Proper Tail Recursion and Space Efficiency

William D Clinger
Northeastern University
will@ccs.neu.edu

Abstract

The IEEE/ANSI standard for Scheme requires implementations to be *properly tail recursive*. This ensures that portable code can rely upon the space efficiency of continuation-passing style and other idioms. On its face, proper tail recursion concerns the efficiency of procedure calls that occur within a tail context. When examined closely, proper tail recursion also depends upon the fact that garbage collection can be asymptotically more space-efficient than Algol-like stack allocation.

Proper tail recursion is not the same as ad hoc tail call optimization in stack-based languages. Proper tail recursion often precludes stack allocation of variables, but yields a well-defined asymptotic space complexity that can be relied upon by portable programs.

This paper offers a formal and implementation-independent definition of proper tail recursion for Scheme. It also shows how an entire family of reference implementations can be used to characterize related safe-for-space properties, and proves the asymptotic inequalities that hold between them.

1 Introduction

Tail recursion is a phrase that has been used to refer to various syntactic notions, to particular techniques for implementing syntactic tail recursion, and to the space efficiency of those techniques. Syntactically, a call is a *tail call* if it appears within a function body that can reduce to the call; this is formalized in Section 2. Since the complete call graph is seldom available, a tail call is often said to be *tail recursive* regardless of whether it occurs within a cycle in the call graph.

Scheme, Standard ML, and several other mostly-functional languages rely heavily on the efficiency of tail recursion. Common idioms, notably continuation-passing style (CPS), would quickly run out of stack space if tail calls were to consume space. To ensure that portable code can rely upon these idioms, the IEEE standard for Scheme [IEE91] says

> Implementations of Scheme are required to be properly tail-recursive [Ste78]. This allows the execution of an iterative computation in constant space, even if the iterative computation is described by a syntactically recursive procedure.

E	::=	`(quote` c`)`	constants
	|	I	variable references
	|	L	lambda expressions
	|	`(if` E_0 E_1 E_2`)`	conditional expressions
	|	`(set!` I E_0`)`	assignments
	|	`(`E_0 E_1 `...)`	procedure calls
L	::=	`(lambda (`I_1 `...)` E`)`	
c	::=	TRUE | FALSE | NUM:z | SYM:I	
	|	VEC:$\langle\alpha_0, \ldots\rangle$ | ...	
z	$\in$	**Number**	
α, β	$\in$	**Location**	
I	$\in$	**Identifier**	

Figure 1: Internal syntax of Core Scheme.

The standard's citation refers to a technical report that uses CPS-conversion to explain what proper tail recursion meant in the context of the first Scheme compiler [Ste78]. That explanation is formally precise, but it is not entirely clear how it applies to an implementation that uses a different algorithm for CPS-conversion or does not use CPS-conversion at all. Most attempts to characterize proper tail recursion in a truly implementation-independent way have been more informal [Ste78]:

> Intuitively, function calls do not "push control stack"; instead, it is argument evaluation which pushes control stack.

Using a style of definition proposed by Morrisett and Harper [MH97], this paper defines a set of asymptotic space complexity classes that characterize proper tail recursion and several related safe-for-space complexity properties. Although these complexity classes are defined in terms of specific reference implementations, they can be used without depending upon the details or even the existence of implementation-dependent data structures such as stacks or heaps. This provides a solid foundation for reasoning about the asymptotic space complexity of Scheme programs, and also provides implementors with a formal basis for determining whether potential optimizations are safe with respect to proper tail recursion.

Program	Language	Lines	Calls	Tail calls	Self-tail calls
Larceny	Scheme	27986	10782	30.0%	$\leq$ 8.3%
lalr A	Scheme	1576	702	31.8%	$\leq$ 22.8%
lalr B	Scheme	1397	569	25.3%	$\leq$ 10.2%
dynamic	Scheme	2343	1139	24.1%	$\leq$ 0.7%
nbody	Scheme	1428	621	18.4%	$\leq$ 7.2%
nucleic2	Scheme	3740	354	9.3%	$\leq$ 1.7%
lcc	C	25477	4492	6.5%	4.2%
jpeg	C	29746	2122	6.2%	5.4%
grep	C	11277	656	4.6%	3.7%

Figure 2: Static frequency of tail calls. These numbers were obtained by instrumenting two compilers: `lcc` and Twobit [FH95, CH94]. The self-tail calls shown for Scheme include all tail calls to known closures, because Twobit has no reason to recognize self-tail calls as a special case. See also Section 14.

2 Tail Calls

Figure 1 shows an internal syntax for the core of Scheme. The external syntax of full Scheme can be converted into this internal syntax by expanding macros and by replacing vector, string, and list constants by references to constant storage.

Definition 1 *The* tail expressions *of a program written in Core Scheme are defined inductively as follows.*

1. *The body of a lambda expression is a tail expression.*
2. *If* (`if` E_0 E_1 E_2) *is a tail expression, then both* E_1 *and* E_2 *are tail expressions.*
3. *Nothing else is a tail expression.*

[KCR98] extends this definition to the syntax of full Scheme.

Definition 2 *A* tail call *is a tail expression that is a procedure call.*

Figure 2 shows that tail calls are much more common than the special case of self-tail calls, in which a procedure calls itself tail recursively.

3 The Essence of Proper Tail Recursion

The essence of proper tail recursion is that a procedure can return by performing a tail call to any procedure, including itself. This is a kind of dual to the informal characterization quoted in Section 1. A tail call does not cause an immediate return, but passes the responsibility for returning from the procedure that performs the tail call to the procedure it is calling. In other words, the activation of a procedure extends from the time that it is called to the time that it performs either a return or a tail call.

This is fundamentally different from the traditional view of procedure calls, in which the activation of a procedure encompasses the activations of all procedures that it calls.

4 An Example

Figure 3 shows a procedure definition that contains three tail calls, of which the last is a self-tail call. Given a predicate, a binary tree, and a failure continuation of no arguments, `find-leftmost` searches for the leftmost leaf that satisfies the predicate. If such a leaf is found, then it is returned normally. Otherwise the procedure returns by performing a tail call to the failure continuation or to itself.

```
(define (find-leftmost predicate? tree fail)
  (if (leaf? tree)
      (if (predicate? tree)
          tree                          ;return
          (fail))                       ;tail call
      (let ((continuation
             (lambda ()
               (find-leftmost           ;tail call
                predicate?
                (right-child tree)
                fail))))
        (find-leftmost predicate?       ;tail call
                       (left-child tree)
                       continuation))))
```

Figure 3: An example with three tail calls.

Although `find-leftmost` uses an explicit failure continuation, it is not a pure example of continuation-passing style, because its fourth line returns `tree` to the implicit continuation. Returning is equivalent to performing an implicit tail call to the implicit continuation.

In Scheme, it is perfectly feasible to write large programs in which no procedure ever returns, and all calls are tail calls.[1] This is pure continuation-passing style. Proper tail recursion guarantees that implementations will use only a bounded amount of storage to implement all of the calls that are performed by a program written in this style.

To make this precise, we need a proper model of space consumption. This model should allow us to reason about the space needed to run a program, independent of implementation. For this to be tractable, the space model should take the form of an asymptotic upper bound on the space consumed by an implementation.

Proper tail recursion constrains but does not determine the space model. To obtain a complete model, we must also model the space consumed by variables and data, and specify the roots that a garbage collector would use to determine whether a variable or datum is reachable. Garbage collection then completes the model. With a reasonable garbage collector, the asymptotic space required for variables and data is $O(N)$, where N is the largest number of words occupied by reachable variables and data at any point in the program [App92].

For example, a Scheme programmer can tell that the space required by `find-leftmost` is independent of the num-

[1] Some compilers do this routinely, using CPS Scheme as their target language.

ber of right edges in the tree, and is proportional to the maximal number of left edges that occur within any directed path from the root of the tree to a leaf. If every left child is a leaf, then `find-leftmost` runs in constant space, no matter how large the tree.

5 Retention versus Deletion

A deletion strategy reclaims storage at statically determined points in the program, whereas a retention strategy retains storage until it is no longer needed, as determined by dynamic means such as garbage collection [Fis72]. Algol-like stack allocation is the most important deletion strategy.

By allowing the lifetime of a variable or value to extend beyond the lifetime of the block in which it was declared or created, retention strategies support more flexible programming styles. It is less well-known that retention strategies can also reclaim storage *sooner* than a deletion strategy, and can have better asymptotic space efficiency. This is crucial for proper tail recursion.

Deletion strategies interfere with proper tail recursion [Ste78, Cha88, App92, ASwS96]. For the example of Section 4, allocating `continuation` on a stack would require $O(n)$ space instead of $O(1)$ space, even when every left child is a leaf.

Nevertheless many compilers for Scheme, Standard ML, and similar languages employ optimizations that allocate some variables on a stack [Ste78, KKsR+86, Han90, SF96, Ser97, Sis97]. Some researchers have gone so far as to suggest that a static deletion strategy could be used to replace dynamic garbage collection altogether, for some programs at least [TT94, AFL95, Sis97].

In Scheme, stack allocation and other deletion strategies can be used only when they do not destroy the property of proper tail recursion. This is not always easy to determine [Cha88, SF96]. It has not been made easier by the informality with which proper tail recursion has been defined.

It is fairly easy to define proper tail recursion formally for a particular implementation, but an implementation-independent formalization cannot refer to implementation-dependent structures such as a stack. Standard ML of New Jersey, for example, allocates continuation frames on a heap, and does not distinguish them from closures that are allocated for explicit lambda expressions [App92]. Algol-like implementations use a single stack to represent three distinct kinds of runtime structure: continuations, environments, and the store. The distinction between storage allocated for continuations, environments, and the store is not always clear even to the implementor.

6 Formal Definitions

The largest number of machine words that a (possibly nondeterministic) implementation X might consume when running a program P on an input D can be described by a (deterministic) function S_X such that $S_X(P,D) \in \Re \cup \{\infty\}$, where $\Re$ stands for the real numbers. I will refer to S_X as the space consumption function of X. To compare the asymptotic behavior of such functions, I will use big-O notation as defined below. For real-valued functions of the natural numbers, this definition is equivalent to the usual definition as given in [Knu73, CLR90].

Definition 3 (asymptotic complexity, $O(f)$) *If A is any set, and $f : A \to \Re \cup \{\infty\}$, then the asymptotic (upper bound) complexity class of f is $O(f)$, which is defined as the set of all functions $g : A \to \Re \cup \{\infty\}$ for which there exist real constants c_1 and c_0 such that $c_1 > 0$ and*

$$\forall a \in A \,.\, g(a) \leq c_1 f(a) + c_0$$

Configuration	::=	$\langle v, \sigma \rangle$
	|	$\langle E, \rho, \kappa, \sigma \rangle$
	|	$\langle v, \rho, \kappa, \sigma \rangle$
$v \in$ **Value**	::=	c
	|	UNSPECIFIED
	|	UNDEFINED
	|	PRIMOP:ψ
	|	ESCAPE:$\langle \alpha, \kappa \rangle$
	|	CLOSURE:$\langle \alpha, L, \rho \rangle$
κ	::=	**halt**
	|	**select**:$\langle E_1, E_2, \rho, \kappa \rangle$
	|	**assign**:$\langle I, \rho, \kappa \rangle$
	|	**push**:$\langle \langle E, \ldots \rangle, \langle v, \ldots \rangle, \pi, \rho, \kappa \rangle$
	|	**call**:$\langle \langle E, \ldots \rangle, \kappa \rangle$
ρ	$\in$	**Identifier** $\xrightarrow{\text{fin}}$ **Location**
σ	$\in$	**Location** $\xrightarrow{\text{fin}}$ **Value**
π	$\in$	**Permutation**

Figure 4: Syntax of configurations.

Each space consumption function S_X induces an asymptotic space complexity class $O(S_X)$. Sections 7 through 10 describe a family of reference implementations that formalize several important models of space efficiency. Their space consumption functions S_{stack}, S_{tail}, S_{evlis}, and S_{sfs} are defined in Section 12. Their induced complexity classes are related by the proper inclusions

$$O(S_{\text{sfs}}) \subset O(S_{\text{evlis}}) \subset O(S_{\text{tail}}) \subset O(S_{\text{stack}})$$

Definition 4 (conventional space leaks) *An implementation has a* conventional space leak *iff its space consumption is not in $O(S_{\text{stack}})$. An implementation* has no conventional space leaks *iff its space consumption is in $O(S_{\text{stack}})$.*

Definition 5 (properly tail recursive) *An implementation is* properly tail recursive *iff its space consumption is in $O(S_{\text{tail}})$. An implementation has a* stack-like space leak *iff it has no conventional space leaks but is not properly tail recursive. An implementation with a stack-like space leak is also known as* improperly tail recursive.

Definition 6 (evlis tail recursive, safe for space) *An implementation is* evlis tail recursive *[Wan80, Que96] iff its space consumption is in $O(S_{\text{evlis}})$. An implementation is* safe for space complexity *in the sense of Appel [App92] iff its space consumption is in $O(S_{\text{sfs}})$.*

Among these properties of an implementation, proper tail recursion is especially important because it is easy to construct programs that should run in small constant space but are likely to require at least linear space in implementations that are not properly tail recursive.

Reduction rules:

$\langle$(quote c), $\rho, \kappa, \sigma\rangle \rightarrow \langle c, \rho, \kappa, \sigma\rangle$

$\langle I, \rho, \kappa, \sigma\rangle \rightarrow \langle \sigma(\rho(I)), \rho, \kappa, \sigma\rangle$
if $I \in \text{Dom } \rho$, $\rho(I) \in \text{Dom } \sigma$,
and $\sigma(\rho(I)) \neq$ UNDEFINED

$\langle L, \rho, \kappa, \sigma\rangle \rightarrow \langle$CLOSURE:$\langle \alpha, L, \rho\rangle, \rho, \kappa, \sigma[\alpha \mapsto$ UNSPECIFIED$]\rangle$
if α does not occur within L, ρ, κ, σ

$\langle$(if E_0 E_1 E_2), $\rho, \kappa, \sigma\rangle \rightarrow \langle E_0, \rho,$ **select:**$\langle E_1, E_2, \rho, \kappa\rangle, \sigma\rangle$

$\langle$(set! I E_0), $\rho, \kappa, \sigma\rangle \rightarrow \langle E_0, \rho,$ **assign:**$\langle I, \rho, \kappa\rangle, \sigma\rangle$

$\langle$(E_0 E_1 ...), $\rho, \kappa, \sigma\rangle \rightarrow \langle E_0', \rho,$ **push:**$\langle\langle E_1', \ldots\rangle, \langle\rangle, \pi, \rho, \kappa\rangle, \sigma\rangle$
if $\langle E_0', E_1', \ldots\rangle = \text{reverse}(\pi^{-1}\langle E_0, E_1, \ldots\rangle)$

Continuation rules:

$\langle v, \{\}, \textbf{halt}, \sigma\rangle \rightarrow \langle v, \sigma\rangle$

$\langle v, \rho',$ **halt**$, \sigma\rangle \rightarrow \langle v, \{\},$ **halt**$, \sigma\rangle$

$\langle v, \rho',$ **select:**$\langle E_1, E_2, \rho, \kappa\rangle, \sigma\rangle \rightarrow \langle E_1, \rho, \kappa, \sigma\rangle$
if $v \neq$ FALSE

$\langle$FALSE$, \rho',$ **select:**$\langle E_1, E_2, \rho, \kappa\rangle, \sigma\rangle \rightarrow \langle E_2, \rho, \kappa, \sigma\rangle$

$\langle v, \rho',$ **assign:**$\langle I, \rho, \kappa\rangle, \sigma\rangle \rightarrow \langle$UNSPECIFIED$, \rho, \kappa, \sigma[\rho(I) \mapsto v]\rangle$

$\langle v_0', \rho',$ **push:**$\langle\langle E_1', E_2', \ldots\rangle, \langle v_1', \ldots\rangle, \pi, \rho, \kappa\rangle, \sigma\rangle \rightarrow \langle E_1', \rho,$ **push:**$\langle\langle E_2', \ldots\rangle, \langle v_0', v_1', \ldots\rangle, \pi, \rho, \kappa\rangle, \sigma\rangle$

$\langle v_0', \rho',$ **push:**$\langle\langle\rangle, \langle v_1', \ldots\rangle, \pi, \rho, \kappa\rangle, \sigma\rangle \rightarrow \langle v_0, \rho,$ **call:**$\langle\langle v_1, \ldots\rangle, \kappa\rangle, \sigma\rangle$
if $\langle v_0, v_1, \ldots\rangle = \pi\langle v_0', v_1', \ldots\rangle$

$\langle$CLOSURE:$\langle \alpha, L, \rho\rangle, \rho',$ **call:**$\langle\langle v_1, \ldots, v_n\rangle, \kappa\rangle, \sigma\rangle \rightarrow \langle E, \rho'', \kappa, \sigma'\rangle$
if $L =$ (lambda (I_1 ...) E)
and $\beta_1, \ldots, \beta_n$ do not occur within E, ρ, κ, σ
and $\rho'' = \rho[I_1, \ldots, I_n \mapsto \beta_1, \ldots, \beta_n]$
and $\sigma' = \sigma[\beta_1, \ldots, \beta_n \mapsto v_1, \ldots, v_n]$

Garbage collection rule:

$\langle v, \rho, \kappa, \sigma[\beta, \ldots \mapsto v', \ldots]\rangle \rightarrow \langle v, \rho, \kappa, \sigma\rangle$
if $\{\beta, \ldots\}$ is nonempty
and $\beta, \ldots$ do not occur within v, ρ, κ, σ

Figure 5: Properly tail recursive semantics.

7 Proper Tail Recursion ($\mathcal{I}_{\text{tail}}$)

To expose its space requirements, while retaining the simplicity needed for proofs, the properly tail recursive reference implementation is expressed as a small-step operational semantics of the kind known as a CEKS machine [NN92, MFH95]. The asymptotic space required by a reference implementation must be at least as great as the space required by any reasonable implementation of Scheme, including pure interpreters, so this implementation is resolutely non-clever. It is also nondeterministic, to reflect rampant underspecification in the definition of Scheme.

A configuration of this semantics, as shown in Figure 4, is either a final configuration consisting of a value and a store, or an intermediate configuration consisting of an expression or value, an environment, a continuation, and a store. An intermediate configuration represents the state of an abstract machine with three registers—accumulator, environment (static link), continuation (dynamic link)—and a garbage-collected heap. When the first component of a configuration is an expression, the accumulator acts as a program counter. When the first component is a value, the continuation register acts as the program counter. The initial configurations are described in Section 11, which also defines the observable answer represented by a final configuration.

The core transition rules are shown in Figure 5. These core rules must be supplemented by additional rules, mainly for primitive procedures, which are not specified in this paper.

The six reduction rules say that:

- A quoted datum evaluates to the datum.
- An identifier evaluates to its R-value; if $I \notin \text{Dom } \rho$, $\rho(I) \notin \text{Dom } \sigma$, or $\sigma(\rho(I)) = \text{UNDEFINED}$, then the transition rule cannot be applied, and the computation will be *stuck*.
- A lambda expression evaluates to a closure. A bug in the design of Scheme requires that a location α be allocated to tag the closure [Ram94].
- A conditional expression is evaluated by evaluating the test with a continuation that will select one of the alternatives.
- The right-hand side of an assignment is evaluated with a continuation that will store its R-value into the L-value of the left-hand side.
- A procedure call is evaluated by nondeterministically choosing a permutation π of its operator and operand expressions; the first of the permuted expressions is then evaluated with a continuation that will evaluate the others and then perform the call.

Other sources of nondeterminism include the choice of locations to allocate when a closure is called, and the choice of whether and when to use the garbage collection rule.

The conceptual difference between proper tail recursion and ad hoc tail call optimization permeates this semantics. Proper tail recursion affects every rule shown in Figure 5 except for the first three reduction rules and the first two continuation rules.

In a properly tail recursive implementation, continuations are created to evaluate subexpressions, not to call a procedure [Ste78]. A procedure call is just a goto that changes the environment register. Notice that *every* call is a goto, not just tail calls. Hence the last continuation rule, which shows how closures are called, does not create a new continuation. In particular, a procedure call does not create or pass a new return address, push the environment ρ', or allocate a stack frame.

The continuations that are tagged by **select**, **assign**, and **push** include an environment, which is restored when the continuation is invoked. This allows the test part of a conditional, the right hand side of an assignment, and an operator or operand expression to destroy the environment by performing a procedure call.

The garbage collection rule allows unreachable storage to be recycled.[2] If there exists a nonempty set of locations that are not reachable via the active store from the locations mentioned by v, ρ, and κ, then those locations may be removed from the active store and become available for future steps of the computation.

The garbage collection rule allows but does not require garbage collection. To prevent improper tail recursion from being masked by uncollected garbage, we must require the garbage collection rule to be used sufficiently often. In Section 12 we will require the garbage collection rule to be used whenever garbage remains to be collected. In a real implementation the garbage collector would run much less often, but would use no more than some fixed constant R times the space required when collecting after every computation step ([App92], Section 12.4). Usually $R \leq 3$.

8 Improper Tail Recursion ($\mathcal{I}_{\text{gc}}$, $\mathcal{I}_{\text{stack}}$)

An improperly tail recursive reference implementation $\mathcal{I}_{\text{gc}}$ is obtained by replacing the last continuation rule in Figure 5 by

$$\begin{aligned}&\langle \text{CLOSURE:}\langle \alpha, L, \rho\rangle, \rho', \textbf{call:}\langle\langle v_1, \ldots, v_n\rangle, \kappa\rangle, \sigma\rangle \\ &\quad\rightarrow\ \langle E, \rho'', \textbf{return:}\langle \rho', \kappa\rangle, \sigma'\rangle \\ &\qquad\qquad \text{if } L = \texttt{(lambda (}I_1 \ \ldots\ I_n\texttt{) } E\texttt{)} \\ &\qquad\qquad \text{and } \beta_1, \ldots, \beta_n \text{ do not occur in } E, \rho, \kappa, \sigma \\ &\qquad\qquad \text{and } \rho'' = \rho[I_1, \ldots, I_n \mapsto \beta_1, \ldots, \beta_n] \\ &\qquad\qquad \text{and } \sigma' = \sigma[\beta_1, \ldots, \beta_n \mapsto v_1, \ldots, v_n]\end{aligned}$$

and by adding the continuation rule

$$\langle v, \rho, \textbf{return:}\langle \rho', \kappa\rangle, \sigma\rangle \ \rightarrow\ \langle v, \rho', \kappa, \sigma\rangle$$

By creating a continuation for every procedure call, these rules waste space for no reason [Ste78]. They would look a lot less silly if they implemented a deletion strategy for local variables. In Scheme, however, a deletion strategy can create dangling pointers. As mentioned in Section 4, most Scheme compilers nonetheless use a deletion strategy as an optimization for variables whose lifetimes can be statically bounded. To see what would happen if such optimizations were pursued without regard for their effect on tail recursion, let $\mathcal{I}_{\text{stack}}$ be the semantics obtained by replacing the last continuation rule in Figure 5 by

$$\begin{aligned}&\langle \text{CLOSURE:}\langle \alpha, L, \rho\rangle, \rho', \textbf{call:}\langle\langle v_1, \ldots, v_n\rangle, \kappa\rangle, \sigma\rangle \\ &\quad\rightarrow\ \langle E, \rho'', \textbf{return:}\langle A, \rho', \kappa\rangle, \sigma'\rangle \\ &\qquad\qquad \text{if } L = \texttt{(lambda (}I_1 \ \ldots\ I_n\texttt{) } E\texttt{)} \\ &\qquad\qquad \text{and } \beta_1, \ldots, \beta_n \text{ do not occur in } E, \rho, \kappa, \sigma \\ &\qquad\qquad \text{and } \rho'' = \rho[I_1, \ldots, I_n \mapsto \beta_1, \ldots, \beta_n] \\ &\qquad\qquad \text{and } \sigma' = \sigma[\beta_1, \ldots, \beta_n \mapsto v_1, \ldots, v_n] \\ &\qquad\qquad \text{and } A \subseteq \{\beta_1, \ldots, \beta_n\}\end{aligned}$$

[2] No sequence of reduction steps allocates more than one storage location, so no garbage collection rule is needed for configurations whose first component is an expression.

$$\langle v, \rho, \mathbf{return}{:}\langle A, \rho', \kappa\rangle, \sigma'\rangle \\ \quad \to \langle v, \rho', \kappa, \sigma\rangle \\ \qquad \text{if no } \beta \in A \text{ occurs within } v,\ \rho',\ \kappa,\ \sigma \\ \qquad \text{and } \sigma = \sigma' \mid (\mathrm{Dom}\ \sigma' \setminus A)$$

The nondeterministic choice of A subsumes any static analysis for an optimization that allocates variables on a stack, provided the optimization does not create dangling pointers, and does not extend the lifetime of a garbage variable beyond that of Algol-like stack allocation.

For the Algol-like subset of Scheme, it is always possible to choose $A = \{\beta_1, \ldots, \beta_n\}$, which results in the space required by Algol-like stack allocation of variables. This choice of A always consumes the most space, so it determines the space consumption S_{stack} on such programs. Hence S_{stack} also characterizes the space consumed by Algol-like implementations.

9 Evlis Tail Recursion ($\mathcal{I}_{\text{evlis}}$)

Although $\mathcal{I}_{\text{tail}}$ uses less space than $\mathcal{I}_{\text{gc}}$ or $\mathcal{I}_{\text{stack}}$, it too wastes space needlessly. For example, it is not necessary to preserve the environment across the evaluation of the last subexpression to be evaluated during the evaluation of a procedure call. Even a pure interpreter can take advantage of this technique, which is known as *evlis tail recursion* [Wan80, Que96]. Let $\{\,\}$ denote an empty environment, and let $\mathcal{I}_{\text{evlis}}$ be the semantics obtained from $\mathcal{I}_{\text{tail}}$ by replacing the first continuation rule for **push** by the two rules

$$\langle v_0', \rho', \mathbf{push}{:}\langle\langle E_1', E_2', E_3', \ldots\rangle, \langle v_1', \ldots\rangle, \pi, \rho, \kappa\rangle, \sigma\rangle \\ \quad \to \langle E_1', \rho, \mathbf{push}{:}\langle\langle E_2', E_3', \ldots\rangle, \langle v_0', v_1', \ldots\rangle, \pi, \rho, \kappa\rangle, \sigma\rangle \\ \langle v_1', \rho', \mathbf{push}{:}\langle\langle E_0'\rangle, \langle v_2', \ldots\rangle, \pi, \rho, \kappa\rangle, \sigma\rangle \\ \quad \to \langle E_0', \rho, \mathbf{push}{:}\langle\langle\rangle, \langle v_1', v_2', \ldots\rangle, \pi, \{\,\}, \kappa\rangle, \sigma\rangle$$

10 Safe for Space Complexity ($\mathcal{I}_{\text{free}}$, $\mathcal{I}_{\text{sfs}}$)

In the implementations that have been described so far, a lambda expression is closed over all variables that are in scope, regardless of whether those variables actually occur free within the lambda expression. This is typical of interpretive and Algol-like implementations, but it sometimes forces programmers to write awkward code to avoid space leaks. Compiled implementations often close over only the free variables, which improves the space complexity of some programs.

Let $\mathcal{I}_{\text{free}}$ be the semantics obtained from $\mathcal{I}_{\text{tail}}$ by replacing the reduction rule for lambda expressions by

$$\langle L, \rho, \kappa, \sigma\rangle \\ \quad \to \langle \text{CLOSURE}{:}\langle \alpha, L, \rho'\rangle, \rho, \kappa, \sigma[\alpha \mapsto \text{UNSPECIFIED}]\rangle \\ \qquad \text{if } \alpha \text{ does not occur within } L,\ \kappa,\ \rho,\ \sigma \\ \qquad \text{and } \rho' = \rho \mid (\mathrm{Dom}\ \rho \cap \mathrm{FV}(L))$$

Let $\mathcal{I}_{\text{sfs}}$ be the semantics obtained from $\mathcal{I}_{\text{tail}}$ by replacing the reduction rule for lambda expressions as above, and by replacing the last three reduction rules and the next-to-last continuation rule by

$$\langle (\texttt{if}\ E_0\ E_1\ E_2), \rho, \kappa, \sigma\rangle \\ \quad \to \langle E_0, \rho, \mathbf{select}{:}\langle E_1, E_2, \rho', \kappa\rangle, \sigma\rangle \\ \qquad \text{if } \rho' = \rho \mid (\mathrm{Dom}\ \rho \cap (\mathrm{FV}(E_1) \cup \mathrm{FV}(E_2))) \\ \langle (\texttt{set!}\ I\ E_0), \rho, \kappa, \sigma\rangle \\ \quad \to \langle E_0, \rho, \mathbf{assign}{:}\langle I, \rho', \kappa\rangle, \sigma\rangle \\ \qquad \text{if } \rho' = \rho \mid \{I\}$$

$$\langle (E_0\ E_1\ \ldots), \rho, \kappa, \sigma\rangle \\ \quad \to \langle E_0', \rho, \mathbf{push}{:}\langle\langle E_1', \ldots\rangle, \langle\rangle, \pi, \rho', \kappa\rangle, \sigma\rangle \\ \qquad \text{if } \langle E_0', E_1', \ldots\rangle = \mathrm{reverse}(\pi^{-1}\langle E_0, E_1, \ldots\rangle) \\ \qquad \text{and } \rho' = \rho \mid (\mathrm{Dom}\ \rho \cap (\mathrm{FV}(E_1') \cup \ldots)) \\ \langle v_0', \rho', \mathbf{push}{:}\langle\langle E_1', E_2', \ldots\rangle, \langle v_1', \ldots\rangle, \pi, \rho, \kappa\rangle, \sigma\rangle \\ \quad \to \langle E_1', \rho, \mathbf{push}{:}\langle\langle E_2', \ldots\rangle, \langle v_0', v_1', \ldots\rangle, \pi, \rho', \kappa\rangle, \sigma\rangle \\ \qquad \text{if } \rho' = \rho \mid (\mathrm{Dom}\ \rho \cap (\mathrm{FV}(E_2') \cup \ldots))$$

The asymptotic space efficiency of $\mathcal{I}_{\text{sfs}}$ represents safe-for-space complexity in the sense defined by Appel [App92, AS96], whereas the asymptotic space efficiency of $\mathcal{I}_{\text{free}}$ represents a weaker but still useful sense of safe-for-space.

11 Equivalence of Implementations

This section proves that all of the reference implementations compute the same answers. Since some of the implementations use more space than others, this result requires a countably infinite set of locations $\mathbf{Location} = \{\alpha_i \mid i \geq 0\}$.

Lemma 7 *If C is a configuration of $\mathcal{I}_{\text{stack}}$, and C' is the configuration of $\mathcal{I}_{\text{gc}}$ obtained by replacing all continuations of the form* $\mathbf{return}{:}\langle A, \rho, \kappa\rangle$ *by* $\mathbf{return}{:}\langle \rho, \kappa\rangle$*, and $C \to^* \langle v, \sigma\rangle$, then $C' \to^* \langle v, \sigma\rangle$.*

Proof: By induction on the number of computation steps. The locations that occur within C' are a subset of those that occur within C, so all of the nondeterministic choices that are made in the original computation can also be made by the computation in $\mathcal{I}_{\text{gc}}$. □

The next two lemmas are proved similarly.

Lemma 8 *If C is a configuration of $\mathcal{I}_{\text{gc}}$, and C' is the configuration of $\mathcal{I}_{\text{tail}}$ obtained by replacing all continuations of the form* $\mathbf{return}{:}\langle \rho, \kappa\rangle$ *by κ, and $C \to^* \langle v, \sigma\rangle$, then $C' \to^* \langle v, \sigma\rangle$.*

Lemma 9 *If $C \to^* \langle v, \sigma\rangle$ by the rules of $\mathcal{I}_{\text{tail}}$, then $C \to^* \langle v, \sigma\rangle$ by the rules of $\mathcal{I}_{\text{evlis}}$.*

Definition 10 (initial configurations) *An* initial configuration *is a configuration of the form $\langle E, \rho_0, \mathbf{halt}, \sigma_0\rangle$ such that*

- *If α occurs in E, ρ_0, or σ_0, then $\alpha \in \mathrm{Dom}\ \sigma_0$.*
- *If an identifier I occurs free in E, then $I \in \mathrm{Dom}\ \rho_0$.*
- *If α occurs in E, then $\alpha \notin \mathrm{Ran}\ \rho_0$.*

Definition 11 (answers) *The observable* answer *represented by a final configuration $\langle v, \sigma\rangle$ is the possibly infinite sequence of output tokens* $\mathrm{answer}(v, \sigma)$ *where*

$$\begin{aligned}
\mathrm{answer}(v, \sigma) &= \mathrm{ans}(v, \sigma, \texttt{EOF}) \\
\mathrm{ans}(\text{TRUE}, \sigma, s) &= \texttt{\#t}, \mathrm{ans}(s, \sigma, \texttt{EOF}) \\
\mathrm{ans}(\text{FALSE}, \sigma, s) &= \texttt{\#f}, \mathrm{ans}(s, \sigma, \texttt{EOF}) \\
\mathrm{ans}(\text{NUM}{:}z, \sigma, s) &= z, \mathrm{ans}(s, \sigma, \texttt{EOF}) \\
\mathrm{ans}(\text{SYM}{:}I, \sigma, s) &= I, \mathrm{ans}(s, \sigma, \texttt{EOF}) \\
\mathrm{ans}(\text{VEC}{:}\langle \alpha_0, \ldots\rangle, \sigma, s) &= \texttt{\#(}, \mathrm{ans}(\langle \sigma(\alpha), \ldots\rangle, \sigma, s) \\
\mathrm{ans}(\text{ESCAPE}{:}\langle \alpha, \kappa\rangle, \sigma, s) &= \texttt{\#<PROC>}, \mathrm{ans}(s, \sigma, \texttt{EOF}) \\
\mathrm{ans}(\text{CLOSURE}{:}\langle \alpha, L, \rho\rangle, \sigma, s) &= \texttt{\#<PROC>}, \mathrm{ans}(s, \sigma, \texttt{EOF}) \\
&\vdots \\
\mathrm{ans}(\langle\rangle, \sigma, s) &= \texttt{)}, \mathrm{ans}(s, \sigma, \texttt{EOF}) \\
\mathrm{ans}(\langle v_0, v_1, \ldots\rangle, \sigma, s) &= \mathrm{ans}(v_0, \sigma, \langle v_1, \ldots\rangle, s)) \\
\mathrm{ans}(\texttt{EOF}, \mathit{store}, s) &= \texttt{;End of output}
\end{aligned}$$

Lemma 12 *If*

- *X is $\mathcal{I}_{\text{tail}}$ and Y is $\mathcal{I}_{\text{free}}$, or*
- *X is $\mathcal{I}_{\text{evlis}}$ and Y is $\mathcal{I}_{\text{sfs}}$, or*
- *X is $\mathcal{I}_{\text{free}}$ and Y is $\mathcal{I}_{\text{sfs}}$,*

and $C \to^* \langle v, \sigma \rangle$ by the rules of X, then there exist v' and σ' such that $\text{answer}(v, \sigma) = \text{answer}(v', \sigma')$ *and $C \to^* \langle v', \sigma' \rangle$ by the rules of Y.*

Proof: At each step of the computation, the analogous rule of Y can be used with the same nondeterministic choices, because the only differences between two corresponding configurations are that some of the environments that occur in the configuration of Y have been restricted to the free variables of the expression(s) that will be evaluated using that environment. The final configurations differ only in the environment components of closures and of continuations that have been captured as part of an escape procedure; these differences are not observable. □

Definition 13 (α-convertible) *Configurations C_1 and C_2 are α-convertible, written $C_1 \stackrel{\alpha}{=} C_2$, iff there exists a one-to-one function R : **Location** $\to$ **Location** such that C_2 is the configuration obtained from C_1 by renaming its locations according to R.*

The following lemma shows that α-convertible configurations are semantically equivalent.

Lemma 14 *If $C \stackrel{\alpha}{=} C'$, and $C \to^* \langle v, \sigma \rangle$, then there exist v' and σ' such that* $\text{answer}(v, \sigma) = \text{answer}(v', \sigma')$ *and $C' \to^* \langle v', \sigma' \rangle$.*

Sketch of proof: None of the rules treat any location specially. This must be verified for the primitive operations as well, whose rules are omitted from this paper. □

The next few definitions and lemmas establish the existence of two canonical forms for computations: without garbage collection, and with maximal garbage collection.

Definition 15 (gc-transform of a configuration) *If C is a configuration, then its* gc-transform *is D defined as follows.*

- *If C is a final configuration $\langle v, \sigma \rangle$, then $D = \langle v, \sigma' \rangle$ where $\langle v, \{\}, \textbf{halt}, \sigma' \rangle$ is the gc-transform of $\langle v, \{\}, \textbf{halt}, \sigma \rangle$.*
- *If the garbage collection rule does not apply to C, and C is not a final configuration, then $D = C$.*
- *Otherwise D is the unique configuration such that $C \to D$ by the garbage collection rule, and the garbage collection rule does not apply to D.*

Lemma 16 *If $C \to C'$, and D and D' are the gc-transforms of C and C', then $D = D'$, or $D \to D'$, or there exists a configuration D'' (unique up to α-convertibility) such that $D \to D'' \to D'$.*

Definition 17 (gc-transform of a computation) *If $C_0 \to C_1 \to \cdots$ is a computation, and for each i the gc-transform of C_i is D_i, then the* gc-transform *of the computation is the computation obtained from the sequence $D_0, D_1, \ldots$ as follows. If $D_i = D_{i+1}$, then D_{i+1} is erased from the sequence. If $D_i \to D_{i+1}$, then D_i and D_{i+1} remain adjacent. If $D_i \to D'' \to D_{i+1}$, then D'' is inserted into the sequence between D_i and D_{i+1}.*

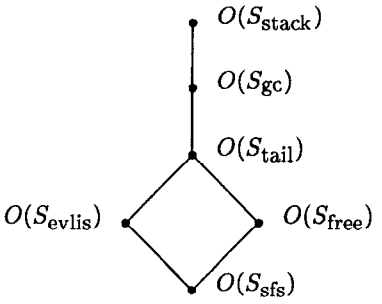

Figure 6: A hierarchy of space complexity classes.

Theorem 18 (canonical forms) *If $C_0 \to C_1 \to \cdots$ is any computation, then its gc-transform is (pointwise) α-convertible to the gc-transform of a computation that never uses the garbage collection rule.*

Theorem 19 *If an answer can be computed from some initial configuration by the rules of $\mathcal{I}_{\text{sfs}}$, then it can be computed from that configuration by the rules of $\mathcal{I}_{\text{stack}}$.*

Proof: Given a computation in $\mathcal{I}_{\text{sfs}}$, Theorem 18 can be used to lift it to a structurally similar computation that does not use the garbage collection rule. This gc-free computation can be transformed into a computation in $\mathcal{I}_{\text{tail}}$ by using ρ instead of ρ' at each use of a rule for which $\mathcal{I}_{\text{sfs}}$ differs from $\mathcal{I}_{\text{tail}}$. The resulting computation can then be transformed into a computation in $\mathcal{I}_{\text{gc}}$ by adding **return** continuations at each use of the **call** rule, and by inserting uses of the **return** rule into the computation. The resulting computation can be transformed into a computation in $\mathcal{I}_{\text{stack}}$ by adding an empty set to each **return** continuation. □

Corollary 20 *All of the reference implementations compute the same answers.*

12 Space Consumption

This section gives a formal definition of the space required by a configuration, and defines functions S_{tail}, S_{gc}, S_{stack}, S_{evlis}, S_{free}, and S_{sfs} that characterize the space required to run a program on an input as the worst case among all execution sequences that have a certain property. These functions induce the hierarchy of space complexity classes that is shown in Figure 6.

To define the space consumed by an implementation, I will take inputs to be expressions, and programs to be expressions that evaluate to a procedure of one argument. The constants of a program P must not share storage with the constants of an input D, nor may P or D share storage with the standard library. The easiest way to ensure this is to forbid vector, string, and list constants. This entails no loss of generality, because all such constants can be replaced by references to global variables that are initialized using the standard library procedures that allocate new vectors, strings, and lists.

Let ρ_0 and σ_0 be some fixed initial environment and initial store that contain Scheme's standard procedures, as described in Section 6 of [IEE91]. Let **Program** and **Input** both denote the set of Core Scheme expressions that contain no locations, and whose free variables are bound in ρ_0.

$$
\begin{aligned}
\text{space}(\langle v, \sigma\rangle) &= \text{space}(v) + \text{space}(\sigma)\\
\text{space}(\langle E, \rho, \kappa, \sigma\rangle) &= |\text{Dom } \rho| + \text{space}(\kappa) + \text{space}(\sigma)\\
\text{space}(\langle v, \rho, \kappa, \sigma\rangle) &= \text{space}(v) + |\text{Dom } \rho| + \text{space}(\kappa) + \text{space}(\sigma)\\
\text{space}(\sigma) &= \sum_{\alpha \in \sigma} (1 + \text{space}(\sigma(\alpha)))\\
\text{space}(\text{TRUE}) = \text{space}(\text{FALSE}) = \text{space}(\text{SYM:}I) &= 1\\
\text{space}(\text{VEC:}\langle \alpha_0, \ldots, \alpha_{n-1}\rangle) &= 1 + n\\
\text{space}(\text{NUM:}z) &= 1 + \log_2 z \qquad \text{if } z \text{ is an exact positive integer [IEE91]}\\
&\vdots\\
\text{space}(\text{CLOSURE:}\langle \alpha, L, \rho\rangle) &= 1 + |\text{Dom } \rho|\\
\text{space}(\textbf{halt}) &= 1\\
\text{space}(\textbf{select:}\langle E_1, E_2, \rho, \kappa\rangle) &= 1 + |\text{Dom } \rho| + \text{space}(\kappa)\\
\text{space}(\textbf{assign:}\langle I, \rho, \kappa\rangle) &= 1 + |\text{Dom } \rho| + \text{space}(\kappa)\\
\text{space}(\textbf{push:}\langle\langle E_1, \ldots, E_m\rangle, \langle v_1, \ldots, v_n\rangle, \pi, \rho, \kappa\rangle) &= 1 + m + n + |\text{Dom } \rho| + \text{space}(\kappa)\\
\text{space}(\textbf{call:}\langle\langle v_1, \ldots, v_m\rangle, \kappa\rangle) &= 1 + m + \text{space}(\kappa)\\
\text{space}(\textbf{return:}\langle \rho, \kappa\rangle) &= 1 + |\text{Dom } \rho| + \text{space}(\kappa)\\
\text{space}(\textbf{return:}\langle A, \rho, \kappa\rangle) &= 1 + |\text{Dom } \rho| + \text{space}(\kappa)
\end{aligned}
$$

Figure 7: The space consumed by a configuration (using flat environments).

The space consumed by a configuration is defined in Figure 7. This definition corresponds to the use of flat (copied) environments, because the purpose of this definition is to establish upper bounds, not lower bounds. For a comparison of flat environments with linked (shared) environments, see Section 13.

Definition 21 *A* space-efficient computation in $\mathcal{I}_X$ *is a finite or countably infinite sequence of configurations* $\{C_i\}_{i \in I}$ *such that*

- C_0 *is an initial configuration.*
- *If the sequence is finite, then it ends with a final configuration* C_n.
- *For each* $i \in I$, *if* $i > 0$ *then* $C_{i-1} \to C_i$ *by the rules of* $\mathcal{I}_X$.
- *If the garbage collection rule is applicable to* C_i, *then* $C_i \to C_{i+1}$ *by the garbage collection rule.*

This definition eliminates incomplete or stuck computations from consideration. A stuck computation represents a program error or, in the case of $\mathcal{I}_{\text{stack}}$, a stack allocation that creates a dangling pointer.

Definition 22 (supremum) *If* $R \subseteq \Re$, *then the supremum* $\sup R$ *is the least upper bound of* R, *or* ∞ *if there is no upper bound in* $\Re$.

Definition 23 (space consumption S_X) *The space consumption function of* $\mathcal{I}_X$ *is* S_X : **Program** × **Input** → $\Re \cup \{\infty\}$ *defined by*

$$
\begin{aligned}
S_X(P, D) = {} & |P| + \\
& \sup\{\sup\{\text{space}(C_i) \mid i \in I\} \mid \\
& \qquad \{C_i\}_{i \in I} \text{ is a space-efficient} \\
& \qquad \text{computation in } \mathcal{I}_X, \text{ with} \\
& \qquad C_0 = \langle (P\ D), \rho_0, \textbf{halt}, \sigma_0\rangle\}
\end{aligned}
$$

where $|P|$ *is the number of nodes in the abstract syntax tree of* P.

Theorem 24 *For all* $P \in$ **Program** *and* $D \in$ **Input**,

- $S_{\text{tail}}(P, D) \le S_{\text{gc}}(P, D) \le S_{\text{stack}}(P, D)$
- $S_{\text{sfs}}(P, D) \le S_{\text{evlis}}(P, D) \le S_{\text{tail}}(P, D)$
- $S_{\text{sfs}}(P, D) \le S_{\text{free}}(P, D) \le S_{\text{tail}}(P, D)$

Proof: To prove $S_X(P, D) \le S_Y(P, D)$, consider an arbitrary space-efficient computation in $\mathcal{I}_X$. This computation is pointwise α-convertible to the gc-transform of a computation that does not use the garbage collection rule. This gc-free computation can be lifted to $\mathcal{I}_Y$ as in the proof of Theorem 19. The gc-transform of this lifted computation is a space-efficient computation in $\mathcal{I}_Y$ that is equivalent to the original computation in $\mathcal{I}_X$, and consumes at least as much space. □

Theorem 25 *All set inclusions shown in Figure 6 are proper, and* $O(S_{\text{evlis}})$ *and* $O(S_{\text{free}})$ *are incomparable.*

Proof: To show $O(S_Y) \not\subseteq O(S_X)$, it suffices to give an example of a program P such that

$$\lambda N.\, S_Y(P, \texttt{(quote } N\texttt{)}) \notin O(\lambda N.\, S_X(P, \texttt{(quote } N\texttt{)}))$$

For readability, I will write each program in full Scheme as a procedure definition. Except for the program that distinguishes S_{tail} from S_{gc}, these programs consume quadratic space in one implementation but only linear in the other. (In Scheme the linear programs are actually $O(N \log_2 N)$ because of unlimited precision arithmetic, but would be $O(N)$ with fixed precision arithmetic.)

To show $O(S_{\text{stack}}) \not\subseteq O(S_{\text{gc}})$:

```
(define (f n)
  (let ((v (make-vector n)))
    (if (zero? n)
        0
        (f (- n 1)))))
```

To show $O(S_{\text{gc}}) \not\subseteq O(S_{\text{tail}})$:

```
(define (f n) (if (zero? n) 0 (f (- n 1))))
```

To show $O(S_{\text{tail}}) \not\subseteq O(S_{\text{evlis}})$, $O(S_{\text{free}}) \not\subseteq O(S_{\text{evlis}})$, $O(S_{\text{free}}) \not\subseteq O(S_{\text{sfs}})$:

```
(define (f n)
  (define (g)
    (begin (f (- n 1))
           (lambda () n)))
  (let ((v (make-vector n)))
    (if (zero? n)
        0
        ((g)))))
```

To show $O(S_{\text{tail}}) \not\subseteq O(S_{\text{free}})$, $O(S_{\text{evlis}}) \not\subseteq O(S_{\text{free}})$, $O(S_{\text{evlis}}) \not\subseteq O(S_{\text{sfs}})$:

```
(define (f n)
  (let ((v (make-vector n)))
    (if (zero? n)
        0
        ((lambda ()
           (begin (f (- n 1)) n))))))
```

□

13 Flat versus Linked Environments

A definition of space consumption that corresponds to linked environments can be obtained by counting each binding (of an identifier I to a location α) only once per configuration, regardless of how many environments contain that binding. Recall that ρ is a finite function, so it can be viewed as a subset of **Identifier** × **Location**. Let graph(ρ) be that set of ordered pairs. Figure 8 then defines the space consumed by a configuration using linked environments.

The space consumed by an implementation $\mathcal{I}_{\text{X}}$ that uses linked environments can then be defined in terms of the space consumption function U_{X} that differs from S_{X} by using Figure 8 instead of Figure 7 in Definition 23. It is easy to see that analogues of Theorems 24 and 25 hold for linked environments, and that $U_{\text{X}} \leq S_{\text{X}}$ for each implementation $\mathcal{I}_{\text{X}}$.

In this sense, linked environments are more space-efficient than flat environments. Unfortunately, $\mathcal{I}_{\text{free}}$ and $\mathcal{I}_{\text{sfs}}$ cannot always use linked environments; in general, they require flat environments. Hence U_{free} and U_{sfs} have no practical meaning. The asymptotic relationships that hold between U_{tail}, U_{evlis}, S_{free}, and S_{sfs} therefore become a question of some practical interest.

Theorem 26 *Both $O(U_{\text{tail}})$ and $O(U_{\text{evlis}})$ are incomparable with both $O(S_{\text{free}})$ and $O(S_{\text{sfs}})$.*

Proof: Appel [App92] has given examples that show

$$O(U_{\text{evlis}}) \not\subseteq O(S_{\text{free}})$$

Since $U_{\text{evlis}} \subset U_{\text{tail}}$ and $S_{\text{sfs}} \subset S_{\text{free}}$, Appel's examples establish half of the theorem.

The other half of the theorem is established by using an example to show $O(S_{\text{sfs}}) \not\subseteq O(U_{\text{tail}})$. For each natural number k, let $E_{0,k}$ be

```
(let ((x0 n))
  (define (loop i thunks)
    (if (zero? i)
        ((list-ref thunks (random (length thunks))))
        (loop (- i 1)
              (cons (lambda ()
                      (list i x0 x1 ... xk))
                    thunks))))
  (loop n '()))
```

For each $j > 0$ let $E_{j,k} \equiv$ `(let ((xj (- n j))) `$E_{j-1,k}$`)`. Let $P_k \equiv$ `(define (f n) `$E_{k,k}$`)`. Then

$$\lambda N.\, U_{\text{tail}}(P_N, \texttt{(quote } N\texttt{)}) \in O(N \log_2 N)$$

and would be linear with fixed precision arithmetic, but

$$\lambda N.\, S_{\text{sfs}}(P_N, \texttt{(quote } N\texttt{)}) \in \Theta(N^2)$$

□

This theorem reveals an important difference between the formulation of space complexity in Section 6 of this paper and the formulation used by Appel in [App92, AS96]. Appel's formulation allows the constants of Definition 3 to be chosen separately for each program. This effectively ignores the extra space that is consumed by flat environments and closures, as well as the space consumed by a large class of compiler optimizations such as inlining and loop unrolling. As formulated here, the space safety properties allow any bounded increase in space due to inlining, loop unrolling, shared closures, and similar techniques, but the bound must be independent of the source program.

Appel's formulation has led to a perception that flat closures (over free variables only) are asymptotically more space-efficient than linked closures (that close over all variables in scope); as formulated here, however, the asymptotic space complexities of flat and linked closures are incomparable. This is consistent with the intuition of implementors who have argued that it does not make sense to impose stringent safe for space complexity requirements without also bounding the increases in space consumption that result from a compiler's transformation of the program.

14 Sanity Check

The idea of defining proper tail recursion as a space complexity class is new, so it is intensionally not the same as informal notions of proper tail recursion. Extensionally, however, it should coincide with the consensus of the Scheme and Standard ML programming communities concerning which implementations are properly tail recursive.

To the best of my knowledge, most implementations of Scheme and Standard ML are properly tail recursive by the formal definition of this paper. Some implementations are not, for one of the following reasons:

- space leaks caused by bugs.
- space leaks caused by conservative garbage collection [Boe93].
- space leaks caused by over-aggressive stack allocation or other optimizations.
- a target architecture that makes proper tail recursion difficult, hence slow, so the implementors have deliberately sacrificed proper tail recursion for speed and/or compatibility.

$$
\begin{aligned}
\text{space}(\langle v, \sigma\rangle) &= \text{space}(v) + \text{space}(\sigma)\\
\text{space}(\langle E, \rho, \kappa, \sigma\rangle) &= |\text{graph}(\rho) \cup \text{bindings}(\kappa)| + \text{space}(\kappa) + \text{space}(\sigma)\\
\text{space}(\langle v, \rho, \kappa, \sigma\rangle) &= \text{space}(v) + |\text{graph}(\rho) \cup \text{bindings}(\kappa)| + \text{space}(\kappa) + \text{space}(\sigma)\\
&\vdots\\
\text{space}(\text{CLOSURE:}\langle \alpha, L, \rho\rangle) &= 1\\
\text{space}(\textbf{halt}) &= 1\\
\text{space}(\textbf{select:}\langle E_1, E_2, \rho, \kappa\rangle) &= 1 + \text{space}(\kappa)\\
&\vdots\\
\text{bindings}(\text{CLOSURE:}\langle \alpha, L, \rho\rangle) &= \text{graph}(\rho)\\
\text{bindings}(\textbf{halt}) &= \{\,\}\\
\text{bindings}(\textbf{select:}\langle E_1, E_2, \rho, \kappa\rangle) &= \text{graph}(\rho) \cup \text{bindings}(\kappa)\\
\text{bindings}(\textbf{assign:}\langle I, \rho, \kappa\rangle) &= \text{graph}(\rho) \cup \text{bindings}(\kappa)\\
\text{bindings}(\textbf{push:}\langle\langle E_1, \ldots, E_m\rangle, \langle v_1, \ldots, v_n\rangle, \pi, \rho, \kappa\rangle) &= \text{graph}(\rho) \cup \text{bindings}(\kappa)\\
\text{bindings}(\textbf{call:}\langle\langle v_1, \ldots, v_m\rangle, \kappa\rangle) &= \text{bindings}(\kappa)\\
\text{bindings}(\textbf{return:}\langle \rho, \kappa\rangle) &= \text{graph}(\rho) \cup \text{bindings}(\kappa)\\
\text{bindings}(\textbf{return:}\langle A, \rho, \kappa\rangle) &= \text{graph}(\rho) \cup \text{bindings}(\kappa)
\end{aligned}
$$

Figure 8: The space consumed by a configuration (using linked environments).

For example, Larceny v0.29 [CH94] sometimes retains a temporary for too long within a stack frame. As with conservative garbage collection, few users would notice this bug, but it causes rare failures of proper tail recursion.

On most target architectures, for languages like Scheme that require garbage collection anyway, proper tail recursion is considerably faster than improper tail recursion. Unfortunately, the standard calling conventions of some machines[3] make proper tail recursion difficult or inefficient, so the implementor must choose between

- improper tail recursion, or
- proper tail recursion using a nonstandard and possibly slower calling convention.

This dilemma arises most often when the target architecture is ANSI C. As explained by the Bigloo user's manual [Ser97]:

> Bigloo produces C files. C code uses the C stack, so some programs can't be properly tail recursive. Nevertheless all simple tail recursions are compiled without stack consumption.

Thus Bigloo and similar implementations fail with continuation-passing style and with the `find-leftmost` example of Section 4, but most tail calls to known procedures consume no space. Implementations of this kind often assume that the global variable defined by a top-level procedure definition is never assigned, which increases the percentage of calls to known procedures beyond that shown in the last column of Figure 2.

[3]Examples include the SPARC and PowerPC. Their standard calling conventions pass some arguments in a stack frame created by the caller. If the callee performs a tail call to a procedure whose arguments require more space than was allocated in the caller's frame, then the callee must either allocate a new frame, which causes improper tail recursion, or must increase the size of its caller's frame, which is impossible on the PowerPC and slow on the SPARC. Both architectures perform very well in properly tail recursive implementations that use slightly nonstandard calling conventions.

One of the standard techniques for generating properly tail recursive C code is to allocate stack frames for all calls, but to perform periodic garbage collection of stack frames as well as heap nodes [Bak95]. A definition of proper tail recursion that is based on asymptotic space complexity allows this technique. To my knowledge, no other formal definitions do.

15 Previous Work

Scheme was invented by Steele and Sussman for the purpose of understanding Hewitt's actor model, which made heavy use of continuation-passing style and was among the first attempts to formalize proper tail recursion [SS75, Hew77]. Proper tail recursion was then popularized by Steele, Sussman, and others [SS75, WF78, SS76, Ste76, Ste77, Ste78, SS78b, SS78a, ASwS96].

Proper tail recursion is one of the safety properties considered by Chase in his analysis of optimizations that can increase the asymptotic space complexity of programs [Cha88]. The close connection between proper tail recursion, garbage collection, and asymptotic space complexity was pointed out by Appel in Chapter 12 of [App92].

Most discussions of proper tail recursion have been informal, but there have been several formal definitions within the context of a particular implementation [Ste78, Cli84, App92, FSDF93, Ram97].

My definition is essentially the same as a definition proposed by Morrisett and Harper [MH97]. My treatment of garbage collection is closely related to that of Morrisett, Felleisen, and Harper [MFH95].

16 Future Work

The reference implementations described here can be related to the denotational semantics of Scheme by proving that every answer that is computed by the denotational semantics is computed by the reference implementations.

It should be possible to prove that implementations such as MacScheme and VLISP are properly tail recursive [Cli84, Lig90, GW95]. These proofs should be much easier than proofs of their correctness, and might not be much more difficult than the proofs in Sections 11 and 12.

It should also be possible to formulate and to prove the soundness of a formal system for reasoning about the space complexity of programs written in Scheme, Standard ML, and similar languages.

17 Conclusions

The space efficiency that is required of properly tail recursive implementations can be stated in a precise and implementation-independent manner. Other kinds of safe-for-space properties can be defined in similar fashion.

The space complexity classes that are defined in this paper justify formal, implementation-independent reasoning about the space required by programs. Three of the space complexity classes defined in this paper (S_{stack}, S_{tail}, and S_{sfs}) correspond to the models that working programmers already use to reason about the space required by Algol-like, Scheme, and Standard ML programs respectively.

These complexity classes can be used to classify space leaks created by some optimizations. Algol-like stack allocation can itself be viewed as an optimization that usually delivers a small constant improvement over the space required by a garbage collector, while risking asymptotic increases in the space required to run some programs.

Acknowledgements

This paper owes much to electronic communications with Henry Baker, Hans Boehm, Olivier Danvy, Matthias Felleisen, Greg Morrisett, Kent Pitman, John Ramsdell, and Jeffrey Mark Siskind, but the mistakes are mine.

References

[AFL95] A. Aiken, M. Fähndrich, and R. Levien. Better static memory management: Improving region-based analysis of higher-order languages. In *1995 Conference on Programming Language Design and Implementation*, pages 174–185, San Diego, California, June 1995.

[App92] Andrew W. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.

[AS96] Andrew W. Appel and Zhong Shao. An empirical and analytic study of stack vs. heap cost for languages with closures. *Journal of Functional Programming*, 6(1):47–74, 1996.

[ASwS96] Hal Abelson and Gerald Jay Sussman with Julie Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, Cambridge, MA, second edition, 1996.

[Bak95] Henry Baker. Cons should not cons its arguments, part II: Cheney on the M.T.A. *ACM SIGPLAN Notices*, 30(9):17–20, September 1995.

[Boe93] Hans-Juergen Boehm. Space-efficient conservative garbage collection. In *Proc. 1993 ACM Conference on Programming Language Design and Implementation*, pages 197–206, 1993.

[CH94] William D. Clinger and Lars Thomas Hansen. Lambda, the ultimate label, or a simple optimizing compiler for Scheme. In *Proc. 1994 ACM Symposium on Lisp and Functional Programming*, pages 128–139, 1994.

[Cha88] David R. Chase. Safety considerations for storage allocation optimizations. In *Proc. 1988 ACM Conference on Programming Language Design and Implementation*, pages 1–10, 1988.

[Cli84] William Clinger. The Scheme 311 compiler: an exercise in denotational semantics. In *Proc. 1984 ACM Symposium on Lisp and Functional Programming*, pages 356–364, August 1984.

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* MIT Press/McGraw-Hill, 1990.

[FH95] Christopher W. Fraser and David R. Hanson. *A Retargetable C Compiler: Design and Implementation.* Benjamin/Cummings, 1995.

[Fis72] Michael J. Fischer. Lambda calculus schemata. In *Proceedings of the ACM Conference on Proving Assertions about Programs*, pages 104–109. ACM SIGPLAN, SIGPLAN Notices, Vol. 7, No 1 and SIGACT News, No 14, January 1972.

[FSDF93] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *Proc. 1993 ACM Conference on Programming Language Design and Implementation*, pages 237–247, 1993.

[GW95] Joshua D. Guttman and Mitchell Wand (editors). *VLISP: A Verified Implementation of Scheme.* Kluwer, Boston, 1995. Originally published as a special double issue of the journal *Lisp and Symbolic Computation* (Volume 8, Issue 1/2).

[Han90] Chris Hanson. Efficient stack allocation for tail-recursive languages. In *Proc. 1990 ACM Symposium on Lisp and Functional Programming*, 1990.

[Hew77] Carl Hewitt. Viewing control structures as patterns of passing messages. *Artificial Intelligence*, 8:323–364, 1977.

[IEE91] IEEE Computer Society, New York. *IEEE Standard for the Scheme Programming Language*, IEEE standard 1178-1990 edition, 1991.

[KCR98] Richard Kelsey, William Clinger, and Jonathan Rees (eds.). Revised^5 report on the algorithmic language Scheme.

`ftp://ftp.nj.nec.com/pub/kelsey/r5rs.ps`, February 1998.

[KKsR+86] David A. Kranz, Richard Kelsey, Jonathan A. Rees, Paul Hudak, James Philbin, and Norman I. Adams. Orbit: An optimizing compiler for Scheme. In *Proceedings SIGPLAN '86 Symposium on Compiler Construction*, 1986. *SIGPLAN Notices 21*(7), July, 1986, 219-223.

[Knu73] Donald A. Knuth. *The Art of Programming, Volume 1 / Fundamental Algorithms.* Addison-Wesley, Boston, second edition, 1973.

[Lig90] Lightship Software. *MacScheme Manual and Software.* The Scientific Press, 1990.

[MFH95] Greg Morrisett, Matthias Felleisen, and Robert Harper. Abstract models of memory management. In *Proc. 1995 ACM Conference on Functional Programming and Computer Architecture*, pages 66–77, 1995.

[MH97] G. Morrisett and R. Harper. Semantics of memory management for polymorphic languages. In A. D. Gordon and A. M. Pitts, editors, *Higher Order Operational Techniques in Semantics*, Publications of the Newton Institute, pages 175–226. Cambridge University Press, 1997.

[NN92] Hanne Riis Nielson and Flemming Nielson. *Semantics with Applications: a Formal Introduction.* John Wiley & Sons, 1992.

[Que96] Christian Queinnec. *Lisp in Small Pieces.* Cambridge University Press, 1996.

[Ram94] John D. Ramsdell. Scheme: the next generation. *ACM Lisp Pointers*, 7(4):13–14, Oct-Dec 1994.

[Ram97] John D. Ramsdell. The tail recursive SECD machine. To appear, 1998.

[Ser97] Manuel Serrano. Bigloo user's manual. Part of the Bigloo 1.9 distribution available via

`http://cuiwww.unige.ch/~serrano/bigloo.html`, 1997.

[SF96] Manuel Serrano and Marc Feeley. Storage use analysis and its applications. In *Proc. 1996 ACM International Conference on Functional Programming*, pages 50–61, 1996.

[Sis97] Jeffrey Mark Siskind. Stalin - a STAtic Language ImplementatioN. Software available via

`http://www.neci.nj.nec.com/homepages/qobi/`, 1997.

[SS75] Gerald J. Sussman and Guy L. Steele, Jr. Scheme: An interpreter for extended lambda calculus. Artificial Intelligence Memo 349, Massachusetts Institute of Technology, Cambridge, MA, December 1975.

[SS76] Guy L. Steele, Jr. and Gerald Jay Sussman. Lambda: The ultimate imperative. Artificial Intelligence Memo 353, Massachusetts Institute of Technology, Cambridge, MA, March 1976.

[SS78a] Guy L. Steele, Jr. and Gerald Jay Sussman. The art of the interpreter or, the modularity complex (parts zero, one and two). Artificial Intelligence Memo 453, Massachusetts Institute of Technology, Cambridge, MA, May 1978.

[SS78b] Guy L. Steele, Jr. and Gerald Jay Sussman. The revised report on SCHEME. Artificial Intelligence Memo 452, Massachusetts Institute of Technology, Cambridge, MA, January 1978.

[Ste76] Guy L. Steele, Jr. Lambda: The ultimate declarative. Artificial Intelligence Memo 379, Massachusetts Institute of Technology, Cambridge, MA, October 1976.

[Ste77] Guy Lewis Steele, Jr. Debunking the "expensive procedure call" myth. In *ACM National Conference*, pages 153–162, Seattle, October 1977. Revised as MIT AI Memo 443, October 1977.

[Ste78] Guy L. Steele, Jr. Rabbit: A compiler for Scheme. Technical Report 474, Massachusetts Institute of Technology, Cambridge, MA, May 1978.

[TT94] Mads Tofte and Jean-Pierre Talpin. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *Proceedings 21st Annual ACM Symposium on Principles of Programming Languages*, pages 188–201, 1994.

[Wan80] Mitchell Wand. Continuation-based program transformation strategies. *Journal of the ACM*, 27:164–180, 1980.

[WF78] Mitchell Wand and Daniel P. Friedman. Compiling lambda expressions using continuations and factorizations. *Journal of Computer Languages*, 3:241–263, 1978.