Liveness-Based Garbage Collection for Lazy Languages

Double Blind Review

Abstract

We consider the problem of reducing the memory required to run lazy first-order functional programs. To do this, we analyze programs for liveness of heap-allocated data. The accompanying garbage collector is modified to use the result of the analysis to preserve only live data—a subset of reachable data—during garbage collection. This results in an increase in the garbage reclaimed and consequently reduces the memory footprint of programs. While this technique has already been shown to yield benefits for eager first-order languages, the lack of a statically determinable execution order and the presence of closures at runtime pose additional challenges for lazy languages. This requires changes both in the earlier liveness analysis and also in the garbage collection scheme required to reclaim memory that is not live.

The implementation of our analysis annotates each potential garbage collection point in the program with a set of deterministic finite-state automata (DFA) describing the liveness at that point. We also implement a copying collector that uses the DFA to preserve live objects. Our experiments confirm that liveness-based collection always increases the garbage reclaimed, consequently decreasing the number of collections. In addition, it reduces the execution time of several programs. Experiments also reveal that the best results are obtained for a mixed garbage collection scheme that does a liveness-based collection for fully evaluated data and reachability-based collection for closures. On our benchmark programs, we obtain a decrease of up to 45% in the number of collections, and a decrease of up to 88% in the number of dragged cells. Even though liveness based collector uses an expensive traversal scheme, the execution time is competitive (0.2X - 2.1X) when compared to a reachability based collector.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Memory Management (Garbage Collection), Optimizations; F.3.2 [*Logic and Meanings Of Programs*]: Semantics of Programming Languages—Program Analysis

General Terms Algorithms, Languages, Theory

Keywords Heap Analysis, Liveness Analysis, Memory Management, Garbage Collection, Lazy Languages

1. Introduction

Functional programs use dynamically allocated memory extensively. The allocation is either explicit (while using constructors, for example) or implicit (as while creating closures). Programs written in lazy functional languages put additional demands on memory as they require closures to be carried from the point of creation to the point of evaluation.

While the runtime system of most functional languages includes a garbage collector (GC) to efficiently reclaim memory, empirical studies on Scheme [**?**] and, more importantly, on Haskell [**?**] programs have shown that GCs leave uncollected a large number of memory objects that are reachable but not live (here *live* means the object can potentially be used by the program at a later stage). This results in unnecessary memory retention.

In this paper, we propose the use of liveness analysis of heap cells for garbage collection in a lazy first-order functional language. The central notion in our analysis is a generalization of liveness called *demand*—the pattern of future uses of the value of an expression. The analysis has two parts. First, each function is summarized in a context-sensitive manner as a *demand transformer* that transforms a *symbolic* demand on its body to demands on its arguments. This summary is then used to step through function calls during analysis. Second, the concrete demand on a function body is obtained through a 0-CFA-like conservative approximation that combines the demands on all the calls to the function. The result of the analysis is an annotation of each program point with finite-state automata capturing the liveness of variables at that point. Depending on the program point where garbage collection has been triggered, the GC consults a set of automata to restrict reachability during marking. This results in an increase in the garbage reclaimed and consequently in fewer garbage collections.

While this idea has been shown to be effective [**?**] for a first-order *eager* language, a straightforward extension of the earlier technique is not possible for lazy languages, where a heap-allocated object may be an unevaluated expression or closure. Since data is made live by evaluation of closures, and in lazy languages the place in the program where this evaluation takes place cannot be statically determined, laziness complicates liveness analysis itself. In addition, apart from evaluated data, we extend liveness-based collection to closures—closures with no future use should also be collected. This involves tracking closures to their points of creation at runtime from where we obtain their liveness. This requires significant modification to the earlier GC scheme.

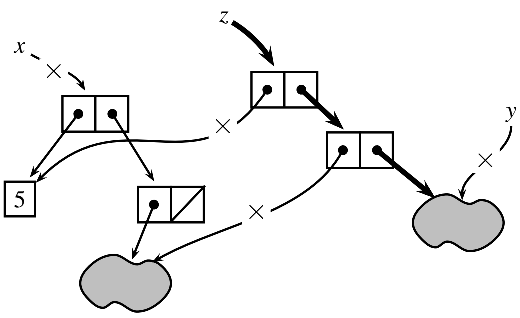
Experiments with a single generation copying collector (Section **??**) confirm the expected performance benefits. Liveness-based collection results in an increase in garbage reclaimed. As a consequence, there is a decrease in the number of collections (up to 45%) and a decrease (up to 88%) in the number of *dragged* cells (cells that survive a GC after their last use). Besides, there is a reduction in the overall execution time for some programs. Our experiments show that a mixed strategy, where we do liveness based GC up to the closure boundary and reachability based GC for the arguments (free variables) of the closure, gives us similar benefits as a fully liveness based GC (where the arguments of the closure are also collected using liveness) with lesser overhead (Section **??**).

```
(define (length l)
    (if (null? l) 0 (+ 1 (length(cdr l)))))

(define (append l1 l2)
    (if (null? l1) l2
        (cons (car l1) (append (cdr l1) l2))))

(let x ← (cons 5 (cons (cons 6 nil) nil)) in
    (let y ← (cons 3 nil) in
        (let z ← (append x y) in
            (if (null? (car z)) 0 π: (length z))))))
```

(a) Example program.



(b) Memory graph at π. denotes a closure. Thick edges denote live links. Traversal stops at edges marked × during garbage collection for an ideal collector.

Figure 1. Example Program and its Memory Graph

1.1 Motivating Example

Figure **??** shows an example program and the state of the heap at the program point π, i.e. just before the evaluation of (**length** z). The heap is represented by a graph in which a node either represents atomic values (**nil**, integers etc.), or a **cons** cell containing **car** and **cdr** fields, or a chunk of memory representing an unevaluated value also called a *closure* (represented by hatched boxes). Edges in the graph are *references* and emanate from variables or fields. The figure shows the lists x and z partially evaluated. The evaluation was triggered by the test (**null**? (**car** z)) in the **if** expression.

The edges shown by thick arrows are those which are live at π. Ideally, a cell should be preserved during garbage collection only if it is reachable from the root set through a path of live edges. All other cells can be reclaimed. Thus if a garbage collection takes place at π with the heap shown in Figure **??**(b), an ideal liveness-based collector (LGC) will preserve only the cells referenced by z, (**cdr** z) and (the cells constituting) the closure referenced by (**cdr** (**cdr** z)). In contrast, a reachability-based collector (RGC) would have preserved all cells.

While an ideal garbage collector is impractical, in this work we show that static analysis of heap data can help practical garbage collectors in reclaiming more garbage. The specific contributions of this paper are

- We formulate a liveness analysis for a lazy first order functional language and prove its correctness. We show how to compute function summaries in a context independent manner to achieve a context sensitive analysis. We further describe the transformation of the liveness equations to grammar rules in a way that the language accepted by the resulting grammar describes live heap locations.

$$
\begin{array}{rcll}
p \in Prog & ::= & d_1 \ldots d_n\ e_{\mathbf{main}} & \textit{— program} \\
df \in Fdef & ::= & (\mathbf{define}\ (f\ x_1\ \ldots\ x_n)\ e) & \textit{— function def} \\
e \in Expr & ::= & \begin{cases} (\mathbf{if}\ x\ e_1\ e_2) & \textit{— conditional} \\ (\mathbf{let}\ x \leftarrow s\ \mathbf{in}\ e) & \textit{— let binding} \\ (\mathbf{return}\ x) & \textit{— return from function} \end{cases} & \\
s \in App & ::= & \begin{cases} k & \textit{— constant (numeric or } \mathbf{nil}\textit{)} \\ (\mathbf{cons}\ x_1\ x_2) & \textit{— constructor} \\ (\mathbf{car}\ x) & \\ (\mathbf{cdr}\ x) & \textit{— selectors} \\ (\mathbf{null?}\ x) & \\ (+\ x_1\ x_2) & \textit{— tester/generic arithmetic} \\ (f\ x_1\ \ldots\ x_n) & \textit{—function application} \end{cases} &
\end{array}
$$

Figure 2. The syntax of our language

- We prove the undecidability of the grammar that results from the analysis. To overcome this, we present an algorithm to transform grammars to finite state automata (DFA) such that the language accepted by the DFA over-approximates the liveness (and hence a safe approximation).
- We have implemented a garbage collector that can use the DFA to retain live cells that also include closures. We have done extensive experiments with this garbage collector and obtained empirical results that show the effectiveness of liveness-based garbage collection.

1.2 Organization of the paper

Section **??** introduces the syntax of the programming language considered in our analysis and a small-step operational semantics for it. We present our liveness analysis for this lazy language in Section **??**. Section **??** describes the formulation of the liveness as a grammar and its approximation to DFA. It also contains a proof showing the undecidability of the precise liveness in our formulation which necessitates the approximation to DFA. Section **??** discusses different ways of handling closures and their pros and cons. We report our experimental results in Section **??** along with some observations. Section **??** discusses previous work related to garbage collection and liveness and Section **??** discusses possible extensions and concludes the paper.

2. The target language—syntax and semantics

Figure **??** describes the syntax of our language. It is a first order language with lazy semantics. Programs are restricted to be in Administrative Normal Form (ANF) [**?**] where all actual parameters to functions are variables. While this restriction does not affect expressibility, this form has the benefit of making explicit the creation of closures through the **let** construct. **let**s in our language are lazy; in the expression **let** $x \leftarrow s$ **in** e, x may occur in s. This enables creation of graph-like structures in a pure language. The restriction of **let** to a single definition is for ease of exposition—generalization to multiple definitions does not add conceptual difficulties. We further restrict each variable in a program to be distinct, so that no scope shadowing occurs—this simplifies the reasoning about the program. The **return** and **if** expressions and strict operators trigger evaluation of closures.

The body of a function f is denoted as e_f. We shall extend this convention to the main expression $e_{\mathbf{main}}$. We assume that each program has a distinguished function **main** with the definition (**define** (**main**) $e_{\mathbf{main}}$) and the execution of the program starts with the call to **main**. We write $\pi\!:\!e$ to associate the label π (not part of the language syntax) with the program point just before expression e.

Premise	Transition	Rule name
	$\rho,\, (\rho',\ell,e):S,\, H,\, \kappa \rightsquigarrow \rho',\, S,\, H[\ell := \kappa],\, e$	CONST
	$\rho,\, (\rho',\ell,e):S,\, H,\, (\textbf{cons}\; x\; y) \rightsquigarrow \rho',\, S,\, H[\ell := (\rho(x),\rho(y))],\, e$	CONS
$H(\rho(x))$ is $\langle s,\rho'\rangle$	$\rho,\, S,\, H,\, (\textbf{car}\; x) \rightsquigarrow \rho',\, (\rho, x, (\textbf{car}\; x)):S,\, H,\, s$	CAR-CLO
$H(\rho(x))$ is $(\langle s,\rho'\rangle,d)$	$\rho,\, S,\, H,\, (\textbf{car}\; x) \rightsquigarrow \rho',\, (\rho, addr(\langle s,\rho,\rangle), (\textbf{car}\; x)):S,\, H,\, s$	CAR-1-CLO
$H(\rho(x))$ is (v,d)	$\rho,\, (\rho',\ell,e):S,\, H,\, (\textbf{car}\; x) \rightsquigarrow \rho',\, S,\, H[\ell := v],\, e$	CAR-SELECT
$H(\rho(x)), H(\rho(y)) \in \mathbb{N}$	$\rho,\, (\rho',\ell,e):S,\, H,\, (+\; x\; y) \rightsquigarrow \rho',\, S,\, H[\ell := H(\rho'(x)) + H(\rho'(y))],\, e$	PRIM-ADD
$H(\rho(x))$ is $\langle s,\rho'\rangle$	$\rho,\, S,\, H,\, (+\; x\; y) \rightsquigarrow \rho',\, (\rho, \rho(x), (+\; x\; y)):S,\, H,\, s$	PRIM-1-CLO
$H(\rho(y))$ is $\langle s,\rho'\rangle$	$\rho,\, S,\, H,\, (+\; x\; y) \rightsquigarrow \rho', (\rho,\, \rho(y),\, (+\; x\; y)):S,\, H,\, s$	PRIM-2-CLO
f defined as $(\textbf{define}\; (f\; \vec{y})\; e_f)$	$\rho,\, S,\, H,\, (f\; \vec{x}) \rightsquigarrow [\vec{y} \mapsto \rho(\vec{x})],\, S,\, H,\, e_f$	FUNCALL
ℓ is a new location	$\rho,\, S,\, H,\, (\textbf{let}\; x \leftarrow s\; \textbf{in}\; e) \rightsquigarrow \rho \oplus [x \mapsto \ell],\, S,\, H[\ell \mapsto \langle s, \lfloor\rho\rfloor_{FV(s)} \oplus [x \mapsto \ell]\rangle],\, e$	LET
$H(\rho(x)) \neq 0$	$\rho,\, S,\, H,\, (\textbf{if}\; x\; e_1\; e_2) \rightsquigarrow \rho,\, S,\, H,\, e_1$	IF-TRUE
$H(\rho(x)) = 0$	$\rho,\, S,\, H,\, (\textbf{if}\; x\; e_1\; e_2) \rightsquigarrow \rho,\, S,\, H,\, e_2$	IF-FALSE
$H(\rho(x)) = \langle s,\rho'\rangle$	$\rho,\, S,\, H,\, (\textbf{if}\; x\; e_1\; e_2) \rightsquigarrow \rho',\, (\rho, \rho(x), (\textbf{if}\; x\; e_1\; e_2)):S,\, H,\, s$	IF-CLO
$H(\rho(x))$ is WHNF with value v	$\rho,\, (\rho',\ell,e):S,\, H,\, (\textbf{return}\; x) \rightsquigarrow \rho',\, S,\, H[\ell := v],\, e$	RETURN-WHNF
$H(\rho(x)) = \langle s,\rho'\rangle$	$\rho,\, S,\, H,\, (\textbf{return}\; x) \rightsquigarrow \rho',\, (\rho, \rho(x), (\textbf{return}\; x)):S,\, H,\, s$	RETURN-CLO

Figure 3. The small-step semantics for the language.

2.1 Semantics

We now give a small-step semantics for our language. We first specify the domains used by the semantics:

$$\begin{array}{rcll}
\mathrm{H} : \mathit{Heap} & = & \mathit{Loc} \to \mathit{Data} & \text{– Heap} \\
d : \mathit{Data} & = & \mathit{Val} + \mathit{Clo} & \text{– Values \& Closures} \\
v : \mathit{Val} & = & \mathbb{N} + \{\textbf{nil}\} + \mathit{Data} \times \mathit{Data} & \text{– Values} \\
c : \mathit{Clo} & = & (\mathit{App} \times \mathit{Env}) & \text{– Closures} \\
\rho : \mathit{Env} & = & \mathit{Var} \to \mathit{Loc} & \text{– Environment}
\end{array}$$

Here *Loc* is a countable set of locations in the heap. These either contain a value in *WHNF* (a number, the empty list **nil**, or a **cons** cell with possibly unevaluated constituents) or a *closure*. A closure is a pair $\langle s,\rho\rangle$ in which s is an unevaluated application, and ρ maps free variables of s to their respective locations. Since all data objects are boxed, we model an environment as a mapping from the set of variables of the program *Var* to locations in the heap.

The semantics of expressions (and applications[1]) are given by transitions of the form $\rho, \mathrm{S}, \mathrm{H}, e \to \rho', \mathrm{S}', \mathrm{H}', e'$. Here S is a stack of continuation frames. Each continuation frame is of the form (ρ, ℓ, e), signifying that the location ℓ has to be updated with the value of the currently evaluating expression, and e is to be evaluated next in the environment ρ. Note that the state of the transition system consists of the current evaluation context, given by ρ and e, and the suspended evaluation contexts (represented as continuation frames) in the stack S. The heap H is common to all contexts.

The initial state of the transition system described above is $([\,]_\rho, ([\,]_\rho, \texttt{answer}, (\textbf{print}\; \texttt{answer})) : [\,]_S, [\,]_H, (\textbf{main}))$, in which $[\,]_\rho$ and $[\,]_H$ are the initial environment and the initial heap. The initial stack consists of a single continuation frame in which `answer` is a distinguished variable that would eventually be updated with the value of (**main**). In addition, **print** is a function modeling a printing mechanism—a standard run-time support assumption for lazy languages—that prints the value of (**main**). Notice the use of the operator : for adding elements to the top of the stack.

[1] In most contexts, we shall use the term 'expression' and the notation e for both expressions and applications.

The notation $[\vec{x} \mapsto \vec{\ell}]$ represents an environment that maps variables x_i to locations ℓ_i and $\mathrm{H}[\ell := d]$ indicates the updating of a heap H at ℓ with d. $\rho \oplus \rho'$ represents the environment ρ shadowed by ρ' and $\lfloor\rho\rfloor_X$ represents the environment restricted to the variables in X. Finally $FV(s)$ represents the free variables in the application s and $addr(c)$ gives the address of the closure c in the heap.

The small-step semantics is shown in Figure **??**. As a sample, consider the three rules for $(\textbf{car}\; x)$. If x is a closure, it is evaluated to WHNF, say (d_1, d_2). This is given by the rule CAR-CLO. If d_1 is not in WHNF, it is also evaluated (CAR-1-CLO). The address $(addr(d_1))$ to be updated with the evaluated value is recorded in a continuation frame. This is required for lazy evaluation, else, in the presence of sharing, d_1 may be evaluated more than once [?]. Only after this is the actual selection done (CAR-SELECT).

3. Liveness

A variable is *live* if there is a possibility of its value being used in future computations and dead if it is definitely not used. Heap-allocated data needs a richer model of liveness which talks about liveness of references. Using $\mathbf{0}$, $\mathbf{1}$ to represent access using **car** and **cdr** fields, the liveness of the structure reachable from a variable can be represented by a set of *access paths* i.e. prefix-closed set of strings from $\{\mathbf{0},\mathbf{1}\}^*$. As an example, if x is a list with liveness $\{\varepsilon, \mathbf{1}, \mathbf{10}, \mathbf{11}, \mathbf{110}\}$, then future computations can only refer up to the second and third members of x. A *liveness environment* is a mapping from variables to subsets of $\{\mathbf{0},\mathbf{1}\}^*$, but often expressed as a set, for example by writing $\{x.\varepsilon, x.\mathbf{1}, x.\mathbf{11}, y.\varepsilon\}$ instead of $[x \mapsto \{\varepsilon, \mathbf{1}, \mathbf{11}\}, y \mapsto \{\varepsilon\}, z \mapsto \{\}]$. In this notation, $x \mapsto \{\varepsilon\}$ represents access using x itself and $x \mapsto \{\}$ indicates that x is dead. We associate liveness environments with program points. The liveness at $\pi : e$ is the liveness just before executing e.

A notion related to liveness is *demand*. While liveness represents the future use of variables, a demand represents the future use of the value of an expression. The demand on an expression e is again a set of access paths—that subset of $\{\mathbf{0},\mathbf{1}\}^*$ which the context of e may explore of e's result. To see the need for demands, consider the let expression $\pi : (\textbf{let}\; x \leftarrow (\textbf{cdr}\; y)\; \textbf{in}\; \pi' : \textbf{return}\; x)$. Assume that the context of this expression produces the liveness

$$
\begin{array}{rcl}
\mathit{ref}(\kappa,\sigma,\mathsf{LF}) & = & \{\},\ \text{for } \kappa \text{ a constant, including } \mathbf{nil} \\
\mathit{ref}((\mathbf{cons}\, x\, y),\sigma,\mathsf{LF}) & = & \{x.\alpha \mid \mathbf{0}\alpha \in \sigma\} \cup \{y.\alpha \mid \mathbf{1}\alpha \in \sigma\} \\
\mathit{ref}((\mathbf{car}\, x),\sigma,\mathsf{LF}) & = & \begin{cases} \{x.\varepsilon\} \cup \{x.\mathbf{0}\alpha \mid \alpha \in \sigma\}, & \text{if } \sigma \neq \emptyset \\ \emptyset & \text{otherwise} \end{cases} \\
\mathit{ref}((\mathbf{cdr}\, x),\sigma,\mathsf{LF}) & = & \begin{cases} \{x.\varepsilon\} \cup \{x.\mathbf{1}\alpha \mid \alpha \in \sigma\}, & \text{if } \sigma \neq \emptyset \\ \emptyset & \text{otherwise} \end{cases} \\
\mathit{ref}((+\, x\, y),\sigma,\mathsf{LF}) & = & \begin{cases} \{x.\varepsilon, y.\varepsilon\}, & \text{if } \sigma \neq \emptyset \\ \emptyset & \text{otherwise} \end{cases} \\
\mathit{ref}((\mathbf{null?}\, x),\sigma,\mathsf{LF}) & = & \begin{cases} \{x.\varepsilon\}, & \text{if } \sigma \neq \emptyset \\ \emptyset & \text{otherwise} \end{cases} \\
\mathit{ref}((f\, \vec{x}),\sigma,\mathsf{LF}) & = & \bigcup_{i=1}^{n} x_i.\mathsf{LF}_f^i(\sigma) \\
\mathcal{L}((\mathbf{return}\ \psi : x),\sigma,\mathsf{LF}) & = & [\mathsf{ep} \mapsto x.\sigma],\ \text{where } \mathsf{ep} \text{ is a new e-path terminating with } \psi \\
\mathcal{L}((\mathbf{if}\ \psi : x\ e_1\ e_2),\sigma,\mathsf{LF}) & = & \begin{cases} \mathcal{L}(e_1,\sigma,\mathsf{LF}) \uplus \mathcal{L}(e_2,\sigma,\mathsf{LF}) \uplus [\{\mathsf{ep} \mapsto \{x.\varepsilon\}\}], & \text{if } \sigma \neq \emptyset \\ \emptyset_{\mathcal{M}} & \text{otherwise} \end{cases} \\
 & & \text{where } \mathsf{ep} \text{ is a new e-path terminating with } \psi \\
\mathcal{L}(\mathbf{let}\ x \leftarrow\ s\ \mathbf{in}\ e),\sigma,\mathsf{LF}) & = & \lambda \mathsf{ep}.\ \mathsf{LF}_f(\mathsf{L}(x)) \cup \mathsf{L}\ \text{ where } \mathcal{M} = \mathcal{L}(e,\sigma,\mathsf{LF}),\ \mathsf{L} = \mathcal{M}(\mathsf{ep}),\ \text{and } f \text{ is the new function} \\
 & & \quad (\mathbf{define}\ (f\, x_1\, x_2 \ldots x_n)\ (\mathbf{if} * x\ s[(f\, x_1\, x_2 \ldots x_n)/x])), \\
 & & \qquad \text{where } x_1 \ldots x_n \text{ are the variables in } s
\end{array}
$$

$$
\frac{\mathcal{L}(e_f,\sigma,\mathsf{LF}) = \bigcup_{i=1}^{n} z_i.\mathsf{LF}_f^i(\sigma) \text{ for each } f \text{ and } \sigma}{\mathit{df}_1 \ldots \mathit{df}_k\ \vdash^{l}\ \mathsf{LF}} \qquad (\textsc{live-define})
$$
$$
\text{where } (\mathbf{define}\ (f\ z_1\ \ldots\ z_n)\ \ e_f) \text{ is a member of } \mathit{df}_1 \ldots \mathit{df}_k
$$

Figure 4. Liveness equations and judgement rule

$[x \mapsto \{\varepsilon, \mathbf{0}\}]$ at π'. Due to the **let** definition which binds x to (**cdr** y), the liveness of x at π' now becomes the demand on (**cdr** y). This, in turn, generates the liveness $\{y.\varepsilon, y.\mathbf{1}, y.\mathbf{10}\}$ at π. These are the set of y-rooted accesses required to explore $\{\varepsilon, \mathbf{0}\}$ paths of the result of (**cdr** y). In strong liveness analysis (dual of classical faint variable analysis [**?**]), y and z are live at the entry $\pi : x := y + z$, if and only if x is live at exit of π. In our terminology, the liveness $\{\varepsilon\}$ of x at the exit from n becomes the demand on $y + z$, and this, in turn generates the liveness $\{y.\varepsilon, z.\varepsilon\}$ at the entry of n.

We follow the notations appearing in [**?**]. In particular, We use σ to range over demands, α to range over access paths and L to range over liveness environments. The notation $\sigma_1\sigma_2$ denotes the set $\{\alpha_1\alpha_2 \mid \alpha_1 \in \sigma_1, \alpha_2 \in \sigma_2\}$. Often we shall abuse notation to juxtapose an edge label and a set of access paths: $\mathbf{0}\sigma$ is a shorthand for $\{\mathbf{0}\}\sigma$.

3.1 Liveness Analysis for lazy languages

Figure **??** describes our analysis which has two parts. The function *ref*, takes an application s and a demand σ, and returns the incremental liveness generated for the free variables of s due to the application. The function $\mathcal{L}$ uses *ref* to propagate liveness across expressions.

Since in a lazy language, an expression is not evaluated unless required. The null demand ($\emptyset$) does not generate liveness in any of the rules defining *ref* or $\mathcal{L}$. A non-null demand of σ on (**cdr** x), is transformed to the liveness $\{x.\varepsilon, x.\mathbf{1}\sigma\}$. In an opposite sense, the demand of $\mathbf{1}\sigma$ on (**cons** y z) is transformed to the demand σ on z. Since **cons** does not dereference its arguments, there is no ε demand on y and z. The rules for $(+\ \mathrm{x}\ \mathrm{y})$ and (**null**? x) are similar. Constants do not generate any liveness.

For the case of a function call we use a third parameter LF that represents the summaries of all functions in the program. LF_f (the component of LF for a specific function f), expresses how the demand σ on a call to f is transformed into the liveness of its parameters at the beginning of the call. LF is determined by the judgement $\mathit{Prog}\ \vdash^{l}\ \mathsf{LF}$ using inference rule (LIVE-DEFINE). This rule describes the fixed-point property to be satisfied by LF, namely, the demand transformation assumed for each function in the program should be the same as the demand transformation calculated from its body. As we shall see in Section **??**, we convert the rule into a grammar and the the language generated by this grammar is the least solution satisfying the rule. We prefer the least solution since it ensures the safe collection of the greatest amount of garbage.

We next describe the function $\mathcal{L}$ that propagates liveness across expressions. Note that, unlike eager languages, the syntactic occurrence of an expression in a program denotes the point of creation of its closure, not necessarily its evaluation. To see the consequences of this, consider Figure **??** which shows an analysis of the body of a function **length** with a demand σ. Clearly, this is also the demand on the return value z giving the liveness at the program point π_8 as $\{\mathrm{z}.\sigma\}$. The liveness at π_8 is propagated backwards as in traditional liveness analysis.

Since z is $(1+\mathrm{y})$, the liveness of σ for z translates into a demand of σ on $(1+\mathrm{y})$, which in turn generates a liveness of $\{\mathrm{y}.\varepsilon\}$ at π_7. We now make the following observations:

1. While the liveness of y should be killed at π_6 since it is beyond the scope of the definition of y, we do not do so. However, this does not cause any imprecision in the way of the GC marking more cells as live, since, during execution, y is yet to come into existence at π_6, and its liveness will not be considered by the garbage collector at this point.
2. In an eager language, the heap location referred by y would not have been live at π_8, a program point beyond its last use. In a lazy language, however, the definition $\mathrm{z} \leftarrow (1+\mathrm{y})$, does not result in an evaluation of $(1+\mathrm{y})$; instead a closure is created. The evaluation actually takes place while evaluating (**return** z).

We therefore record the liveness of y at the program point π_8, indeed at every program point from π_8 up to the beginning of the program π_1 (since we do not kill liveness).

We thus regard the body of the function as an expression tree and consider paths from the root of this tree to the nodes corresponding to the program points which trigger evaluation[2]—**if** conditions and **return** expressions. We call such paths *e-paths*. For example, for the function **length**, there are three such paths $\mathsf{ep}_1 = \langle \pi_1, \pi_2, \psi_1 \rangle$, $\mathsf{ep}_2 = \langle \pi_1, \pi_2, \pi_3, \pi_4, \psi_2 \rangle$ and $\mathsf{ep}_3 = \langle \pi_1, \pi_2, \pi_5, \pi_6, \pi_7, \pi_8, \psi_3 \rangle$.

Given a function f and a demand σ, the liveness analysis computes two maps: $\mathcal{L}(e_f, \sigma, \mathsf{LF})$ computes a map that takes an e-path in f and returns the liveness environment arising out of all uses of variables along the e-path. A second map $\mathcal{P}_f$ maps any program point in f to the set of e-paths sharing the point. If a program point happens to fall on more than one such e-path, then the liveness environment of the program point is the variable-wise union of the liveness environments of the individual e-paths. In the example program, since $\mathcal{P}_{\mathbf{length}}(\pi_1) = \{\mathsf{ep}_1, \mathsf{ep}_2, \mathsf{ep}_3\}$, the liveness at π_1 is $\mathcal{M}(\mathsf{ep}_1) \cup \mathcal{M}(\mathsf{ep}_2) \cup \mathcal{M}(\mathsf{ep}_3)$, where $\mathcal{M} = \mathcal{L}(e_f, \sigma, \mathsf{LF})$. In the sequel we shall only define $\mathcal{L}$. The formulation of $\mathcal{P}$ is easy and we do not elaborate it any further.

Consider the $\mathcal{L}$-rules for **let**, **if**, and **return**. The expression (**return** $\psi : x$) with a demand σ forces an evaluation of x and hence a new map $[\mathsf{ep} \mapsto \{x.\sigma\}]$ needs to be created, where ep is a fresh evaluation path terminating in ψ. The map for the expression (**if** $\psi : x\ e_1\ e_2$) is a point-wise-union of the maps of e_1 and e_2. In addition, since the condition also triggers an evaluation of x, the map $[\mathsf{ep} \mapsto \{x.\varepsilon\}]$ is created and added to the union. Also notice a consequence of laziness: the entire expression including the condition is not evaluated if the demand on it is $\emptyset$. This results in the empty map, which is denoted by $\emptyset_{\mathcal{M}}$.

The liveness of **let** has an elegant formulation. Recollect that our language permits lazy **let**s whose definitions can be recursive, i.e. in the expression **let** $x \leftarrow\ s$ **in** e, x can occur in s. We model the definition $x \leftarrow\ s$ as a possibly recursive function (named f in the rule) which captures an arbitrary number of "unrollings" of the definition[3]. We pass to the demand transformer LF_f of this function the demand arising out of the liveness of x at the beginning of e. This gives the liveness of the variables of s.

Section **??** shows how the demand transformers LF for a program (representing a fully context-sensitive analysis) can be safely approximated by a *procedure summary* for each function. The summary is in the form of a demand transformer that maps a demand on a call to the function to demands on each of its arguments.

3.2 Correctness of liveness analysis

We shall now outline a proof of correctness of the liveness analysis presented in Section **??**. This proof technique is similar to the technique used in [**?**].

The idea behind the proof is as follows: We modify the standard semantics to a semantics called *minefield semantics* that models a liveness based garbage collection at each transition. Before each transition, we start from the root-set and insert a special value $\perp$ at each location in the heap that contains a reference and is reachable but not live. The proof is based on a context-sensitive form of liveness of which our calculated liveness is an over-approximation. The state of the transition system is augmented to carry enough information to calculate the liveness environment at the current evaluation context and each of the suspended evaluation contexts on the stack. To simulate garbage collection, before each transition, we start from the root-set and insert a special value $\perp$ at each location in the heap that contains a reference and is reachable but not live by our analysis. Any attempt to dereference such locations during the transition will result in entering a special state denoted BANG. The main proof is to show that no program enters the BANG state in the minefield semantics.

[2] Program points which trigger evaluation are labeled with ψ.

[3] The $*$ in the definition of f represents non-deterministic choice. During liveness analysis it can be assumed to generate no liveness.

1. Do you need to carry the function body?
2. Minefield doesn't insert bottom inside closures. Separate rule to be inserted.
3. Examine Proposition 4.1, See that different cases are disjoint and together cover all possible cases
4. Suppose the statement that we want to prove is this: Every step (step = GC + $\rightsquigarrow$) goes without a bang. How to divide a (n+1)-step transition into 1 and n. What is the exact invariant required before each step.

4. Minefield semantics for the language

To set up the minefield semantics, we follow these steps:

1. We enrich the abstract machine state ρ, S, H, e to ρ, S, H, e, σ, e_f, where σ is the demand on the expression e, and e_f is the body of the function. As the section on liveness analysis shows, the entire function body e_f and σ are needed to calculate the liveness environment at different program points. The demand sigma is "dynamic" in the sense that it arises from the actual function calls made during execution. The 0-CFA demand is an over-approximation of this demand. The information in a suspended evaluation context is also similarly augmented with the demand on it and the function body of which the suspended evaluation is a part. Thus a stacked entry now takes the form $(\rho, x, e, \sigma, e_f)$. A modification of the small step semantics which carries the extra information is shown in Figure **??**.
2. Given the current evaluation context $\rho, S, H, e, \sigma, e_f$ and $\mathcal{LF}$, we can construct a liveness environment as follows:
   (a) If e is an expression, then:
   $$\mathsf{L} = \bigcup_{\psi \in \mathcal{P}(e)} \mathcal{M}(\psi), \text{where } \mathcal{M} = \mathcal{L}(e_f, \sigma, \mathsf{LF})$$
   (b) If e is an application, then:
   $$\mathsf{L} = \mathit{ref}(e, \sigma, \mathsf{LF})$$
   We can define in the same way the liveness environment for each of the suspended evaluation contexts in S giving a stack of liveness environments that we shall denote SL.
3. *GC* models a liveness based garbage collection: $GC(\rho, S, H, \mathcal{L}, \mathsf{SL}) = (\rho', S', H')$
   (a) For each $x \in \mathit{domain}(\rho)$, $\rho'(x) = \perp$ iff $x.\varepsilon \notin \mathsf{L}$.
   (b) For each stack entry $(\rho, \_, \_, \_, \_)$ in S with L' as the corresponding liveness environment in SL, and for each $x \in \mathit{domain}(\rho)$, $\rho'(x) = \perp$ iff $x.\varepsilon \notin \mathsf{L}'$.
   (c) For each location ℓ, $H'(\ell) = \perp$, iff there is no x in either the current environment or one of the stacked environments ρ such that for some α, $H[x.\alpha] = \ell$ and $x.\alpha \in \mathsf{L}$, .

Starting from the initial state $([\,]_\rho, ([\,]_\rho, \texttt{answer}, (\textbf{print}\ \texttt{answer})) : [\,]_S, [\,]_H, (\textbf{main})$, every transition is preceded by a garbage collection using *GC*. We consider a garbage collection followed by a transition as a step. We show by induction that, for programs without function calls, starting from the initial state that has $e_{\mathbf{main}}$ as the first expression to be evaluated instead of **main**, any transition of n steps occurs without a BANG.

Premise	Transition	Rule name
	$\rho, (\rho', x, e, \sigma', e'_f) : S, H, \kappa, \sigma, e_f$ $\rightsquigarrow \rho', S, H[\rho'(x) := \kappa], e, \sigma', e'_f$	CONST
	$\rho, (\rho', z, e, \sigma', e'_f) : S, H, (\mathbf{cons}\ x\ y), \sigma, e_f$ $\rightsquigarrow \rho', S, H[\rho'(z) := (\rho(x), \rho(y))], e, \sigma', e'_f$	CONS
$H(\rho(x))$ is (l_1, l_2)	$\rho, (\rho', z, e, \sigma', e'_f) : S, H, (\mathbf{car}\ x), \sigma, e_f$ $\rightsquigarrow \rho', S, H[\rho'(z) := H(l_1)], e, \sigma', e'_f$	CAR-WHNF
$H(\rho(x))$ is $\langle s, \rho' \rangle$	$\rho, S, H, (\mathbf{car}\ x), \sigma, e_f$ $\rightsquigarrow \rho', (\rho, x, (\mathbf{car}\ x), \sigma, e_f) : S, H, s, (\mathbf{2} \cup \mathbf{0})\sigma, \_$	CAR-CLO
$H(\rho(x)), H(\rho(y)) \in \mathbb{N}$	$\rho, (\rho', z, e, \sigma', e'_f) : S, H, (+\ x\ y), \sigma, e_f$ $\rightsquigarrow \rho', S, H[\rho'(z) := H(\rho'(x)) + H(\rho'(y))], e, \sigma', e'_f$	PRIM-WHNF
$H(\rho(x))$ is $\langle s, \rho' \rangle$	$\rho, S, H, (+\ x\ y), \sigma, e_f$ $\rightsquigarrow \rho', (\rho, x, (+\ x\ y), \sigma, e_f) : S, H, s, \mathbf{2}\sigma, \_$	PRIM-1-CLO
$H(\rho(y))$ is $\langle s, \rho' \rangle$	$\rho, S, H, (+\ x\ y), \sigma, e_f$ $\rightsquigarrow \rho', (\rho, y, (+\ x\ y), \sigma, e_f) : S, H, s, \mathbf{2}\sigma, \_$	PRIM-2-CLO
g defined as $(\mathbf{define}\ (g\ \vec{y})\ e_g)$	$\rho, S, H, (g\ \vec{x}), \sigma, e_f$ $\rightsquigarrow [\vec{y} \mapsto \rho(\vec{x})], S, H, e_g, \sigma, e_g$	FUNCALL
ℓ is a new location	$\rho, S, H, (\mathbf{let}\ x \leftarrow s\ \mathbf{in}\ e), \sigma, e_f$ $\rightsquigarrow \rho \oplus [x \mapsto \ell], S, H[\ell \mapsto \langle s, \lfloor \rho \rfloor_{FV(s)} \oplus [x \mapsto \ell] \rangle], e, \sigma, e_f$	LET
$H(\rho(x)) \neq 0$	$\rho, S, H, (\pi : \mathbf{if}\ \psi : x\ e_1\ e_2), \sigma, e_f$ $\rightsquigarrow \rho, S, H, e_1, \sigma, e_f$	IF-TRUE
$H(\rho(x)) = 0$	$\rho, S, H, (\pi : \mathbf{if}\ \psi : x\ e_1\ e_2), \sigma, e_f$ $\rightsquigarrow \rho, S, H, e_2, \sigma, e_f$	IF-FALSE
$H(\rho(x)) = \langle s, \rho' \rangle$	$\rho, S, H, (\pi : \mathbf{if}\ \psi : x\ e_1\ e_2), \sigma, e_f$ $\rightsquigarrow \rho', (\rho, x, (\mathbf{if}\ x\ e_1\ e_2), \sigma, e_f) : S, H, s, \mathbf{2}\sigma, e_f$	IF-CLO
$H(\rho(x))$ is whnf with value v	$\rho, (\rho', z, e, \sigma', e'_f) : S, H, (\mathbf{return}\ x), \sigma, e_f$ $\rightsquigarrow \rho', S, H[\rho'(z) := v], e, \sigma', e'_f$	RETURN-WHNF
$H(\rho(x)) = \langle s, \rho' \rangle$	$\rho, S, H, (\psi : \mathbf{return}\ x), \sigma, e_f$ $\rightsquigarrow \rho',\ (\rho, x, (\mathbf{return}\ x), \sigma, e_f) : S, H, s, \sigma, e_f$	RETURN-CLO

Figure 5. Minefield semantics.

Observe that the transitions that dereference and therefore can potentially lead to BANG are those which have a $H(\ldots)$ term in the transition rule. Of these, terms of the form $H(\ldots)$ dereference a root variable, and the term $H(l)$ dereferences a heap location. In both the cases we show that the dereferencing cannot lead to a BANG state in programs without function calls.

We need the following proposition that connects liveness analysis with minefield semantics transitions.

PROPOSITION 4.1. *1. Consider a* **let** *definition:* π: (**let** $x \leftarrow\ \ldots$ **in** e)*, such that there is a use of x in e, i.e. an occurrence of* **let** $y \leftarrow s$ *in…, where s is* (**car** x)*,* (**cdr** x) *or* $(+\ x\ y)$*, or* (**if** $x\ e1\ e2$) *or* (**return** x)*. Then the demand on x at* π *is non-null.*

2. Assume that a **let** *definition* π : (**let** $z \leftarrow$ (**cons** $x\ y$) **in** e) *has an occurrence of* (**car** z) *in e. Then the liveness of x at* π *includes* ε*. Similarly if e contains* (**cdr** z)*, then the liveness of y at* π *includes* ε*.*

In the second case for example, the liveness of x due to the **cons** is $\bar{\mathbf{0}}\sigma$, where σ is the demand on z. Since σ includes the demand $\mathbf{0}\alpha$ due to its use in (**car** z), the liveness on x includes ε. We thus have the following lemma:

LEMMA 4.2. *A program without function calls cannot enter the* BANG *state in the minefield semantics.*

Proof For a program executing according to *minefield semantics*, if a BANG occurs then some variable which is dereferenced must be set to $\perp$, between its creation and use points. This can only happen if the liveness of the variable being dereferenced was null. From the Proposition **??** it is clear that this cannot happen. ☐

For a program with functions, we use the techniques in [**?**]. We assume as a contradiction that the program can enter a BANG state and inline the functions called to obtain a program that is without functions but has the same execution behavior and demands as the original program. We then use Lemma **??** to show that the resulting functionless program cannot enter a bang state.

5. Computing liveness and its encoding as DFA

Section **??** gave a context-sensitive liveness analysis and proved it correct with reference to a *minefield* semantics. In our analysis, $\mathcal{L}$ and $\mathcal{T}_f$ together described a liveness set for each program point in a function in terms of a given σ and LF. However, we still have to describe how to obtain demand transformers LF from the rule LIVE-DEFINE and how to compute the specific demand σ on each function. To answer these questions, it is convenient to cast the equations arising from *ref* and $\mathcal{L}$ as the rules of a grammar. To do so, we need to modify the rules themselves to a different form.

5.1 Modifying liveness rules

The *ref* rule for **cons**, shown in Figure **??**, requires us to remove the leading **0** and **1** from the access paths in σ. Similarly, the rules for **car**, **cdr**, $+$, **null**?, and **if** require us to return $\emptyset$, if σ itself is $\emptyset$ and $\{\varepsilon\}$ otherwise. To realize these rules σ needs to be known. This creates difficulties since we want to solve the equations arising from liveness symbolically.

The solution is to also treat the operations mentioned above symbolically. We introduce three new symbols: $\bar{\mathbf{0}}$, $\bar{\mathbf{1}}$, $\mathbf{2}$. These symbols are defined as a relation $\hookrightarrow$ between sets of access paths:

$$\bar{\mathbf{0}}\sigma \hookrightarrow \sigma' \text{ where } \sigma' = \{\alpha \mid \mathbf{0}\alpha \in \sigma\}$$
$$\bar{\mathbf{1}}\sigma \hookrightarrow \sigma' \text{ where } \sigma' = \{\alpha \mid \mathbf{1}\alpha \in \sigma\}$$

Thus $\bar{\mathbf{0}}$ selects those entries in σ that have leading $\mathbf{0}$, and removes the leading $\mathbf{0}$ from them. The symbol $\mathbf{2}$ reduces the set of strings following it to a set containing only ε. It filters out, however, the empty set of strings.

$$\mathbf{2}\sigma \hookrightarrow \begin{cases} \emptyset & \text{if } \sigma = \emptyset \\ \{\varepsilon\} & \text{otherwise} \end{cases}$$

We can now rewrite the **cons** and the **car** rules of *ref* as:

$$ref((\mathbf{cons}\ x\ y), \sigma, \mathsf{LF}) = x.\bar{\mathbf{0}}\sigma \cup y.\bar{\mathbf{1}}\sigma, \text{ and}$$
$$ref((\mathbf{car}\ x), \sigma, \mathsf{LF}) = x.\mathbf{2}\sigma \cup x.0\sigma$$

and the $\mathcal{L}$ rule for **if** as:

$$\mathcal{L}((\mathbf{if}\ x\ e_1\ e_2), \sigma, \mathsf{LF}) = \mathcal{L}(e_1, \sigma, \mathsf{LF}) \uplus \mathcal{L}(e_2, \sigma, \mathsf{LF}) \uplus [\mathsf{ep} \mapsto x.\mathbf{2}\sigma]$$

The rules for **cdr**, $+$ and **null**? are also modified in a manner similar to **car**.

When there are multiple occurrences of $\bar{\mathbf{0}}$, $\bar{\mathbf{1}}$ and $\mathbf{2}$, $\hookrightarrow$ is applied from right to left. The reflexive transitive closure of $\hookrightarrow$ will be denoted as $\overset{*}{\hookrightarrow}$. The following proposition relates the original and the modified liveness rules.

PROPOSITION 5.1. *Assume that a liveness computation based on the original set of rules gives the liveness of the variable x at a program point π_i as σ (symbolically, $\mathsf{L}_i^x = \sigma$). Further, let $\mathsf{L}_i^x = \sigma'$ when the modified rules are used instead of $\mathcal{L}$. Then $\sigma' \overset{*}{\hookrightarrow} \sigma$.*

For an explanation of why the proposition holds for the modified **cons** rule, we refer the reader to [?]. The proposition also holds for other modified rules for similar reasons.

5.2 Generating Liveness Equations

Given a function f, we now describe how to generate equations for the demand transformation LF_f. The program in Figure **??** serves as a running example. Starting with a symbolic demand σ, we determine $\mathcal{L}(e_f, \sigma, \mathsf{LF})$. In particular, we consider $\mathsf{L}_1^{x_i}$, the liveness of the i^{th} parameter x_i at the program point at the beginning of e_f (assumed to be π_1). By the rule LIVE-DEFINE, this should be the same as $\mathsf{LF}_f^i(\sigma)$. Applying this to **length**, we have:

$$\mathsf{L}_1^{\mathsf{l}} = \mathbf{2}\sigma \cup \mathbf{1}\mathsf{LF}^1_{\mathbf{length}}(\mathbf{2}\sigma) \cup \mathbf{2}\mathsf{LF}^1_{\mathbf{length}}(\mathbf{2}\sigma)$$

and the only equation defining $\mathsf{LF}_{\mathbf{length}}$ is:

$$\mathsf{LF}^1_{\mathbf{length}}(\sigma) = \mathbf{2}\sigma \cup \mathbf{1}\mathsf{LF}^1_{\mathbf{length}}(\mathbf{2}\sigma) \cup \mathbf{2}\mathsf{LF}^1_{\mathbf{length}}(\mathbf{2}\sigma)$$

In general, the equations for LF are recursive since, as in the case of **length**, L_1 may have been defined in terms of LF_f. However, it is desirable to have a closed form solution for LF_f. As mentioned in [?], each of the liveness rules modifies a demand by prefixing it with symbols in the alphabet $\{\mathbf{0}, \mathbf{1}, \bar{\mathbf{0}}, \bar{\mathbf{1}}, \mathbf{2}\}$, and therefore we can assume that $\mathsf{LF}_f^i(\sigma)$ has the closed form:

$$\mathsf{LF}_f^i(\sigma) = \mathsf{D}_f^i\sigma \tag{1}$$

where D_f^i are sets of strings over the alphabet mentioned above. Substituting the guessed form in the equation describing LF_f, and factoring out σ, we get an equation for D_f^i that is independent of σ. Any solution for D_f^i yields a solution for LF_f. Applied to $\mathsf{LF}_{\mathbf{length}}$, we get:

$$\mathsf{LF}^1_{\mathbf{length}}(\sigma) = \mathsf{D}^1_{\mathbf{length}}\sigma, \text{ and}$$
$$\mathsf{D}^1_{\mathbf{length}} = \mathbf{2} \cup \mathbf{1}\mathsf{D}^1_{\mathbf{length}}\mathbf{2} \cup \mathbf{2}\mathsf{D}^1_{\mathbf{length}}\mathbf{2}$$

Note that this equation can also be viewed as a CFG with $\{\mathbf{1}, \mathbf{2}\}$ as terminal symbols and $\mathsf{D}^1_{\mathbf{length}}$ as the sole non-terminal.

5.3 Generating liveness equations L for function bodies

To avoid analyzing the body of a function for each call, we calculated the liveness for the arguments and the variables in a function with respect to a symbolic demand σ. To get the actual liveness we calculate an over-approximation of the actual demands made by all the calls and calculate the liveness at each GC point inside the function based on this approximation. The 0-CFA-style *summary demand* is calculated by taking a union of the demands at every call site of a function.

Consider a function g containing a call to f at a site π, say π: (**let** $x \leftarrow (f\, y_1 \ldots y_n)$ **in** $\pi_i : e$). Let the demand on g be σ_g and, based on this demand, the liveness of x at π_i be L_i^x. By the **let** rule of Figure **??**, the call at π contributes L_i^x to the demand σ_f. Let us denote the contribution of a call site π in a function g to the overall demand on the function f as $\delta_f(\pi, g)$. Assuming that there are k call sites to function f, π^1 (in function g^1) $\ldots \pi^k$ (in function g^k), the over-approximation of σ_f is $\delta_f(\pi^1, g^1) \cup \cdots \cup \delta_f(\pi^k, g^k)$. The distinguished function **main** is a special case. We assume it is called through a printing mechanism with demand $\sigma_{\mathbf{main}} = \{\mathbf{0}, \mathbf{1}\}^*$ (denoted σ_{all}) if **main** returns a structure and ε if it returns a base value.

For the running example, **length** has calls from **main** at π_{12} and a recursive call at π_6. So $\sigma_{\mathbf{length}} = \delta_{\mathbf{length}}(\pi_{12}, \mathbf{main}) \cup \delta_{\mathbf{length}}(\pi_6, \mathbf{length})$. Filling in the values gives:

$$\sigma_{\mathbf{length}} \quad = \quad \{\varepsilon\} \cup \mathbf{2}\sigma_{\mathbf{length}}$$

As examples, the liveness of $\mathsf{L}_1^{\mathsf{l}}$ and $\mathsf{L}_9^{\mathsf{a}}$ in terms of $\sigma_{\mathbf{length}}$ are:

$$\mathsf{L}_1^{\mathsf{l}} = (\mathbf{2} \cup \mathbf{1}\mathsf{D}^1_{\mathbf{length}}\mathbf{2} \cup \mathbf{2}\mathsf{D}^1_{\mathbf{length}}\mathbf{2})\sigma_{\mathbf{length}}$$
$$\mathsf{L}_9^{\mathsf{a}} = \mathbf{2}\bar{\mathbf{0}}\mathsf{D}^1_{\mathbf{length}}\{\varepsilon\}$$

In summary, the equations generated during liveness analysis are:

1. For each function f, equations defining D_f^i for use by LF_f.
2. For each function f, an equation defining the summary demand σ_f on e_f.
3. For each function f (including **main** for $e_{\mathbf{main}}$) an equation defining liveness at each GC point of e_f.

5.4 Solving liveness equations—the grammar interpretation

The equations above can now be re-interpreted as a context-free grammar (CFG) on the alphabet $\{\mathbf{0}, \mathbf{1}, \bar{\mathbf{0}}, \bar{\mathbf{1}}, \mathbf{2}\}$. Let $\langle X\rangle$ denote the non-terminal for a variable X occurring on the LHS of the equations generated from the analysis. We can think of the resulting productions as being associated with several grammars, one for each non-terminal $\langle \mathsf{L}_i^x\rangle$ regarded as a start symbol. As an example, the grammar for $\langle \mathsf{L}_1^{\mathsf{l}}\rangle$ comprises the following productions:

$$\begin{array}{rcl}
\langle \mathsf{L}_1^{\mathsf{l}}\rangle & \rightarrow & \mathbf{2}\langle\sigma_{\mathbf{length}}\rangle \mid \mathbf{1}\langle\mathsf{D}^1_{\mathbf{length}}\rangle\mathbf{2}\langle\sigma_{\mathbf{length}}\rangle \\
 & & \mid \mathbf{2}\langle\mathsf{D}^1_{\mathbf{length}}\rangle\mathbf{2}\langle\sigma_{\mathbf{length}}\rangle \\
\langle\mathsf{D}^1_{\mathbf{length}}\rangle & \rightarrow & \mathbf{2} \mid \mathbf{1}\langle\mathsf{D}^1_{\mathbf{length}}\rangle\mathbf{2} \mid \mathbf{2}\langle\mathsf{D}^1_{\mathbf{length}}\rangle\mathbf{2} \\
\langle\sigma_{\mathbf{length}}\rangle & \rightarrow & \varepsilon \mid \mathbf{2}\langle\sigma_{\mathbf{length}}\rangle \\
\langle\mathsf{L}_9^{\mathsf{a}}\rangle & \rightarrow & \mathbf{2}\bar{\mathbf{0}}\langle\mathsf{D}^1_{\mathbf{length}}\rangle
\end{array}$$

```
(define (length l)
    π1:(let x ← (null? l) in
        π2:(if ψ1: x
            π3:(let v ← 0 in
                π4: (return ψ2 : v)
            π5:(let u ← (cdr l) in
                π6:(let y ← (length u) in
                    π7:(let z ← (1 + y) in
                        π8: (return ψ3 : z))))))))

(define (main)
    π9: (let a ← ( [a BIG closure] ) in
        π10: (let b ← (+ a 1) in
            π11: (let c ← (cons b nil) in
                π12:(let w ← (length c) in
                    π13: (return ψ4 : w)))))
```
(a)

$$
\begin{aligned}
\mathcal{L}(e_{\mathbf{length}},\sigma,\mathsf{LF}) &= [\,\mathsf{ep}_1 \mapsto \{\mathtt{x}.\mathbf{2}\sigma,\mathtt{l}.\mathbf{2}\sigma\},\ \mathsf{ep}_2 \mapsto \{\mathtt{v}.\sigma\},\\
&\quad \mathsf{ep}_3 \mapsto \{\mathtt{z}.\sigma,\mathtt{y}.\mathbf{2}\sigma,\mathtt{u}.\mathsf{LF}^{1}_{\mathbf{length}}(\mathbf{2}\sigma),\\
&\qquad \mathtt{l}.\{\mathbf{1}\mathsf{LF}^{1}_{\mathbf{length}}(\mathbf{2}\sigma)\cup\mathbf{2}\mathsf{LF}^{1}_{\mathbf{length}}(\mathbf{2}\sigma)\}\}]\\
\mathcal{P}_{\mathbf{length}} &= [\,\pi_1 \mapsto \{\mathsf{ep}_1,\mathsf{ep}_2,\mathsf{ep}_3\},\ \pi_2 \mapsto \{\mathsf{ep}_1,\mathsf{ep}_2,\mathsf{ep}_3\},\\
&\quad \pi_3 \mapsto \{\mathsf{ep}_2\},\pi_4 \mapsto \{\mathsf{ep}_2\},\pi_5 \mapsto \{\mathsf{ep}_3\},\\
&\quad \pi_6 \mapsto \{\mathsf{ep}_3\},\pi_7 \mapsto \{\mathsf{ep}_3\},\pi_8 \mapsto \{\mathsf{ep}_3\}\,]\\
\mathcal{L}(e_{\mathbf{main}},\sigma,\mathsf{LF}) &= [\,\mathsf{ep}_4 \mapsto \{\mathtt{w}.\sigma,\mathtt{c}.\mathsf{LF}^{1}_{\mathbf{length}}(\sigma),\\
&\qquad \mathtt{b}.\bar{\mathbf{0}}\mathsf{LF}^{1}_{\mathbf{length}}(\sigma),\mathtt{a}.\mathbf{2}\bar{\mathbf{0}}\mathsf{LF}^{1}_{\mathbf{length}}(\sigma)\}]\\
\mathcal{P}_{\mathbf{main}} &= [\,\pi_9 \mapsto \{\mathsf{ep}_4\},\ \pi_{10} \mapsto \{\mathsf{ep}_4\},\ \pi_{11} \mapsto \{\mathsf{ep}_4\},\\
&\quad \pi_{12} \mapsto \{\mathsf{ep}_4\},\pi_{13} \mapsto \{\mathsf{ep}_4\}\,]
\end{aligned}
$$
(b)

Figure 6. (a) Example program and (b) its liveness maps.

Other equations can be converted similarly. The language generated by $\langle \mathsf{L}_i^x \rangle$, denoted $\mathscr{L}(\langle \mathsf{L}_i^x \rangle)$, is the desired solution of L_i^x. However, note that the strings in the language are over alphabet $\{\mathbf{0},\mathbf{1},\bar{\mathbf{0}},\bar{\mathbf{1}},\mathbf{2}\}$, but we want *forward* access paths only, i. e., access paths over the alphabet $\{\mathbf{0},\mathbf{1}\}$. In other words, the decision problem that we are interested in during garbage collection at a program point π_i is:

> Let $x.\alpha$ be a forward access path consisting only of edges $\mathbf{0}$ and $\mathbf{1}$ (but not $\bar{\mathbf{0}}$, $\bar{\mathbf{1}}$ or $\mathbf{2}$). Let $\mathscr{L}(\langle \mathsf{L}_i^x \rangle) \overset{*}{\hookrightarrow} \sigma$, where σ consists of forward access paths only. Then does $\alpha \in \sigma$?

We now model the above problem as one of deciding the membership of a context-free grammar augmented with a fixed set of unrestricted productions.

DEFINITION 5.2. *Consider the grammar $(N,T,p_1 \cup p_2,S)$ in which N is a set of non-terminals, $T = \{\mathbf{0},\mathbf{1},\bar{\mathbf{0}},\bar{\mathbf{1}},\mathbf{2},\$\}$, p_1 is a set of context-free productions that contains the distinguished production $S \to \alpha\$$, α is a string of grammar symbols that does not contain S, and p_2 is the fixed set of unrestricted productions $\bar{\mathbf{0}}\mathbf{0} \to \varepsilon$, $\bar{\mathbf{1}}\mathbf{1} \to \varepsilon$, $\mathbf{2}\mathbf{0} \to \mathbf{2}$, $\mathbf{2}\mathbf{1} \to \mathbf{2}$, and $\mathbf{2}\$ \to \varepsilon$.*

From Sections **??** and **??**, it is clear that the results of liveness analysis of any program can be modeled by the kind of grammar described above. The following proposition shows that the converse also holds.

PROPOSITION 5.3. *Given a grammar G of the form described in Definition **??**, it is possible to construct a program p with program points π_i and variables x such that the liveness analysis of p is same as G except for a change in non-terminal names which now have the form $\langle \mathsf{L}_i^x \rangle$.*

We now show that the decision problem that is required to be answered at garbage collection time is undecidable.

LEMMA 5.4. *Consider a grammar G of the kind described in Definition **??** and a forward access path α consisting of symbols $\mathbf{0}$ and $\mathbf{1}$ only. The decision problem $\alpha \in \mathscr{L}(G)$ is undecidable.*

Proof Given a Turing machine and an input $w \in (1+0)^*$, we construct a grammar G such that the machine will halt on w if and only if $\varepsilon \in \mathscr{L}(G)$. The grammar includes the fixed set of unrestricted productions in Definition **??**.

We shall denote the configuration of the Turing machine as $w_l(\mathsf{S},c)w_r$, where w_l is the string to the left of the head, w_r is the string to the right, c is the symbol under the head and S is the current state of the machine. For each combination of state and symbol (S,c), the grammar will contain the non-terminal S^c. We shall synchronize each move of the machine to a derivation step using a context free production, followed, if possible, by a derivation step using either $\bar{\mathbf{0}}\mathbf{0} \to \varepsilon$ or $\bar{\mathbf{1}}\mathbf{1} \to \varepsilon$. After each synchronization, we shall establish the following invariant relation between the machine configuration and the sentential form:

> If the configuration of the machine is $w_l(\mathsf{S},c)w_r$, then the sentential form will be $\overline{w_l}\mathsf{S}^c w_r$, where $\overline{w_l}$ is the same as w_l but with each symbol d in w_l replaced by $\overline{d}$.

Assume that the TM starts in a state S_{init} with a tape cw and the head positioned on the symbol c. Then the sentential form corresponding to the initial configuration is $\mathsf{S}^c_{init}w$ (we can assume that there is a production $\mathsf{S} \to \mathsf{S}^c_{init}w$, where S is the start symbol of the grammar). Further correspondences between the Turing machine moves and the grammar productions are as follows:

1. For each transition $(S_i,c) \to (S_j,c',L)$, there are two productions $\mathsf{S}^c_i \to \mathbf{0}\mathsf{S}^{\mathbf{0}}_j c'$ and $\mathsf{S}^c_i \to \mathbf{1}\mathsf{S}^{\mathbf{1}}_j c'$.
2. For each transition $(S_i,c) \to (S_j,c',R)$, there are two productions $\mathsf{S}^c_i \to c\mathsf{S}^{\mathbf{0}}_j\bar{\mathbf{0}}$ and $\mathsf{S}^c_i \to c\mathsf{S}^{\mathbf{1}}_j\bar{\mathbf{1}}$.

The idea behind the productions is explained with an example: Assume that the current sentential form is $\bar{\mathbf{0}}\bar{\mathbf{1}}\mathsf{S}^{\mathbf{0}}_i\mathbf{00}$. Assume that the machine has a transition $(S_i,\mathbf{0}) \to (S_j,\mathbf{1},L)$. Since the next corresponding step in the derivation has to be done without any prior knowledge of whether the symbol to the left of the tape is a $\mathbf{0}$ or a $\mathbf{1}$, two productions are provided, and the invariant will be maintained only if the production $\mathsf{S}^{\mathbf{0}}_i \to \mathbf{1}\mathsf{S}^{\mathbf{1}}_j\mathbf{1}$ is chosen for the next step in the derivation. This gives the configuration $\bar{\mathbf{0}}\bar{\mathbf{1}}\mathbf{1}\mathsf{S}^{\mathbf{1}}_i\mathbf{100}$. Simplification with the production $\bar{\mathbf{1}}\mathbf{1} \to \varepsilon$ yields $\bar{\mathbf{0}}\mathsf{S}^{\mathbf{1}}_i\mathbf{100}$, which exactly corresponds to the changed configuration of the machine. Notice carefully that a wrong choice breaks the invariant and it cannot be recovered subsequently by any choice of productions.

After the Turing machine has halted, there are further "cleanup" derivations that derive ε only if the invariant has been maintained so far. For every symbol c, we introduce a non-terminal S^c_{final} where S_{final} is the final state of the Turing machine. We add productions $\mathsf{S}^c_{final} \to \mathbf{0}\mathsf{S}^c_{final}$ and $\mathsf{S}^c_{final} \to \mathbf{1}\mathsf{S}^c_{final}$ for cleaning up the $\bar{\mathbf{0}}$ and $\bar{\mathbf{1}}$ symbols on the left of the head and $\mathsf{S}^c_{final} \to \mathsf{S}^c_{final}\bar{\mathbf{0}}$ and $\mathsf{S}^c_{final} \to \mathsf{S}^c_{final}\bar{\mathbf{1}}$ for cleaning up $\mathbf{0}$ and $\mathbf{1}$ on the right of the tape head. This completes the reduction. □

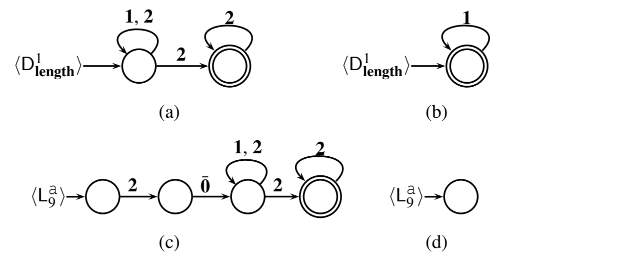


Figure 7. (a) The grammar rules for $\langle \mathsf{D}^1_{\mathbf{length}} \rangle$ converted into an automaton, and (b) its DFA. The same for $\langle \mathsf{L}^{\mathsf{a}}_9 \rangle$ are in (c) and (d).

We circumvent the problem of undecidability by over- approximating the CFG by non-deterministic finite state automata (NFA). The NFAs are then simplified using the $\hookrightarrow$ rules. Finally the simplified NFAs are converted to DFAs.

5.4.1 From CFGs to GC-ready DFA

We use the algorithm by Mohri and Nederhof [?] to approximate a CFG by a *strongly regular* grammar. For example, the grammar fragment for the non-terminal $\langle \mathsf{D}^1_{\mathbf{length}} \rangle$ after the Mohri-Nederhof transformation is:

$$\begin{aligned}\langle \mathsf{D}^1_{\mathbf{length}} \rangle &\rightarrow \mathbf{2}\langle \mathsf{D}^{1'}_{\mathbf{length}} \rangle \mid \mathbf{1}\langle \mathsf{D}^1_{\mathbf{length}} \rangle \mid \mathbf{2}\langle \mathsf{D}^1_{\mathbf{length}} \rangle \\ \langle \mathsf{D}^{1'}_{\mathbf{length}} \rangle &\rightarrow \mathbf{2}\langle \mathsf{D}^{1'}_{\mathbf{length}} \rangle \mid \varepsilon\end{aligned}$$

The strongly regular grammar is converted into a set of NFAs, one for each $\langle \mathsf{L}^x_i \rangle$. The $\hookrightarrow$ simplification is now done on the NFAs by repeatedly introducing ε edges to bypass pairs of consecutive edges labeled $\bar{\mathbf{0}}\mathbf{0}$ or $\bar{\mathbf{1}}\mathbf{1}$ and constructing the ε-closure. Since the ε-closure step may create more adjacent $\bar{\mathbf{0}}\mathbf{0}$ and $\bar{\mathbf{1}}\mathbf{1}$ edges, the simplification is continued till a fixed point is reached, after which the edges labeled $\bar{\mathbf{0}}$ and $\bar{\mathbf{1}}$ are deleted. The details of the algorithm, its correctness and termination proofs are given by Karkare et. al. [? ?]. The resulting automaton has edges labeled with **0**, **1** and **2** only. In this automaton, for every edge labeled **2**, we check if the source node of the edge has a valid path to a final final state. If yes, we mark the mark the source node as final. Finally, we remove all the edges labeled **2** and convert the automaton into a deterministic automaton. This effectively implements the $\hookrightarrow$ simplification rules for $\bar{\mathbf{0}}$, $\bar{\mathbf{1}}$, and **2** to obtain forward paths. While checking for liveness during GC, a forward access-path is valid only if it can reach a final state. Figure **??**(a) shows the NFA that is obtained from the grammar for $\langle \mathsf{D}^1_{\mathbf{length}} \rangle$. The final DFA is shown in Figure **??**(b). This expectedly says that given a demand ε on **length**, the liveness of its argument is $\mathbf{1}^*$ (the spine of the list is traversed). Similarly, Figure **??**(c) shows the NFA for $\langle \mathsf{L}^{\mathsf{a}}_9 \rangle$. The DFA shown in Figure **??**(d) does not accept any forward paths. This reflects the lazy nature of our language. Since **length** does not evaluate the member elements of the argument list, the closure for a is never evaluated and is reclaimed whenever liveness based GC is triggered beyond π_9.

6. The Garbage Collection Scheme

Our experimental setup consists of an interpreter for our language, a liveness analyzer, and a single generation copying collector. The garbage collector can be easily configured to work on the basis of reachability (RGC mode) or use the liveness DFA (LGC mode). While exploring the activation record of a function, LGC uses the DFA at the latest program point traversed in the function's body. Thus, if LGC is invoked at the program point π, it would use the liveness DFA at π itself for exploring the root set of the current activation record. For any other activation record in the stack, say for a function f calling g in the call-chain, the evaluation point in f which resulted in the call to g would be used.

```
procedure lgc ( ) :
    for each reference ref in root set:
        lgcCopy (ref);
    copyReferencesOnPrintStack();
procedure lgcCopy (ref) :
    if ref is found live using dfa:
        newRef = dupHeapCell(ref);
        if cellType(ref) is cons:
            newCar = lgcCopy (ref.car);
            newRef.car = newCar;
            newCdr = lgcCopy (ref.cdr);
            newRef.cdr = newCdr;
        if cellType(ref) is closure:
            newRef.arg1 = lgcCopy (cell.arg1);
            newRef.arg2 = lgcCopy (cell.arg2);
```

Algorithm 1: LGC with closure collection based on liveness.

We shall call a unit allocatable memory as a *cell*. A cell can hold a basic value (*bas*), the constructor **cons** (*cons* arg_1 arg_2) or a closure. The closure, in turn, can be one of (*unop* arg_1), (*binop* arg_1 arg_2) and function application (*app* arg_1 arg_2). Here each arg_i is a reference to another heap cell or terminated through the function *makenull*.

While exploring the heap, the GC may encounter closures. In RGC mode, the closure will be explored as usual and copied by traversing the references corresponding to the arguments. Since our liveness analysis predicts the liveness of data in its evaluated form only, closures present a challenge for LGC. LGC will need to map the liveness of the free variables of the closure to their evaluation point. This is not straight forward as variables may escape their creation scope and need to be handled in an approximate way.

6.1 A liveness-based garbage collection scheme

Algorithm **??** describes a liveness-based garbage collection scheme. Starting with the root set, each live reference is explored using *lgcCopy*. Copying **cons** cells is simple—it just involves copying the cell itself and recursively copying the **car** and the **cdr** fields. On the other hand copying closures requires determining the liveness of its arguments at run-time.

Let us consider determining the liveness of a closure at runtime. Consider a closure $(app\ f\ y)$ has a reference from the root set variable x. If x is live, then we know that the cell $(app\ f\ y)$ has to be copied. To copy the heap structure rooted at y, we need the liveness of y at the program point which triggered the evaluation of $(app\ f\ y)$. However, this is only possible if the activation record of the function which created this closure is still on the activation stack. This does not always happen since a function could pack a closure in a **cons** cell and return the **cons** cell.

An alternative could be to carry with each closure, extra information about the program point which triggered its evaluation. This gives us precise liveness information of the free variables of a closure but involves the overhead of updating the program point information during execution. To avoid this overhead we go for a safe approximation by storing the *creation point of the closure* and using it to check liveness. This is safe since the liveness at the creation point of the closure dominates all possible evaluation points of the closure, but it might be imprecise as the actual liveness might be way less than what is found at the creation point.

```
procedure lgc():
    for each reference ref in root set:
        lgcCopy(ref);
    copyReferencesOnPrintStack();
procedure lgcCopy(ref):
    if cellType(ref) is cons:
        if ref is found live using dfa and has not been copied
        using reachability:
            newRef = dupHeapcell(ref);
            newCar = lgcCopy(ref.car);
            newRef.car = newCar;
            newCdr = lgcCopy(ref.cdr);
            newRef.cdr = newCdr;
        else:
            makenull(ref);
    else:
        if cellType(ref) is closure and has not been copied:
            newRef = rgcCopy(ref);
```

Algorithm 2: LGC with reachability-based closure collection.

Note that when a reachability-based collector visits a cell for the first time, it is marked. Explorations starting from the cell are curtailed if the cell is visited subsequently due to sharing. Since a liveness-based collector may reach the same cell multiple times, each time with a different automaton state, this curtailment cannot be done in the case of LGC. This is an inherent drawback of a liveness-based collector. This can be mitigated by maintaining a list of liveness automata states that a particular heap cell was traversed with and avoid repeated traversals if we reach the cell in one of these states. However this also involves storage and run-time penalty.

To avoid performance penalty, we experimented with a mixed-mode garbage collection scheme which uses liveness for evaluated data and reachability for closures.

6.2 Liveness-based collector with reachability-based closure collection

Table 1. Time taken by LGC for different strategies. The programs are modified (smaller) instances of original benchmarks.

Program	#GC	Closure collected using	
		Reachability	Liveness
nqueens	183	0.009 sec	0.228 sec
lambda	11	0.001 sec	4.992 sec
lcss	1	0.000 sec	34.668 sec

Algorithm **??** describes a garbage collection scheme where the liveness automata is consulted only for **cons** cells; and assumes everything under a closure to be live and copies everything that is reachable.

A consequence of copying everything under a closure is that we might reach a non-live cell that has been reclaimed earlier by liveness based GC. This can happen if a **cons** cell having partially live sub-structure is copied using liveness during a collection and it subsequently becomes part of a live closure. During a subsequent collection, the collector may try to copy everything under the closure including the dead sub-structure of the **cons** cell. To avoid this, we ensure that any non-live references are not carried across GC's. We do this by scanning the live part of the buffer after every GC and nullifying any reference which is not live. This is correct because any reference which is dead during a collection is guaranteed not be dereferenced any further by the program.

Another advantage of this mixed garbage collection scheme is that it avoids the problem of multiple explorations from the same cell. This is done by maintaining a flag in each cell to indicate whether it was copied using reachability.

6.3 Garbage collection for references on print stack

In lazy languages the evaluation of closures is driven by the print function [**?**]. Printing of atomic values is simple and does not trigger a garbage collection. In case of **cons** cells, the **car** is first printed followed by **cdr**. In a lazy language, both **car** and **cdr** could be closures requiring evaluation, and this may trigger a garbage collection. When this happens. we have to consider any references that might be on the print stack and copy them. We extend the liveness analysis to the print function and use its result during garbage collection.

7. Experimental Evaluation

To test the utility of our method we have implemented a garbage-collector for a simple scheme-like first order language having non-strict semantics. It consists of an interpreter, liveness analyzer and a garbage-collector that can optionally use reachability or liveness. Our benchmark consists of programs from nofib [**?**] suite and some external programs manually converted to ANF. All our benchmarks were executed on a machine having 8 core Intel(R) Core(TM) i7-4770 3.40GHz CPU , 8192 KB L2 cache, 16 GB RAM running 64 bit Ubuntu 14.04.

The process of liveness-based garbage collection involves going through each activation record on the stack and exploring each variable in the current activation record (root set). The liveness of each variable is determined using the program point and the variable name. The liveness of all variables are stored as DFA. These DFA are then encoded as a table. A variable is live only if an entry is found in this table. All cells that are live are copied to the live semi-space. In case of **cons** cells the **car** and **cdr** pointers are chased and if they are live the copied **cons** cell will get the updated addresses of its **car** and **cdr** fields.

7.1 Copying closure using liveness vs copying closures using reachability

To compare the effect of using reachability to copy closures instead of liveness we take some representative programs from our benchmark suite and execute them with both. As shown in Table **??**, the number of garbage collections in both cases are same but copying closures using liveness takes a lot more time compared to reachability based copying.

The number of garbage collections is the same since we approximate the liveness of a closure to the liveness at its creation point. The liveness we get is usually grossly over approximated. Using this liveness may not give too much of a benefit compared to reachability as nearly everything would be live at its point of creation.

As mentioned in Section **??**, using liveness for closures may entail multiple traversals over the same structure consuming more time. The advantage of avoiding multiple traversals can be clearly seen from the lcss example where a single GC takes less than a millisecond for reachability based closure collection where as the liveness based collection is of the order of seconds. Lazy languages tend to have a lot of sharing and hence using a liveness based collection for closures is impractical.

Therefore we believe that using reachability for closures is a sweet spot in terms of performance for liveness based garbage collectors. For the remainder of the section all LGC results we discuss uses a mixed-mode collector–liveness for evaluated data and reachability for closures.

Table 2. Statistics for liveness analysis and garbage collection

Program	lambda	nperm	treejoin	lcss	sudoku	fibheap	nqueens	knightstour	gc_bench
#CFG Nonterminals	1078	794	709	839	1758	780	529	659	417
#CFG Rules	1823	1038	1610	1752	2509	1358	748	909	486
#DFA States	3312	2055	2332	2340	5672	2254	1229	1742	729
#DFA Transitions	5792	3632	4235	4261	10857	4076	2078	3101	1124
DFA Generation Time (sec)	14.7	1.1	2690.5	13.3	798.5	49.8	0.5	5.1	0.1

(a) Data for Liveness Analysis

Program	#Cells collected/GC		#Cells touched/GC		#GCs		#Drag Cells			GC time (sec)		Speedup
	RGC	LGC	RGC	LGC	RGC	LGC	RGC	LGC	$\frac{RGC}{LGC}$	RGC	LGC	$\frac{RGC}{LGC}$
lambda	7271.7	7517.1	13194.2	95644.2	775	750	654155	520799	1.25	0.17	0.93	0.17
nperm	4684.4	8633.2	22744.2	18808.5	710	386	172987	92033	1.87	0.24	0.16	1.45
treejoin	50284.2	72317.5	1566250.0	1553000.0	116	81	1851014	1643767	1.12	4.16	4.13	1.00
lcss	8064.7	10103.6	14177.9	12542.8	30	24	32100	31472	1.01	0.01	0.01	0.86
sudoku	983.9	1048.8	3091.0	5650.9	169	159	24349	22451	1.08	0.01	0.02	0.34
fibheap	4610.8	4736.6	33389.9	35890.1	1002	975	1442001	1424078	1.01	1.08	4.83	0.22
nqueens	17048.1	21993.1	5680.9	985.9	511	396	230378	166310	1.38	0.05	0.02	2.04
knightstour	229973.0	289223.0	450027.0	454813.0	412	328	1148037	1083066	1.05	4.52	8.77	0.51
gc_bench	21080.6	204817.0	183769.0	33.0	34	4	204874	31	6608.83	0.12	0.00	0.00

(b) Comparing RGC with LGC

7.2 Results

The statistics related to compile time liveness analysis are shown in Table **??**(a). We observe that the analysis requires a reasonable time except for `treejoin` and `sudoku`. The bottleneck in our analysis is the NFA to DFA conversion (worst-case exponential behaviour). To avoid this compile-time overhead, we have added a caching mechanism in our simulator that can reuse the DFA from earlier analysis if the program has not changed..

Table **??**(b) compares the garbage collection statistics for RGC and LGC. We report the number of GC events, average number of cells reclaimed per GC, average number of cells touched per GC, total number of dragged cells and total time to perform all collections. It is no surprise that number of cells reclaimed per GC is higher (and consequently the number of GCs is lower) for LGC based collection. LGC also reduces the number of dragged cells significantly. However, the cost for LGC is that per GC execution time is more for LGC, which results in higher overall execution time for garbage collection, even with reduced number of collections. However, LGC run time is still competitive (slowdown within 5X of RGC for worst case), and is better for 3 benchmarks (best case 2X speedup). Note that `gc_bench` [**?**] is a synthetic benchmark that allocates complete binary trees of various sizes. These binary trees are never used by the program. As a result, the benchmark highly favours LGC. We have included this benchmark for completeness only and the numbers for it are not considered representative of practical programs.

Memory usage graphs for select benchmarks are shown Figure **??**, Column (a). As the number of garbage collections tend to be very large, for each benchmark we also show a window which is representative of the behavior for that particular benchmark (Column (b)). In all the programs we can see that the curve corresponding to LGC (red line) regularly dips below the RGC curve (blue line), the dip is specially visible for the benchmarks where LGC outperforms RGC substantially. The graphs also include curve for reachable cells (black, obtained approximately by forcing RGC to run at a very high frequency) and the live cells for particular run (lightblue, obtained by post processing the heap access at the end of a program).

One area of concern is the huge gap between the actual liveness and the liveness perceived by our collector. In case of LGC for eager languages [**?**] the gap was very narrow and almost touched the actual liveness curve. In case of lazy languages, this gap is largely due to approximate reachability based collection of closures. To implement liveness based collection of closures, we need to record liveness data for each closure and update it at each evaluation point. These space and run time overheads make it infeasible to incorporate liveness based GC for closures in a practical garbage collector.

8. Related Work

Although augmenting garbage collection with liveness information has been studied earlier, it was mainly in the context of imperative languages [**? ? ?**]. Due to the presence of global variables and mutation, practical utility of simple liveness based techniques are found to be inefficient.

In the space of functional languages, improving space efficiency has been mainly studied as a compile time activity. Either through rewriting methods such as deforestation [**? ? ?**], sharing analysis based reallocation [**?**], region based analysis [**?**], or through insertion of compile time nullifying statements such as [**? ? ?**]. The compile time marking approaches all rely on an efficient and precise alias analysis and in the absence of it cannot provide notable improvement. Another important approach advocated by Hofmann [**?**] is to use linear typing to analyze first-order programs for heap usage and annotated them. The system then uses these annotations to re-use allocated memory cells instead of requesting for newer cells. This requires the user to write programs in a specific way and hence may not be practical

Simplifiers [**?**], described as lightweight daemons, improve the efficiency of the program in general and the garbage collector in particular. Most of the simplifications mentioned in this work are subsumed by a liveness based collector. We feel this work is orthogonal and can augment our liveness based collector. The closest to our approach is the liveness based garbage collector implemented in [**?**]. We extend their work to handle lazy evaluation and closures.

9. Conclusions and Future Work

We extended the liveness based garbage collection to lazy languages and shown its benefit for garbage collection in practical programs. We defined a context sensitive liveness analysis that uses context independent summaries of functions obtained using a sym-

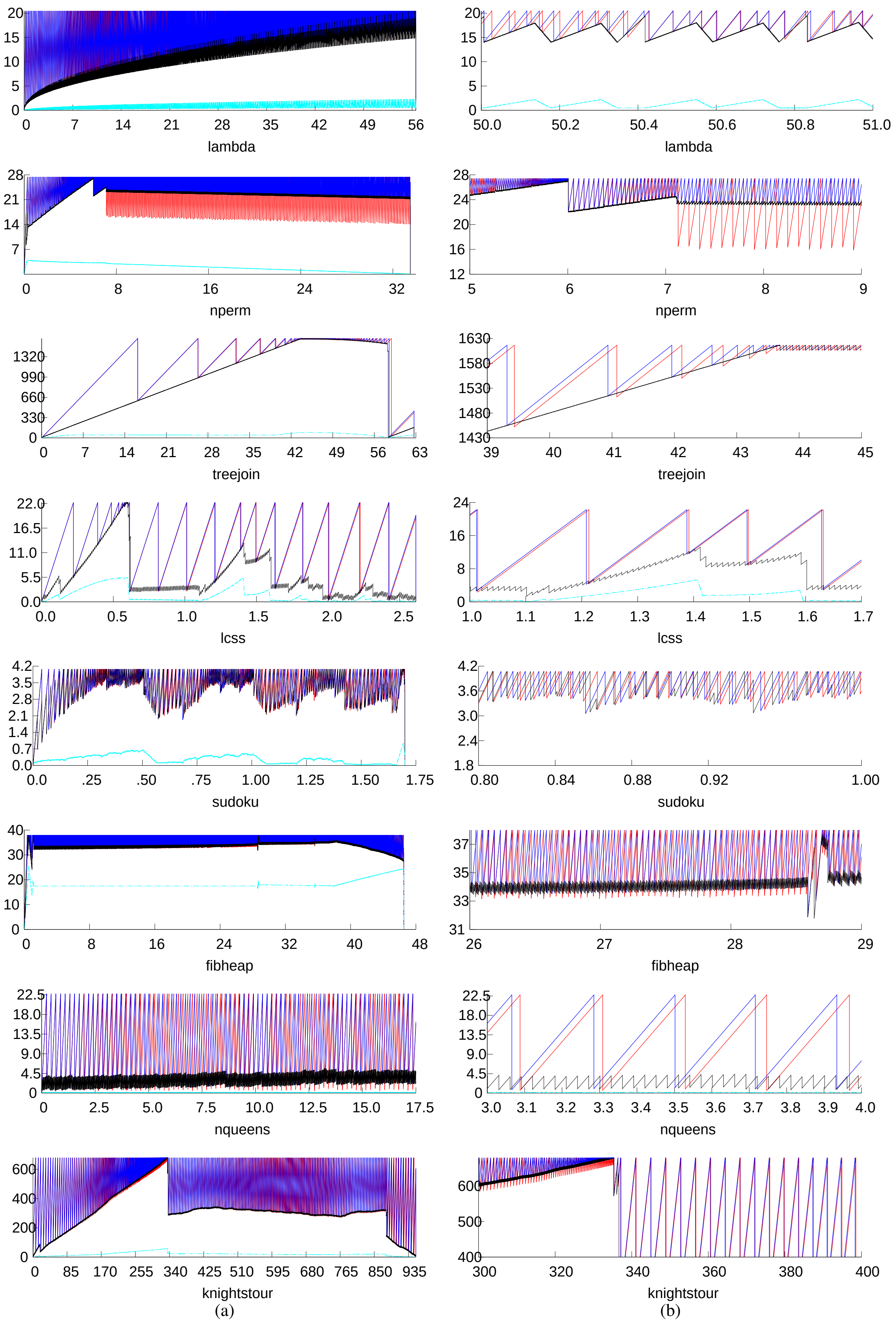


Figure 8. Memory usage. Column (a) shows complete usage, column (b) enlarges a part of the usage. The blue and the red curves indicate the number of cons cells in the active semi-space for RGC and LGC respectively. The black curve represents the number of reachable cells and the lightblue curve represents the number of cells that are actually live (of which liveness analysis does a static approximation). x-axis is the time measured in number of cons-cells allocated (scaled down by factor 10^5). y-axis is the number of cons-cells (scaled down by 10^3).

bolic demand. We showed that obtaining precise solution to these equations is undecidable in our formulation, and hence we safely approximate the result using DFA. These DFA are consulted by the garbage collector to improve collection.

Liveness of closures presented further challenges to our implementation. We compared different approximate strategies for handling liveness of closures, and found that a mixed strategy—reachability based collection of closure arguments, liveness based collection of everything else—works best in practice as it avoids runtime and space overheads. With mixed-mode garbage collection scheme, we were able to collect more cells than reachability based collectors at a reasonable overhead.

Although we provide an implementation for a liveness based garbage collection scheme for lazy language, we do not provide a formal proof of its correctness. A formal proof of correctness would describe how closures are handled during garbage collection and liveness is propagated inside closures.

Another interesting exercise is to narrow the gap between the actual liveness and the perceived liveness of our garbage collector. Our experiments (not reported here) show that a significant number of dead cells get trapped inside closures. Using strictness analysis to eagerly evaluate closures might release more of these dead cells to be garbage collected. Since our garbage collector can handle closures (suspended evaluations), another major challenge is to extend it to support higher order programs.

Orthogonally, we plan to improve the efficiency of the liveness based garbage collector using heuristics such as limiting the depth of DFA, merging nearly-equivalent states and using better representation and algorithms for automata manipulation. We also need to investigate the interaction of liveness with other collection schemes, such as incremental and generational collection. In summary, we need to investigate ways to make liveness based garbage collection attractive for practical collectors.