text{Dumps}
\end{array}$$

The term represented by a closure is given by the function `real`, which uses the auxiliary function `butt`. It can easily be shown that the function `real` produces an answer when applied to a closure.

$$\begin{aligned}
\text{real} &: \text{terms} \times \text{environments} \rightarrow \text{terms} \\
\text{real}(M, \text{nil}) &= M \\
\text{real}(M, L :: E) & \\
&= \text{real}(\text{subst}(M, \text{butt}(\text{real } L, E), 1), E)
\end{aligned}$$

$$\begin{aligned}
\text{butt} &: \text{terms} \times \text{environments} \rightarrow \text{terms} \\
\text{butt}(M, \text{nil}) &= M \\
\text{butt}(M, L :: E) &= \text{bump}(\text{butt}(M, E), 0)
\end{aligned}$$

3.1 The SECD Machine

Figure 4 presents the function `step`, the transition function for the SECD machine.

$$\begin{aligned}
&\text{step} : \text{dumps} \rightarrow \text{dumps} \\
&\text{step}(L :: S, E, \text{ret}, (S', E', C', D')) = (L :: S', E', C', D') \\
&\text{step}(S, E, N :: C, D) = (\text{lookup}(N, E) :: S, E, C, D) \\
&\text{step}(S, E, K :: C, D) = ((K, \text{nil}) :: S, E, C, D) \\
&\text{step}(S, E, \lambda M :: C, D) = ((\lambda M, E) :: S, E, C, D) \\
&\text{step}((\lambda M, E') :: L :: S, E, \text{call} :: C, D) \\
&\quad = (\text{nil}, L :: E', M :: \text{ret}, (S, E, C, D)) \\
&\text{step}((K, E') :: (K', E'') :: S, E, \text{call} :: C, D) \\
&\quad = ((\delta(K, K'), \text{nil}) :: S, E, C, D) \\
&\text{step}(S, E, (MM') :: C, D) = (S, E, M' :: M :: \text{call} :: C, D)
\end{aligned}$$

Figure 4: SECD Machine

The function `step` uses the function `lookup` to find the closure associated with a variable.

$$\begin{aligned}
&\text{lookup} : \text{variables} \times \text{environments} \rightarrow \text{closures} \\
&\text{lookup}(1, L :: E) = L \\
&\text{lookup}(N, L :: E) = \text{lookup}(N - 1, E), \quad (N > 1)
\end{aligned}$$

A run of the SECD machine produces an answer when an accepting state is detected. The function `run` is undefined if it reaches an error state, or fails to reach an accepting state for any other reason.

$$\begin{aligned}
&\text{run} : \text{dumps} \rightarrow \text{answers} \\
&\text{run}(L :: S, E, \text{ret}, \text{halt}) = \text{real } L \\
&\text{run } D = \text{run}(\text{step } D)
\end{aligned}$$

3.2 Correctness Proof

The correctness of the SECD machine is expressed as follows:

Theorem 1 (Correctness) *If M is a closed term, then*

$$\text{eval } M = \text{run}(\text{nil}, \text{nil}, M :: \text{ret}, \text{halt}).$$

Some of the important lemmas used to prove the theorem follow. To control the complexity of the proofs, the correctness proof is factored into a stage that verifies the correctness of environments and substitution, and a

later stage that verifies the correctness of the control structures. Figure 5 shows an evaluation function in which substitution is replaced by environment extension.

$$\begin{aligned}
 &\text{reduce} : \text{terms} \times \text{environments} \rightarrow \text{closures} \\
 &\text{reduce}(K, E) = (K, \text{nil}) \\
 &\text{reduce}(\lambda M, E) = (\lambda M, E) \\
 &\text{reduce}(N, E) = \text{lookup}(N, E) \\
 &\text{reduce}((MM'), E) = \text{reduce}(M'', L :: E'), \\
 &\quad (\text{reduce}(M, E) = (\lambda M'', E') \text{ and } \text{reduce}(M', E) = L) \\
 &\text{reduce}((MM'), E) = (\delta(K, K'), \text{nil}), \\
 &\quad (\text{reduce}(M, E) = (K, E') \text{ and } \text{reduce}(M', E) = (K', E''))
 \end{aligned}$$

Figure 5: Reduction Function

Lemma 1 gives the key property associated with the replacement of substitution by environment extension.

Lemma 1 (Substitution) *If $\text{real}(\lambda M, E) = \lambda M'$, then*

$$\text{subst}(M', \text{real } L, 1) = \text{real}(M, L :: E).$$

$$\begin{aligned}
 &\text{isbound} : \text{terms} \times \text{naturals} \rightarrow \text{booleans} \\
 &\text{isbound}(K, n) = \text{true} \\
 &\text{isbound}(\lambda M, n) = \text{isbound}(M, n + 1) \\
 &\text{isbound}(N, n) = N \leq n \\
 &\text{isbound}((MM'), n) = \text{isbound}(M, n) \text{ and } \text{isbound}(M', n)
 \end{aligned}$$

Figure 6: Bound Predicate

The analog of an answer is called an answer closure. It is defined in terms of answer environments.

Definition 1 (Answer Environment) *An answer environment is defined inductively.*

1. *nil is an answer environment*

2. $(A, E) :: E'$ is an answer environment if

- $\text{isbound}(A, \text{length } E)$
- both E and E' are answer environments.

A term M is said to be closed in environment E iff E is an answer environment and $\text{isbound}(M, \text{length } E)$. A closure (A, E) is said to be an answer closure iff A is closed in E .

The following lemma ends the stage of the proofs that verifies the correctness of environments and substitution.

Lemma 2 (Reduction) *Assume E is an answer environment and M is a term closed in E . If $\text{eval}(\text{real}(M, E))$ is undefined, then $\text{reduce}(M, E)$ is undefined, otherwise*

1. $\text{eval}(\text{real}(M, E)) = \text{real}(\text{reduce}(M, E))$,
2. $\text{reduce}(M, E)$ is an answer closure.

The stage that verifies the correctness of the control structures consists of two main lemmas.

Lemma 3 (Adequacy) *If E is a value environment, M is a term closed in E , and $\text{reduce}(M, E)$ is defined, then*

$$\text{run}(S, E, M :: C, D) = \text{run}(\text{reduce}(M, E) :: S, E, C, D).$$

Lemma 4 (Faithfulness) *If E is a value environment, M is a term closed in E , and $\text{reduce}(M, E)$ is not defined, then $\text{run}(S, E, M :: C, D)$ is not defined.*

The proof of Theorem 1 has two cases. If M is a closed term and $\text{reduce}(M, \text{nil})$ is defined, then

$$\begin{aligned} & \text{run}(\text{nil}, \text{nil}, M :: \text{ret}, \text{halt}) \\ &= \text{run}(\text{reduce}(M, \text{nil}) :: \text{nil}, \text{nil}, \text{ret}, \text{halt}) && \text{by Lemma 3} \\ &= \text{real}(\text{reduce}(M, \text{nil})) && \text{by function run} \\ &= \text{eval}(\text{real}(M, \text{nil})) && \text{by Lemma 2} \\ &= \text{eval } M && \text{by function real} \end{aligned}$$

If M is a closed term and $\text{reduce}(M, \text{nil})$ is not defined, then Lemma 4 and Lemma 2 apply.

3.3 The TR-SECD Machine

Objects manipulated by the TR-SECD machine differ only in the fact that code sequences differ. In the TR-SECD machine, no code may follow the call symbol.

$$\bar{C} \rightarrow \text{ret} \mid \text{call} \mid \bar{M} :: \bar{C} \quad \text{Code}$$

The function step' is the transition function for the TR-SECD machine. Notice the first four clauses are identical to the ones that define the SECD transition function given in Figure 4.

$$\begin{aligned} \text{step}' : \text{dumps} &\rightarrow \text{dumps} \\ \text{step}'(L :: S, E, \text{ret}, (S', E', C', D')) &= (L :: S', E', C', D') \\ \text{step}'(S, E, N :: C, D) &= (\text{lookup}(N, E) :: S, E, C, D) \\ \text{step}'(S, E, K :: C, D) &= ((K, \text{nil}) :: S, E, C, D) \\ \text{step}'(S, E, \lambda M :: C, D) &= ((\lambda M, E) :: S, E, C, D) \\ \text{step}'((\lambda M, E') :: L :: S, E, \text{call}, D) &= (S, L :: E', M :: \text{ret}, D) \\ \text{step}'((K, E') :: (K', E'') :: S, E, \text{call}, D) &= ((\delta(K, K'), \text{nil}) :: S, E, \text{ret}, D) \\ \text{step}'(S, E, (MM') :: \text{ret}, D) &= (S, E, M' :: M :: \text{call}, D) \\ \text{step}'(S, E, (MM') :: C, D) &= (\text{nil}, E, M' :: M :: \text{call}, (S, E, C, D)), \quad (C \neq \text{ret}) \end{aligned}$$

Figure 7: Tail-Recursive SECD Machine

When the code is a combination, one of the last two clauses may apply. The last clause pushes a dump, so name it the push dump combination rule, and name the penultimate clause the tail combination rule. In the TR-SECD machine, iterative processes will be produced by computations that use only the tail combination rule.

The statement of the correctness theorem, as well as the supporting lemmas remains unchanged with the exception of the Adequacy lemma.

Lemma 5 (TR-SECD Adequacy) *If E is a value environment, M is a term closed in E , and $\text{reduce}(M, E)$ is defined, then*

$$\text{run}(S, E, M :: C, D) = \text{run}(\text{reduce}(M, E) :: S, E', C, D)$$

where $E' = \text{runenv}(M, E, C = \text{ret})$ and the function runenv is defined in Figure 8.

$$\begin{aligned}
\text{runenv} &: \text{terms} \times \text{environments} \times \text{booleans} \rightarrow \text{environments} \\
\text{runenv}(K, E, B) &= E \\
\text{runenv}(\lambda M, E, B) &= E \\
\text{runenv}(N, E, B) &= E \\
\text{runenv}((MM'), E, \text{true}) &= \text{runenv}(M'', L :: E', \text{true}), \\
&\quad (\text{reduce}(M, E) = (\lambda M'', E') \text{ and } \text{reduce}(M', E) = L) \\
\text{runenv}((MM'), E, \text{true}) &= E, \\
&\quad (\text{reduce}(M, E) = (K, E') \text{ and } \text{reduce}(M', E) = (K', E'')) \\
\text{runenv}((MM'), E, \text{false}) &= E
\end{aligned}$$

Figure 8: Run Environment Function

4 The Boyer-Moore Theorem Prover

The logic of the Boyer-Moore theorem prover (NQTHM) is a quantifier free, first-order logic with recursive functions. The lack of quantifiers was handled by replacing each existentially quantified variable by a function that produces a witness.

All the functions used in the proofs are total, yet most of the functions presented in preceding sections are partial. Many of the partial functions can be extended to become total functions by adding clauses that produce an error value distinct from the ordinary values.

Since zero is often a default value in NQTHM, it was usually selected as the error value in the encoding of the descriptions used in NQTHM. For example, the function `lookup` is made total by returning zero when the function is given something other than a positive integer as a variable. The special status of zero in NQTHM is part of the reason variables are encoded as the positive integers, not natural numbers as one might expect.

There is no extension of the function `eval` that is total. Instead, the function `ev` in Figure 9 is used.

The function `eval` is defined at M if `ev`(M, n) is an answer for some natural number n . Furthermore, `eval` $M = \text{ev}(M, n)$ whenever `ev`(M, n) is an answer.

The statement of the correctness theorems includes an implicit quantification over all δ functions that have certain properties. The Boyer-Moore theorem prover has a `CONSTRAIN` event suited to this task.

Given the lack of support for partial functions in the Boyer-Moore theorem prover, one might think another theorem prover would be better for this

$$\begin{aligned}
\text{ev} &: \text{terms} \times \text{naturals} \rightarrow \text{answers} \cup \{0\} \\
\text{ev}(M, 0) &= 0 \\
\text{ev}(K, n) &= K \\
\text{ev}(\lambda M, n) &= \lambda M \\
\text{ev}((MM'), n) &= \text{ev}(\text{subst}(M'', A, 1), n - 1), \\
&\quad (\text{ev}(M, n - 1) = \lambda M'' \text{ and } \text{ev}(M', n - 1) = A) \\
\text{ev}((MM'), n) &= \delta(K, K'), \\
&\quad (\text{ev}(M, n - 1) = K \text{ and } \text{ev}(M', n - 1) = K')
\end{aligned}$$

Figure 9: Total Evaluation Function

job. The Boyer-Moore theorem prover's ability to find inductive proofs was exploited and some of the proofs about the lambda calculus were taken verbatim from a proof of the Church-Rosser theorem [16], which comes with the theorem prover. A proof of the correspondence between the lambda calculus in standard notation and in de Bruijn notation is part of the Church-Rosser proof.

4.1 The Boyer-Moore Proofs

Each of the two proofs is divided into the same six topics. The first section defines λ -terms using de Bruijn notation, substitution, and evaluation. The second section defines a total function based on `reduce` by following the procedure used to create `ev` from `eval` and concludes with a proof of the Substitution lemma and the Reduction lemma. The first two sections of each proof are identical.

The third section defines the SECD machine. The TR-SECD machine is defined by the transition function `step'`, and the original SECD machine is defined by the transition function `step`. The encoding of the function `run` maps a dump and a natural number to a dump. The natural number is called the step count.

$$\begin{aligned}
\text{run}' &: \text{dumps} \times \text{naturals} \rightarrow \text{dumps} \\
\text{run}'(D, 0) &= D \\
\text{run}'(D, n) &= \text{run}'(\text{step } D, n - 1)
\end{aligned}$$

The fourth section defines a timed version of the total function based on `reduce`. This function yields both a closure and the step count required to

compute the closure. The major lemma in this section, the Timed Reduction lemma, establishes that the closure produced by the timed reduction function is the same as the one produced by `reduce`. The TR-SECD machine version of timed reduction is slightly more complex than the original SECD machine version because the step count differs depending on which rule is used for a combination. The two combination rules are the last two clauses of the transition function `step'` in Figure 7.

The fifth section contains a proof of the TR-SECD or the SECD Adequacy lemma. The sixth and final section contains a proof of the Faithfulness lemma.

4.2 A Comparison of the Proofs

The differences between the correctness proofs of the two machines can be put into three categories.

1. Some of the function definitions must differ to account for the tail-recursiveness of the TR-SECD machine. The function defining the transition function and the function defining timed reduction are examples.
2. The hints used to tell the theorem prover how to perform complex inductions must differ because the ones for the TR-SECD machine require additional cases for tail calls. An understanding of the TR-SECD machines makes the definition of the hints straightforward.
3. Occasionally, functions with no counterpart are required for the TR-SECD correctness proof. The function `runenv` is an example. The required definition of each of these functions was obvious.

While the differences between the inputs to the theorem prover are not large, the difference in the number of cases examined by the theorem prover is large. This fact is reflected in the CPU time used while performing the proofs. The correctness proof for the TR-SECD machine requires roughly 50% more CPU time than that used by the proof for the SECD machine. The theorem prover easily handles the extra cases, but the cases take a substantial amount of additional time.

The authors of NQTHM suggest that proofs of difficult theorems be decomposed into proofs of many small lemmas in order to control the theorem

Lemma	CPU time (minutes)	
	TR-SECD	SECD
Reduction	3.3	3.3
Adequacy	3.1	1.6
Timed Reduction	2.4	1.1
Entire proof	11.4	7.4

Table 1: Proof Times

prover. The authors state that the theorem prover should be “kept on a short leash” in Chapter 9 of [2]. In particular, at no time should the theorem prover be allowed to spend a long time proving one lemma.

The SECD and TR-SECD proofs are organized around setting up the proof of several time consuming lemmas. The proofs of the Reduction, Adequacy, and Timed Reduction lemmas each use over a minute of CPU time on a 200 MHz. Pentium Pro-based PC running Linux. Table 1 gives the CPU time required to prove selected lemmas.

Readers with experience with the Boyer-Moore theorem prover are invited to read the file of events for the TR-SECD [15] and SECD [14] correctness proofs. The section on the Adequacy lemma succinctly provides support to the thesis of this work.

4.3 Proofs and Tail Recursion

The proofs of the adequacy lemmas reveals the intuition behind calling the TR-SECD machine tail recursive. For both machines, the `ret` rule is the only way to reduce the depth of the dump. In the SECD machine, a dump is pushed by a use of the `call` abstraction rule. While specifying the details of the SECD Adequacy lemma, one finds that each use of a `call` abstraction rule is matched by a use of a `ret` rule.

In the TR-SECD machine, the `call` abstraction rule does not push dumps. Instead, one of the two combination rules pushes dumps. As one might expect, the proof of the TR-SECD Adequacy lemma reveals that each use of the push dump combination rule is matched by a use of a `ret` rule.

Recall that the timed reduce function produces a pair consisting of a closure and a step count. The step count assigned to a combination by the timed reduce function for the TR-SECD machine depends on the combination rule

used. The function uses an additional parameter to encode this information. To avoid extraneous details associated with computing step counts, a timed reduce function that returns just a closure is used in this presentation. The key clause is displayed.

$$\begin{aligned} & \text{reduce}' : \text{terms} \times \text{environments} \times \text{booleans} \rightarrow \text{closures} \\ & \dots \\ & \text{reduce}'((MM'), E, B) = \text{reduce}'(M'', L :: E', \text{true}), \\ & \quad (\text{reduce}'(M, E, \text{false}) = (\lambda M'', E') \text{ and } \text{reduce}'(M', E, \text{false}) = L) \end{aligned}$$

The Boolean parameter is used only to compute the step count—the closure computed by the timed reduce function does not depend on the parameter. The change to the adequacy lemma is simple.

Lemma 6 (Timed TR-SECD Adequacy) *If E is a value environment, M is a term closed in E , and $\text{reduce}'(M, E, C = \text{ret})$ is defined, then*

$$\text{run}(S, E, M :: C, D) = \text{run}(\text{reduce}'(M, E, C = \text{ret}) :: S, E', C, D)$$

where $E' = \text{runenv}(M, E, C = \text{ret})$.

The proof of this lemma makes it clear that when reduce' is applied to a combination, the tail combination rule is used when the Boolean parameter is true, otherwise the push dump combination rule is used.

The form of the definition of the function reduce' allows a stronger statement about the selection between the two combination rules. The combination rule used for each combination in a term can be determined by a simple analysis of the term. The following grammar partitions combinations into two categories. Tail combinations are marked with angle brackets while push dump combinations retain the original syntax for combinations.

$$\begin{aligned} \bar{T} & \rightarrow \bar{V} \mid \langle \bar{U}\bar{U} \rangle && \text{tail combinations and values} \\ \bar{U} & \rightarrow \bar{V} \mid (\bar{U}\bar{U}) && \text{push dump combinations and values} \\ \bar{V} & \rightarrow \bar{K} \mid \lambda \bar{T} \mid \bar{N} && \text{values} \\ \bar{M} & \rightarrow \bar{V} \mid \langle \bar{U}\bar{U} \rangle \mid (\bar{U}\bar{U}) && \text{marked terms} \end{aligned}$$

With this added structure, the reduction function becomes

$$\begin{aligned} & \text{reduce}'' : \text{marked terms} \times \text{environments} \times \text{booleans} \rightarrow \text{closures} \\ & \dots \\ & \text{reduce}''(\langle UU' \rangle, E, \text{true}) = \text{reduce}''(T, L :: E', \text{true}), \\ & \quad (\text{reduce}''(U, E, \text{false}) = (\lambda T, E') \text{ and } \text{reduce}''(U', E, \text{false}) = L) \\ & \text{reduce}''((UU'), E, \text{false}) = \text{reduce}''(T, L :: E', \text{true}), \\ & \quad (\text{reduce}''(U, E, \text{false}) = (\lambda T, E') \text{ and } \text{reduce}''(U', E, \text{false}) = L) \end{aligned}$$

As expected, when `reduce''` is applied to a tail combination, the Boolean parameter is true, and when `reduce''` is applied to a push dump combination, the Boolean parameter is false.

Once terms are marked, the Boolean parameter to the timed reduce function is redundant because the step count for values does not depend on the Boolean parameter.

The tail combinations identified by the above grammar play distinguished roles in other works. For example, Figure 3 in [5] presents a CPS transform that treats tail combinations specially.

Contexts provide an alternative to a grammar for identifying tail combinations. A combination context is a λ -term when its hole is replaced by a combination. The set of combination contexts is defined by

$$\bar{X} \rightarrow [] \mid \lambda\bar{X} \mid (\bar{X}\bar{M}) \mid (\bar{M}\bar{X})$$

The term that results from substituting combination (MM') into the hole in context X is $X[(MM')]$.

A proper subset of combination contexts is the set \bar{Y} , the set of tail contexts:

$$\begin{aligned} \bar{Y} &\rightarrow [] \mid \bar{Z} \\ \bar{Z} &\rightarrow \lambda\bar{Y} \mid (\bar{Z}\bar{M}) \mid (\bar{M}\bar{Z}) \end{aligned}$$

Given a term $M = X[(M'M'')]$, $(M'M'')$ is a tail combination if X is a tail context, otherwise it is a push dump combination. Tail contexts motivate the form of the proposed text for identifying tail calls in Scheme presented in Section 6.

5 The CEK Machine

The adequacy proof for the TR-SECD machine shows that a simple syntactic analysis is all that is needed to identify which of its two combination rules is applicable to a combination. The focus of this section is a tail-recursive machine that does not seem to distinguish between the two types of combinations, however, a careful analysis of the machine shows the distinction is important for it too.

The CEK machine [6] is similar to the SECD machine but closer to a denotation semantics of ISWIM. The states of the machine are triples composed of a term, an environment, and a continuation code which represents

the remainder of the computation. Figure 10 defines a CEK machine assuming terms are unmarked.

$$\begin{array}{ll} \bar{R} & \rightarrow \text{halt} \mid (\text{fun}, \bar{M}, \bar{E}, \bar{R}) \mid (\text{arg}, \bar{L}, \bar{R}) & \text{continuation code} \\ \bar{W} & \rightarrow (\bar{M}, \bar{E}, \bar{R}) & \text{CEK state} \end{array}$$

$$\begin{array}{l} \text{cekstep} : \text{CEK state} \rightarrow \text{CEK state} \\ \text{cekstep}((MM'), E, R) = (M', E, (\text{fun}, M, E, R)) \\ \text{cekstep}(A, E, (\text{fun}, M, E', R)) = (M, E', (\text{arg}, (A, E), R)) \\ \text{cekstep}(\lambda M, E, (\text{arg}, L, R)) = (M, L :: E, R) \\ \text{cekstep}(K, E, (\text{arg}, (K', E'), R)) = (\delta(K, K'), \text{nil}, R) \\ \text{cekstep}(N, E, R) = (A, E', R) \quad ((A, E') = \text{lookup}(N, E)) \end{array}$$

$$\begin{array}{l} \text{cekrun} : \text{CEK states} \rightarrow \text{answers} \\ \text{cekrun}(A, E, \text{halt}) = \text{real}(A, E) \\ \text{cekrun } W = \text{cekrun}(\text{cekstep } W) \end{array}$$

Figure 10: CEK Machine

The results in [6] and [13] prove the CEK Correctness theorem.

Theorem 2 (CEK Correctness) *If M is a closed unmarked term,*

$$\text{eval } M = \text{cekrun}(M, \text{nil}, \text{halt}).$$

5.1 Tail Recursion

A hint as to how the CEK machine distinguishes between the two types of combinations comes from studying its relation to the TR-SECD machine. In TR-SECD machine runs that produce a value from the prescribed initial state, the length of the stack is never greater than two. As a result, the TR-SECD machine pushes dumps in only one of two forms:

$$\begin{array}{l} (\text{nil}, E, M :: \text{call}, D) \\ (L :: \text{nil}, E, \text{call}, D) \end{array}$$

These two forms correspond to continuing by evaluating the function, and continuing by applying the function value to the argument value—the two ways of continuing in the CEK machine. In the TR-SECD machine, the rule

used by push dump combinations is the only means of increasing the depth of the dump. Perhaps transitions involving push dump combinations are the only way that the CEK machine increases some measure of the depth of its continuation code.

The trick to showing that the CEK machine distinguishes between the types of combinations is to use both marked terms and marked continuation codes. A mark continuation that behaves as an identity continuation has been added.

$$\begin{array}{ll}
 \bar{P} \rightarrow \text{halt} \mid (\text{mark}, \bar{Q}) & \text{marking codes} \\
 \bar{Q} \rightarrow (\text{fun}, \bar{U}, \bar{E}, \bar{P}) \mid (\text{arg}, \bar{L}, \bar{P}) & \text{combination codes} \\
 \bar{R} \rightarrow \bar{P} \mid \bar{Q} & \text{continuation codes}
 \end{array}$$

Notice that the continuation in a mark code is a combination code, and the continuation in a combination code is a marking code. As a result, a mark code separates every pair of combination codes. The depth of a continuation code is the number of mark continuations it contains.

$$\begin{array}{l}
 \text{cekstep}' : \text{marked CEK state} \rightarrow \text{marked CEK state} \\
 \text{cekstep}'(\langle UU' \rangle, E, P) = (U', E, (\text{fun}, U, E, P)) \\
 \text{cekstep}'(\langle UU' \rangle, E, Q) = (U', E, (\text{fun}, U, E, (\text{mark}, Q))) \\
 \text{cekstep}'(A, E, (\text{fun}, U, E', P)) = (U, E', (\text{arg}, (A, E), P)) \\
 \text{cekstep}'(\lambda T, E, (\text{arg}, L, P)) = (T, L :: E, P) \\
 \text{cekstep}'(K, E, (\text{arg}, (K', E'), P)) = (\delta(K, K'), \text{nil}, P) \\
 \text{cekstep}'(N, E, R) = (A, E', R) \quad ((A, E') = \text{lookup}(N, E)) \\
 \text{cekstep}'(A, E, (\text{mark}, Q)) = (A, E, Q)
 \end{array}$$

Figure 11: Marked CEK Machine

The transition function of a CEK machine that uses both marked terms and marked continuation codes is given in Figure 11. A state of the Marked CEK machine with a tail combination always has a marking continuation, and a state with a push dump combination always has a combination continuation. As with the TR-SECD machine, the depth of the continuation code is only increased by the push dump combination rule. In the Marked CEK machine, iterative processes will be produced by computations that use only its tail combination rule.

Erasing the marks in the Marked CEK machine produces the original CEK machine. The implication is that CEK machine treats both kinds of

combinations differently even though it has one rule for both kinds of combinations. In particular, only push dump combinations can lead to unbounded growth in the depth of the continuation code.

5.2 The Boyer-Moore Proof Attempts

Both the CEK and the TR-SECD machines are tail recursive, but the CEK machine seems to have many advantages. Its transition function has fewer rules and it manipulates fewer classes of objects. Both of these features lead to a reduction in the number of cases required by automated proofs.

The TR-SECD machine is similar to the abstract machines used in the Vliisp Project [9], a project that produced a verified implementation of the Scheme programming language in the sense that the algorithms used by the implementation were formally verified. Since the author contributed to this effort, the TR-SECD machine was a natural starting point for this work. Furthermore, the TR-SECD machine would be the natural starting point of any effort to refine the implementation verified at the algorithm level into one verified at the code level.

It is lucky that the TR-SECD machine was chosen for this work. Attempts to produce an NQTHM correctness proof of the CEK machine of the form produced for the SECD machines failed.

The adequacy lemma is an equation. The inductive proof involves a case for combinations in which the function part reduces to a lambda term. In this case, there are three induction hypotheses, one for the evaluation of the argument, one for the evaluation of the function, and one for the application of the lambda term to the argument value. The hypotheses must be used in the order presented.

In the proof of the TR-SECD Adequacy lemma, NQTHM has no other options but to use the hypotheses in the correct order. The existence of the `ret` rule makes it impossible to use them in the reverse order.

Attempts by the author to prove the CEK Adequacy lemma failed because NQTHM chose to use the induction hypotheses in reverse order. The elegance of the CEK machine increases the opportunities for a wrong step, and the large step nature of the proof makes it hard to construct the required constraint on the theorem prover.

6 Scheme and Tail Recursion

This section relates previous work to this work. The programming language Scheme [11] was the subject of the work relevant to this section. In what follows, a simplification of the Scheme's abstract syntax will be used to present Scheme related results. The syntax of a core of Scheme is

$$\begin{aligned}\bar{M} &\rightarrow \bar{V} \mid (\text{set! } \bar{I} \bar{M}) \mid (\text{if } \bar{M}_1 \bar{M}_2 \bar{M}_3) \mid (\bar{M} \bar{M}_1 \dots \bar{M}_n) \\ \bar{V} &\rightarrow \bar{K} \mid \bar{I} \mid (\text{lambda } (\bar{I}_1 \dots \bar{I}_n) \bar{M})\end{aligned}$$

where \bar{K} is the syntactic category of Scheme constants and \bar{I} is the syntactic category of identifiers.

The grammar that partitions Scheme procedure calls into two categories is specified by three production rules. The start symbol is \bar{T} .

$$\begin{aligned}\bar{T} &\rightarrow \bar{V} \mid (\text{set! } \bar{I} \bar{U}) \mid (\text{if } \bar{U}_1 \bar{T}_1 \bar{T}_2) \mid \langle \bar{U} \bar{U}_1 \dots \bar{U}_n \rangle \\ \bar{U} &\rightarrow \bar{V} \mid (\text{set! } \bar{I} \bar{U}) \mid (\text{if } \bar{U}_1 \bar{U}_2 \bar{U}_3) \mid (\bar{U} \bar{U}_1 \dots \bar{U}_n) \\ \bar{V} &\rightarrow \bar{K} \mid \bar{I} \mid (\text{lambda } (\bar{I}_1 \dots \bar{I}_n) \bar{T})\end{aligned}$$

The syntax defined by \bar{T} is the same as the original syntax except that procedure calls generated by \bar{T} are delimited using angle brackets. The marked procedure calls are the tail calls. Mitchell Wand [17] used a grammar of this form to produce a syntax directed compiler that is tail recursive by omitting the push of a stack frame when compiling tail calls.

Another syntactic approach to identifying tail calls is presented in [7]. The paper presents a set of source-to-source reduction rules that transform a core of Scheme into something called A-Normal Form, which is a good intermediate representation for compilers. Procedure calls are naturally partitioned into two classes when transformed into A-Normal Form.

It appears that the set of procedure calls that are transformed into A-Normal Form tail calls is the same as the set of procedure calls marked by \bar{T} . No formal proof has been attempted, however, the following observation provides support for the proposition.

A core Scheme expression may be translated into A-Normal Form via a sequence of A-reductions. An evaluation context is used to identify an expression to which an A-reduction applies. The use of Scheme expressions marked by the above grammar makes it is easy to see that A-reductions apply to non-value expressions generated by the syntactic category \bar{U} . Furthermore, A-reductions transform procedure calls generated by \bar{U} into A-Normal Form calls that are not tail calls.

Several different efforts have identified the same set of procedure calls within an expression as tail calls. The efforts include correctness proofs of tail-recursive machines, tail-recursive CPS transformations, tail-recursive syntax directed compilers, and, most likely, transformation into A-Normal Form. It is time that programming language definitions that require tail-recursive implementations do so by identifying tail calls and specifying constraints on how they may be implemented.

Tail calls could be identified using a grammar as above, however, tail contexts presented in Section 4.3 suggest a better method. Tail contexts naturally suggest that the surrounding in which a procedure call occurs determines whether it is a tail call.

Given a Scheme program M , the goal is to identify the subexpressions of M that surround tail calls. Name these subexpressions tail contexts. An inductive definition gives the subexpressions that are the tail contexts of a given Scheme program M .

- The expression M is a tail context of M .
- If an expression of the form $(\text{lambda } (I \dots) T)$ occurs in M , then the expression T is a tail context of M .
- If an expression of the form $(\text{if } M' T T')$ is a tail context of M , then both T and T' are ones.
- If an expression of the form $(M' M'' \dots)$ is a tail context of M , then it is a tail call.

This work served as the motivation for the author's request that the new revision of the Scheme Report [12] include text that clearly defines the requirement that its implementations be tail recursive. By the time this document was revised, the Scheme community added text that uses tail contexts to identify procedures that must be implemented using tail calls. Furthermore, William Clinger gave a formal definition of the requirement [4].

7 Conclusion

Tail-recursive abstract machines provide efficient support for iterative processes via the ordinary procedure call mechanism. This document showed that the use of tail-recursive abstract machines incurs only a small increase

in theorem-proving burden when compared with what is required when using ordinary abstract machines. The position was supported by comparing correctness proofs performed using the Boyer-Moore theorem prover. A careful study of the correctness proof of tail-recursive abstract machines resulted in a method for identifying tail calls using tail contexts. This method is intuitive and easily understood.

Acknowledgements

This document was substantially improved as a result of correspondence with Profs. Matthias Felleisen, William Clinger, and Robert Boyer. Joshua Guttman provided valued detailed comments that also improved this document.

References

- [1] H. P. Barendregt. *The Lambda Calculus, Its Syntax and Semantics*. North-Holland, Amsterdam, revised edition, 1984.
- [2] Robert S. Boyer and J Strother Moore. *A Computational Logic Handbook*. Academic Press, 1988.
- [3] William D. Clinger. The Scheme 311 compiler: An exercise in denotational semantics. In *1984 ACM Symposium on Lisp and Functional Programming*, pages 356–364, New York, August 1984. The Association for Computing Machinery, Inc.
- [4] William D. Clinger. Proper tail recursion and space efficiency. *ACM SIGPLAN Notices*, 33(5):174–185, May 1998. Proc. SIGPLAN '98 Conference on Programming Language Design and Implementation.
- [5] Olivier Danvy and Andrzej Filinski. Representing control, a study of the CPS transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, December 1992.
- [6] M. Felleisen and D. Friedman. Control operators, the SECD-machine, and the lambda calculus. In M. Wirsing, editor, *Formal Description of Programming Concepts III*, pages 193–217. Elsevier Science Publishers B. V. (North-Holland), Amsterdam, 1987.

- [7] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. *ACM SIGPLAN Notices*, 28(6):237–247, June 1993. Proc. SIGPLAN '93 Conference on Programming Language Design and Implementation.
- [8] Brian T. Graham. *The SECD Machine: A Verification Case Study*. SECS178. Kluwer Academic Publishers, Dordrecht, 1992.
- [9] Joshua D. Guttman and Mitchell Wand (eds.). *VLISP: A Verified Implementation of Scheme*. Kluwer Academic Publishers, Dordrecht, 1995. Contents identical to *Lisp and Symbolic Computation*, Vol. 8, Nos. 1 & 2, special double issue devoted to the results of VLISP.
- [10] John Hannan. Making abstract machines less abstract. In J. Hughes, editor, *Lecture Notes in Computer Science*, volume 524, pages 618–635. Springer-Verlag, 1991. Proc. of the 5th ACM Conference on Function Programming Languages and Computer Architecture.
- [11] IEEE Std 1178-1990. *IEEE Standard for the Scheme Programming Language*. Institute of Electrical and Electronic Engineers, Inc., New York, NY, 1991.
- [12] Richard Kelsey, William Clinger, and Jonathan Rees, editors. Revised⁵ report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, 1998.
- [13] Gordon D. Plotkin. Call-by-name, call-by-value and the λ -calculus. *Theoretical Computer Science*, 1:125–159, 1975.
- [14] John D. Ramsdell. SECD events. NQTHM event file, September 1996. <ftp://ftp.cs.utexas.edu/pub/boyer/nqthm/trsecd/secd.events>.
- [15] John D. Ramsdell. TR-SECD events. NQTHM event file, September 1996. <ftp://ftp.cs.utexas.edu/pub/boyer/nqthm/trsecd/trsecd.events>.
- [16] N. Shankar. *Metamathematics, Machines, and Goedel's Proof*. Cambridge University Press, revised edition, 1994.
- [17] Mitchel Wand. Correctness of procedure representations in higher-order assembly language. In S. Brookes, M. Main, A. Melton, M. Mislove, and D. Schmidt, editors, *Lecture Notes in Computer Science*, volume 529, pages 294–311. Springer-Verlag, 1991. Mathematical Foundations of Programming Semantics.