Garbage Collection using a Finite Liveness Domain

Aman Bansal*
IIT Bombay
India
amanb@cse.iitb.ac.in

Saksham Goel*
IIT Bombay
India
saksham.goel@cse.iitb.ac.in

Preey Shah*
IIT Bombay
India
preey@cse.iitb.ac.in

Amitabha Sanyal
IIT Bombay
India
as@cse.iitb.ac.in

Prasanna Kumar
IIT Bombay
India
prasannak@cse.iitb.ac.in

Abstract

Functional languages manage heap data through garbage collection. Since static analysis of heap data is difficult, garbage collectors conservatively approximate the *liveness* of heap objects by *reachability* i.e. every object that is reachable from the root set is considered live. Consequently, a large amount of memory that is reachable but not used further during execution is left uncollected by the collector.

Earlier attempts at liveness-based garbage collection for languages supporting structured types were based on analyses that considered arbitrary liveness values, i.e. they assumed that any substructure of the data could be potentially live. This made the analyses complex and unscalable. However, functional programs traverse structured data like lists in identifiable patterns. We identify a set of eight usage patterns that we use as liveness values. The liveness analysis that accompanies our garbage collector is based on this set; liveness arising out of other patterns of traversal are conservatively approximated by this set.

This restriction to a small set of liveness values reaps several benefits – it results in a simple liveness analysis which scales to much larger programs with minimal loss of precision, enables the use of a faster collection technique, and is extendable to higher-order programs.

Our experiments with a purely functional subset of Scheme show a reduction in the analysis time by orders of magnitude. In addition, the minimum heap size required to run programs is comparable with a liveness-based collector with unrestricted liveness values, and in situations where memory is limited, the garbage collection time is lower than its reachability counterpart.

CCS Concepts: • **Software and its engineering** → **Garbage collection**; *Functional languages*; • **Theory of computation** → **Program analysis**.

Keywords: Garbage Collection, Liveness Analysis, Access Paths, Functional Programming

ACM Reference Format:
Aman Bansal, Saksham Goel, Preey Shah, Amitabha Sanyal, and Prasanna Kumar. 2020. Garbage Collection using a Finite Liveness Domain. In *Proceedings of the 2020 ACM SIGPLAN International Symposium on Memory Management (ISMM '20), June 16, 2020, London, UK*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3381898.3397208

1 Introduction

Modern programming languages have been increasingly adopting automated memory management through garbage collection because of the safety and precision that it offers. Functional languages, where all memory is dynamically allocated on the heap and accessed through references from the stack, have always had garbage collectors as an essential part of their runtime support. Therefore, it is important to investigate possibilities of increasing the effectiveness of garbage collection, especially for use in functional programming languages. Most garbage collectors work by marking heap cells that are reachable from the activation stack and registers (called the *root set*) and collecting unmarked cells as garbage. However, studies across functional programming languages [3, 15, 22] have shown that programs often allocate considerable amount of memory that remains reachable beyond their last use, i.e. even after they cease to be live. Thus a liveness-based garbage collector (LGC) can be expected to collect more garbage in the same program state than a reachability based collector (RGC). Use of a LGC results in several benefits—programs run with lesser memory, there are fewer collections, and, in the case of copying collectors, which copy useful cells instead of garbage, the collections are, on the average faster, since fewer memory cells have

*Sorted lexicographically by last name

ISMM '20, June 16, 2020, London, UK

ACM ISBN 978-1-4503-7566-5/20/06…$15.00
https://doi.org/10.1145/3381898.3397208

to copied. Therefore, not only do programs run with lesser heap space, they often run in lesser time.

The primary reason why LGCs have not found acceptance in spite of their claimed and empirically demonstrated benefits [9] is the difficulty of liveness analysis itself[1]. When the analysis involves scalars only, liveness has two values—may-be-live or dead. When the language supports structured data, as it does in the case of functional programming languages with algebraic data types, any reasonably precise analysis has to go beyond this binary classification and identify parts of the structure that are live. Liveness of structured data is commonly represented by sets of paths in the heap [3, 11, 12, 17]. These paths represent parts of the data that can be potentially traversed in the future. However, manipulating arbitrary sets of paths in an analysis that also strives for precision is cumbersome, and the resulting methods do not scale to programs consisting of more than a few functions. The complexity of analysis also inhibits analysis of programs with higher order functions, a staple of most functional programs. Additionally, since the garbage collector has to refer to liveness of root set variables at runtime, the complexity of liveness values and their representation adds overheads to the garbage collection algorithm. This partially nullifies the apparent advantages of a liveness based collection. Finally, the storage of liveness data has its own overheads, as this information has to be stored for every potential garbage collection point.

It is our observation that most functional programs that use datatypes such as lists and trees access the heap in regular identifiable patterns. Therefore, in this paper, we restrict ourselves to a small fixed set of liveness values, each value representing a common pattern of heap accesses. All other values of liveness are safely approximated to a value from this fixed set. This simplifies liveness analysis and leads to dramatic improvements in the scalability of the method. In addition, it also allows us to handle higher-order functions. While the approximation introduced could have led to a reduction in the collected garbage, we largely make up for the loss in precision in two ways, both attributable to our choice of the small fixed set of liveness values.

Firstly, for the body of a function, we maintain separate liveness information for each distinct calling context, where the context includes the liveness value at the call-site. This is different from most analyses[2]. We exploit this added precision by maintaining the calling context at runtime, and using the liveness corresponding to the context in case of a garbage collection in the body of the function.

[1] [9] obtained their liveness information from program traces and not from a static analysis.

[2] While most analyses avoid infeasible interprocedural paths by maintaining separate information regarding call-sites in their calling context, the analysis information (the dataflow values) at the call-sites are generally not part of the calling context and are merged while analysing the body of a function. [20] is one of the few exceptions.

Secondly, our garbage collector has a marking scheme that leverages the fact that the liveness values can be ordered by inclusion. We use this fact in two ways: First we are able to avoid a drawback of most liveness based collectors, namely repeated revisits to the same memory location while marking. And secondly, we are able do a breadth first traversal during marking, thereby avoiding the stack-space cost of a depth first search.

A Motivating Example. As a motivating example, consider the program shown in Figure 1 written in a restricted subset of Scheme. The program defines two functions, **append** (lines 1-8) and **main** (lines 9-13). The function **append** concatenates its list arguments, and **main** creates a pair of lists using **makelist** (definition omitted) and uses **append** to concatenate the pair. The labels denoted by π that annotate some of the expressions refer to *program points* and are not part of the program. Semantically, a program point represents an instant of time just before the execution of the expression that it labels.

For concreteness, assume that the list `x` passed to **append** from **main** has four elements indicated by the triangles A, B, C and D in that order (Figure 2). The reference to this list from the activation frame AF_1 is called x_1; we differentiate between incarnations of the same variable in different function calls using subscripts. The triangles in the figure indicate structures whose details are not important. Rectangles indicate **cons** cells, the basic building blocks of lists with pointers to the head (**car**) and the tail (**cdr**) of the list. The crossed rectangle is an empty list.

The snapshot in the figure follows a sequence of events in which **main** calls **append**, and **append**, in turn, calls itself recursively four more times, of which the last three calls have returned. As a result, the last two elements of `x` are appended to the list `y` and the result bound to `t` (the t_2 incarnation in AF_2). Also assume that the currently active call to **append** has progressed further and invoked the **cons** operation at line 7. If this operation, which requires a fresh **cons** cell to be allocated, fails because the allocator has run out of memory, the garbage collector is invoked. At this time, the topmost activation frame, AF_2, on the stack contains references represented by the set of variables $\{x_2, y_2, d_2, t_2, a_2\}$, and the next frame, AF_1, contains the set $\{x_1, y_1, d_1\}$. In garbage collection parlance, these are called *root set variables*. Note that `c` is bound to a boolean value and not a reference.

Our liveness-based garbage collector begins a traversal from, say, the topmost frame AF_2. It consults the liveness of the root set variables at program point π_4, because this is where the program will resume after garbage collection. Our liveness analysis would have discovered the liveness of the root set variables: x_2, y_2 and d_2 as "not live" (denoted $\bot$) as they do not appear in the function body after π_4, and the liveness of t_2 and a_2 as "fully live" (denoted $\top$), which means that *all* **cons** cells reachable from these variables are likely to

```
1  (define (append x y)
2    (let c← (null? x) in                    ; if x is empty
3      (if c (return y)                      ; return y
4        (let d← (cdr x) in                  ; else bind d to tail of x
5          (let t ← (append d y) in          ; and t to (append d y)
6            π3: (let a← (car x)  in         ; bind a to head of x
7              π4: (let ans ← (cons a t)  in ; return a list
8                    (return ans))))))))     ; with head a, tail t
9  (define (main)
10   (let p← (makelist ...) in               ; make a list and bind to p
11     π1: (let q← (makelist ...) in         ; similarly for q
12       π2: (let final← (append p q) in
13             (return final)))))            ; return (append p q)
```

Figure 1. An example annotated program to illustrate our method.

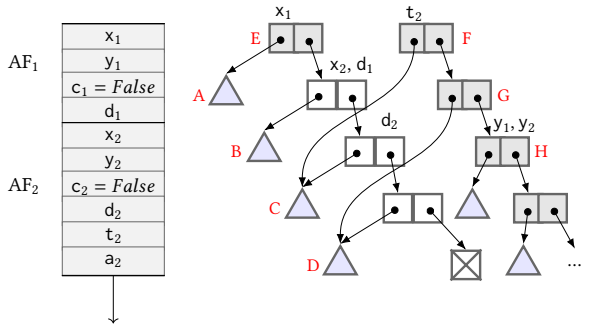


Figure 2. A snapshot of the stack and the heap during execution. Stack references and heap data are assumed to be connected through common variable names instead of arrows. Only the activation frames of **append** are shown.

be used in the future. This is understandable since both these variables and the structures reachable from them are part of the result of the program, and we assume that the agent that invoked this program would use the result in its entirety. Assuming that the underlying garbage collector is a copying collector (though the same idea can also be replayed on other collectors), it would scavenge all the cells in the structures B, C and D, the cons cells F, G and H, and the entire structure under H. Similarly, for garbage collecting the root set in AF_1, the liveness at program point π_3 is consulted. This is because π_3 is the program point where the call corresponding to AF_1 would resume after the return of the inner call to **append**. Because of the expression (**car** x) that occurs in the function after π_3, the liveness analysis will determine that the cell pointed by x_1 and the whole substructure under its **car** to be live. We denote this pattern of liveness ("root-and-**car**" live) as $\top_{0_\epsilon}$. The variables d_1 and y_1 on the other hand have $\bot$ as their liveness values. Traversal from x_1 results in the scavenging of E and all the cells in A. The heap data that will be scavenged in this round of garbage collection are indicated by shading, the remaining cells that are not shaded, while reachable, can be garbage collected and reused for subsequent allocation.

This example shows the benefits of liveness-based garbage collection. In fact, a reachability based collector would not have been able to reuse any of the **cons** cells of the original lists and would have required four extra cells to append the two lists. Carried to its limit, if we can tolerate a garbage collection for every **cons** cell allocated by the program, then the two lists, however long, can be appended entirely by reusing the cells of the original lists.

This example uses just three liveness values, $\bot$, $\top$ and $\top_{0_\epsilon}$. Our complete set consists of five more liveness values. As an example of yet another liveness value, consider a list passed to a function that calculates its length. Such a list is traversed without touching any of the elements. We denote such a liveness value ("spine-only live") as $\mathbf{1}^*$. The complete set of liveness values that we use is described in Section 2.

The main contributions of the paper are:

1. We propose a liveness-based garbage collection method for a restricted and pure subset of Scheme that allows higher order functions and has lists as the primary data structure.
2. The garbage collection relies on a liveness analysis that is simple and scalable. The scalability derives from our choice of a small fixed set of liveness values that embodies common patterns of heap traversal. Interestingly, experiments on our benchmarks demonstrate that the limited set of liveness values is able to capture heap liveness with a precision that is comparable to a previous method [3] that employs arbitrary liveness values.
3. The small set of liveness values allows us to separate out the liveness of function bodies based on contexts. The context information is carried over to runtime at negligible space costs. However, it enhances the precision of liveness analysis enabling more cells to be identified as garbage.
4. Our choice of liveness patterns also enables the design of a garbage collection mechanism that avoids revisits of memory cells. It also enables breadth-first traversal of the heap, saving the space-cost of a depth-first traversal.
5. We have implemented an interpreter and a copying collector based on our ideas. Our experiments confirm that compared to to an LGC based on arbitrary liveness patterns, our analysis is orders of magnitude faster, comparable in precision and in the number of garbage collections and on the average faster.

However, our study also has its limitations:

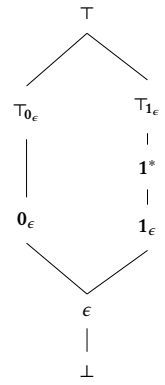
1. As mentioned earlier, our method in its current form is limited to lists, and because lists in Scheme are heterogeneous, also to binary trees. Lists and trees are widely used

$$
\begin{array}{rcll}
p \in Prog & ::= & d_1 \ldots d_n \;\; e_{\textbf{main}} & -- \textit{program} \\
df \in Fdef & ::= & (\textbf{define}\ (f\ G_1 \ldots G_m\ x_1 \ldots x_n)\ e_f) & -- \textit{function def} \\
e \in Expr & ::= & \begin{cases} (\textbf{if}\ x\ e_1\ e_2) & -- \textit{conditional} \\ (\textbf{let}\ x \leftarrow a\ \textbf{in}\ e) & -- \textit{let binding} \\ (\textbf{return}\ x) & -- \textit{return from function} \end{cases} \\
a \in App & ::= & \begin{cases} k & -- \textit{constant (numeric or } \textbf{nil}) \\ (\textbf{cons}\ x_1\ x_2) & -- \textit{constructor} \\ (\textbf{car}\ x) & -- \textit{selects first field of } \textbf{cons} \\ (\textbf{cdr}\ x) & -- \textit{selects second field of } \textbf{cons} \\ (\textbf{null?}\ x) & -- \textit{returns true if } x \textit{ is } \textbf{nil} \\ (+\ x_1\ x_2) & -- \textit{generic arithmetic} \\ (f\ g_1 \ldots g_m\ x_1 \ldots x_n) & -- \textit{function application} \end{cases}
\end{array}
$$

(a)

Demand	Set of access paths
$\bot$	$\emptyset$
ϵ	$\{\epsilon\}$
$\mathbf{0}_\epsilon$	$\{\epsilon, \mathbf{0}\}$
$\mathbf{1}_\epsilon$	$\{\epsilon, \mathbf{1}\}$
$\mathbf{1}^*$	$\{\epsilon, \mathbf{1}, \mathbf{11}, \ldots\}$
$\top_{\mathbf{0}_\epsilon}$	$\{\epsilon\} \cup \mathbf{0}(\mathbf{0}+\mathbf{1})^*$
$\top_{\mathbf{1}_\epsilon}$	$\{\epsilon\} \cup \mathbf{1}(\mathbf{0}+\mathbf{1})^*$
$\top$	$(\mathbf{0}+\mathbf{1})^*$

(b)



(c)

Figure 3. (a) The syntax of the language. (b) Demands and their meaning. (c) The lattice of demand patterns ordered by inclusion.

in functional programs. Other data types in our benchmarks were encoded in terms of lists.

2. While the chosen liveness values have been surprisingly effective for the programs considered, we do not know whether these are general enough to serve a wider range of benchmarks: (i) programs with possibly different heap traversal patterns, (ii) programs using different algebraic data types, and (iii) programs that are larger in size.

2 The Target Language

Figure 3(a) shows the target language. It is an eager, higher-order language with a syntax similar to Scheme. For ease of presentation, we restrict the language to Administrative Normal Form (ANF) [23]. In this form, function calls can only have variables as arguments. To avoid dealing with scope-shadowing, we assume that all variables in a program are distinct. Neither of these restrictions affects the expressibility of our language. As mentioned earlier, we sometimes use a label π to refer to the program point before an expression e, such as π:e. However the label is not part of the language.

A program in our language is a collection of function definitions followed by a call to a distinguished function called **main**. Applications (denoted by the syntactic category *App*) consist of functions or operators applied to their arguments. Constants are regarded as 0-ary functions. Expressions (*Expr*) are either an **if** expression, a **let** expression, or a **return** expression. A let expression (**let** $x \leftarrow a$ **in** e) creates a local binding for x and evaluates the expression e with this binding. The **return** keyword is used to mark the end of a function.

All functions are named, the language does not support lambda expressions. However, functions can be higher-order, i.e. they can take other functions as arguments. In the function definition (**define** $(f\ G_1 \ldots G_m\ x_1 \ldots x_n)\ e_f$), G_1 to G_m are function parameters, x_1 to x_n are non-function parameters, and e_f is the function body. Functions have to be fully applied, they cannot be applied to some but not all of their arguments. Partial application is allowed in many functional languages (e.g. Haskell), but not in Scheme. However, while describing liveness analysis, we shall refer to partial applications such as $(f\ g_1 \ldots g_m)$. This is a first order function that has been derived by applying a higher-order function to all and only its function arguments. Note that partial applications will be used only to describe the analysis. It is not permitted by the language and cannot occur in programs.

3 Liveness and its Analysis

A scalar variable is live if there is a possibility of its value being used in future computations and dead if it is definitely not used. In contrast, variables that are bound to heap-allocated structured data need a richer model than the may-be-live or dead model of classical liveness—a model which talks about future accesses of parts of a structure. In this paper, we restrict ourselves to lists. Borrowing notations from [3], we represent an access using the **car** selector by **0** and the **cdr** selector by **1**. An *access path* is any sequence of selectors including the empty sequence ϵ. Given a structure, a *demand* is a set of access paths that represent how the structure is possibly accessed. As an example, if x is a variable with value (**cons** 1 (**cons** 2 3)). then the accesses of the values in the substructure (**cons** 2 3) with respect to x is given by the demand $\{\epsilon, \mathbf{1}, \mathbf{10}, \mathbf{11}\}$. The access paths in this demand start with the root cons cell of x (represented by ϵ), the path to access the cons cell forming the root of (**cons** 2 3) (represented by **1**), and the paths to the leaves 2 and 3 (**10** and **11**). Notice that demands are prefix-closed; a node in a structure cannot be accessed from a root-set variable without going through its ancestors.

	σ							
$E(\sigma)$	$\bot$	ϵ	$\mathbf{0}_\epsilon$	$\mathbf{1}_\epsilon$	$\mathbf{1}^*$	$\top_{\mathbf{0}_\epsilon}$	$\top_{\mathbf{1}_\epsilon}$	$\top$
$\epsilon \cup \mathbf{0}\sigma$	ϵ	$\mathbf{0}_\epsilon$	$\top_{\mathbf{0}_\epsilon}$	$\top_{\mathbf{0}_\epsilon}$	$\top_{\mathbf{0}_\epsilon}$	$\top_{\mathbf{0}_\epsilon}$	$\top_{\mathbf{0}_\epsilon}$	$\top_{\mathbf{0}_\epsilon}$
$\epsilon \cup \mathbf{1}\sigma$	ϵ	$\mathbf{1}_\epsilon$	$\top_{\mathbf{1}_\epsilon}$	$\top_{\mathbf{1}_\epsilon}$	$\mathbf{1}^*$	$\top_{\mathbf{1}_\epsilon}$	$\top_{\mathbf{1}_\epsilon}$	$\top_{\mathbf{1}_\epsilon}$
$\bar{\mathbf{0}}\sigma$	$\bot$	$\bot$	ϵ	$\bot$	$\bot$	$\top$	$\bot$	$\top$
$\bar{\mathbf{1}}\sigma$	$\bot$	$\bot$	$\bot$	ϵ	$\mathbf{1}^*$	$\bot$	$\top$	$\top$

Figure 4. The approximation function $\mathcal{S}$.

While a demand can in general be an arbitrary set of access paths, recall that the claim of this paper is that restriction to a carefully chosen small set of demands makes liveness analysis more effective with only marginal loss of precision. Towards this we have chosen the following set of demands $\Sigma = \{\bot, \epsilon, \mathbf{0}_\epsilon, \mathbf{1}_\epsilon, \mathbf{1}^*, \top_{\mathbf{0}_\epsilon}, \top_{\mathbf{1}_\epsilon}, \top\}$. These correspond to the sets of access paths shown in Figure 3(b). These demands form a lattice that is ordered by set inclusion, and the join operation on this lattice is denoted by $\sqcup$. Any other demand that may arise during liveness analysis is approximated by a demand from this fixed set Σ. Notice that each of the sets is prefix closed. The element $\mathbf{0}_\epsilon$, for example, is a demand that denotes the set of access paths $\{\epsilon, \mathbf{0}\}$ and not simply $\{\mathbf{0}\}$.

A *liveness* of a variable at a program point is a demand that approximates all possible future accesses of the data bound to that variable. A variable x having a liveness of $\mathbf{0}_\epsilon$ at program point π means that future accesses to x starting from π are limited to the root of the structure of x and (the referent of) its **car** field. A *liveness environment* is a mapping from variables to demands, for example $\{x \mapsto \mathbf{0}_\epsilon, y \mapsto \epsilon, z \mapsto \bot\}$. The liveness of $z \mapsto \bot$ indicates that z is dead. Note that we use the empty sequence of selectors ϵ to stand for the root of a structure. Thus base values such as integers have ϵ as the only substructure.

Access paths are denoted by α and demands by σ. Given two access paths α_1 and α_2, we use $\alpha_1\alpha_2$ to denote their concatenation. We extend this notation to the concatenation of a symbol with a demand: $\mathbf{0}\sigma$ is a shorthand for $\{\mathbf{0}\alpha \mid \alpha \in \sigma\}$. However the set of access paths represented by $\mathbf{0}\sigma$ may not be exactly representable in Σ and may need to be approximated by the least element in the lattice of demands that contains it. This approximation is done by the function $\mathcal{S}$. Figure 4 defines $\mathcal{S}$ for all possible demand expressions $E(\sigma)$ that can be encountered during liveness analysis.

3.1 Liveness Analysis

Liveness analysis determines the liveness environment, i.e. the mapping of variables to demands, at each program point. The analysis is described in terms of the three functions $\mathcal{A}$, $\mathcal{L}$ and $\mathcal{FS}$ shown in figure 5. Given a demand σ representing the future uses of an application, the function $\mathcal{A}$ returns a liveness environment that describes the uses of the variables in the application. $\mathcal{L}$ plays a similar role for an expression—given a demand, it returns a liveness environment for the *free variables* of the expression. Both $\mathcal{A}$ and $\mathcal{L}$ are backward analyses—they transform a demand on the result of an application or an expression to demands on their arguments.

We start by explaining $\mathcal{A}$. Consider the application (**car** x) with a demand σ. A program accessing σ part of (**car** x) has to start with the root of x, select the **car**, and then access σ part of the resulting structure. Thus the set of paths that are accessed relative to x is $\epsilon \cup \mathbf{0}\sigma$. However, the demand $\epsilon \cup \mathbf{0}\sigma$ may not be representable in our domain of patterns and may have to be approximated by $\mathcal{S}$. This gives the liveness environment of (**car** x) as $\{x \mapsto \mathcal{S}(\epsilon \cup \mathbf{0}\sigma)\}$. The application (**null**? x) produces the rule $\{x \mapsto \epsilon\}$ because a list can be tested for nullity by just accessing its root cell. Similarly, a scalar is represented by a single (root) cell which contains its value. Finally, to evaluate $(+\ x\ y)$ for any demand, including $\bot$, only the roots of x and y have to be accessed. Thus the liveness environment of $(+\ x\ y)$ is $\{x \mapsto \epsilon, y \mapsto \epsilon\}$[3].

To explain the rule for **cons**, notice that a demand of $\mathbf{0}_\epsilon$ (or $\{\epsilon, \mathbf{0}\}$ in expanded form) on (**cons** $x\ y$) means that future uses of (**cons** $x\ y$) will only access the root and the cell obtained by a **car** selection. Since **car** of (**cons** $x\ y$) is x itself, it follows that the demand on x is ϵ. Generalising this argument, if the demand on (**cons** $x\ y$) is σ, then the demand on x is $\{\alpha \mid \mathbf{0}\alpha \in \sigma\}$. Denoting $\{\alpha \mid \mathbf{0}\alpha \in \sigma\}$ as $\bar{\mathbf{0}}\sigma$ and approximating this demand using $\mathcal{S}$, we get the liveness of x as $\mathcal{S}(\bar{\mathbf{0}}\sigma)$. A similar reasoning gives the demand on y as $\mathcal{S}(\bar{\mathbf{1}}\sigma)$, where the symbol $\bar{\mathbf{1}}$ is defined as $\bar{\mathbf{1}}\sigma = \{\alpha \mid \mathbf{1}\alpha \in \sigma\}$. Observe that for the same demand $\mathbf{0}_\epsilon$, the demand on y is $\mathcal{S}(\bar{\mathbf{1}}\mathbf{0}_\epsilon)$ or $\bot$, as it should be. The behaviour of $\mathcal{A}$ for a function call will be explained after we describe the function $\mathcal{L}$.

Similar to $\mathcal{A}$, the function $\mathcal{L}$ transforms a demand on an expression to a liveness environment involving the free variables in the expression. The rule for (**return** x) is obvious. The liveness environment of (**if** $x\ e_1\ e_2$) is the pointwise union of the liveness environment that arises out of testing the condition x, and the liveness environments of e_1 and e_2. The liveness environments of (**let** $x \leftarrow a$ **in** e) is the pointwise union of the liveness environments of the application a and the expression e. The liveness environment L_1 of e is determined with respect to the demand σ on the **let** expression, and the liveness environment of a is determined with respect to the demand on x in L_1. The liveness of x is erased from the union because the scope of x is local to e. This is similar to the backward propagation of liveness across an assignment $x = e$ in classical dataflow analysis of imperative languages.

The liveness environment arising from a function call $(f\ g_1 \ldots g_m\ x_1 \ldots\ x_n)$ is described using the function $\mathcal{FS}$.

[3]Interestingly, the situation would have been different for a lazy language, which does not evaluate an expression unless required. For the $\bot$ demand, the liveness environment for $(+\ x\ y)$ would have been $\{x \mapsto \bot, y \mapsto \bot\}$.

$$\boxed{\mathcal{A} :: (\text{App} \times \text{Demand}) \rightarrow \text{LivenessEnvironment}}$$

$$\begin{aligned}
\mathcal{A}(\kappa, \sigma) &= \bot, \text{ for all constants } \kappa \text{ including } \mathbf{nil} \\
\mathcal{A}((\mathbf{null?}\ x), \sigma) &= \{x \mapsto \epsilon\} \\
\mathcal{A}((+\ x\ y), \sigma) &= \{x \mapsto \epsilon, y \mapsto \epsilon\} \\
\mathcal{A}((\mathbf{car}\ x), \sigma) &= \{x \mapsto \mathcal{S}(\epsilon \cup \mathbf{0}\sigma)\} \\
\mathcal{A}((\mathbf{cdr}\ x), \sigma) &= \{x \mapsto \mathcal{S}(\epsilon \cup \mathbf{1}\sigma)\} \\
\mathcal{A}(\pi{:}(\mathbf{cons}\ x\ y), \sigma) &= \{x \mapsto \mathcal{S}(\bar{\mathbf{0}}\sigma), y \mapsto \mathcal{S}(\bar{\mathbf{1}}\sigma)\} \\
\mathcal{A}((f\ g_1 \ldots g_m\ x_1 \ldots\ x_n), \sigma) &= \bigcup_{i=1}^{n}\{x_i \mapsto \sigma_i\}, \text{where} \\
& \quad (\sigma_1, \ldots, \sigma_n) = \mathcal{FS}\ ((f\ g_1 \ldots g_m), \sigma)
\end{aligned}$$

$$\boxed{\mathcal{L} :: (\text{Exp} \times \text{Demand}) \rightarrow \text{LivenessEnvironment}}$$

$$\begin{aligned}
\mathcal{L}((\mathbf{return}\ x), \sigma) &= \{x \mapsto \sigma\} \\
\mathcal{L}((\mathbf{if}\ x\ e_1\ e_2), \sigma) &= \{x \mapsto \epsilon\} \boxplus \mathcal{L}(e_1, \sigma) \boxplus \mathcal{L}(e_2, \sigma) \\
\mathcal{L}((\mathbf{let}\ x\ \leftarrow\ a\ \mathbf{in}\ e), \sigma) &= \mathsf{L}, \text{where} \\
& \quad \mathsf{L}_1 = \mathcal{L}(e, \sigma), \\
& \quad \sigma' = \mathsf{L}_1(x), \\
& \quad \mathsf{L}_2 = \mathcal{A}(a, \sigma'), \text{ and} \\
& \quad \mathsf{L} = (\mathsf{L}_1 \boxplus \mathsf{L}_2) - \{x \mapsto \mathsf{L}_1(x)\}
\end{aligned}$$

$$\boxed{\mathcal{FS} :: (\text{FirstOrderFunc} \times \text{Demand}) \rightarrow (\text{Demand}_1 \times \ldots \times \text{Demand}_n))}$$

$$\begin{aligned}
\mathcal{FS}\ ((f\ g_1\ \ldots\ g_m), \sigma) &= (\mathsf{L}(x_1), \ldots, \mathsf{L}(x_n)), \text{where} \\
& \quad f \stackrel{\text{def}}{=} (\mathbf{define}\ (f\ G_1\ \ldots\ G_m\ x_1\ \ldots\ x_n)\ \ e_f), \text{and} \\
& \quad \mathsf{L} = \mathcal{L}(e_f[G_1/g_1, \ldots, G_m/g_m], \sigma)
\end{aligned}$$

Figure 5. Liveness Analysis

$\mathcal{FS}$ takes a first-order function $(f\ g_1 \ldots g_m)$[4] and a demand σ, and returns a tuple of demands $(\sigma_1, \ldots, \sigma_n)$. The function $\mathcal{A}$ maps this tuple to the variables $x_1 \ldots\ x_n$ to yield the liveness environment of the application $(f\ g_1 \ldots g_m\ x_1 \ldots\ x_n)$. $(\sigma_1, \ldots, \sigma_n)$ is computed by applying $\mathcal{L}$ to the body e_f of f after substituting the formal parameters $G_1 \ldots G_n$ by concrete functions $g_1 \ldots g_m$. $\mathcal{FS}$ yields an equation for each function f and each tuple of functions $(g_1 \ldots g_m)$ that f may be called with in the program. We have used the control flow analysis from a well-known method for *firstification* [19] to determine all possible concrete functions that a higher order function may be called with.

The equations derived from $\mathcal{FS}$ are, in general, mutually recursive, and are solved by computing the fixed point of a series of approximations. We denote the n^{th} approximation of $\mathcal{FS}$ applied to $(f\ g_1 \ldots g_m)$ as $\mathcal{FS}^n(f\ g_1 \ldots g_m)$. $\mathcal{FS}^0(f\ g_1 \ldots g_m)$ is the function $\lambda\sigma.(\bot, \ldots, \bot)$, the function that maps any demand to a tuple of empty demands. $\mathcal{FS}^n(f\ g_1 \ldots g_m)$ is computed from $\mathcal{FS}^{n-1}(f'\ g'_1 \ldots g'_m)$, where $(f'\ g'_1 \ldots g'_m\ y_1 \ldots\ y_k)$ is a function call encountered during the computation of $\mathcal{L}(e_f[G_1/g_1, \ldots, Gm/g_m])$. In practice we use these approximation for computing $\mathcal{FS}^n(f\ g_1 \ldots g_m)$ for each possible demand σ in Σ and cache the results to be used for the computation of liveness for the body of functions.

[4] As mentioned earlier, the first order function is, in general, obtained by partially applying a higher order function to its function parameters.

The compute intensive part of the analysis is the iterative calculation of $\mathcal{FS}$. We use a work-list based algorithm to find the fixed point for $\mathcal{FS}$. The work-list is initialised with all function definitions. For any function, say f, if $\mathcal{FS}^n \neq \mathcal{FS}^{n-1}$, all functions which call f are pushed onto the work-list.

We calculate a bound on the number of worklist iterations with the following argument. For a particular σ , $\mathcal{FS}^n \neq \mathcal{FS}^{n-1}$ if there exists an argument x_i such that $\mathsf{L}^n(x_i) \neq \mathsf{L}^{n-1}(x_i)$, where L is as defined within the equation for $\mathcal{FS}$ in Figure 5. Note that for this iterative procedure, L is a monotonically increasing function and hence the number of updates is bounded by the height of the lattice h. Extending this argument to all functions, it follows that $\mathcal{FS}$ can update atmost $kN_f \times h$ times per demand, where k is the average number of arguments in a function, before reaching the fixed point. The number of functions which call any function is loosely bounded by N_f, which proves that the worklist can have atmost $kN_f \times h \times N_f$ *pushes* (and consequently, *pops*) per demand, or $kN_f^2 \times h \times l$ in total where l is $|\Sigma|$, the number of lattice points. This is far superior in comparison to the $O(N_f.2^{N_f})$ complexity of the liveness-based method with arbitrary precision in [3]. The 2^{N_f} factor arises from a NFA to DFA conversion that is a part of the method. We substantiate this result with experiments in section 5.

3.2 Liveness Propagation

While the functions $\mathcal{A}$ and $\mathcal{L}$ describe the liveness of an application or an expression with respect to an arbitrary symbolic demand σ, for garbage collection we need to compute the concrete liveness environment at program points. In particular, we have to compute the liveness environment only at certain distinguished program points, called *gc-points*, where the garbage collector can be invoked. The liveness environments at these points are cached for use during garbage collection. We shall, for the moment, assume that a collection of program points called gc-points has already been identified, and define these later.

We now describe how to calculate the liveness environments at gc-points.

1. We begin by calculating $\mathcal{L}(e_{\textbf{main}}, \top)$, i.e. the liveness environment of $e_{\textbf{main}}$, the body of **main**, for the demand $\top$. The demand on $e_{\textbf{main}}$ is $\top$, because we assume that the entire result returned by the program would be required. While doing this we propagate liveness to the entire program in the following manner:
   a. Each time we compute $\mathcal{L}((f\ g_1 \ldots g_m\ x_1 \ldots x_n), \sigma)$, we also calculate the liveness environment of $(e_f[G_1/g_1, \ldots, Gm/g_m])$ for the same demand σ. The pair $((g_1, \ldots g_m), \sigma)$ is called the *calling context* for the call $((f\ g_1 \ldots g_m\ x_1 \ldots x_n)$.
   b. Consider a gc-point π in the body of f. While computing $\mathcal{L}(e_f[G_1/g_1, \ldots, Gm/g_m], \sigma)$, we merge the liveness environments at π arising out of different calls only if their calling contexts are the same. Notice that liveness at a program point for different calls to f are not merged unconditionally as is done in most interprocedural analyses. One of the key aspects of our method is that we track the calling context at runtime. As a consequence, our garbage collector is guided by a more precise analysis.
   c. The liveness environment at gc-points is recorded in a table called **gctable** that is consulted during garbage collection. **gctable**$[\pi, f, (g_1, \ldots, g_m), \sigma]$ contains the merged liveness environment at π, for all calls to f with the context $((g_1, \ldots g_m), \sigma)$.

An Example. Consider the same **append** function now expressed as an instance of a higher order function **foldr** (also called **reduce** in some languages). The function **append** is now defined by instantiating **foldr** to appropriate arguments. As before, **main** constructs a pair of lists p and q, and appends them. There are six garbage collection points π_1-π_6. Of these, we would like to know the liveness environments at π_2, π_3 and π_4. We first compute $\mathcal{FS}$ for each first order function (either declared or made first order by applying a higher order function to its function parameters). For the gc-points of interest, these are **append** and **foldr**. In fact, since **append** calls **foldr** with a swap of its non-function arguments, it is enough to analyse **foldr** and swap the elements of the resulting tuple.

```
1  (define (foldr G id l)
2    (let c← (null? l) in
3      (if c (return id)
4            (let d← (cdr l) in
5              (let t← (foldr G id d) in
6                π3: (let a← (car l)  in
7                      π4: (let ans← (G a t)  in
8                            (return ans))))))))
9  (define (append x y)
10   (let res ← (foldr cons y x) in
11     π5: (return res)))
12 (define (main)
13   (let p← (makelist …) in
14     π1: (let q← (makelist …) in
15       π2: (let final ← (append p q) in
16             π6: (return final)))))
```

Figure 6. Higher order version of **append**. It uses the higher order function **foldr**

A control flow analysis reveals that the only function that can be passed for G in **foldr** is **cons**. Thus one has to solve for $\mathcal{FS}$ $((\textbf{foldr cons}), \sigma)$ for all possible values of σ[5]. In the exposition below, if any entity E has a tuple as a value, we use the notation $\langle E \rangle_i$ to denote the i^{th} element of the tuple. The $\mathcal{FS}$ rule directly yields the following equation for $\mathcal{FS}$ $((\textbf{foldr cons}), \sigma)$:

$$\begin{aligned} \mathcal{FS}\ ((\textbf{foldr cons}), \sigma) &= (\mathtt{L}(\mathtt{id}), \mathtt{L}(\mathtt{l})), \\ \text{where } \mathtt{L} &= \mathcal{L}(e_{\textbf{foldr}}[\mathrm{G}/\textbf{cons}], \sigma) \end{aligned}$$

This, in turn, expands into the following:

$$\begin{aligned} \mathcal{FS}\ ((\textbf{foldr cons}), \sigma) &= (\sigma_{\mathtt{id}}, \sigma_{\mathtt{l}}), \text{where} \\ \sigma_{\mathtt{id}} &= \sigma \sqcup \langle \mathcal{FS}\ ((\textbf{foldr cons}), \mathcal{S}(\bar{\mathbf{1}}\sigma)) \rangle_1 \\ \sigma_{\mathtt{l}} &= \epsilon \sqcup \\ &\quad \mathcal{S}(\epsilon \cup \mathbf{1}\langle \mathcal{FS}\ ((\textbf{foldr cons}), \mathcal{S}(\bar{\mathbf{1}}\sigma)) \rangle_2) \sqcup \\ &\quad \mathcal{S}(\epsilon \cup \mathbf{0}\mathcal{S}(\bar{\mathbf{0}}\sigma)) \end{aligned}$$

Let us explain the equation for $\sigma_{\mathtt{l}}$. The demands on the three lines joined by $\sqcup$ correspond to the livenesses of l occurring in lines 2, 4 and 6. The liveness ϵ is directly from the **null**? rule. The liveness of ans is σ because of the **return** rule, and this is transformed to a liveness of $\mathcal{S}(\bar{\mathbf{0}}\sigma)$ for a and $\mathcal{S}(\bar{\mathbf{1}}\sigma)$ for t by the **let** and the **cons** rules. The liveness of a now becomes the demand on (**car** l) at line 6, and this gives the liveness of this occurrence of l as $\mathcal{S}(\epsilon \cup \mathbf{0}\mathcal{S}(\bar{\mathbf{0}}\sigma))$. The liveness of t, on the other hand, is propagated to the second (non-function) argument of **foldr**, namely d, to give a liveness of $\langle \mathcal{FS}\ ((\textbf{foldr cons}), \mathcal{S}(\bar{\mathbf{1}}\sigma)) \rangle_2$ for d. This is further propagated through (**cdr** l) to give the liveness of l at 3 as $\mathcal{S}(\epsilon \cup \mathbf{1}\langle \mathcal{FS}\ ((\textbf{foldr cons}), \mathcal{S}(\bar{\mathbf{1}}\sigma)) \rangle_2)$.

[5] We shall shortly see that solving for $\mathcal{FS}$ $((\textbf{foldr cons}), \top)$ is enough.

Let us calculate the value of $\mathcal{FS}$ ((**foldr cons**), σ), when $\sigma = \top$. We have

$$\begin{aligned}
\mathcal{FS}^0\,((\textbf{foldr cons}), \top) &= (\bot, \bot), \text{and} \\
\mathcal{FS}^1\,((\textbf{foldr cons}), \top) &= (\sigma^1_{\mathtt{id}}, \sigma^1_1), \text{where} \\
\sigma^1_{\mathtt{id}} &= \top \sqcup \langle \mathcal{FS}^0\,((\textbf{foldr cons}), \mathcal{S}(\bar{\mathbf{1}}\top))\rangle_1 \\
&= \top \sqcup \langle \mathcal{FS}^0\,((\textbf{foldr cons}), \top)\rangle_1 \\
&= \top \sqcup \bot = \top, \textit{and} \\
\sigma^1_1 &= \epsilon \sqcup \mathcal{S}(\epsilon \cup \mathbf{1}\langle \mathcal{FS}^0\,((\textbf{foldr cons}), \mathcal{S}(\bar{\mathbf{1}}\top))\rangle_2) \\
&\quad \sqcup \mathcal{S}(\epsilon \cup \mathbf{0}\mathcal{S}(\bar{\mathbf{0}}\top)) \\
&= \epsilon \sqcup \mathcal{S}(\epsilon \cup \mathbf{1}\langle \mathcal{FS}^0\,((\textbf{foldr cons}), \top)\rangle_2) \\
&\quad \sqcup \mathcal{S}(\epsilon \cup \mathbf{0}\top) \\
&= \epsilon \sqcup \mathcal{S}(\epsilon \cup \mathbf{1}\bot)) \sqcup \top_{\mathbf{0}_\epsilon} \\
&= \epsilon \sqcup \mathcal{S}(\epsilon \cup \bot) \sqcup \top_{\mathbf{0}_\epsilon} = \epsilon \sqcup \epsilon \sqcup \top_{\mathbf{0}_\epsilon} = \top_{\mathbf{0}_\epsilon}
\end{aligned}$$

It is clear that the value of $\sigma_{\mathtt{id}}$ converges to $\top$. However, to find the convergence point of σ_1, we iterate once more.

$$\begin{aligned}
\mathcal{FS}^2\,((\textbf{foldr cons}), \top) &= (\top, \sigma^2_1), \text{where} \\
\sigma^1_2 &= \epsilon \sqcup \mathcal{S}(\epsilon \cup \mathbf{1}\langle \mathcal{FS}^1\,((\textbf{foldr cons}), \mathcal{S}(\bar{\mathbf{1}}\top))\rangle_2) \\
&\quad \sqcup \mathcal{S}(\epsilon \cup \mathbf{0}\mathcal{S}(\bar{\mathbf{0}}\top)) \\
\sigma^1_2 &= \epsilon \sqcup \mathcal{S}(\epsilon \cup \mathbf{1}\langle \mathcal{FS}^1\,((\textbf{foldr cons}), \top)\rangle_2) \sqcup \top_{\mathbf{0}_\epsilon} \\
\sigma^1_2 &= \epsilon \sqcup \mathcal{S}(\epsilon \cup \mathbf{1}\top_{\mathbf{0}_\epsilon}) \sqcup \top_{\mathbf{0}_\epsilon} = \epsilon \sqcup \top_{\mathbf{1}_\epsilon} \sqcup \top_{\mathbf{0}_\epsilon} = \top
\end{aligned}$$

Since σ_1 converges to $\top$, $\mathcal{FS}^2$ ((**foldr cons**), $\top$) = ($\top$, $\top$). Armed with this information, we calculate the liveness environments at the three gc-points π_2, π_3 and π_4 as follows:

1. The liveness at π_2 is $\{\mathtt{p} \mapsto \mathcal{FS}^1\,((\textbf{append}), \top), \mathtt{q} \mapsto \mathcal{FS}^2\,((\textbf{append}), \top)\}$. This is the same as $\{\mathtt{p} \mapsto \mathcal{FS}^2\,((\textbf{foldr cons}), \top), \mathtt{q} \mapsto \mathcal{FS}^1\,((\textbf{foldr cons}), \top)\}$ and further simplifies to $\{\mathtt{p} \mapsto \top, \mathtt{q} \mapsto \top\}$.
2. The liveness at π_3 is $\{\mathtt{a} \mapsto \mathcal{S}(\bar{\mathbf{0}}\top), \mathtt{t} \mapsto \mathcal{S}(\bar{\mathbf{1}}\top), \mathtt{l} \mapsto \mathcal{S}(\epsilon \cup \mathcal{S}(\bar{\mathbf{0}}\top))$. This simplifies to $\{a \mapsto \top, \mathtt{t} \mapsto \top, \mathtt{l} \mapsto \top_{\mathbf{1}_\epsilon}\}$.
3. The liveness at π_4 is $\{\mathtt{a} \mapsto \bot, \mathtt{t} \mapsto \bot, \mathtt{l} \mapsto \bot, \mathtt{ans} \mapsto \top\}$.

The liveness of l at π_3 and π_4 are the same as in our informal description in Section 1.

GC-Points. Before proceeding further we explain what gc-points are. As mentioned earlier, these are program points where a program can potentially resume after pausing for garbage collection.

1. First consider the template π_1: (**let** $a \leftarrow$ (**cons** x y) **in** e). Clearly the garbage collection happens because of the failed allocation of a cons cell. After garbage collection is over, the execution resumes at π_1 to complete the evaluation of the **let** expression. Thus a program point before a **let** expression that binds a variable to a **cons** cell is a gc-point.
2. Next consider the template (**let** $a \leftarrow (f\,x_1 \ldots x_n)$ **in** π_2: e). A garbage collection could potentially take place during the call to f. After garbage collection, the execution will resume in the current scope at program point π_2. Thus a program point just after a function call is a gc-point.

A tricky case is the function call π_1: (**let** $a \leftarrow (G\,x_1\,x_2)$ *in* π_2 : e), where G is a parameter that may be bound to **cons**. This is a situation where the idea of a separate liveness analysis for each calling context comes to use. If the calling context binds G to **cons**, then the gc-point is identified as π_1, else it is π_2. Thus gc-point is not a purely syntactic notion but could also depend on the calling context.

4 The Garbage Collector

We can think of a liveness based traversal as a generalization of reachability. Given a node n in the heap and a demand σ associated with it, we define a node n' to be *σ-reachable* from n, if n' can be reached from n by following an access path in σ. Thus an RGC traverses all cells that are $\top$-reachable from the referents of the root set.

Our experimental setup includes an interpreter for the language shown in Figure 3 along with a single-generational copying collector. Assume that the program has been paused for garbage collection. Consider an activation frame in the stack and assume that it corresponds to the function call $(f\ g_1 \ldots g_m\ x_1 \ldots x_n)$. The activation frame will contain, along with other information, references corresponding to the root set variables $x_1 \ldots x_n$ and the function names $g_1 \ldots g_m$. As additional information, we also maintain the name of the function f itself, the gc-point π inside the function body where the program is paused and also the demand σ on the function call during analysis. Thus the activation frame has all information for the garbage collector to consult **gctable**$[\pi, f, (g_1, \ldots, g_m), \sigma]$ and read the liveness of the root set variables at π. This information is used by the garbage collector for traversal during copying.

LGCs [3, 21] with arbitrary demand patterns suffer from a drawback over RGCs during runtime. On visiting a memory cell n that has already been marked (i.e. visited before), an RGC does not initiate further traversals from n. This is safe because all cells reachable from n would have already been visited earlier. This property does not hold for LGC. An earlier visit to n with a demand, say σ, would have resulted in visits to only σ-reachable cells from n. Since there is no guarantee that the next visit to n will be accompanied by a demand that is a subset of σ, LGC methods with arbitrary patterns may need to initiate further traversals from an already visited cell.

With a set of finite demand patterns and an ordering on them, we are able to address this issue. The garbage collector creates a sequence of bins for the demand patterns $\top$, $\mathbf{1}^*$, $\mathbf{1}_\epsilon$, $\mathbf{0}_\epsilon$ and ϵ, and populates them by variables references in the root set. The order of the patterns in the sequence is important. The traversal for garbage collection now takes place by picking references from the bins in the sequence above. For example, while traversing a reference from the bin $\mathbf{1}^*$, if we visit a marked cell n, we know that this cell had been visited earlier with either a $\mathbf{1}^*$ pattern itself or a $\top$ pattern. Since the $\mathbf{1}^*$-reachable cells from n would have already been visited earlier, further traversals from n are not necessary. The situation is slightly different however, if we

reach a marked cell with a $\mathbf{0}_\epsilon$ pattern. The marking could have been due to an earlier arrival with a $\mathbf{1}_\epsilon$ or $\mathbf{1}^*$ pattern, and neither of these two patterns subsume $\mathbf{0}_\epsilon$ (i.e. $\mathbf{0}_\epsilon \not\sqsubseteq \mathbf{1}_\epsilon$ and $\mathbf{0}_\epsilon \not\sqsubseteq \mathbf{1}^*$). Therefore, we have to traverse a single step from n, but this step may be redundant if n had been visited earlier with $\mathbf{0}_\epsilon$ or $\top$.

For obvious reasons, the demand $\bot$ does not have a bin. Also notice the absence of bins for $\top_{\mathbf{0}_\epsilon}$ and $\top_{\mathbf{1}_\epsilon}$. If a root set variable, say z, has the liveness $\top_{\mathbf{0}_\epsilon}$, then, while initializing bins, the reference corresponding to z is inserted in the bin for $\mathbf{0}_\epsilon$ and the reference corresponding to (**car** z) is inserted in the bin for $\top$. Note that the traversal with very few revisits was made possible because of a total order amongst the bins corresponding to the infinite demands $\mathbf{1}^*$ and $\top$.

A second drawback of LGC addressed in our method is that we do a breadth-first traversal of the heap, thus avoiding the space-costs that come from using a stack for depth-first traversal. Earlier methods that had arbitrary sets of access paths as demands [3, 21] used an automaton for representing liveness, and collection involved a joint depth-first traversal of the heap and the automaton. Our collector uses the property that if the liveness σ of a root set variable is one of the infinite demands $\top$ or $\mathbf{1}^*$, then every σ-reachable cell from this root set variable also has the same demand. So we can copy these in breadth-first order without having to expend space to explicitly keep track of their liveness.

The Garbage Collection Algorithm. Algorithm 1 shows the garbage collection algorithm. As in any other single generation copying collector, the available memory is divided into two semispaces, their starting locations denoted by the variables fromSpace and toSpace. A round of garbage collection starts with a swap of roles of these variables, so that the occupied memory starts with fromSpace, from where live cells are copied into toSpace.

The garbage collector works as follows. The helper function copy(r) copies the content of r to toSpace if it is not already copied, and, in any case, returns a pointer to the copy. The collector function gc picks up root sets in order of bins. If the current bin corresponds to a demand σ, then all cells that are σ-reachable from r are copied into toSpace, if they are not already copied. The referent of each root in the bins is copied first, and the rest of the cells that are σ-reachable from r are either copied within the **if** (σ == $\mathbf{0}_\epsilon$) or in the copy function (for other values of σ). startp is a logical variable unrelated to garbage collection. Its only role is in describing an invariant that would be used in arguing the correctness of the collector.

```
Function gc():
    swap(fromSpace, toSpace)
     allocp ← beginning of toSpace
    for each bin σ in {⊤, 1*, 1ε, 0ε, ε} do
        for each root r in σ do
            scanp ← allocp; startp ← allocp
            r ← copy(r)
            if σ == 0ε then
                car(r) ← copy(car(r))
                scanp ← allocp
            while scanp < allocp do
                if scanp points to a cons cell then
                    switch σ do
                        case ⊤ do
                            car(scanp) ←
                             copy(car(scanp))
                            cdr(scanp) ←
                             copy(cdr(scanp))
                            scanp ←
                             scanp + size(scanp)
                        case 1* do
                            cdr(scanp) ←
                             copy(cdr(scanp))
                            scanp ←
                             scanp + size(scanp)
                        case 1ε do
                            cdr(scanp) ←
                             copy(cdr(scanp))
                            scanp ← allocp
                        case ε do
                            scanp ← allocp
                else
                    /*scanp points to nil or integer*/
                    scanp ← allocp
Function copy(r):
    if r has no forwarding address then
        r′ ← allocp
        copy *r to *r′
        allocp ← allocp + size(r)
        set forwarding address of r as r′
    return (forwarding address of r)
```

Algorithm 1: The garbage collection algorithm

Correctness of the Algorithm. We use a variation of the coloring abstraction that is often used to argue about the soundness of garbage collectors. We colour memory cells as follows: A cell is black if it and its σ-reachable children have been copied to toSpace. A cell is grey, if it has been copied into toSpace, but at least one of its σ-reachable children has not been copied. Finally, a cell is white if the cell itself has not been copied to toSpace. The correctness of the algorithm follows from the following invariant at the beginning of the while loop:

1. startp ≤ scanp ≤ allocp, *and*
2. *all cells with starting addresses within* [startp, scanp), *are black and are σ-reachable from* r.

Consider a bin σ and a root variable r from the σ selected by the headers of the two **for** loops. While we do

not prove the invariant stated above, we show as its consequence that when the while loop terminates, the algorithm would have correctly copied all cells that are σ-reachable from r. First notice that just after the termination of the while loop, scanp = allocp, and thus there are no grey cells. Now assume to the contrary that there is a cell n that is σ-reachable from the root and is not colored black. Since the root has been copied before the while loop, we can find a cell in [startp, scanp) with at least one white σ-reachable child. Thus the color of this cell is grey, which is contradictory to the loop invariant which says that all cells such that within [startp, scanp) are black.

5 Experimental Results

To demonstrate the effectiveness of our method, we have built an interpreter for the target language and integrated it separately with the following three collectors: Our collector based on the finite liveness domain Σ (referred to as LGC_F), an implementation of the collector based on arbitrary liveness values [3] (referred to as LGC_A) and a reachability based garbage collector (referred to as RGC). The hardware platform for our experiments was a third generation *i*7 processor with 8GB RAM. Any difference in the reported timings in [3] and what we report for the same benchmark is attributable to two reasons: the input sizes of the benchmark and the hardware platform (both not reported in [3]). The collectors were benchmarked with programs from various sources including the standard no-fib suite. In addition, LGC_F was tested with four medium-sized higher-order programs: Barnes-Hut simulation, a list based floating point arithmetic program (IEEE 754), a functional parser, and curves—a program to generate fractals.

Results. The graphs in figure 7 indicate memory usage patterns of some of the benchmarks. The x-axis measures progress in execution in terms of the cumulative number of cells allocated, which we call *GC Ticks*. The y-axis is the number of cells in toSpace. The black line denotes the actual number of live cells at every tick. This metric is computed by recording the GC-ticks at creation and at last use of each allocated cell at runtime. Note that the graphs represent memory usage of programs when run with the heap size set to the minimum required for execution under RGC. The saw-tooth nature of the curves is explained by the number of cells in toSpace growing one-on-one with each GC-Tick and then instantaneously falling during the release of cells during garbage collection. A line connecting the lowest points of the curve for a GC scheme represents the *analysed liveness* of the analysis accompanying the scheme

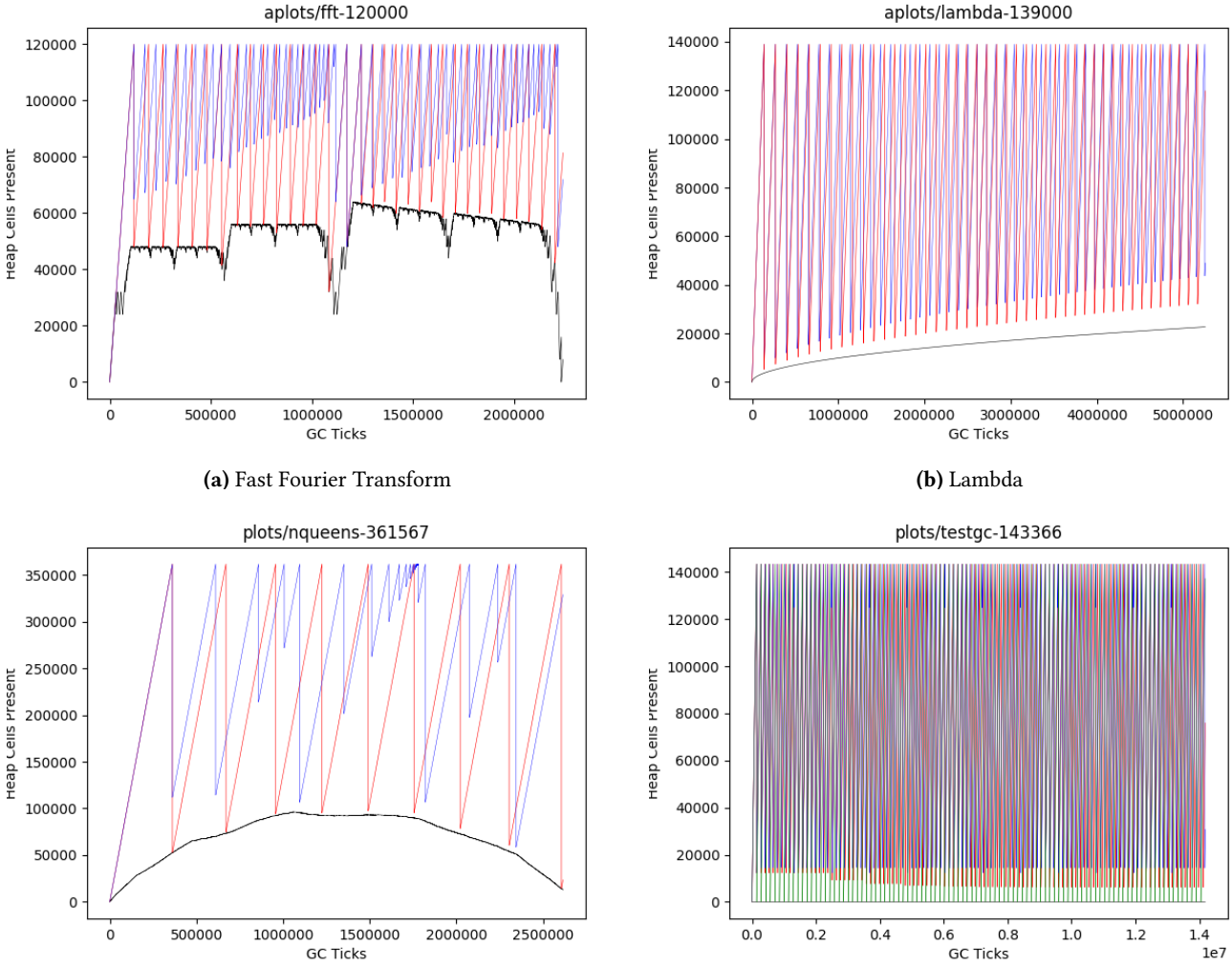
Precision of Liveness and Minimum Heap Requirement. Figure 7 shows the precision of liveness analysis. This shows up as the extent to which the analysed liveness comes close to real liveness during garbage collection. The effectiveness of LGC_F over RGC in identifying live memory is evident—LGC_F comes much closer to the real liveness curve than RGC. As expected, the gaps in the curves between analysed and real liveness for `lambda` and `testgc` also show that liveness analysis in LGC_F is an approximation. The memory usage plots also show that saw-toothed pattern for LGC_F is more widely spaced because of fewer GC invocations.

We avoid plotting memory usage for LGC_A, since it is indistinguishable from LGC_F for all benchmarks except `testgc`. Thus our restricted set of liveness values captures liveness almost as precisely as LGC_A. The program `testgc` (Figure 7 (d)), is a notable exception. It is a crafted program with a irregular memory access pattern that cannot be captured by our set of liveness values.

Since LGC_F, in general, releases as much memory (and often more) than RGC during each collection, it is natural to expect programs executing under LGC_F to run with smaller heap sizes. This is confirmed by Table 1. The minimum heap requirements of LGC_F and LGC_A are comparable on most programs. For the program `testgc` LGC_F falls short because of imprecise liveness analysis. Surprisingly, LGC_F performs better that LGC_A for `gcbench`, a synthetic program from the no-fib benchmark suite which creates a tree but does not use any of its parts. Given the comparable precision of liveness analysis for both LGC_F and LGC_A for this program, the superior minheap performance of LGC_F can only be attributed to the additional precision from the context-based separation of analysis of function bodies in our method.

Table 1. Minimum Heap Cells (in bytes) required for execution under various GC schemes.

	RGC	**LGC_F**	**LGC_A**
lambda	138340	102500	102500
fft	113500	64502	64502
testgc*	143366	12291	41
treejoin	525488	7150	7150
fibheap	254520	13558	13558
nperm	930264	8920	8920
huffman	150041	50029	50023
gc_bench*	131071	102401	131071
nqueens	361567	98531	98531
knightstour	508225	307091	307090
lcss	5M+	< 50000	< 50000
paraffins	892090	891194	891194
sudoku	160277	85363	85347
deriv	491462	491446	491446
curves	204000	96000	-
parser	262216	131096	-
barneshut	6830	1530	-
IEEE-754	39360	122	-



(a) Fast Fourier Transform

(b) Lambda

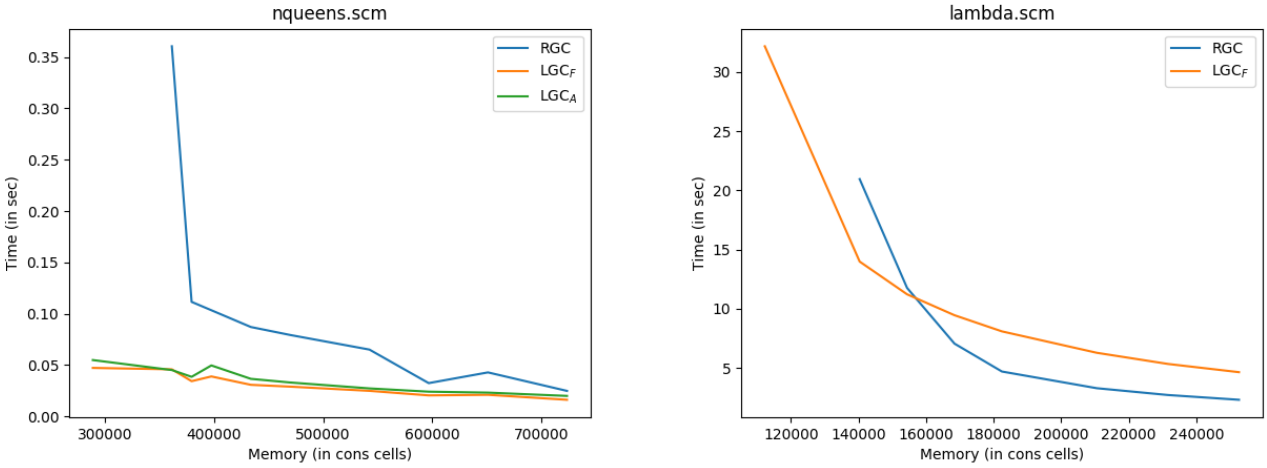
(c) N-Queens

(d) TestGC

Figure 7. Memory plots of programs: x-axis is the cumulative number of **cons** cells allocated and y-axis is the number of **cons** cells in toSpace. The black curve represents real liveness, the blue and the orange curves are the number of cells in toSpace for RGC and LGC_F respectively. The LGC_A curve is indistinguishable from LGC_F for programs (a) to (c). The exception amongst all benchmarks is `testgc` shown in (d). The green curve representing LGC_A in this plot reaches real liveness at each GC, whereas LGC_F does not.

Analysis Time. A comparison of the time taken for liveness analysis for the two liveness-based collectors is in Table 2. LGCF has much lower analysis time with some examples showing a difference of several orders of magnitude. This is expected since, as explained at the end of Section 3.1, the work-list algorithm of LGC_F with its short lattice height is computationally much less expensive than the analysis of LGC_A.

Going beyond the table that shows the analysis time of our benchmark programs, we believe that the analysis will scale for larger programs for the following reasons. The N_f^2 term in the theoretical worst case bound of the expensive $\mathcal{FS}$ computation arises out of the assumption that each function can potentially call every other function in the program. However, if we make the more reasonable assumption that the number of functions called by a function is bounded by a constant, the bound is linear in N_f. We have experimentally observed that, during the computation of $\mathcal{FS}$, the number of iterations per function over all programs in our benchmarks is less than 10. Similarly, the worst time per iteration over our benchmark programs is 0.25 ms. Considering these figures to be representative over arbitrary programs, a realistic yet conservative estimate of the analysis time for a large program with *1000* functions is $0.25 * 10 * 1000 = 2.5secs$.



(a) N-Queens **(b)** Lambda

Figure 8. Time taken by garbage collector vs the amount of heap memory available. In Figure (b) the LGC$_A$ plot is avoided due to difference in scale.

Table 2. Time taken for compile-time analysis.

	LGC$_F$ (in ms)	LGC$_A$ (in ms)
lambda	6.38	6067.00
fft	17.24	546.17
testgc	4.32	29.19
treejoin	5.71	3278.70
fibheap	30.53	2049.89
nperm	48.58	174.67
huffman	11.18	1459.72
gc_bench	18.75	7.61
nqueens	3.86	296.11
knightstour	4.90	432.86
lcss	14.59	980.44
paraffins	33.56	3784.40
sudoku	23.52	187162.00
deriv	6.80	327.75
curves	37.40	-
parser	11.44	-
barneshut	152.76	-
IEEE-754	98.52	-

Time for Garbage Collection. The graphs in Figure 8 plot the average garbage collection time (gc-time) for different collectors over a range of heap sizes. The time measurements are accurate modulo the vagaries of capturing CPU Time. We reiterate the factors that affect gc-time overheads in LGC$_F$. First, as mentioned in Section 4, LGC$_F$ does a binning operation that incurs the costs of a bucket sort during garbage collection time. On the other hand, given the precision of our liveness analysis arrived by several means, the number of copied cells is generally lower than RGC. In addition, the number of collector invocations is also lower for LGC$_F$. All of these add up to significant gc-time gains for LGC$_F$ over RGC in memory constrained regions.

In contrast, the regions where memory is abundantly available, the difference between number of collections is not significant (in many cases, non-existent) and we find the benefits diminishing. For systems where memory is readily available, performing such an analysis is not computationally beneficial and RGC performs better, as seen in Figure 8 `lambda`.

An issue related to gc-time is the number of redundant traversals of the heap during collection. This value will be high if there are repeated heap traversal during a garbage collection. LGC$_F$ has an advantage over LGC$_A$ in that the average number of visits per copied cell is much lower (Figure 9). For LGC$_F$, this ratio is close to 1, and almost always matches RGC. Some programs are not represented in this plot because their LGC$_A$ ratios go beyond scale.

Since our method carries over large amounts of liveness data from analysis to execution, a natural concern would be the size of this data. The amount of space needed to store the analysis data is $N_p * (N_l * (l+1))$ bytes, where N_p is the number of the garbage collection points, N_l is the average number of live variables at each garbage collection point and l is the number of points in the lattice. Empirically, the size of this data is in the range of a few (around 5) KBs for all programs, with the only outlier being `sudoku`, which requires around 20KB.

6 Related Work

There is a substantial body of work related to the the use of static and dynamic analyses for improving usage of heap memory. The empirical study in [9], done in the context of

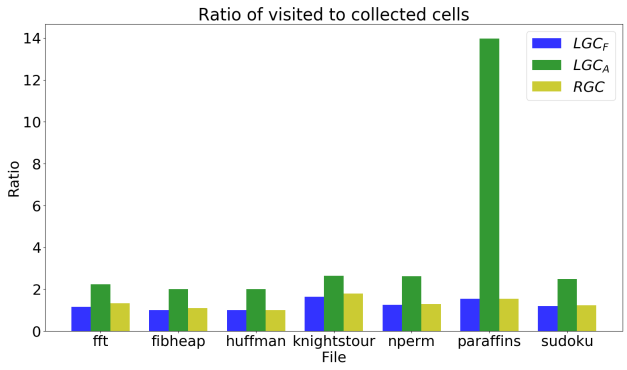


Figure 9. Ratio of total heap cell visited to total heap cells collected, both figures cumulative over all GC invocations. A higher ratio points towards repeated traversal of heap during collection.

C, C++ and Eiffel, was the first to recommend the use of liveness to enhance the effectiveness of garbage collection. The garbage collector was based on a liveness analysis obtained from program traces. The authors discovered that heaps often contained a large number of cells that are reachable but not live (upto 32%). Subsequent studies by [3, 15, 22, 24] have confirmed this feature across several languages: Java, Haskell and Scheme. The conclusion in [9] is that for liveness-based garbage collection to be effective, one must analyze global variables with interprocedural analysis. This message is relevant for us since our analysis *is* interprocedural, and instead of globals we have heap-allocated data that, like globals, is shared across different functions.

We first discuss the use of liveness in garbage collection for imperative languages. The heap safety analysis in [25] suggests a method for answering queries regarding the safety of inserting a `free` statement for a variable or nullifying a reference at a given location, so that the referent object can be collected subsequently. The approach in [16] is similar in its end goals. However, while heap safety analysis merely solves a decision problem—it answers queries regarding whether a reference can be nullified at a program point, the approach in [16] also identifies candidate references which can be nullified at each program point. Both these studies are in the context of Java programs. Similar approaches that either verify the safety of explicitly freed memory or insert memory freeing statements are [6, 8].

In the space of functional languages, many of the approaches are aimed at reallocation through analysis followed by code insertion, also called *compile time garbage collection*. The most well known of these is, of course, deforestation [4, 5, 27]. Deforestation methods essentially identify commonly occurring patterns of code that allocate a chunk of memory and then consume it. Such patterns are fused so that the allocation-consumption takes places at a finer level of granularity, and a small constant amount of memory is repeatedly allocated and consumed immediately. Other approaches to compile time garbage collection are [12], [18] and [7]. In addition, there are sharing analysis based reallocation [14] and region based analysis [26]. Hoffman [10] uses linear types to compile functional programs to C programs that perform in-place updates. This is similar to our motivating example in its end goal. An approach more relevant to garbage collection is [11], which uses linear type system to infer an abstract form of reference counting. However, the method is built on limited datatypes that does not even include lists in their full generality. In addition, the method is not supported by an implementation.

The approaches that are closest to ours are [3] and [17]. The first paper suggests a liveness based garbage collection for a first-order eager language. The second paper reworks the method for a lazy language that also manages to garbage collects parts of closures. However to represent liveness values, they use an unrestricted set of demands (arbitrary subsets of $(\mathbf{0}+\mathbf{1})^*$). A demand is represented as an automaton, and garbage collection is a simultaneous depth first traversal of the heap and the automaton. Construction of the automaton is costly. It involves approximating a context free grammar to a nondeterministic automaton, which is then converted to a deterministic finite automaton—both costly processes. Since the set of demands is infinite with an absence of a total order, they cannot use a binning technique and avoid revisits as we do in our method. The idea of using a finite domain of demands to simplify the analysis comes from strictness analysis [13, 28]. Strictness analysis is similar to liveness, and is used for space optimization of lazy languages. Recollect that a lazy language does not evaluate an expression until it is required. Strictness analysis determines the parts of an argument of a function that are guaranteed to be used inside it. The identified parts can then be evaluated before being passed to the function instead of being passed as space consuming closures. Just as in our case, limiting the strictness values to a small domain enables the analysis to adress higher order functions. The idea of separating the liveness environment by the calling context in a function body also appears in [20], albeit with a different notion of context.

The study in [1] observes that liveness may aid generational garbage collection. Because of the early collection of objects, fewer objects would be moved into the tenured generation and thus result in fewer major collections. We believe that our implementation can be easily modified to work in generational setting; the only additional issue in our case being that inter-generational pointers have to recorded along with their liveness values.

7 Conclusions

The disciplined use of recursive functions on algebraic data types is a common paradigm that is followed while programming in a functional language. This has an interesting

consequence on heap usage. The traversals on the heap are regular and follow identifiable patterns. In this paper we have identified a few such patterns that are used to approximate liveness values of structured data. The liveness values form a lattice and the liveness analysis based on this lattice is much more efficient and as precise as previous methods that use an unrestricted set of liveness values. The use of a restricted set of liveness values also leads to an improvement in the design of the collector which does a breadth-first traversal of the heap and also minimizes redundant traversals. An implementation of a garbage collector based on the restricted liveness set (LGC_F) confirms several benefits over collectors that use an arbitrary set of liveness values (LGC_A), and also over reachability based collectors (RGC): The analysis time of LGC_F is a significant improvement over LGC_A, the minimum memory required for programs to run is comparable with LGC_A, the number of redundant traversals over memory cells reduce to the level of RGC, and garbage collection times are better than both LGC_A and RGC.

Since the choice of liveness values in our case is not backed by sufficient evidence of its applicability to a wider range of programs, an avenue for improvement is a data-driven study in the spirit of [9] to identify a better choice of a lattice of liveness values. The choice could even be specialized to a set of target programs. While it is not very difficult to change the lattice of liveness values in our current implementation, a more careful implementation could fully parameterize the collector implementation with respect to the choice of lattice.

One could also extend the scope of the garbage collector to support a more realistic functional language, and even to a lazy language like Haskell by extending the techniques in [17]. However, whether our technique could be extended to programs in mainstream imperative languages like Java with their state changing features and unrestricted usage of heap, requires investigation.

It has been already been demonstrated in [24] that Java programs too carry considerable data that are dead but reachable. Given this, one could ask whether the techniques used here could carry over to programs written in Java. A possible approach is based on [2]. This paper shows a close relation between SSA, an intermediate form of programs used by most compilers and functional programs of the kind that we have considered in the paper. It does seem possible to take the SSA form of programs written in Java by tapping into the workflow of a compiler and following the technique suggested in [2] convert these into the variant of Scheme considered in this paper. However, given the different styles of programming, whether it would be possible to identify a small set of liveness patterns that can be used to yield an analysis of reasonable precision is an open question.

References

[1] Ole Agesen, David Detlefs, and J. Eliot Moss. 1998. Garbage Collection and Local Variable Type-Precision and Liveness in Java Virtual Machines. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation* (Montreal, Quebec, Canada) *(PLDI '98)*. ACM, New York, NY, USA, 269–279. https://doi.org/10.1145/277650.277738

[2] Andrew W. Appel. 1998. SSA is Functional Programming. *ACM SIGPLAN NOTICES* 33, 4 (1998), 17–20.

[3] Rahul Asati, Amitabha Sanyal, Amey Karkare, and Alan Mycroft. 2014. Liveness-Based Garbage Collection. In *Compiler Construction*, Albert Cohen (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 85–106.

[4] Olaf Chitil. 1999. Typer Inference Builds a Short Cut to Deforestation. In *Fourth ACM SIGPLAN International Conference on Functional Programming* (Paris, France) *(ICFP '99)*. ACM, New York, NY, USA, 249–260. https://doi.org/10.1145/317636.317907

[5] Andrew Gill, John Launchbury, and Simon L. Peyton Jones. 1993. A Short Cut to Deforestation. In *Conference on Functional Programming Languages and Computer Architecture* (Copenhagen, Denmark) *(FPCA '93)*. ACM, New York, NY, USA, 223–232. https://doi.org/10.1145/165180.165214

[6] Samuel Z. Guyer, Kathryn S. McKinley, and Daniel Frampton. 2006. Free-Me: A Static Analysis for Automatic Individual Object Reclamation. In *27th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Ottawa, Ontario, Canada) *(PLDI '06)*. ACM, New York, NY, USA, 364–375. https://doi.org/10.1145/1133981.1134024

[7] G. W. Hamilton. 1995. Compile-Time Garbage Collection for Lazy Functional Languages. In *International Workshop on Memory Management (IWMM '95)*. Springer-Verlag, Berlin, Heidelberg, 119–144.

[8] Matthew Hertz and Emery D. Berger. 2005. Quantifying the Performance of Garbage Collection vs. Explicit Memory Management. *SIGPLAN Not.* 40, 10 (Oct. 2005), 313–326.

[9] Martin Hirzel, Amer Diwan, and Johannes Henkel. 2002. On the Usefulness of Type and Liveness Accuracy for Garbage Collection and Leak Detection. *ACM Trans. Program. Lang. Syst.* 24, 6 (Nov. 2002), 593–624. https://doi.org/10.1145/586088.586089

[10] Martin Hofmann. 2000. A Type System for Bounded Space and Functional In-Place Update–Extended Abstract. In *Programming Languages and Systems, 9th European Symposium on Programming, ESOP 2000 (Lecture Notes in Computer Science)*, Gert Smolka (Ed.), Vol. 1782. Springer, New York, NY, USA, 165–179. https://doi.org/10.1007/3-540-46425-5_11

[11] Atsushi Igarashi and Naoki Kobayashi. 2000. Garbage Collection Based on a Linear Type System.

[12] Katsuro Inoue, Hiroyuki Seki, and Hikaru Yagi. 1988. Analysis of Functional Programs to Detect Run-Time Garbage Cells. *ACM Trans. Program. Lang. Syst.* 10, 4 (Oct. 1988), 555–578. https://doi.org/10.1145/48022.48025

[13] Kristian Damm Jensen, Peter Hjæresen, and Mads Rosendahl. 1994. Efficient strictness analysis of Haskell. In *Static Analysis*, Baudouin Le Charlier (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 346–362.

[14] Simon B. Jones and Daniel Le Métayer. 1989. Compile-Time Garbage Collection by Sharing Analysis. In *Fourth International Conference on Functional Programming Languages and Computer Architecture* (Imperial College, London, United Kingdom) *(FPCA '89)*. ACM, New York, NY, USA, 54–74. https://doi.org/10.1145/99370.99375

[15] Amey Karkare, Amitabha Sanyal, and Uday Khedker. 2006. Effectiveness of Garbage Collection in MIT/GNU Scheme. http://arxiv.org/abs/cs/0611093.

[16] Uday P. Khedker, Amitabha Sanyal, and Amey Karkare. 2007. Heap Reference Analysis Using Access Graphs. *ACM Trans. Program. Lang. Syst.* 30, 1 (Nov. 2007), 1–es. https://doi.org/10.1145/1290520.1290521

[17] Prasanna Kumar K., Amitabha Sanyal, and Amey Karkare. 2016. Liveness-Based Garbage Collection for Lazy Languages. In *2016 ACM SIGPLAN International Symposium on Memory Management* (Santa Barbara, CA, USA) *(ISMM 2016)*. ACM, New York, NY, USA, 122–133. https://doi.org/10.1145/2926697.2926698

[18] Oukseh Lee, Hongseok Yang, and Kwangkeun Yi. 2003. Inserting Safe Memory Reuse Commands into ML-like Programs. In *10th International Conference on Static Analysis* (San Diego, CA, USA) *(SAS'03)*. Springer-Verlag, Berlin, Heidelberg, 171–188.

[19] Neil Mitchell and Colin Runciman. 2009. Losing Functions without Gaining Data: Another Look at Defunctionalisation. In *2nd ACM SIGPLAN Symposium on Haskell* (Edinburgh, Scotland) *(Haskell '09)*. ACM, New York, NY, USA, 13–24. https://doi.org/10.1145/1596638.1596641

[20] Rohan Padhye and Uday P. Khedker. 2013. Interprocedural Data Flow Analysis in Soot Using Value Contexts. In *2nd ACM SIGPLAN International Workshop on State Of the Art in Java Program Analysis* (Seattle, Washington) *(SOAP 13)*. ACM, New York, NY, USA, 31–36.

[21] K Prasanna Kumar, Amitabha Sanyal, Amey Karkare, and Saswat Padhi. 2019. A Static Slicing Method for Functional Programs and Its Incremental Version. In *28th International Conference on Compiler Construction* (Washington, DC, USA) *(CC 2019)*. ACM, New York, NY, USA, 53–64.

[22] Niklas Röjemo and Colin Runciman. 1996. Lag, Drag, Void and Use—Heap Profiling and Space-Efficient Compilation Revisited. In *First ACM SIGPLAN International Conference on Functional Programming* (Philadelphia, Pennsylvania, USA) *(ICFP '96)*. ACM, New York, NY, USA, 34–41. https://doi.org/10.1145/232627.232633

[23] Amr Sabry and Matthias Felleisen. 1992. Reasoning about Programs in Continuation-Passing Style.. In *1992 ACM Conference on LISP and Functional Programming* (San Francisco, California, USA) *(LFP '92)*. ACM, New York, NY, USA, 288–298. https://doi.org/10.1145/141471.141563

[24] Ran Shaham, Elliot K. Kolodner, and Mooly Sagiv. 2000. On Effectiveness of GC in Java. In *2nd International Symposium on Memory Management* (Minneapolis, Minnesota, USA) *(ISMM '00)*. ACM, New York, NY, USA, 12–17. https://doi.org/10.1145/362422.362430

[25] Ran Shaham, Eran Yahav, Elliot K. Kolodner, and Mooly Sagiv. 2003. Establishing Local Temporal Heap Safety Properties with Applications to Compile-Time Memory Management. In *10th International Conference on Static Analysis* (San Diego, CA, USA) *(SAS'03)*. Springer-Verlag, Berlin, Heidelberg, 483–503.

[26] Mads Tofte and Lars Birkedal. 1998. A Region Inference Algorithm. *ACM Trans. Program. Lang. Syst.* 20, 4 (July 1998), 724–767. https://doi.org/10.1145/291891.291894

[27] Philip Wadler. 1988. Deforestation: Transforming programs to eliminate trees. In *ESOP '88*, H. Ganzinger (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 344–358.

[28] Philip Wadler and R. J. M. Hughes. 1987. Projections for strictness analysis. In *Functional Programming Languages and Computer Architecture*, Gilles Kahn (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 385–407.