Safe-for-Space Linked Environments

Matthew Flatt
University of Utah
Salt Lake City, USA
mflatt@cs.utah.edu

Robert Bruce Findler
Northwestern University
Evanston, IL, USA
robby@cs.northwestern.edu

Abstract

Techniques for implementing a call-by-value λ-calculus are well known, including Reynolds's definitional interpreter and techniques that Danvy developed for moving between reduction rules and abstract machines. Most techniques focus on ensuring that an implementation produces the same evaluation result as a model, but time and space properties are also within reach. Proper tail-call handling falls out of Reynolds's approach, for example, but the stronger property of *space safety* is less readily available, particularly if worst-case time complexity matters. In this paper, we explore an approach to space safety that is realized through the garbage collector, instead of the interpreter loop, in the hope of finding a convenient implementation technique that matches the asymptotic time bounds of fast evaluation models and the asymptotic space bounds of compact evaluation models. We arrive within a size-of-source factor of achieving those bounds. Our implementation technique is comparable in complexity to some other interpreter variants; specifically, it requires some library functions for binary trees, specialized environment traversals in the garbage collector, and a compilation pass to gather the free variables of each expression and to rewrite each variable reference to its binding depth.

CCS Concepts: • **Software and its engineering → Semantics**.

Keywords: definitional interpreters, asymptotic complexity

ACM Reference Format:
Matthew Flatt and Robert Bruce Findler. 2025. Safe-for-Space Linked Environments. In *Proceedings of the Workshop Dedicated to Olivier Danvy on the Occasion of His 64th Birthday (OLIVIERFEST '25), October 12–18, 2025, Singapore, Singapore*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3759427.3760379

1 Interpreters in Time and Space

Reynolds's approach to a definitional interpreter [21] provides a clear roadmap to a language implementer who aims to build a simple but practical functional language that runs on conventional hardware. Danvy and collaborators [10, 1, 5, 4, 9, 8], as well as others [20, 24, 12, 14], have explored and generalized the technique, providing implementers with multiple paths to success, as well as some formal assurance that an interpreter following the recipe will compute results that are faithful to a starting model as a specification.

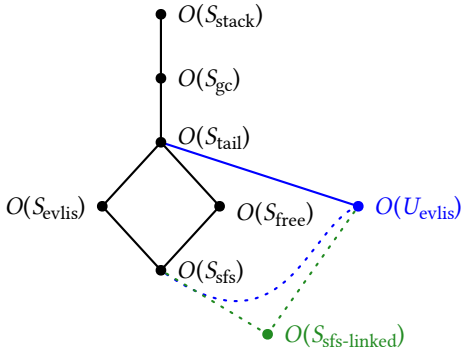


Figure 1. Space-complexity lattice from Clinger [7], where U_{evlis} is described informally there (section 13), while $S_{\text{sfs-linked}}$ is our addition that is below S_{sfs} and $S_{\text{sfs-linked}}$ given a bound on program-source size.

Those guarantees generally concern only the result value, and not the time or space consumption of an implementation that produces the value, however. At a minimum, a programmer following Reynolds's roadmap must supplement his `eval`, `apply`, and `cont` functions with memory management. Fortunately, a semispace copying collector is simple to write and fits naturally with the register-oriented loop of a Reynolds-style interpreter. Even better, a call-by-value λ-calculus implementation will get tail calls right.

Alas, the interpreter might still use more space than a programmer might reasonably expect. In particular, if a variable x, bound to a value v, is not referenced after a certain point, then v should not be retained after that point. An interpreter is *safe for space* if it retains the values of only those variables that appear in a continuation—or, at least, if its asymptotic space consumption is consistent with such an evaluation. *Safe for space* was first described by Appel [2] and was formalized by Clinger [7], who defined a lattice of space complexity shown in Figure 1. The behavior that falls out of Reynolds's approach corresponds roughly to S_{tail}, which includes correct handling of tail calls, but not S_{sfs}, which is safe for space. For many practical purposes, S_{tail} is fine, but

This work is licensed under a Creative Commons Attribution 4.0 International License.
OLIVIERFEST '25, Singapore, Singapore
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2150-2/25/10
https://doi.org/10.1145/3759427.3760379

its space leak relative to S_{sfs} can sometimes matter, such as for applications that use thunks to implement lazy streams.

A Reynolds-style implementation will tend to achieve S_{tail} but not S_{sfs}, because a closure or continuation frame created by `eval` retains the full environment at the point where a λ abstraction is encountered or a `push` frame is created. An implementation that achieves S_{sfs} must keep environment entries for only variables that are mentioned in the function body or pushed expression. The difference is illustrated by a program that uses

```
(let ([f (λ (f) (λ (y)
                  ((f f) (λ (z) z))))])
  ((f f) (λ (x) x)))
```

where retaining `y` in the closure for `(λ (z) z)` will cause space use to grow without bound, while a S_{sfs} implementation will loop in constant space. A typical S_{sfs} implementation copies the environment to *flatten* it into one that contains only needed variables whenever a closure value is produced or an argument expression is pushed onto the continuation. That strategy is simple, but it comes with a new cost: flattening takes time proportional to the size of the flat environment.

Before we move on to a new approach, we must consider yet one more wrinkle. Depending on the way that environments are represented, a Reynolds-style interpreter may actually perform better than S_{tail} and in a way that can be better than S_{sfs}, but that is not consistently better than S_{sfs}. Specifically, a S_{tail} variant with *linked environments* (such as implemented by a linked list), which we designate U_{evlis}, can use less space [7 section 13]. With linked environments, each extension of an environment shares storage with the extended environment. Linked environments are particularly natural when building on a low-level language, because allocating to the front of a linked list is a constant-time operation. Implementing S_{sfs} by flattening loses that sharing, so it potentially increases space usage, after all, compared to U_{evlis}. The difference is illustrated by a program that uses

$$(\lambda\ (x_1) \ldots (\lambda\ (x_N)\ (\lambda\ (y)\ (\lambda\ (z)\ (x_1 \ldots\ x_N)))))$$

and passes curried arguments up to the function that accepts `y`, then calls the function that accepts `y` N times. The N $(\lambda\ (z)\ (x_1 \ldots\ x_N))$ results are all derived from the same environment that has x_1 through x_N, so the results will together use $O(N)$ space with linked environments, but $O(N^2)$ space with flat closures. Overall, S_{sfs} and U_{evlis} are both better than S_{tail}, but this example illustrates how S_{sfs} is sometimes worse than U_{evlis}, while the earlier example is a case where S_{sfs} is better than U_{evlis}.

Shao and Appel [23] describe an optimization approach that starts with S_{sfs} but improves sharing. The optimization moves an implementation closer to the best of S_{sfs} and U_{evlis}, but it is not guaranteed. More generally, since sharing through the optimization is limited to two layers, it cannot achieve enough sharing to ensure that space consumption matches U_{evlis}.

In this paper, we describe an implementation of the call-by-value λ-calculus that is safe for space and that also preserves sharing for linked environments. We get close to an implementation whose asymptotic space and time consumption, $S_{\text{sfs-linked}}$, is bounded by both S_{sfs} and U_{evlis} with linked environments.[1] We do not quite achieve that ideal, because our strategy in the worst case involves an extra factor on both time and space that corresponds to a source program's size. In practice, programs are relatively small, and the relevant factor is usually much smaller.

Instead of flattening environments as part of an interpreter step, which is the way that S_{sfs} is defined, we achieve $S_{\text{sfs-linked}}$ by viewing space safety as a garbage-collection problem. In the same way that a scavenging garbage collector refrains from keeping data that is not reachable structurally, our collector refrains from keeping an environment entry when no expression paired with the environment refers to the entry's variable. The garbage collector may need to traverse the same environment multiple times when the same environment is paired with multiple expressions, but the number of distinct, relevant free-variable sets is bounded by the size of the program's source text. Meanwhile, random access to variables is made logarithmic-time (which we consider effectively constant) by representing an environment or free-variable set as a binary tree; each variable is represented and located in the tree by its binding depth.

We offer the following results:

- an executable abstract-machine model of evaluation and garbage collection that can be used to define both space complexity and time complexity; and
- a machine variant whose asymptotic time and space complexity $S_{\text{sfs-linked}}$ is bounded by both S_{sfs} and U_{evlis}, at least for a bounded program size.

As an evaluation of these results, we offer

- an example implementation in C as the kernel of a tiny variant of Racket, illustrating that the implementation is practical at a low level of abstraction; and
- empirical measurements of the step count and heap size of the model whose curves have the desired shapes.

2 Practical Motivation

A language implementation like Chez Scheme or Racket, which has an optimizing compiler and a sophisticated runtime system, can use various techniques to properly implement tail calls and space safety. The *build* process for Chez Scheme and Racket, however, relies on Zuo, which is an additional, tiny variant of Racket that is implemented as an interpreter in C. Our work here is motivated by the question

[1] We're repurposing space categories like S_{sfs} as characterizing both time and space. The abstract machines that Clinger uses to define S_{tail} and S_{sfs} have a natural interpretation as time complexities by counting reduction steps, except that an environment-closing step in S_{sfs} needs to be expanded to multiple steps. We provide a machine to make that explicit in section 5.

$$
\begin{array}{lll}
e ::= (\lambda\ (x)\ e) \mid x \mid (e\ e) \mid \text{'}lit & K ::= [] \mid k :: K & \sigma ::= integer \\
v ::= \langle \text{obj}\ \sigma \rangle \mid \langle \text{prim}\ lit \rangle & k ::= \langle t\ m \rangle & \rho ::= \sigma \mid \text{mt} \\
m ::= \langle e, \rho \rangle \mid v & t ::= \text{arg} \mid \text{app} & \Sigma ::= \{\sigma{=}b, \ldots\} \\
x ::= variable & \quad \mid \text{ret} & b ::= \langle \text{env}\ \langle x, v, \rho \rangle \rangle \\
 & & \quad \mid \langle \text{clos}\ \langle e, \rho \rangle \rangle
\end{array}
$$

$$(\text{let}\ ([x\ e_{rhs}])\ \ e_{body}) \ = \ ((\lambda\ (x)\ e_{body})\ e_{rhs})$$

Figure 2. Expressions, continuations, and store

of whether a simple interpreter at the C level of abstraction can be made safe-for-space in a way that is well-founded and still simple—analogous to the way that Reynolds's approach provides proper tail-call handling for such interpreters.

Simplicity in this context is particularly practical and specific: Zuo's kernel is implemented in a single C file with no header files other than standard C and system headers, and it needs no libraries other than default C and system libraries. Zuo can be compiled with a C compiler with no command-line arguments other than the source file's path. From there, Zuo scripts build Chez Scheme and Racket. Maintaining yet another Scheme implementation for a build system may sound like overkill, but orchestrating the build of Racket on top of Chez Scheme—each of which has some C-implemented portions and complex dependencies on third-party libraries—has proved challenging. Using Racket itself would be ideal, but relying on a language to build the same language has well-known drawbacks. Our modest investment in Zuo has paid off, as Zuo scripts have successfully replaced more than 10k lines of fragile makefiles and Visual Studio project files that formerly comprised the Racket build process.

Building Chez Scheme and Racket is the only definite use of Zuo. To minimize effort and maintenance, Zuo is a textbook, Reynolds-style interpreter with a semispace collector. The implementation has been stable and perfectly adequate for a build system. At the same time, Zuo seems potentially useful for other purposes, and for a language that is carefully constructed otherwise, it seems a shame that it doesn't get asymptotic space consumption right.

3 Interpreter

Following Clinger [7], we model program evaluation as a small-step operational semantics, which allows us to clearly characterize both the time and space needed to evaluate a program. Clinger's model includes branching, assignment, and multiple-argument functions, which we omit here. We defer branching to primitives, while assignment and multiple-argument functions are not essential for our purposes.

The leftmost column of Figure 2 shows the expression and value grammar for our model's λ-calculus variant:

- For an expression e, in addition to function abstraction, variables, and function application, we include an unspecified set of primitive values that are written as quoted literals, 'lit. Primitives include operations like addition and multiplication, which always accept only primitive arguments but may return bounded, closed function abstractions to implement control flow (e.g., returning $(\lambda\ (x)\ (\lambda\ (y)\ x))$ to represent "true").
- A value v is distinct from an expression e. Our model includes a step to convert a function or literal expression form to a value. Closures for functions are allocated and represented by $\langle \text{obj}\ \sigma \rangle$ and primitive values are represented as $\langle \text{prim}\ lit \rangle$.
- An interpreter mode m is (for now) either an expression with an environment, $\langle e, \rho \rangle$, or a value, v. The $\langle e, \rho \rangle$ mode corresponds to "evaluate" mode in Reynolds's interpreter, while v corresponds to "continue" mode.

The middle column of Figure 2 shows the representation of continuations used by the model. A continuation K is a list of frames k, where every frame has the same shape: a tag t paired with a configuration m. The primary continuation tags are arg and app, which represent a pending argument expression while a function expression is evaluated, and a pending application of a function value while an argument expression is evaluated, respectively. The ret tag will be used for garbage collection. We grow a continuation K by adding a k to the front of the list. We do not allocate continuations or continuation frames, since we do not include an operation to capture a continuation, but the length of a continuation does count toward space consumption.

The rightmost column of Figure 2 shows the store:

- A σ is an allocated address, represented as a natural.
- An environment ρ is either the address of an allocated environment frame, or it is empty: mt.
- A store Σ maps allocated addresses, σ, to records, b.
- An allocated record b is either an environment frame or a closure, since those are the only two entities that we need to allocate explicitly. An environment frame $\langle \text{env}\ \langle x, v, \rho \rangle \rangle$ binds x to v and chains to the rest of the environment ρ. A closure $\langle \text{clos}\ \langle e, \rho \rangle \rangle$ pairs the expression e with its environment ρ.

These pieces (or, at least, most of them) are assembled in the interpreter shown in Figure 3. The evaluation rules in the left-hand side of the figure generally follow Clinger's model. The first four rules are the "reduction" or `interp` steps, while the last three rules are the `continue` steps. The right-hand side of each rule clearly matches the left-hand side of either an `interp` rule or a `continue` rule, so evaluation could be written as tail calls, as in Reynolds [21]'s presentation.

The rules of Figure 3 are mostly standard:

- [lam] allocates a closure for a function expression, which updates the store Σ and moves to `continue` mode. The current size of the store Σ can be used as the newly allocated address.

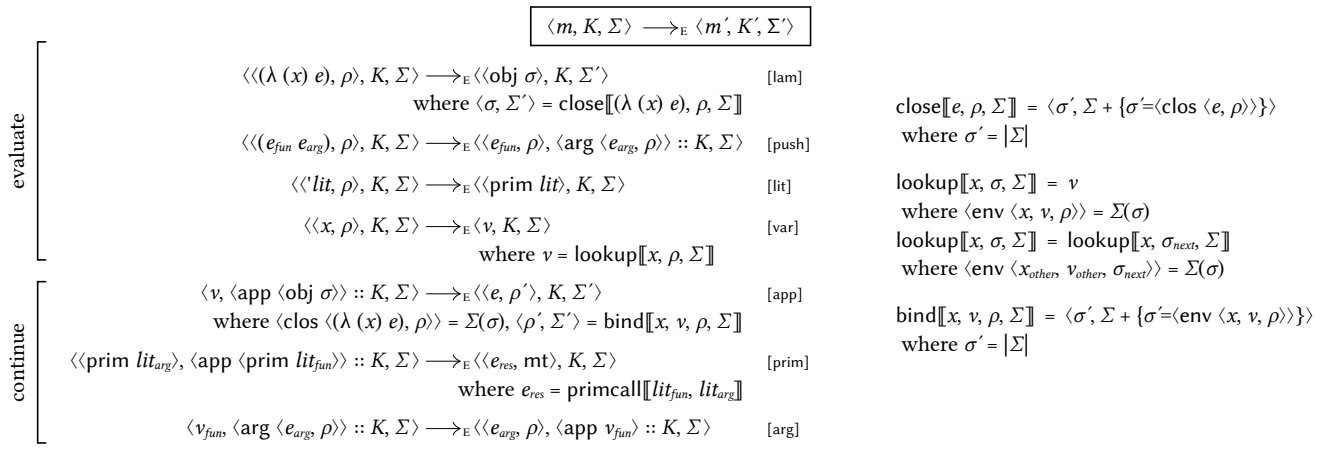


Figure 3. Interpreter

- [push] starts evaluation of the function part of an application form, moving the argument expression into the continuation, and stays in `interp` mode.
- [lit] converts a primitive-literal expression to a primitive-literal value and moves to `continue` mode.
- [var] locates a variable's value in the store and moves to `continue` mode. Internally, the lookup metafunction recurs to follow the environment-linked list to locate a value for the variable x.
- [app] begins evaluation of an applied function's body, binding an argument value through a freshly allocated environment frame, and then moves to `interp` mode.
- [prim] applies a primitive function to a primitive value. The result can be any expression, but it is paired with the mt environment, so we are assuming that primitives produce only closed terms. For example, the result might be a quoted literal for an arithmetic result, a primitive adder that has received its first argument and needs a second, or a function form such as $(\lambda\ (x)\ (\lambda\ (y)\ x))$ to represent a boolean result. In any case, we assume that the representation of primitive values is simple enough to be irrelevant to space and time consumption. Evaluation moves back to `interp` mode to handle the result of the primitive operation.
- [arg] ends evaluation of an applied function expression, and starts evaluation of the argument expression, shifting the function value into the continuation and moving back into the `interp` state.

The right-hand side of Figure 3 provides helper metafunctions for the interpreter. Each metafunction is meant to represent a constant- or $O(log\ n)$-time operation. As written, lookup suggests an $O(n)$-time operation, since it walks through a linked list of environment frames, but we return to this point and describe a tree-based representation of environments in section 7.

4 Garbage Collection

Clinger [7] models garbage collection abstractly by the describing the result heap it must produce. Much other past work similarly characterizes garbage collection [18, 3], sometimes to prove correctness of an implementation [16, 19, 22]. Here, we take the specification as given and focus on a specific implementation in terms of small-step reductions. This choice enables us to measure the concrete state size to characterize space consumption and to count the number of steps to characterize time consumption.

To keep the overall interpreter as simple as possible, we implement a stop-the-world, semispace copying collector [17 section 4.1], and we treat its performance as an upper bound for any reasonable implementation of garbage collection. We do not need to worry about bit-level of encodings of objects, tags, and pointers; a representation of the store as a map from addresses to records is sufficient.

A reminder of how garbage collection is characterized in the *tricolor model* [11, 17 section 2.2]: garbage collection starts by painting all objects white, and only the objects that end up **black** at the end of garbage collection are retained. During a collection, white objects are ones that have not yet been found as reachable, gray objects are reachable with outgoing references that have not yet been swept, and **black** objects are reachable and reference only objects that are also gray or **black**. The garbage collector starts by painting objects reachable from roots as gray. It then iterates by selecting a gray object, painting it **black**, and painting as gray every white object that is immediately referenced by the newly **black** object (leaving as gray or **black** any immediately referenced object that is not still white).

In our semispace copying collector, objects are copied from Σ_{from} to Σ_{to}, obtaining an address σ_{to} in Σ_{to} that may be different from the original address σ_{from} in Σ_{from}. Objects that are only in Σ_{from} count as being painted white. An object at σ_{from} in Σ_{from}

$$A ::= \{\sigma{=}\sigma, \ldots\} \qquad g ::= \langle \text{roots } K\rangle \mid \text{sweep}$$
$$S ::= [] \mid \sigma :: S \qquad n ::= \textit{integer}$$

$$\boxed{\langle g, K, \Sigma_{from}, A, \Sigma_{to}, S\rangle \longrightarrow_{\text{G}} \langle g', K', \Sigma_{from}, A', \Sigma'_{to}, S'\rangle}$$

collect

$$\langle\langle \text{roots } \langle t\ \langle e, \rho_{from}\rangle\rangle :: K_{root}\rangle, K, \Sigma_{from}, A, \Sigma_{to}, S\rangle \longrightarrow_{\text{G}} \langle\langle \text{roots } K_{root}\rangle, K +\!\!+ [\langle t\ \langle e, \rho_{to}\rangle\rangle], \Sigma_{from}, A', \Sigma_{to}, S'\rangle \quad \text{[root-env]}$$
$$\text{where } \langle \rho_{to}, A', S'\rangle = \text{retain-env}[\![\rho_{from}, A, S]\!]$$

$$\langle\langle \text{roots } \langle t\ v\rangle :: K_{root}\rangle, K, \Sigma_{from}, A, \Sigma_{to}, S\rangle \longrightarrow_{\text{G}} \langle\langle \text{roots } K_{root}\rangle, K +\!\!+ [\langle t\ v'\rangle], \Sigma_{from}, A', \Sigma_{to}, S'\rangle \quad \text{[root-val]}$$
$$\text{where } \langle v', A', S'\rangle = \text{retain-val}[\![v, A, S]\!]$$

$$\langle\langle \text{roots } []\rangle, K, \Sigma_{from}, A, \Sigma_{to}, S\rangle \longrightarrow_{\text{G}} \langle \text{sweep}, K, \Sigma_{from}, A, \Sigma_{to}, S\rangle \quad \text{[roots-done]}$$

$$\langle \text{sweep}, K, \Sigma_{from}, A, \Sigma_{to}, \sigma_{from} :: S\rangle \longrightarrow_{\text{G}} \langle \text{sweep}, K, \Sigma_{from}, A'', \Sigma_{to} + \{\sigma_{to}{=}\langle \text{env } \langle x, v', \rho_{to}\rangle\rangle\}, S''\rangle \quad \text{[sweep-env]}$$
$$\text{where } \sigma_{to} = A(\sigma_{from}),\ \langle \text{env } \langle x, v, \rho_{from}\rangle\rangle = \Sigma_{from}(\sigma_{from}),\ \langle v', A', S'\rangle = \text{retain-val}[\![v, A, S]\!],$$
$$\langle \rho_{to}, A'', S''\rangle = \text{retain-env}[\![\rho_{from}, A', S']\!]$$

$$\langle \text{sweep}, K, \Sigma_{from}, A, \Sigma_{to}, \sigma_{from} :: S\rangle \longrightarrow_{\text{G}} \langle \text{sweep}, K, \Sigma_{from}, A', \Sigma_{to} + \{\sigma_{to}{=}\langle \text{clos } \langle e, \rho_{to}\rangle\rangle\}, S'\rangle \quad \text{[sweep-clos]}$$
$$\text{where } \sigma_{to} = A(\sigma_{from}),\ \langle \text{clos } \langle e, \rho_{from}\rangle\rangle = \Sigma_{from}(\sigma_{from}),\ \langle \rho_{to}, A', S'\rangle = \text{retain-env}[\![\rho_{from}, A, S]\!]$$

$$\text{retain-env}[\![\text{mt}, A, S]\!] = \langle \text{mt}, A, S\rangle$$
$$\text{retain-env}[\![\sigma_{from}, A, S]\!] = \text{retain}[\![\sigma_{from}, A, S]\!]$$

$$\text{retain-val}[\![\langle \text{prim } \textit{lit}\rangle, A, S]\!] = \langle\langle \text{prim } \textit{lit}\rangle, A, S\rangle$$
$$\text{retain-val}[\![\langle \text{obj } \sigma_{from}\rangle, A, S]\!] = \langle\langle \text{obj } \sigma_{to}\rangle, A', S'\rangle \quad \text{where } \langle \sigma_{to}, A', S'\rangle = \text{retain}[\![\sigma_{from}, A, S]\!]$$

$$\text{retain}[\![\sigma_{from}, A, S]\!] = \langle \sigma_{to}, A, S\rangle \quad \text{where } \sigma_{to} = A(\sigma_{from})$$
$$\text{retain}[\![\sigma_{from}, A, S]\!] = \langle \sigma_{to}, A + \{\sigma_{from}{=}\sigma_{to}\}, S +\!\!+ [\sigma_{from}]\rangle \quad \text{where } \sigma_{from} \notin \text{dom}(A),\ \sigma_{to} = |A|$$

Figure 4. Garbage collector

is painted gray by reserving an address σ_{to} in Σ_{to}. An object at σ_{from} is conceptually painted **black** by ensuring that σ_{to} has a copy of the object, and by updating the immediate references in the copy to also point into Σ_{to}—which may involve painting some of those referenced objects gray by allocating addresses for them. A low-level implementation of a copying collector uses some of the space at σ_{from} in Σ_{from} to store a corresponding σ_{to} for a gray object, but our rules instead record forwarding for gray objects in an "allocation" map, A. Also, a low-level implementation finds a next gray object by iterating through Σ_{from} [6], but our evaluation uses a separate "sweep" queue, S. Figure 4 defines A and S.

Overall, the garbage collector's state has six components:

- g: The current mode of the collector, which is either roots for finding initial gray objects, or sweep for sweeping gray objects to **black**. A garbage collection starts in roots and eventually moves to sweep, and it completes when sweep is done.
  The roots mode has a list of roots yet to process, which can be represented simply as a continuation K. The interpreter's current environment or value as m must also count as a root, but our model will shift that component into the continuation as a ret frame and then move it back out afterward, so we only need to consider a K as the roots of a garbage collection.
  The sweep mode will continue as long as the S queue of gray objects is non-empty.
- K: Reconstructed roots (i.e continuation frames), which have store references changed from σ_{from} to σ_{to}. This component starts out as the empty continuation. As roots mode processes each continuation frame, it accumulates an updated frame. After sweep mode, the K component changes no further, and it can be used directly at the end of sweep mode to resume evaluation.
- Σ_{from}: The store from before garbage collection starts. This component does not change, and it is consulted to find the content of a gray object as it is copied into Σ_{to} to paint it **black**.
- A: The forwarding map from Σ_{from} addresses to Σ_{to} addresses. This component changes each time an object goes from white to gray. Although adding to A conceptually corresponds to allocating in Σ_{to}, our model defers updating Σ_{to} until the corresponding object is copied in transition from gray to **black**.
- Σ_{to}: An accumulating store to use after garbage collection completes. This new store contains an entry for each object that has been painted **black**, and it changes as a gray object is swept to paint it **black**.
- S: A queue of gray objects yet to be painted **black**. The queue contains addresses in Σ_{from} so that an object's content can be copied to Σ_{to} at an address previously recorded in A. Every address in S represents a unique object, since an object is painted gray at most once. Garbage collection competes when S is empty.

Figure 4 shows the garbage-collection rules. Aside from the transition rule [roots-done], there are two roots rules and two sweep rules, because we have two kinds of allocated objects. The retain-env metafunction must ignore mt, and

retain-val must similarly ignore primitive literals, otherwise they both defer to retain. The retain function checks whether the given address is already allocated, returning the new address if so. Otherwise it allocates an address for the object and paints it gray by adding to both A and S.

Since allocated objects have a fixed immediate size, each step in the garbage collector can be straightforwardly implemented as a constant-time operation. Each root is processed once by a roots rule, and each reachable object is processed once by a sweep rule, so the total time is limited by the continuation length plus the store size. The use of S as a queue justifies its low-level implementation as a pointer that advances through newly allocated addresses σ_{to} within Σ_{to}. In that case, each record at σ_{from} in Σ_{from} must be copied eagerly into Σ_{to} by retain, so that the content is available for [sweep-rule] or [sweep-clos] using only σ_{to}; we leave that adjustment as an exercise for the reader.

Figure 5 combines the interpreter and garbage-collector rules. On each interpreter step, a counter is decremented, and when the counter reaches -1, the reduction rule switches to garbage-collection mode. In garbage-collection mode, the counter increments with each step. The counter value at the end of a garbage collection becomes the fuel for the next evaluation phase, which means that garbage collection does not change a program's asymptotic time complexity: garbage-collection work can be charged to a matching evaluation step, so the number of steps is at most doubled. Meanwhile, each allocation during evaluation costs a step, so space complexity S_{tail} as defined by Clinger is also preserved: the store can grow to no more than twice as large as constantly performing a zero-cost abstract garbage collection.

Since environments are explicitly linked in our representation, asymptotic space complexity is actually better than $O(S_{\text{tail}})$, and we call it $O(U_{\text{evlis}})$. The example from the introduction illustrates a case where U_{evlis} is better than S_{tail}. Here is the example again, written more fully using library abbreviations that are defined in the appendix:

```
(let ([D (λ (x1) ... (λ (xN) (λ (y) (λ (z) (x1 ... xN)))))])
  (let ([R ((D '1) ... 'N)])
    (((foldn 'N) (λ (a) ((cons a) (R '0)))) empty)))
```

The closure for R will have an environment with N variables, and the result of calling R will be a fresh closure that extends that environment. If the environment is duplicated for every closure, as implicitly in the S_{tail}'s model [7], then the N-element list constructed by the fold will use $O(N^2)$ space. When the same allocated environment for R is used for every result of R, then the list will use $O(N)$ space.

5 Safe-for-Space Evaluation

The interpreter defined by Figure 5 benefits from using linked closures, but it is not safe for space. The [lam] rule in Figure 3 keeps the full environment ρ in the newly allocated closure, and the [arg] rule in Figure 3 keeps the full environment ρ in the newly allocated arg continuation frame. A typical strategy for space safety is to *flatten* an environment to keep only the free variables of an expression when the environment is paired with the expression.

Figure 6 provides replacement [lam] and [arg] rules to flatten environments. Flattening cannot be performed in a single step, because it can involve an arbitrary number of free variables. Evaluation by other rules is therefore paused by wrapping the current expression with flat, combining it with a set of variables that starts out as the expression's free variables. We assume that free variables have been computed once before evaluation starts, and the free-vars metafunction looks up the free variables of a particular expression.

In addition to the original expression and a set of variables, the flat state (added to m at the top of Figure 6) includes two environments: one that starts out as the relevant portion of the original environment and shrinks as variables are processed, and another that starts as the empty environment and accumulates new bindings as variables are processed. At each step, the skip-to metafunction jumps to the remaining portion of the relevant environment based on the remaining variables to process. The [lam] and [arg] rules use skip-to to find ρ_{next}, which is the relevant portion of the original ρ. The [arg] rule starts flattening for the argument expression to eventually move it into the continuation, but the function expression's evaluation (which happens before the argument expression's evaluation), will continue to use the current ρ, so it it saved in a ret continuation frame.

Overall, this handling of flat mode is implemented by three new reduction rules:

- The [flat] rule peels off one variable from the set that started out as all free variables. It finds the value of that variable in the remaining relevant portion of the original ρ, and it creates a new environment frame to accumulate onto ρ_{flat}. It also advances ρ to ρ_{next} based on the remaining variables to process.
- The [flat-lam] rule recognizes when bindings for all free variables have been copied for a function expression, and it allocates the closure with the flattened ρ_{flat}.
- The [flat-push] rule similarly recognizes when bindings for all free variables have been copied for an argument expression. It creates an arg continuation frame using the flattened ρ_{flat}, and then it shifts evauation back to the saved function expression with its saved original environment, ρ.

Reducing the ρ in a flat form while also reducing the set of variables is not strictly necessary. In the case of a flat form created by [arg], the original ρ is retained in a ret continuation frame, anyway. In the case of a flat form created by [lam], the original ρ would be kept only a little while longer as ρ_{flat} is built up. We nevertheless use skip-to to reduce ρ in flat because it sets up an idea for our next interpreter. Like the lookup metafunction of Figure 3, skip-to is shown in terms of

$$\begin{array}{rl} \textit{config} ::= & \langle \text{eval}, n, m, K, \Sigma \rangle \\ | & \langle \text{gc}, n, g, K, \Sigma, A, \Sigma, S \rangle \end{array}$$

$$\boxed{\textit{config} \longrightarrow \textit{config}'}$$

$$\langle \text{eval}, n, m, K, \Sigma \rangle \longrightarrow \langle \text{eval}, n{-}1, m', K', \Sigma' \rangle \quad \text{[eval]}$$
$$\text{where } n \geq 0, \langle m, K, \Sigma \rangle \longrightarrow_{\text{E}} \langle m', K', \Sigma' \rangle$$

$$\langle \text{eval}, \text{-1}, m, K, \Sigma \rangle \longrightarrow \langle \text{gc}, 0, \langle \text{roots } \langle \text{ret } m \rangle :: K \rangle, [], \Sigma, \emptyset, \emptyset, [] \rangle \quad \text{[start-gc]}$$

$$\langle \text{gc}, n, g, K, \Sigma_{from}, A, \Sigma_{to}, S \rangle \longrightarrow \langle \text{gc}, n{+}1, g', K', \Sigma'_{from}, A', \Sigma'_{to}, S' \rangle \quad \text{[gc]}$$
$$\text{where } \langle g, K, \Sigma_{from}, A, \Sigma_{to}, S \rangle \longrightarrow_{\text{G}} \langle g', K', \Sigma'_{from}, A', \Sigma'_{to}, S' \rangle$$

$$\langle \text{gc}, n_{gc}, \text{sweep}, \langle \text{ret } m \rangle :: K, \Sigma_{from}, A, \Sigma_{to}, [] \rangle \longrightarrow \langle \text{eval}, n_{gc}, m, K, \Sigma_{to} \rangle \quad \text{[end-gc]}$$

Figure 5. Combined interpreter and garbage collector

$$m ::= .... | \langle \text{flat } \langle e, \{x, ...\}, \rho, \rho \rangle \rangle$$

$$\boxed{\langle m, K, \Sigma \rangle \longrightarrow_{\text{E}} \langle m', K', \Sigma' \rangle}$$

evaluate

$$\langle \langle (\lambda\ (x)\ e), \rho \rangle, K, \Sigma \rangle \longrightarrow_{\text{E}} \langle \langle \text{flat } \langle (\lambda\ (x)\ e), \{x_{free}, ...\}, \rho_{live}, \text{mt} \rangle \rangle, K, \Sigma \rangle \quad \text{[lam]}$$
$$\text{where } \{x_{free}, ...\} = \text{free-vars}[\![(\lambda\ (x)\ e)]\!], \rho_{live} = \text{skip-to}[\![\rho, \{x_{free}, ...\}, \Sigma]\!]$$

$$\langle \langle (e_{fun}\ e_{arg}), \rho \rangle, K, \Sigma \rangle \longrightarrow_{\text{E}} \langle \langle \text{flat } \langle (e_{fun}\ e_{arg}), \{x_{free}, ...\}, \rho_{live}, \text{mt} \rangle \rangle, \langle \text{ret } \langle e_{fun}, \rho \rangle \rangle :: K, \Sigma \rangle \quad \text{[push]}$$
$$\text{where } \{x_{free}, ...\} = \text{free-vars}[\![e_{arg}]\!], \rho_{live} = \text{skip-to}[\![\rho, \{x_{free}, ...\}, \Sigma]\!]$$

continue

$$\langle \langle \text{flat } \langle e, \{x, x_{rest}, ...\}, \rho, \rho_{flat} \rangle \rangle, K, \Sigma \rangle \longrightarrow_{\text{E}} \langle \langle \text{flat } \langle e, \{x_{rest}, ...\}, \rho_{next}, \rho' \rangle \rangle, K, \Sigma' \rangle \quad \text{[flat]}$$
$$\text{where } v = \text{lookup}[\![x, \rho, \Sigma]\!], \langle \rho', \Sigma' \rangle = \text{bind}[\![x, v, \rho_{flat}, \Sigma]\!], \rho_{next} = \text{skip-to}[\![\rho, \{x_{rest}, ...\}, \Sigma]\!]$$

$$\langle \langle \text{flat } \langle (\lambda\ (x)\ e), \emptyset, \text{mt}, \rho_{flat} \rangle \rangle, K, \Sigma \rangle \longrightarrow_{\text{E}} \langle \langle \text{obj } \sigma \rangle, K, \Sigma' \rangle \quad \text{[flat-lam]}$$
$$\text{where } \langle \sigma, \Sigma' \rangle = \text{close}[\![(\lambda\ (x)\ e), \rho_{flat}, \Sigma]\!]$$

$$\langle \langle \text{flat } \langle (e_{fun}\ e_{arg}), \emptyset, \text{mt}, \rho_{flat} \rangle \rangle, \langle \text{ret } \langle e_{fun}, \rho \rangle \rangle :: K, \Sigma \rangle \longrightarrow_{\text{E}} \langle \langle e_{fun}, \rho \rangle, \langle \text{arg } \langle e_{arg}, \rho_{flat} \rangle \rangle :: K, \Sigma \rangle \quad \text{[flat-push]}$$

$$\begin{array}{lll} \text{skip-to}[\![\text{mt}, \emptyset, \Sigma]\!] & = & \text{mt} \\ \text{skip-to}[\![\sigma, \{x_{live}, ...\}, \Sigma]\!] & = & \sigma \quad \text{where } \langle \text{env } \langle x, v, \rho_{next} \rangle \rangle = \Sigma(\sigma), x \in \{x_{live}, ...\} \\ \text{skip-to}[\![\sigma, \{x_{live}, ...\}, \Sigma]\!] & = & \text{skip-to}[\![\rho_{next}, \{x_{live}, ...\}, \Sigma]\!] \quad \text{where } \langle \text{env } \langle x, v, \rho_{next} \rangle \rangle = \Sigma(\sigma), x \notin \{x_{live}, ...\} \end{array}$$

Figure 6. Space safety by environment flattening, replaces first two rules of Figure 3 and adds three rules

collect

$$\langle \langle \text{roots } \langle t\ \langle \text{flat } \langle e, \{x, ...\}, \rho, \rho_{flat} \rangle \rangle \rangle :: K \rangle, K_{done}, \Sigma_{from}, A, \Sigma_{to}, S \rangle \longrightarrow_{\text{G}} \langle \langle \text{roots } K \rangle, K_{done} +\!\!+ [\langle t\ \langle \text{flat } \langle e, \{x, ...\}, \rho', \rho'_{flat} \rangle \rangle \rangle], \Sigma_{from}, A'', \Sigma_{to}, S'' \rangle \quad \text{[root-flat]}$$
$$\text{where } \langle \rho', A', S' \rangle = \text{retain-env}[\![\rho, A, S]\!], \langle \rho'_{flat}, A'', S'' \rangle = \text{retain-env}[\![\rho_{flat}, A', S']\!]$$

Figure 7. Rule to extend Figure 4 for flat

a linked list, so it can take time proportional to the number of bindings in the original ρ. Later, we will show a binary-tree representation that allows skip-to to work in logarithmic time.

In case the garbage collector is triggered during flat mode, the rule in Figure 7 adds support for processing a flat form as a root. At this point, the garbage collector does not try to take into account that the ρ in a flat form (as opposed to ρ_{flat}) may retain more bindings than will be needed. The idea that it *could* take the scheduled set of variables into account leads us to an alternative implementation of space safety in section 6.

The new evaluator rules, when combined with the augmented garbage-collection rules as in Figure 5, improve the space performance of some programs. The example from the introduction,

```
(let ([f (λ (f) (λ (y)
                  ((f f) (λ (z) z))))])
  ((f f) (λ (x) x)))
```

uses unbounded space with the evaluation rules of Figure 3, because the closure formed for (λ (z) z) will capture the argument y, which is bound to the closure formed the previous time around, and so on. The new rules in Figure 6 form the closure for (λ (z) z) with an empty environment, since (λ (z) z)

has no free variables, and so the example runs in bounded space.

The example at the end of section 4, however, is back to $O(N^2)$ space instead of the $O(N)$ space used by linked closures. With flattening, each time that R is called, a closure is formed for (λ (z) (x_1 ... x_N)), which has N free variables that are copied out of R's environment, and keeping N of those new closures uses $O(N^2)$ space. Space performance in this particular example could be improved by noting that the closure will capture all variables in the environment *except* for y, which was just added; therefore, using the rest of the environment as referenced by the current environment would be safe and avoid copying. Leaving out a small number of other x_is from (λ (z) (x_1 ... x_N)) could have asymptotically sufficient sharing by using a tree structure instead of a list structure for environments. Even that strategy will fail if the number of omitted x_is is around $N/2$—at least, if they are in an interleaved order in the tree relative to preserved x_is. To defeat any strategy that involves ordering carefully, D could be changed to have multiple possible results from R that each capture arbitrary subsets of the x_is.

The potential space regression of this safe-for-space implementation is mirrored by a time regression. Forming a closure over N free variables takes $O(N)$ time.[2] Although that time could be reduced in some cases by the same strategies that improve sharing, the general case is resistant to improvement in both space and time.

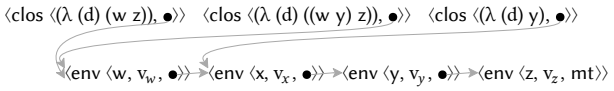
Overall, the problem with space safety through closure flattening is that a flattening rule does not have enough global context about what closures have been created or could still be created in the future, and so it cannot determine optimal sharing. Gathering the needed global context is the job of garbage collection, instead of evaluation.

6 Space Safety with Linked Environments via Garbage Collection

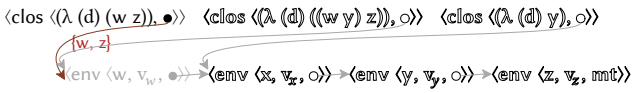
We can obtain space safety while preserving sharing of environments among closures by implementing a form of flattening in the garbage collector instead of the evaluator. Instead of flattening to create a fresh environment with only the free variables of an expression, the garbage collector can prune a shared environment by dropping bindings for variables that are not among the free variables of any expression paired with the environment. Operationally, the garbage collector can tentatively map a chain of environment frames to a new chain that omits bindings for so-far unreachable variables. It can later add back frames if they are discovered to be usable from some other closure.

[2]Similar to lookup, skip-to as written suggests time proportional to the size of the source environment, not the copied environment, since it has to traverse a linked list. We explain a tree alternative in section 7.

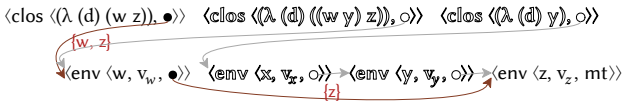
For example, suppose that a garbage collection starts with a store that contains a set of closures that share a set of environment frames:



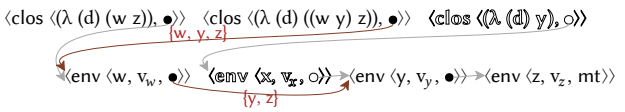
When the first closure is found reachable, then sweeping will follow the environment reference knowing that only w and z are used by the closure's expression:



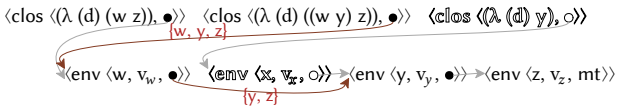
When sweeping the environment frame for w, since only z is left, the reference to the next frame is updated to skip over the variables that are unused, so far:



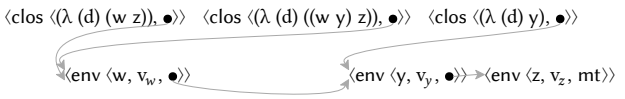
When the second closure is found reachable, sweeping will again consider the first environment frame, but this time with w, y, and z. Since y is relevant after all, the first environment's frame is adjusted again so that it only skips x:



When the third closure is found reachable, sweeping considers the third environment frame with just y. Although z is not in the relevant set for sweeping, the environment frame for z has already been found reachable, so it is kept:



After garbage collection completes, the unreached frame for x can be discarded:



Unlike flattened closures, the first closure with free variables w and z still has y in its environment—but that doesn't create a space leak, because y is needed for the second and third closures. Meanwhile, the first two closures usefully share the environment chain that has both w and z, instead of duplicating that combination.

For the garbage collector to take into account free variables when sweeping environments, the sweep queue S will need to keep a set of variables with each address to sweep. As shown at the top of Figure 8, we adjust S to be a queue of tuples, instead of a queue of plain addresses. Along similar lines, we must no longer skip sweeping of an environment merely because it has been reached before, but only when it

$$S ::= [] \mid \langle \sigma, \{x, \ldots\} \rangle :: S \qquad A ::= \{\sigma = \langle \sigma, L \rangle, \ldots\} \qquad L ::= \{\{x, \ldots\}, \ldots\}$$

$$\mathsf{mt} \vee \rho = \rho \qquad \sigma \vee \rho = \sigma$$

$$\begin{aligned}
&\text{kept-skip-to}[\![\mathsf{mt}, \emptyset, \Sigma, A]\!] = \langle \mathsf{mt}, \mathsf{mt}, \mathsf{mt} \rangle \\
&\text{kept-skip-to}[\![\sigma, \{x_{live}, \ldots\}, \Sigma, A]\!] = \langle \sigma, \mathsf{mt}, \sigma \rangle \\
&\quad \text{where } \langle \mathsf{env}\ \langle x, v, \rho_{next} \rangle \rangle = \Sigma(\sigma),\ x \in \{x_{live}, \ldots\} \\
&\text{kept-skip-to}[\![\sigma, \{x_{live}, \ldots\}, \Sigma, A]\!] = \text{kept-skip-to}[\![\rho_{next}, \{x_{live}, \ldots\}, \Sigma, A]\!] \\
&\quad \text{where } \sigma \notin \mathrm{dom}(A),\ \langle \mathsf{env}\ \langle x, v, \rho_{next} \rangle \rangle = \Sigma(\sigma),\ x \notin \{x_{live}, \ldots\} \\
&\text{kept-skip-to}[\![\sigma, \{x_{live}, \ldots\}, \Sigma, A]\!] = \langle \sigma, \rho_{prev} \vee \sigma, \rho_{live} \rangle \\
&\quad \text{where } \sigma \in \mathrm{dom}(A),\ \langle \mathsf{env}\ \langle x, v, \rho_{next} \rangle \rangle = \Sigma(\sigma),\ x \notin \{x_{live}, \ldots\},\ \langle \rho_{kept}, \rho_{prev}, \rho_{live} \rangle = \text{kept-skip-to}[\![\rho_{next}, \{x_{live}, \ldots\}, \Sigma, A]\!]
\end{aligned}$$

$$\begin{aligned}
&\text{retain-skip-env}[\![\mathsf{mt}, \emptyset, \Sigma_{from}, A, \Sigma_{to}, S]\!] = \langle \mathsf{mt}, \mathsf{mt}, A, \Sigma_{to}, S \rangle \\
&\text{retain-skip-env}[\![\sigma_{from}, \emptyset, \Sigma_{from}, A, \Sigma_{to}, S]\!] = \langle \rho_{kept\text{-}to}, \mathsf{mt}, A, \Sigma_{to}, S \rangle \\
&\quad \text{where } \langle \rho_{kept\text{-}from}, \rho_{prev}, \mathsf{mt} \rangle = \text{kept-skip-to}[\![\sigma_{from}, \emptyset, \Sigma_{from}, A]\!],\ \rho_{kept\text{-}to} = A(\rho_{kept\text{-}from}) \\
&\text{retain-skip-env}[\![\sigma, \{x, \ldots\}, \Sigma_{from}, A, \Sigma_{to}, S]\!] = \langle \sigma_{kept\text{-}to}, \sigma_{live\text{-}to}, A', \Sigma'_{to}, S' \rangle \\
&\quad \text{where } \langle \sigma_{kept\text{-}from}, \rho_{prev}, \sigma_{live\text{-}from} \rangle = \text{kept-skip-to}[\![\sigma, \{x, \ldots\}, \Sigma_{from}, A]\!],\ \langle \sigma_{live\text{-}to}, A', S' \rangle = \text{retain-for}[\![\sigma_{live\text{-}from}, \{x, \ldots\}, A, S]\!], \\
&\qquad \Sigma'_{to} = \text{refine}[\![\rho_{prev}, \sigma_{live\text{-}to}, A', \Sigma_{to}]\!],\ \sigma_{kept\text{-}to} = A'(\sigma_{kept\text{-}from})
\end{aligned}$$

$$\begin{aligned}
&\text{retain-for}[\![\sigma_{from}, \{x, \ldots\}, A, S]\!] = \langle \sigma_{to}, A, S \rangle \\
&\quad \text{where } \langle \sigma_{to}, L \rangle = A(\sigma_{from}),\ \{x, \ldots\} \in L \\
&\text{retain-for}[\![\sigma_{from}, \{x, \ldots\}, A, S]\!] = \langle \sigma_{to}, A + \{\sigma_{from} = \langle \sigma_{to}, L \cup \{\{x, \ldots\}\} \rangle\}, S +\!\!+ [\langle \sigma_{from}, \{x, \ldots\} \rangle] \rangle \\
&\quad \text{where } \langle \sigma_{to}, L \rangle = A(\sigma_{from}) \\
&\text{retain-for}[\![\sigma_{from}, \{x, \ldots\}, A, S]\!] = \langle \sigma_{to}, A + \{\sigma_{from} = \langle \sigma_{to}, \{\{x, \ldots\}\} \rangle\}, S +\!\!+ [\langle \sigma_{from}, \{x, \ldots\} \rangle] \rangle \\
&\quad \text{where } \sigma_{from} \notin \mathrm{dom}(A),\ \sigma_{to} = |A|
\end{aligned}$$

$$\begin{aligned}
&\text{refine}[\![\mathsf{mt}, \sigma_{to}, A, \Sigma]\!] = \Sigma \\
&\text{refine}[\![\sigma_{prev\text{-}from}, \sigma_{to}, A, \Sigma]\!] = \Sigma \quad \text{where } A(\sigma_{prev\text{-}from}) \notin \mathrm{dom}(\Sigma) \\
&\text{refine}[\![\sigma_{prev\text{-}from}, \sigma_{to}, A, \Sigma]\!] = \Sigma + \{\sigma_{prev\text{-}to} = \langle \mathsf{env}\ \langle x, v, \sigma_{to} \rangle \rangle\} \quad \text{where } \sigma_{prev\text{-}to} = A(\sigma_{prev\text{-}from}),\ \langle \mathsf{env}\ \langle x, v, \rho \rangle \rangle = \Sigma(\sigma_{prev\text{-}to})
\end{aligned}$$

$$\langle g, K, \Sigma_{from}, A, \Sigma_{to}, S \rangle \longrightarrow_{\mathrm{G}} \langle g', K', \Sigma_{from}, A', \Sigma'_{to}, S' \rangle$$

collect

$$\langle \langle \mathsf{roots}\ \langle t\ \langle e, \rho_{from} \rangle \rangle :: K_{root} \rangle, K, \Sigma_{from}, A, \Sigma_{to}, S \rangle \longrightarrow_{\mathrm{G}} \langle \langle \mathsf{roots}\ K_{root} \rangle, K +\!\!+ [\langle t\ \langle e, \rho_{to} \rangle \rangle], \Sigma_{from}, A', \Sigma'_{to}, S' \rangle \quad \text{[root-env]}$$
$$\text{where } \{x, \ldots\} = \text{free-vars}[\![e]\!],\ \langle \rho_{kept\text{-}to}, \rho_{to}, A', \Sigma'_{to}, S' \rangle = \text{retain-skip-env}[\![\rho_{from}, \{x, \ldots\}, \Sigma_{from}, A, \Sigma_{to}, S]\!]$$

$$\langle \langle \mathsf{roots}\ \langle t\ v \rangle :: K_{root} \rangle, K, \Sigma_{from}, A, \Sigma_{to}, S \rangle \longrightarrow_{\mathrm{G}} \langle \langle \mathsf{roots}\ K_{root} \rangle, K +\!\!+ [\langle t\ v \rangle], \Sigma_{from}, A', \Sigma_{to}, S' \rangle \quad \text{[root-val]}$$
$$\text{where } \langle v', A', S' \rangle = \text{retain-val}[\![v, A, S]\!]$$

$$\langle \langle \mathsf{roots}\ [] \rangle, K, \Sigma_{from}, A, \Sigma_{to}, S \rangle \longrightarrow_{\mathrm{G}} \langle \mathsf{sweep}, K, \Sigma_{from}, A, \Sigma_{to}, S \rangle \quad \text{[roots-done]}$$

$$\langle \mathsf{sweep}, K, \Sigma_{from}, A, \Sigma_{to}, \langle \sigma_{from}, \{x_{live}, \ldots\} \rangle :: S \rangle \longrightarrow_{\mathrm{G}} \langle \mathsf{sweep}, K, \Sigma_{from}, A', \Sigma''_{to}, S' \rangle \quad \text{[sweep-env]}$$
$$\begin{aligned}
&\text{where } \sigma_{to} = A(\sigma_{from}),\ \langle \mathsf{env}\ \langle x, v, \rho_{from} \rangle \rangle = \Sigma_{from}(\sigma_{from}),\ \langle v', A_v, S_v \rangle = \text{retain-val}[\![v, A, S]\!], \\
&\quad \{x_{next}, \ldots\} = \{x_{live}, \ldots\} \backslash \{x\}, \\
&\quad \langle \rho_{kept\text{-}to}, \rho_{to}, A', \Sigma'_{to}, S' \rangle = \text{retain-skip-env}[\![\rho_{from}, \{x_{next}, \ldots\}, \Sigma_{from}, A_v, \Sigma_{to}, S_v]\!], \\
&\quad \Sigma''_{to} = \Sigma'_{to} + \{\sigma_{to} = \langle \mathsf{env}\ \langle x, v', \rho_{kept\text{-}to} \rangle \rangle\}
\end{aligned}$$

$$\langle \mathsf{sweep}, K, \Sigma_{from}, A, \Sigma_{to}, \langle \sigma_{from}, \emptyset \rangle :: S \rangle \longrightarrow_{\mathrm{G}} \langle \mathsf{sweep}, K, \Sigma_{from}, A', \Sigma'_{to} + \{\sigma_{to} = \langle \mathsf{clos}\ \langle e, \rho_{to} \rangle \rangle\}, S' \rangle \quad \text{[sweep-clos]}$$
$$\begin{aligned}
&\text{where } \sigma_{to} = A(\sigma_{from}),\ \langle \mathsf{clos}\ \langle e, \rho_{from} \rangle \rangle = \Sigma_{from}(\sigma_{from}),\ \{x, \ldots\} = \text{free-vars}[\![e]\!], \\
&\quad \langle \rho_{kept\text{-}to}, \rho_{to}, A', \Sigma'_{to}, S' \rangle = \text{retain-skip-env}[\![\rho_{from}, \{x, \ldots\}, \Sigma_{from}, A, \Sigma_{to}, S]\!]
\end{aligned}$$

Figure 8. Space safety via garbage collection

has been reached with enough free variables. We therefore update A as at the top of Figure 8 so that it maps a σ_{from} to a tuple with σ_{to} and sets of variable sets. We use L to represent such a set of sets. As a convenience, we continue to sometimes match $A(\sigma_{from})$ to σ_{to} instead of a tuple, which is a shorthand for matching a tuple whose first component is σ_{to}.

It may seem natural to keep a single set of variables for each σ_{from} in A, because we do not need to resweep σ_{from} for a set of variables if it has already been reached with a larger set of variables. On this point, however, we run into a limitation of sets as a data structure. Checking for a subset would take $O(N)$ time for a candidate subset of size N, but we need an operation that is at worst $O(\log N)$ time. Fortunately, the

relevant sets will all be tails of free-variable sets, and the free-variable sets of a program can be gathered and enumerated as part of a compilation pass before the program runs. Checking for a set in a set of sets will therefore reduce to checking for a number in a set of numbers, which takes $O(log\ N)$ time with a binary-tree representation.

To handle the new variants of A and S, we need replacements for skip-to, retain-env, and retain. The new metafunctions are shown in Figure 8:

- kept-skip-to is like skip-to, but it returns three environments: ρ_{kept}, ρ_{prev}, and ρ_{live}. The ρ_{live} environment is the same one that skip-to returns. The ρ_{kept} result can be the same as ρ_{live}, or it can be an environment seen along the way that has already been determined as reachable, so it should be kept in the overall environment chain. The ρ_{prev} result is a store address when ρ_{kept} and ρ_{live} are different addresses, in which case it is an environment frame (either ρ_{kept} or a later frame) just before ρ_{live} that may need updating to refer to ρ_{live}; the refine helper performs that update.
- retain-skip-env replaces retain-env. Instead of keeping all frames in an environment like retain-env does, it uses kept-skip-to to determine the earliest (in the chain) environment frame to keep as well as the next frame relevant for a set of variables.
- retain-for is like retain, but used by retain-skip-env. It accepts a set of variables in addition to an address σ_{from}. Like retain, it adds to A and S when an address has not been allocated before, but retain-for also updates A and enqueues to S in the case that σ_{to} has been allocated but the set of variables was not previously encountered with σ_{from}.
- refine is used by retain-skip-env to update an already copied environment frame. It has no effect if ρ_{prev} as matched in retain-skip-env is mt or if it is an address that has not yet been copied (due to the order that the collector happens to visit allocated objects). If ρ_{prev} is copied later, that copy will incorporate ρ_{live} as kept in the first place.

The updated reduction rules in Figure 8 are essentially the same as the ones in Figure 4, but using free-vars and the new A, S, and metafunctions. When a closure is swept, the "kept" environment is ignored, and the closure references the first relevant environment frame directly; this is a minor optimization that has no consequence for space usage.

The combination of the original interpreter and this garbage collector is safe for space, because an environment frame is never retained unless the frame's variable appears in a closure or continuation frame that references the environment. The example from section 5 runs forever in constant space. At the same time, all environment linking is preserved, since a frame is allocated and added to A at most once. The example from section 4 runs in $O(N)$ space, not $O(N^2)$ space.

In the garbage-collection steps, A needs to store not just a forwarding address, but also a set of sets. The number of sets is bounded for a particular program, like the examples in section 4 and section 5, but it is not a priori bounded for all programs. In the worst case, the number of different sets associated with a σ_{from} in A is proportional to the original program's size, M. Along the same lines, a σ_{from} can appear multiple times in the queue S with different variable sets, so S could be a factor of M larger in the worst case. Thus the asymptotic complexity of space safety of this implementation, $O(S_{\text{sfs-linked}})$, is not dominated by $O(S_{\text{sfs}})$ or $O(U_{\text{evlis}})$; we can construct a program that is large and uncooperative enough so that $S_{\text{sfs-linked}}$ exceeds S_{sfs} or U_{evlis} by an arbitrarily large constant factor. We can only do that, however, by allowing the program itself to be arbitrarily large.

Time complexity similarly may depend on the program size. The number of steps taken by a garbage collection can be larger than for the a non-safe-for-space collector, because an environment can be swept multiple times. This longer time translates into a higher fuel that is given to evaluation mode. More fuel for evaluation implies a higher peak for memory use before a garbage collection is triggered. Thus, the safe-for-space collector not only needs extra memory during the collection phase to store A and S, but it can incur a similarly larger space complexity even when considering only the evaluator's allocation. These effects do not spiral out of control, however, because the extra time and space factor is based on live data and sharing at the point where a garbage collection starts, not based on on the total size of the store when a collection starts.

7 Environments as Trees

Environments in our model are allocated as linked lists. Searching a linked list for lookup or kept-skip-to would take time proportional to the length of the list—but for characterizing time complexity of evaluation, we have been assuming that metafunctions take effectively constant time.

Using a balanced-tree representation instead of a linked list can support $O(log\ N)$ access time. We count that as effectively constant, since the number of bindings N is bounded on realistic hardware. In particular, if each variable is replaced with an integer that represents its binding depth, then a simple binary-tree representation supports efficient access and update. Some sharing will be lost in a tree representation, either through rebalancing or just through reconstruction of a spine to add or update and entry, but for many functional tree implementations, most sharing is preserved for individual operations. On a large enough tree, the asymptotic space complexity of a balance tree can match that of linked lists.

Concretely, consider the representation

$$\begin{aligned} \mathit{tree} ::= &\ \text{empty} \\ &\mid v \\ &\mid \langle \text{branch } n, \mathit{tree}, \mathit{tree} \rangle \end{aligned}$$

where all values are placed in leaves, the n in a branch constructor indicates that the left subtree has 2^n values, and a subtree with no values is always represented by empty. This representation straightforwardly supports functional, single-element access, addition, update, and removal operations in $O(log\ N)$ time. The same representation can be used for environments, variable sets (where membership in the set is represented by an arbitrary value at a leaf), and sets of variable sets (assuming that each relevant variable set is mapped to a unique index). Furthermore, the shape of the tree for a set of variables will match the shape of a tree for environments, which simplifies the operation of finding environment entries relevant for a set of variables.

Iterating through environment entries as in [sweep-env], retain-for, or retain-skip-env is not as convenient in a model as linked lists. That iteration is convenient using recursion, and recursive calls will nest only up to $log\ N$ depth. We take advantage of that limited recursion in our C implementation of space safety via garbage collection for Zuo. When traversing an environment tree with a variable set, our implementation recurs instead of adding to an $\mathcal{S}$-like queue. That recursion implements a stack-based exploration of gray environment objects instead of a queue-based exploration, but either exploration works; we previously emphasized queue-like exploration only to accommodate a semispace collector that represents $\mathcal{S}$ by a pointer in Σ_{to}, and that strategy is easily combined with a specialized, limited-recursion strategy for exploring environment trees.

8 Implementation and Sanity Checks

We have implemented space safety via garbage collection for Zuo using a translation of section 6 with the tree representation of section 7.[3] The implementation was successful in the sense that garbage-collection and interpreter additions are only about 150 lines of C code, assuming a set of binary-tree functions. The implementation remains somewhat unsatisfying, however, because it required the addition of a 300-line compilation pass to convert variables to binding-depth integers and to gather free-variable sets; Zuo otherwise can interpret abstract syntax directly, the same as Reynolds's interpreter or a CEK machine. Also, the binary-tree data structure required an additional 200 lines of code, and supporting extra administrative allocation during a garbage collection required roughly 50 more lines. All together, these additions increased Zuo's implementation size by about 10%.

Executable versions of this paper's models in PLT Redex [13] are available with the Zuo implementation,[4] and this paper's figures are typeset from that executable model. The model also offers us the opportunity to measure the space and time complexity for various examples, because we can use Racket to drive Redex and then count the number of steps and the peak size of the store during the execution.

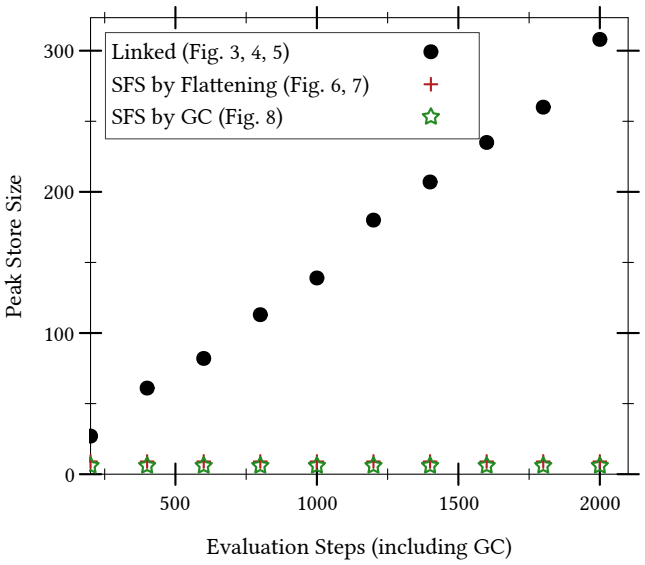


Figure 9. First example in introduction, peak memory use

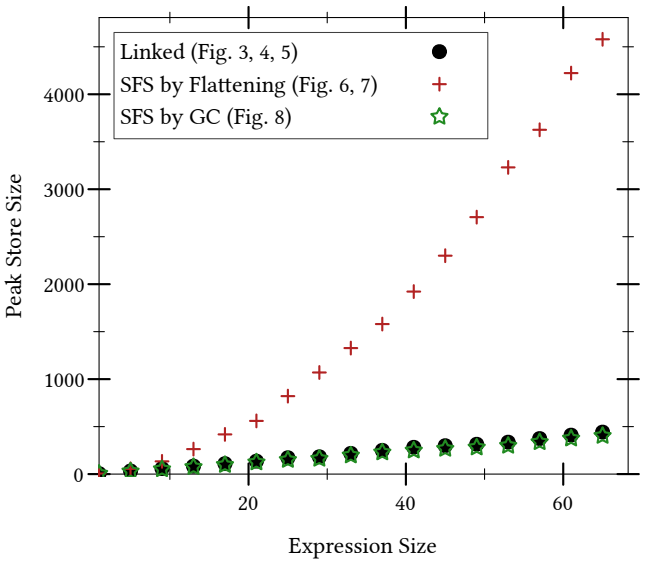


Figure 10. Second example in introduction, peak memory use

Figure 9 shows the peak memory consumption for the first example from the introduction for all three machines, stopping the machine after different numbers of steps, as shown along the x-axis. The y-axis shows the maximum heap size that the computation achieved at those points. The black dots are the original machine from Figure 3 and Figure 4. The red pluses are the safe-for-space machine that amends those figures with the rules in Figure 6 and Figure 7. The green stars

[3] Available as the `sfs` branch of https://github.com/mflatt/zuo.
[4] The `model` directory in the same repository and branch.

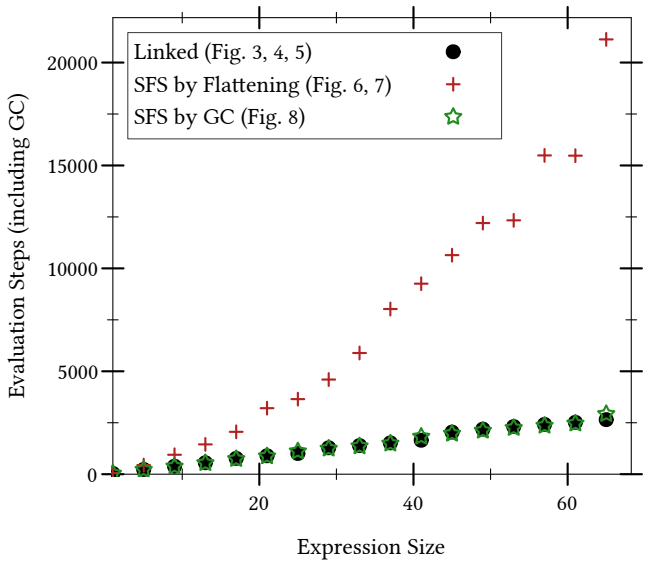


Figure 11. Second example in introduction, time measured as interpretation steps

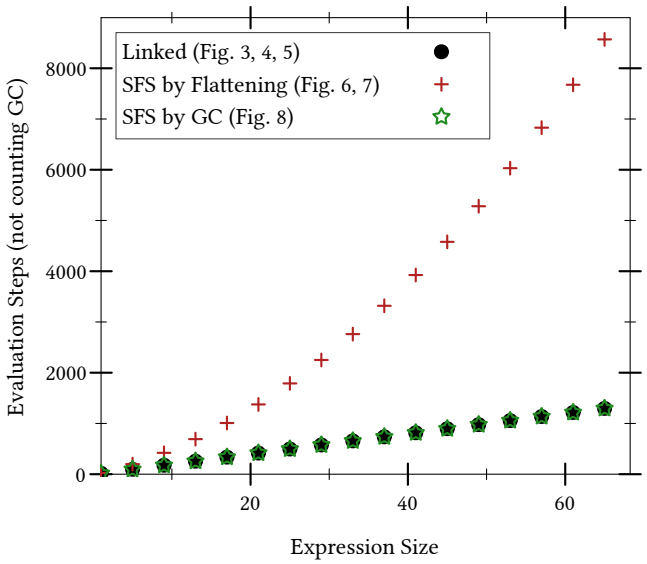


Figure 12. Second example in introduction, evaluation steps only, without including garbage collection

are the safe-for-space machine that uses the garbage collector to eliminate unused parts of the closures, from Figure 8. The original machine's space usage grows without bound, and both safe-for-space variants use a constant amount of space.

Figure 10 shows the peak memory use for the second example from the introduction as N grows. The original machine and the garbage collection-based safe-for-space machine use about the same amount of space, but the flattening-based safe-for-space machine takes asymptotically more space.

Figure 11 shows the number of steps needed to evaluate the second example from the introduction as N grows. These times are step counts that include both evaluation and garbage-collection steps. The original machine and the garbage collection-based safe-for-space machine take linear time. The flattening-based safe-for-space machine takes asymptotically more time due to its eager copying of environments; the curve is easier to discern in Figure 12, which shows only evaluation steps, omitting the jitter that is created by shifting garbage-collection triggers.

9 Future Work

Our models are intended to enable precise reasoning, but we have not yet attempted any formal proofs of correctness or asymptotic complexity, and we offer only sketches of arguments here. Formally completing the idea is a clear direction for future work.

This work was motivated by an implementation context where the garbage collector can be tightly integrated with the interpreter and specialized to the interpreter's representations of expressions and free-variable sets. The same approach could be useful to interpreters that are written in a high-level language, such as Chez Scheme or Racket, which already has a general-purpose garbage collector. In that case, the high-level language would need to provide a suitably general abstraction that is both supported by the garbage collector and available as a primitive datatype to programs; such a mechanism for "masked" reachability could be similar to the way that ephemerons [15] provide a general mechanism for "and" reachability. That interface might involve binary trees and a special pairing constructor, which can be used to combine a free-variable mask and an environment map that are each represented by binary trees.

Appendix

```
true = (λ (x) (λ (y) x))
false = (λ (x) (λ (y) y))
sel = (λ (q) (λ (x) (λ (y) (((q x) y) '0))))
pair = (λ (a) (λ (d) (λ (s) ((s a) d))))
empty = ((pair false) false)
cons = (λ (a) (λ (d) ((pair true) ((pair a) d))))
foldn = (λ (N)
          (λ (R)
            (λ (a)
              ((λ (f)
                 (((f f) a) N))
               (λ (f)
                 (λ (a)
                   (λ (n)
                     (((sel ('zero? n))
                       (λ (d) a))
                      (λ (d)
                        (((f f) (R a)) (('minus n) '1))))))))))))
```

References

[1] Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, and Jan Midtgaard. A Functional Correspondence Between Evaluators and Abstract Machines. In *Proc. Principles and Practice of Declarative Programming*, 2003. doi:10.1145/888251.888254

[2] Andrew W. Appel. Compiling with Continuations. Cambridge University Press, 1992. doi:10.1017/CBO9780511609619

[3] David F. Bacon, Perry Cheng, and V. T. Rajan. A Unified Theory of Garbage Collection. In *Proc. Object-Oriented Programming, Systems, Languages and Applications*, 2004. doi:10.1145/1028976.1028982

[4] Małgorzata Biernacka and Olivier Danvy. A Concrete Framework for Environment Machines. *Transactions on Computational Logic* 9(1), 2007. doi:10.1145/1297658.1297664

[5] Dariusz Biernacki and Olivier Danvy. From Interpreter to Logic Engine by Defunctionalization. In *Proc. Logic Based Program Synthesis and Transformation*, 2003. doi:10.1007/978-3-540-25938-1_13

[6] C. J. Cheney. A Nonrecursive List Compacting Algorithm. *Communications of the ACM* 12(11), 1970. doi:10.1145/362790.362798

[7] William D. Clinger. Proper Tail Recursion and Space Efficiency. In *Proc. Programming Language Design and Implementation*, 1998. doi:10.1145/277650.277719

[8] Olivier Danvy. Defunctionalized Interpreters for Programming Languages. In *Proc. International Conference on Functional Programming*, 2008. doi:10.1145/1411204.1411206

[9] Olivier Danvy and Kevin Millikin. On the Equivalence Between Small-Step and Big-Step Abstract Machines: a Simple Application of Lightweight Fusion. *Information Processing Letters* 106(3), 2008. doi:10.1016/j.ipl.2007.10.010

[10] Olivier Danvy and Lasse R. Nielsen. Defunctionalization at Work. In *Proc. Principles and Practice of Declarative Programming*, 2001. doi:10.1145/773184.773202

[11] Edsgar W. Dijkstra, Leslie Lamport, A. J. Martin, C.S. Scholten, and E. F. M. Steffens. On-the-Fly Garbage Collection: An Exercise in Cooperation. *Communications of the ACM* 21(11), 1978. doi:10.1145/359642.359655

[12] Matthias Felleisen. The Calculi of λ-v-CS Conversion: A Syntactic Theory of Control and State in Imperative Higher-Order Programming Languages. PhD dissertation, Indiana University, 1987.

[13] Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. Semantics Engineering with PLT Redex. MIT Press, 2009.

[14] Jeremy Gibbons. Continuation-Passing Style, Defunctionalization, Accumulations, and Associativity. *The Art, Science, and Engineering of Programming* 6(2), 2022. doi:10.22152/programming-journal.org/2022/6/7

[15] Barry Hayes. Ephemerons: a New Finalization Mechanism. *ACM SIGPLAN Notices* 32(10), 1997.

[16] Paul B. Jackson. Verifying a Garbage Collection Algorithm. In *Proc. International Conference on Theorem Proving in Higher Order Logics*, 1998.

[17] Richard Jones, Antony Hosking, and Eliot Moss. The Garbage Collection Handbook: The Art of Automatic Memory Management. Chapman and Hall/CRC, 2023.

[18] Greg Morrisett, Matthias Felleisen, and Robert Harper. Abstract Models of Memory Management. In *Proc. Functional Programming Languages and Computer Architecture*, 1995. doi:10.1145/224164.224182

[19] Magnus O. Myreen. Reusable Verification of a Copying Collector. In *Proc. International Conference on Verified Software: Theories, Tools, and Experiments*, 2010.

[20] Gordon D. Plotkin. Call-by-Name, Call-by-Value and the λ-Calculus. *Theoretical Computer Science* 1(2), 1975.

[21] John C. Reynolds. Definitional Interpreters for Higher-Order Programming Languages. In *Proc. ACM Annual Conference*, 1972. doi:10.1145/800194.805852

[22] Adam Sandberg Ericsson, Magnus O. Myreen, and Johannes Åman Pohjola. A Verified Generational Garbage Collector for CakeML. *Journal of Automated Reasoning* 63(2), 2019. doi:10.1007/s10817-018-9487-z

[23] Zhong Shao and Andrew W. Appel. Efficient and Safe-for-Space Closure Conversion. *Transactions on Programming Languages and Systems* 22(1), 2000. doi:10.1145/345099.345125

[24] Mitchell Wand. From Interpreter to Compiler: a Representational Derivation. In *Proc. Programs as Data Objects*, 1986. doi:10.1007/3-540-16446-4_17

Received 2025-06-09; accepted 2025-07-31