Liveness-Based Garbage Collection

Rahul Asati[1], Amitabha Sanyal[1], Amey Karkare[2], and Alan Mycroft[3]

[1] IIT Bombay, Mumbai 400076, India
{rahulasati,as}@cse.iitb.ac.in,
[2] IIT Kanpur, Kanpur 208016, India
karkare@cse.iitk.ac.in,
[3] Computer Laboratory, University of Cambridge, CB3 0FD, UK
alan.mycroft@cl.cam.ac.uk

Abstract. Current garbage collectors leave much heap-allocated data uncollected because they preserve data *reachable* from a root set. However, only *live* data—a subset of reachable data—need be preserved.
Using a first-order functional language we formulate a context-sensitive liveness analysis for structured data and prove it correct. We then use a 0-CFA-like conservative approximation to annotate each allocation and function-call program point with a finite-state automaton—which the garbage collector inspects to curtail reachability during marking. As a result, fewer objects are marked (albeit with a more expensive marker) and then preserved (e.g. by a copy phase).
Experiments confirm the expected performance benefits—increase in garbage reclaimed and a consequent decrease in the number of collections, a decrease in the memory size required to run programs, and reduced overall garbage collection time for a majority of programs.

1 Introduction

Most modern programming languages support dynamic allocation of heap data. Static analysis of heap data is much harder than analysis of static and stack data. Garbage collectors, for example, conservatively approximate the liveness of heap objects by their reachability from a set of memory locations called the *root set*. Consequently, many objects that are reachable but not live remain uncollected, causing a larger-than-necessary memory demand. This is confirmed by empirical studies on Haskell [1], Scheme [2] and Java [3] programs.

Here we consider a first-order pure functional language and propose a liveness analysis which annotates various program points with a description of variables and fields whose object references may be dereferenced in the future. The garbage collector then only marks objects pointed to by live references and leaves other, merely reachable, objects to be reclaimed. (Although not strictly necessary, a collector would normally *nullify* dead variables and fields rather than leaving dangling references.) Since there are fewer live objects than reachable objects, more memory is reclaimed. Additionally, since the collector traverses a smaller portion of the heap, the time spent for each collection is also smaller. The work is presented in the context of a stop-the-world non-incremental garbage collector (mark-and-sweep, compacting or copying) for which we also show a monotonicity result: that our technique can never cause more garbage collections to occur in spite of changing the rather unpredictable execution points at which collections occur. We anticipate that our technique is applicable to more modern collectors (generational, concurrent, parallel), but leave such extensions to future work.

We first define a *fully context-sensitive* (in the sense that its results are unaffected by function inlining) liveness analysis and prove it correct. However, fully context-sensitive methods often do not scale, and this analysis would also require us to determine, at run-time, the internal liveness of a function body at each call. Hence, similarly to the 0-CFA approach, we determine a context-independent *summary* of liveness for each function which safely approximates the context-dependence of all possible calls [4–6]. (Note that an intraprocedural context-insensitive method which assumes no information about function callers would be too imprecise for our needs.) In essence our approach sets up interprocedural data-flow equations for the liveness summaries of functions and shows how these can be solved symbolically as context-free grammars (CFGs). We can then determine a CFG for each program point; these are then safely approximated with finite-state automata which are encoded as tables for each program point. For garbage collection purposes only automata corresponding to *GC points* need to be stored. GC points are program points associated with a call to a user function or to **cons**—see Section 4.

We previously proposed an intraprocedural method for heap liveness analysis for a Java-like language [7] which statically inserted statements nullifying dead references to improve garbage collection; by contrast nullification here occurs dynamically (which can work better with aliasing) when the garbage collector acts on liveness annotations to avoid traversing dead references. A workshop paper [8] outlined the basic 0-CFA-style-summary interprocedural approach to functional-program liveness analysis. The current paper adds the context-sensitive analysis and better formalisation along with experimental results.

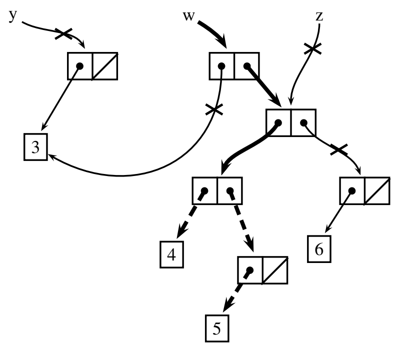
Motivating Example Figure 1(a) shows an example program. The label π of an expression e denotes a program point. During execution of the program, it represents the instant of time just before the evaluation of e. We view the heap as a graph. Nodes in the heap, also called (**cons**) *cells* contain **car** and **cdr** *fields* containing values. Edges in the graph are *references* and emanate from *variables* or fields. Variable and field values may also be atomic values (**nil**, integers etc.) While it is convenient to box these in diagrams, our presented analysis treats them as non-heap values.

Figure 1(b) shows the heap at π. The edges shown by thick continuous arrows are those which are made live by the program. In addition, assuming that the value of any reachable part of the program result may be explored or printed, the edges marked by thick dashed arrows are also live. A cell is marked and 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. We model the liveness properties of the heap as automata and pass these automata to the garbage collector. Thus if a garbage collection happens at π with the heap shown in Figure 1(b), only the cells `w` and (**cdr** `w`), along with (**car** (**cdr** `w`)) and all cells reachable from it, will be marked and preserved.

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

(let z ←(cons (cons 4 (cons 5 nil))
              (cons 6 nil)) in
  (let y ← (cons 3 nil) in
      (let w ← (append y z) in
        π:(car (cdr w)))))
```

(a) Example program.



(b) Memory graph at π. Thick edges denote live links. Traversal stops during garbage collection at edges marked ×.

Fig. 1. Example Program and its Memory Graph.

Organisation of the paper Section 2 gives the syntax and semantics of the language used to illustrate our analysis along with basic concepts and notations. Liveness analysis is described in Section 3 followed by a sketch of a correctness proof relative to a non-standard semantics. Section 4 shows how to encode liveness as finite-state automata. Section 5 reports experimental results and Section 6 proves that a liveness-based collector can never do more garbage collections than a reachability-based collector.

2 The target language—syntax and semantics

We let x, y, z range over variables, f over user-functions and p over primitive functions (**cons**, $+$ etc.). The syntax of our language is shown in Figure 2; it has eager semantics and restricts programs to be in Administrative Normal Form (ANF) [9] where all actual parameters to functions are variables. This restriction does not affect expressibility (and indeed we feel free to ignore it in examples when inessential), but simplifies the analysis formulation. Additionally, as in the three-address instruction form familiar from compiler texts, it forces each temporary to be named and function calls to be serialised (necessary to get an unambiguous definition of liveness). We further require that each variable in a program is distinct, so that no scope shadowing occurs—this simplifies proofs of soundness. In this formulation expressions: either perform a test (**if**), make a computation step (**let**) or return (**return**). The **return** keyword is logically redundant, but we find it clarifies the semantics and analysis.

The body of the program is the expression denoted by $e_{\textbf{main}}$; for analysis purposes it is convenient to regard $e_{\textbf{main}}$ as part of a function definition (**define** (`main`) $e_{\textbf{main}}$) as in C. We write $\pi : e$ to associate the label π (not part of the language syntax) with the program point just before expression e.

In spite of the ANF restrictions it is still possible to inline non-recursive functions (a fact we use to prove the safety of liveness analysis). A user-function call $(\textbf{let}\, x \leftarrow (f\, y_1 \ldots y_n)\; \textbf{in}\; e)$ to a function defined (after renaming its formals and locals to be disjoint from existing variables) by $(\textbf{define}\; (f\; z_1\; \ldots\; z_n)\; e_f)$ is replaced by a sequence of **let**s of the form $z_i \leftarrow (\textbf{id}\; y_i)$ followed by the body e_f but with its $(\textbf{return}\, w)$ expressions replaced by $(\textbf{let}\, x \leftarrow (\textbf{id}\; w)\; \textbf{in}\; e)$. (We prefer to use **id** as a form of no-op

$$
\begin{array}{lll}
p \in \mathit{Prog} ::= d_1 \ldots d_n\, e_{\mathbf{main}} & & \textit{— program} \\
d \in \mathit{Fdef} ::= (\mathbf{define}\ (f\ x_1\ \ldots\ x_n)\ e) & & \textit{— function definition} \\
e \in \mathit{Expr} ::= \left\{ \begin{array}{l} (\mathbf{if}\ x\ e_1\ e_2) \\ (\mathbf{let}\ x \leftarrow s\ \mathbf{in}\ e) \\ (\mathbf{return}\ x) \end{array} \right. & & \begin{array}{l} \textit{— conditional} \\ \textit{— let binding} \\ \textit{— return from function} \end{array} \\
s \in \mathit{Stmt} ::= \left\{ \begin{array}{ll} k & \\ (\mathbf{cons}\ x_1\ x_2) & \\ (\mathbf{car}\ x) & (\mathbf{cdr}\ x) \\ (\mathbf{null?}\ x) & (+\ x_1\ x_2) \\ (\mathbf{id}\ x) & \\ (f\ x_1\ \ldots\ x_n) & \end{array} \right. & & \begin{array}{l} \textit{— constant (numeric or } \mathbf{nil}\textit{)} \\ \textit{— constructor} \\ \textit{— selectors} \\ \textit{— tester and generic arithmetic} \\ \textit{— identity function (for inlining)} \\ \textit{— function application} \end{array}
\end{array}
$$

Fig. 2. The syntax of our language

function rather than introducing the form $(\mathbf{let}\, x \leftarrow w\ \mathbf{in}\ e)$ where the *Stmt* part of **let** is a simple variable.)

Semantics We now give an operational semantics for our language. Later, a refinement of the operational semantics, which we call minefield semantics, will serve to prove liveness analysis correct. We give a small-step semantics because, unlike big-step semantics, correctness for non-terminating programs does not need special treatment. We start with the domains used by the semantics:

$$
\begin{array}{lll}
v : \mathit{Val} & = \mathbb{N} + \{\mathbf{nil}\} + \mathit{Loc} & \text{– Values} \\
\rho : \mathit{Env} & = \mathit{Var} \to \mathit{Val} & \text{– Environment} \\
\mathrm{H} : \mathit{Heap} & = \mathit{Loc} \to (\mathit{Val} \times \mathit{Val}) & \text{– Heap}
\end{array}
$$

Here *Loc* is a countable set of locations which hold **cons** cells. A value is either a number, the empty list **nil**, or a location ℓ. Our liveness analysis does not track numeric values, and thus is neutral as to whether these are boxed or represented as immediates. An environment is a finite mapping from variables to values, and a heap a finite mapping from locations to pairs of values. Finally, S is a stack (using $\bullet$ for push and $[]$ for empty stack) of frames of unfinished function calls. A frame is a triple (e, x, ρ) representing the call site $(\mathbf{let}\, x \leftarrow (f\, y_1\, \ldots\, y_n)\ \mathbf{in}\ e)$ being evaluated in environment ρ. Frames can also be viewed as continuations, in this view the (ORD-RETURN) rule in the small-step operational semantics (Figure 3) invokes them.

The semantics of statements s are given by the judgement form $\rho, \mathrm{H}, s \rightsquigarrow \mathrm{H}', v$ and those for expressions e by the form $\rho, \mathrm{S}, \mathrm{H}, e \to \rho', \mathrm{S}', \mathrm{H}', e'$. The start state is $(\{\}, [], \{\}, e_{\mathbf{main}})$ and the program terminates successfully with result value $\rho(x)$ on reaching the halt state $(\rho, [], \mathrm{H}, (\mathbf{return}\, x))$

Notation: we write $\rho[x \mapsto v]$ for the environment which is as ρ but has value v at x. We also write $[\vec{x} \mapsto \vec{v}]$ which respectively has values $v_1, \ldots, v_n$ at $x_1, \ldots, x_n$ and write $[\vec{x} \mapsto \rho(\vec{y})]$ when $v_1, \ldots, v_n$ are $\rho(y_1), \ldots, \rho(y_n)$.

$$\frac{}{\rho, \mathrm{H}, k \rightsquigarrow \mathrm{H}, k}\ (\text{ORD-CONST}) \qquad \frac{\ell \notin \mathrm{dom}(\mathrm{H}) \text{ is a fresh location}}{\rho, \mathrm{H}, (\mathbf{cons}\, x\, y) \rightsquigarrow \mathrm{H}[\ell \mapsto (\rho(x), \rho(y))], \ell}\ (\text{ORD-CONS})$$

$$\frac{\mathrm{H}(\rho(x)) = (v_1, v_2)}{\rho, \mathrm{H}, (\mathbf{car}\, x) \rightsquigarrow \mathrm{H}, v_1}\ (\text{ORD-CAR}) \qquad \frac{\mathrm{H}(\rho(x)) = (v_1, v_2)}{\rho, \mathrm{H}, (\mathbf{cdr}\, x) \rightsquigarrow \mathrm{H}, v_2}\ (\text{ORD-CDR})$$

$$\frac{}{\rho, \mathrm{H}, (\mathbf{id}\, x) \rightsquigarrow \mathrm{H}, \rho(x)}\ (\text{ORD-ID}) \qquad \frac{\rho(x) \in \mathbb{N} \qquad \rho(y) \in \mathbb{N}}{\rho, \mathrm{H}, (+\, x\, y) \rightsquigarrow \mathrm{H}, \rho(x) + \rho(y)}\ (\text{ORD-PRIM})$$

$$\frac{\rho(x) \neq \mathbf{nil}}{\rho, \mathrm{H}, (\mathbf{null?}\, x) \rightsquigarrow \mathrm{H}, 0} \qquad \frac{\rho(x) = \mathbf{nil}}{\rho, \mathrm{H}, (\mathbf{null?}\, x) \rightsquigarrow \mathrm{H}, 1}\ (\text{ORD-NULL})$$

$$\frac{\rho(x) \in \mathbb{N} \setminus \{0\}}{\rho, \mathrm{S}, \mathrm{H}, (\mathbf{if}\, x\, e_1\ e_2) \longrightarrow \rho, \mathrm{S}, \mathrm{H}, e_1} \qquad \frac{\rho(x) = 0}{\rho, \mathrm{S}, \mathrm{H}, (\mathbf{if}\, x\, e_1\ e_2) \longrightarrow \rho, \mathrm{S}, \mathrm{H}, e_2}\ (\text{ORD-IF})$$

$$\frac{\rho, \mathrm{H}, s \rightsquigarrow \mathrm{H}', v \qquad s \text{ is not } (f\, y_1 \ldots y_n)}{\rho, \mathrm{S}, \mathrm{H}, (\mathbf{let}\, x \leftarrow s\ \mathbf{in}\ e) \longrightarrow \rho[x \mapsto v], \mathrm{S}, \mathrm{H}', e}\ (\text{ORD-LET-NONFN})$$

$$\frac{s \text{ is } (f\, y_1 \ldots y_n) \qquad f \text{ defined as } (\mathbf{define}\ (f\ z_1\ \ldots\ z_n)\ e_f)}{\rho, \mathrm{S}, \mathrm{H}, (\mathbf{let}\, x \leftarrow s\ \mathbf{in}\ e) \longrightarrow [\vec{z} \mapsto \rho(\vec{y})],\ (\rho, x, e) \bullet \mathrm{S},\ \mathrm{H}, e_f}\ (\text{ORD-LET-FNCALL})$$

$$\frac{}{\rho,\ (\rho', x', e') \bullet \mathrm{S},\ \mathrm{H},\ (\mathbf{return}\, x) \longrightarrow \rho'[x' \mapsto \rho(x)], \mathrm{S}, \mathrm{H}, e'}\ (\text{ORD-RETURN})$$

Fig. 3. The small-step operational semantics

Stuck states. Note that certain forms of e do not reduce with $\rightarrow$ (perhaps because $\rightsquigarrow$ could not reduce a contained s). Some of these we eliminate syntactically, e.g. ensuring all variables and functions are defined and are called with the correct number of parameters. Others include (**cdr nil**), (**car** 3), (+ **nil** 4) and (**if nil** e_1 e_2). All but the first can be eliminated with a static type system but, treating our program as dynamically typed, we regard all these as stuck states.

3 Liveness

In classical liveness analysis a variable is either 'live' (its value may be used in future computation) or 'dead' (definitely not used). Semantically, a variable is dead at a given program point if arbitrary changes to its value have no effect on the computation. Later we will use $\bot$ to represent a value which 'explodes' when it is used in a computation; dead variables can safely have their value replaced with $\bot$. For heap-allocated data we need a richer model of liveness in that both variables and fields of **cons** cells may be dead or live. Using **0**, **1** to represent access using **car**, **cdr** respectively, liveness of the structure reachable from a variable is a set of *access paths* which we represent as a subset of $\{\mathbf{0},\mathbf{1}\}^*$, and use conventional grammar notation. Thus the liveness of x being $\{\mathbf{10},\mathbf{110}\}$ means that future computation can only refer to the second and third members of x considered as a list. Semantically, access paths are prefix-closed, as accessing a field requires accessing all the paths from the variable to the field, and hence the above liveness is properly written $\{\varepsilon,\mathbf{1},\mathbf{10},\mathbf{11},\mathbf{110}\}$. The classical notions of a scalar variable being live or dead correspond to $\{\varepsilon\}$ and $\{\}$.

The overall liveness (also written *liveness environment* for emphasis) at a program point is conceptually a mapping from variables to subsets of $\{\mathbf{0},\mathbf{1}\}^*$, but we often abuse notation, for example writing $\{x.\mathbf{01}, x.\mathbf{1}, y.\varepsilon\}$ instead of the map $[x \mapsto \{\varepsilon, \mathbf{0}, \mathbf{01}, \mathbf{1}\}, y \mapsto \{\varepsilon\}, z \mapsto \{\}]$. Analogously to classical liveness, the liveness at program point π in $\pi : e$ is the liveness just before executing e.

A complementary notion to liveness is *demand*. The demand for expression e is again an access path—that subset of $\{\mathbf{0},\mathbf{1}\}^*$ which the context of e may explore of e's result. So, for example given a demand σ and the expression $\pi : (\textbf{return}\ x)$, the liveness at π is exactly $x.\sigma$. The classical analogy of this is in *strong liveness*, where an assignment node $n : x := y+z$ causes y and z to be live on entry to n if (and only if) x is live at exit of n—the liveness of x at exit from n becomes the demand on $y+z$. Note that, for an operation like division which may raise an exception, the assignment $n : x := y/z$ makes y and z live regardless of the liveness of x.

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 $\{\mathbf{0}\}\sigma$. Finally, we use LF to range over *demand transformers*; given user function f, LF_f transforms demands on a call to f into demands on its formal parameters: if f is defined by $(\textbf{define}\ (f\ x_1\ \ldots\ x_n)\ \ e_f)$ and called with demand σ, then $\mathsf{LF}^i_f(\sigma)$ is the liveness of x_i at e_f.

Note that liveness refers to variables and fields, and not to **cons** cells (i.e. to edges in the memory graph, not to locations themselves). Hence liveness of $\{x.\varepsilon, x.\mathbf{0}\}$ means that future computation may refer to the value ℓ of variable x, and also to the **car** field of location ℓ. In the absence of other pointers to heap location ℓ, we are certain that the **cdr** field of ℓ will not be referenced and may hence be corrupted arbitrarily. Note therefore, that while ℓ cannot be garbage collected, any location ℓ' stored in the **cdr** field of ℓ would be garbage (again provided there are no other aliases to ℓ or ℓ').

3.1 Liveness Analysis

First recall the classical formulation of liveness (as sets of simple variables) on three-address instructions, $\mathit{live}_{in}(I) = \mathit{live}_{out}(I) \setminus \mathit{def}(I) \cup \mathit{ref}(I)$, and then note that *strong liveness* needs, when I is the instruction $z := x+y$, that $\mathit{ref}(I)$ be refined to $\{x,y\}$ if $z \in \mathit{live}_{out}(I)$ and $\{\}$ otherwise.

Our liveness analysis formulated in Figure 4 is analogous. Firstly, the function *ref*, when given a statement s, returns the liveness *generated* by s. Because we generalise *strong* liveness, *ref* needs a second parameter, specifying the demand σ on the result of s, to determine which access paths of its free variables are made live. The cases for $(\textbf{id}\ x)$ and $(+\ x\ y)$ exemplify this. A demand of σ on $(\textbf{car}\ x)$ is transformed to the demand $\mathbf{0}\sigma$ on x. In addition, **car** always dereferences its argument (even if its result is never used). This generates the liveness $\{x.\varepsilon\} \cup x.\mathbf{0}\sigma$ (note σ may be $\{\}$). In the opposite sense, the demand of $\mathbf{0}\sigma$ on $(\textbf{cons}\ x\ y)$ is transformed to the demand σ on x. Note that **cons** does not, by itself, dereference its arguments. Thirdly, for the case of a user-function call, a third parameter LF to *ref* expresses how the demand σ on the result is transformed into demands on its parameters. Constants generate no liveness.

$$\begin{aligned}
\mathit{ref}(\kappa,\sigma,\mathsf{LF}) &= \{\}, \text{ for } \kappa \text{ a constant, including } \textbf{nil}\\
\mathit{ref}((\textbf{cons}\,x\,y),\sigma,\mathsf{LF}) &= \{x.\alpha \mid \mathbf{0}\alpha \in \sigma\} \cup \{y.\alpha \mid \mathbf{1}\alpha \in \sigma\}\\
\mathit{ref}((\textbf{car}\,x),\sigma,\mathsf{LF}) &= \{x.\varepsilon\} \cup \{x.\mathbf{0}\alpha \mid \alpha \in \sigma\}\\
\mathit{ref}((\textbf{cdr}\,x),\sigma,\mathsf{LF}) &= \{x.\varepsilon\} \cup \{x.\mathbf{1}\alpha \mid \alpha \in \sigma\}\\
\mathit{ref}((\textbf{id}\,x),\sigma,\mathsf{LF}) &= \{x.\sigma\}\\
\mathit{ref}((+\,x\,y),\sigma,\mathsf{LF}) &= \{x.\varepsilon, y.\varepsilon\}\\
\mathit{ref}((\textbf{null?}\,x),\sigma,\mathsf{LF}) &= \{x.\varepsilon\}\\
\mathit{ref}((f\,y_1\,\cdots\,y_n),\sigma,\mathsf{LF}) &= \textstyle\bigcup_{i=1}^{n} y_i.\mathsf{LF}_f^i(\sigma)
\end{aligned}$$

$$\begin{aligned}
\mathcal{L}((\textbf{return}\,x),\sigma,\mathsf{LF}) &= x.\sigma\\
\mathcal{L}((\textbf{if}\,x\,e_1\;e_2),\sigma,\mathsf{LF}) &= \mathcal{L}(e_1,\sigma,\mathsf{LF}) \cup \mathcal{L}(e_2,\sigma,\mathsf{LF}) \cup \{x.\varepsilon\}\\
\mathcal{L}((\textbf{let}\,x \leftarrow s\;\textbf{in}\;e),\sigma,\mathsf{LF}) &= \mathsf{L} \setminus x.\{\mathbf{0},\mathbf{1}\}^* \cup \mathit{ref}(s,\mathsf{L}(x),\mathsf{LF}), \text{ where } \mathsf{L} = \mathcal{L}(e,\sigma,\mathsf{LF})
\end{aligned}$$

$$\frac{\mathcal{L}(e_f,\sigma,\mathsf{LF}) = \bigcup_{i=1}^{n} z_i.\mathsf{LF}_f^i(\sigma) \text{ for each } f \text{ and } \sigma}{d_1 \ldots d_k \vdash^l \mathsf{LF}} \quad (\textsc{live-define})$$
where $(\textbf{define}\;(f\;z_1\;\ldots\;z_n)\;\;e_f)$ is a member of $d_1 \ldots d_k$

Fig. 4. Liveness equations and judgement rule

The function $\mathcal{L}$ now gives the (total) liveness of an expression e. The cases **return** and **if** are straightforward, but note the liveness $x.\varepsilon$ generated by the latter. The case $(\textbf{let}\;z \leftarrow s\;\textbf{in}\;e')$ resembles a three-address instruction: the liveness of e is given by taking the liveness, L, of e', killing any liveness of z and adding any liveness generated by s. The main subtlety is how the liveness of z in L is converted to a demand $\mathsf{L}(z)$ to be placed on s via $\mathit{ref}(s,\mathsf{L}(z),\mathsf{LF})$.

Finally, the judgement form $\mathit{Prog}\;\vdash^l\;\mathsf{LF}$ is used to determine LF. Analogously to classical liveness being computed as a solution of dataflow equations, we require, via inference rule (LIVE-DEFINE), LF to satisfy the fixed-point property that: when we assume LF to be the demand transformer for the program then the calculated liveness of each function body $\mathcal{L}(e_f,\sigma,\mathsf{LF})$ agrees with the assumed LF_f. As usual, there are often multiple solutions to LF; all are safe (see Section 3.2) but we prefer the least one as giving the least liveness subject to safety—and hence greatest amount of garbage collected.

We make three observations: firstly the rule (LIVE-DEFINE) has a least solution as $\mathcal{L}(\cdot)$ is monotonic in σ; secondly that (LIVE-DEFINE) resembles the rule for type inference of mutually recursive function definitions, and thirdly the asymmetry of demand and liveness (compared to post- and pre-liveness classically) is due to the functional formulation here.

Section 4 shows how the demand transformer LF for a program (representing a fully context-sensitive analysis) can be safely approximated, for each function, by a *procedure summary* (unifying the contexts in the style of 0-CFA). The summary consists of a pair of a single demand and, for this demand, the corresponding tuple of demands the function makes on its arguments.

3.2 Minefield semantics and correctness

This section gives a modified semantics which checks liveness annotations at run time, and 'explodes' when these are found to be inconsistent with execution behaviour, but otherwise behaves as the standard semantics. We show that such explosions never occur and hence run-time checks can be elided. We first assume an arbitrary demand transformer LF (below we assume $Prog \ \vdash^l \ \mathsf{LF}$). We then enrich the abstract machine state $\rho, \mathrm{S}, \mathrm{H}, e$ to $\rho, \mathrm{S}, \mathrm{H}, e, \sigma, \Sigma$. Here σ is the demand to the currently active function, thus the liveness L at e is $\mathcal{L}(e, \sigma, \mathsf{LF})$,[4] and Σ is a stack of demands—one for each function frame pushed in S.

Second, we augment *Val* with a value $\bot$. To model strong liveness $\bot$ may be copied freely, including into a **cons** cell, but explodes when used computationally (in a primitive operation other than a copy). Additionally we define $GC(\mathsf{L}, \Sigma) : (\rho, \mathrm{H}, \mathrm{S}) \mapsto (\rho', \mathrm{H}', \mathrm{S}')$ which determines *live-reachability*[5] using ρ and the ρ's in S as the root set and following links in H *only as far as allowed by* L *and* Σ. Hence $GC(\mathsf{L}, \Sigma)$ replaces live-unreachable values—in ρ, in H and the ρ's in S—with $\bot$. For example, if $x.\varepsilon \notin \mathsf{L}$ then $\rho'(x) = \bot$. Only $GC(\cdots)$ introduces $\bot$.

Third, we update the semantics in four ways: (*i*) we arrange that all $(\rightarrow)$ transitions on expressions e first use $GC(\cdots)$ to update the state and then continue as before; and (*ii*) whenever the value $\bot$ is used computationally in a reduction $(\rightsquigarrow)$, we enter a distinguished stuck state BANG. For example, supposing $\rho(x) = \bot$ then $\rho, \mathrm{H}, s \rightsquigarrow$ BANG if s is $(\mathbf{car}\, x)$, $(\mathbf{cdr}\, x)$ or $(+\ x\ y)$, but not if s is $(\mathbf{id}\ x)$, $(\mathbf{cons}\ x\ y)$ or $(f\ x\ y)$. Finally (*iii*) we make a similar change to the $(\rightarrow)$ reduction for (ORD-IF) and (*iv*) augment the (ORD-LET) rule for primitives to propagate BANG from $(\rightsquigarrow)$ to $(\rightarrow)$.

The resulting minefield semantics behaves identically (identical heap, identical steps, including possible non-termination) to the standard semantics, except for the sole possibility of the minefield semantics going BANG while the standard semantics continues (either to a halt state, to a stuck state, or reduces forever).

We now prove a result that relates liveness analysis to the semantics.

Proposition 1. *Given program P with $P \ \vdash^l \ \mathsf{LF}$, then in the minefield semantics $P \rightarrow$ BANG can never occur (cf. 'well-typed programs do not go wrong').*

Proof outline: Space does not permit a full proof, but we give the two main steps. We proceed by contradiction and assume there is a program P for which $P \ \vdash^l \ \mathsf{LF}$ can enter state BANG. The first step is to construct a program P' with identical behaviour, but with no user-function definitions, using inlining. This is possible because a program which goes BANG does so after a finite number of reductions, and hence even recursive functions have only had a finite number of invocations. We hence repeatedly inline-expand user-function calls in P until we obtain a program P' which behaves identically[6] to P in the standard semantics, but executes no user-function calls. Any remaining, non-executed, calls can be replaced with a new primitive with the same demand-to-liveness transfer function—thus making P' a simple expression e'. Not only do the program

[4] This a simple liveness propagation using $\mathcal{L}(\cdots)$ and $ref(\cdots)$ as LF is assumed given.

[5] Reachability curtailed by liveness information.

[6] Modulo replacement of (ORD-CALL) and (ORD-RETURN) steps with (ORD-ID) steps.

points in reducing e' correspond one-one to states during evaluation of P, but also the liveness associated with a point in e' is identical to the liveness at the corresponding state $(\rho, \mathsf{S}, \mathsf{H}, e, \sigma, \Sigma)$ of P (concatenating the liveness $\mathcal{L}(e, \sigma, \mathsf{LF})$ at e with the liveness, obtained from Σ, of the call sites in S and after renaming the variables correspondingly to the inlining which produced e'). This assertion relies on the analysis being fully context-sensitive, and noting that while the change of scope caused by inlining changes variable visibility between P and e' it does not change the liveness—as local-to-a-function **let**-variables whose scope has been prolonged due to inlining are dead in the prolonged scope.

The second step of the proof is to show that e' cannot go BANG. We proceed by induction. Correctness of the **if** and final **return** forms are immediate; the (**let** $z \leftarrow s$ **in** e) form requires showing the inference rules ensure that any value referenced via z in e was already live-reachable (via another variable and path) in any enclosing expression (so that $GC(\cdots)$ could not re-write it to $\perp$).

4 Computing liveness and its encoding as a table

Section 3 gave a context-sensitive liveness analysis and proved it correct with reference to a *minefield* semantics. For practical use we need to solve the liveness equations *finitely and symbolically*. As expressed mathematically, and given a fixed program, three things are potentially unbounded: (*i*) the number of call strings (and hence arguments σ to LF); (*ii*) the length of access paths $\alpha \in \sigma$ and (*iii*) the number of such access paths.

We commonly first solve (*i*) by reducing the number of distinct calling contexts (e.g. to a single unified context in 0-CFA style). However, it turns out that we can solve the equations for LF symbolically without this reduction, so here we defer this to Section 4.2. (It also allows easier extension to dynamically determined liveness—future work.) We address (*ii*) and (*iii*) by re-interpreting the liveness definitions in Figure 4 symbolically as a grammar rather than a mutually recursive set of equations on sets of access paths.

This requires two ideas. Firstly we have to control the use of functions—they tend to be infinitary and do not occur naturally in CFGs; in particular our L maps names to access paths, and LF maps access paths to a tuple of access paths. The former is achieved by using a separate meta-variable (later non-terminal) L_i^x for each variable x and each program point π_i (L_i^x represents $\mathsf{L}(x)$ at π_i). Section 4.1 shows how the latter LF_f^i is also expressible finitely (it is a linear form).

Secondly, there are also two technical issues in re-interpreting Figure 4 as a grammar. One is the use of the set-difference operator $\setminus$ in "$\cdots \setminus x.\{\mathbf{0},\mathbf{1}\}^*$" reflecting the classical *gen/kill* dataflow formulation. However after separating, as above, liveness environments into per-variable liveness L_i^x the '$\setminus$' operator reduces to the harmless grammar rule $\mathsf{L}_i^x = \{\}$ (assuming i labels the (**let** x ...) expression). The other is that *ref* for **cons** decomposes strings and thus gives a general grammar not a CFG. Below we show how symbols $\bar{\mathbf{0}}$, $\bar{\mathbf{1}}$ can give an equivalent CFG.

Finally, Section 4.3 uses a construction due to Mohri and Nederhof [10] to over-approximate context-free grammars with regular grammars; these are more appropriate for run-time use. Hence the overall 'big picture' view is that each GC point is annotated with a table encoding the DFA for that program point. When garbage collection occurs, each saved return address on the run-time stack identifies the call-site GC point. The DFA annotating each such GC point is then used by the garbage collector to curtail (to access paths accepted by the DFA) its local-variable reachability-based marking.

GC points. Given a call site (an expression $\pi_1\colon(\mathbf{let}\ x \leftarrow (f\ y_1 \ldots y_n)\ \mathbf{in}\ \pi_2\colon e))$ its associated GC point is π_2, as it is liveness at π_2 that should be encoded in the DFA associated with the call.[7] In the case of a call to **cons** as in $\pi_1\colon(\mathbf{let}\ x \leftarrow (\mathbf{cons}\ y_1\ y_2)\ \mathbf{in}\ \pi_2\colon e)$, the situation is slightly more complex. We may either treat **cons** as doing a full procedure call (and mark its formal parameters separately during garbage collection, which again leads to its GC point being π_2), or we may regard **cons** as being inlined, in which case it is vital that liveness of y_1 and y_2 are represented in the DFA (which is achieved by using π_1 rather than π_2 as the GC point). We adopt the latter approach.

Modifying the cons rule. The *ref* rule for **cons**, shown in Figure 4, requires us to remove the leading **0** and **1** from access paths in σ. Mathematically this is fine but causes problems when solving the liveness equations *symbolically* since such decomposition cannot be expressed as a context-free grammar. To handle this, we introduce two new symbols $\bar{\mathbf{0}}$ and $\bar{\mathbf{1}}$ with the properties:

$$\bar{\mathbf{0}}\sigma \triangleq \{\alpha \mid \mathbf{0}\alpha \in \sigma\} \qquad \text{and} \qquad \bar{\mathbf{1}}\sigma \triangleq \{\alpha \mid \mathbf{1}\alpha \in \sigma\}$$

We can now rewrite the **cons** rule as:

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

We call the liveness equations with this modification $\mathcal{L}'$. The definitions of $\bar{\mathbf{0}}$ and $\bar{\mathbf{1}}$ induce the following relation $\hookrightarrow$ over sets of access paths:

$$\sigma_1\bar{\mathbf{0}}\sigma_2 \hookrightarrow \sigma_1\sigma_2', \text{ where } \sigma_2' = \{\alpha \mid \mathbf{0}\alpha \in \sigma_2\}, \quad \text{and}$$
$$\sigma_1\bar{\mathbf{1}}\sigma_2 \hookrightarrow \sigma_1\sigma_2', \text{ where } \sigma_2' = \{\alpha \mid \mathbf{1}\alpha \in \sigma_2\}$$

The reflexive transitive closure of $\hookrightarrow$ will be denoted as $\overset{*}{\hookrightarrow}$. The following proposition relates $\mathcal{L}$ and $\mathcal{L}'$:

Proposition 2. *Assume that a liveness computation based on $\mathcal{L}$ gives the liveness of the variable x at a program point π_i as σ (symbolically, $\mathsf{L}_i^x = \sigma$). Further, suppose $\mathsf{L}_i^x = \sigma'$ when $\mathcal{L}'$ is used for liveness computation instead of $\mathcal{L}$. Then $\sigma' \overset{*}{\hookrightarrow} \sigma$.*

To see why the proposition is true, consider an analysis based on $\mathcal{L}'$ in which σ appears in the context $\mathit{ref}((\mathbf{cons}\ x\ y), \sigma, \mathsf{LF})$. Let $\alpha \in \sigma$. The symbol $\bar{\mathbf{0}}$ (respectively $\bar{\mathbf{1}}$) merely marks a place in α where the original **cons** rule would have erased an immediately following **0** (respectively **1**), or, in absence of such a symbol, would have dropped α itself. Since the application of any rule in $\mathcal{L}'$ merely adds symbols at the beginning of α, the markers and other symbols in α are propagated to other dependent parts of program in their same relative positions. Consequently, the erasure carried out at the end of the analysis with $\overset{*}{\hookrightarrow}$ gives the same result as obtained through $\mathcal{L}$.

[7] A subtlety is that at machine code level the assignment to x does not take place until after the call, and so for garbage-collection purposes the DFA need not represent liveness of x.

```
(define (append l1 l2)
    $\pi_1$:(let test ← (null? l1) in
        $\pi_2$:(if test $\pi_3$:(return l2)
            $\pi_4$:(let tl ← (cdr l1) in
                $\pi_5$:(let rec ← (append tl l2) in
                    $\pi_6$:(let hd ← (car l1) in
                        $\pi_7$:(let ans ← (cons hd rec) in (return ans))))))))

$\pi_{\mathbf{main}}$: ···
    $\pi_8$:(let y ← (append a b) in
        $\pi_9$:(let w ← (append y z) in
            $\pi_{10}$: (let c ← (cdr w) in
                $\pi_{11}$: (let d ← (car c) in (return d)))))))
```

Fig. 5. An example program. GC points are π_6, π_7, π_9 and π_{10}.

4.1 Generating equations for the demand transformer LF

We shall consider the program in Figure 5 as a running example. Unlike the program in Figure 1, this program is in ANF.

To generate the equations defining LF_f, we follow the rule DEFINE-LIVE. We start with a *symbolic demand* σ and determine $\mathsf{L} = \mathcal{L}(e_f, \sigma, \mathsf{LF})$, treating LF as an uninterpreted function symbol. We then generate equations of the form $\mathsf{LF}^i_f(\sigma) = \mathsf{L}(x_i)$ where x_i is the ith formal parameter of f and $\mathsf{L}(x_i)$ is the liveness of x_i. For our example program which has a single function **append**, this generates the following equations:

$$\mathsf{LF}^1_{\mathbf{append}}(\sigma) = \{\varepsilon\} \cup \mathbf{0}\bar{\mathbf{0}}\sigma \cup \mathbf{1}\mathsf{LF}^1_{\mathbf{append}}(\bar{\mathbf{1}}\sigma)$$
$$\mathsf{LF}^2_{\mathbf{append}}(\sigma) = \sigma \cup \mathsf{LF}^2_{\mathbf{append}}(\bar{\mathbf{1}}\sigma)$$

In general, the equations for LF are recursive since L may, in turn, be expressed in terms of LF. We assume that LF_f is expressible in the closed form as:[8]

$$\mathsf{LF}^i_f(\sigma) = \mathsf{I}^i_f \cup \mathsf{D}^i_f\sigma \qquad (1)$$

where I^i_f and D^i_f are sets of strings over the alphabet $\{\mathbf{0}, \mathbf{1}, \bar{\mathbf{0}}, \bar{\mathbf{1}}\}$. The reason why LF has this form is as follows. Recall that $\mathsf{LF}^i_f(\sigma)$ gives the access paths starting from i that have to be dereferenced to produce the sub-structure σ of the result of f. I^i_f represents the access paths that would be dereferenced, but do not contribute to the result. This happens, for instance, when the argument is used only within the condition of an **if**. D^i_f, in contrast, represents the paths that are dereferenced to actually produce the result.

[8] This is similar to solving the differential equation $ay'' + by' + c = 0$, where we guess that the solution has the form $y = e^{rx}$. Substituting the solution in the equation yields a quadratic equation in r, and each solution of r gives rise to a solution of the differential equation (in our setup we can effectively pick the least solution rather than needing linear combinations).

To solve for LF_f, we substitute the guessed form into its equations. $\mathsf{LF}_{\mathbf{append}}$ gives:

$$\mathsf{I}^1_{\mathbf{append}} \cup \mathsf{D}^1_{\mathbf{append}}\sigma = \{\varepsilon\} \cup \mathbf{0}\bar{\mathbf{0}}\sigma \cup \mathbf{1}(\mathsf{I}^1_{\mathbf{append}} \cup \mathsf{D}^1_{\mathbf{append}}\bar{\mathbf{1}}\sigma)$$
$$\mathsf{I}^2_{\mathbf{append}} \cup \mathsf{D}^2_{\mathbf{append}}\sigma = \sigma \cup \mathsf{I}^2_{\mathbf{append}} \cup \mathsf{D}^2_{\mathbf{append}}\bar{\mathbf{1}}\sigma$$

Equating the terms containing σ on the two sides of each equation, and doing the same for the terms without σ, we get equations for I^i_f and D^i_f that are independent of σ.

$$\mathsf{I}^1_{\mathbf{append}} = \{\varepsilon\} \cup \mathbf{1}\mathsf{I}^1_{\mathbf{append}} \qquad \mathsf{I}^2_{\mathbf{append}} = \mathsf{I}^2_{\mathbf{append}}$$
$$\mathsf{D}^1_{\mathbf{append}} = \{\mathbf{0}\bar{\mathbf{0}}\} \cup \mathbf{1}\mathsf{D}^1_{\mathbf{append}}\bar{\mathbf{1}} \qquad \mathsf{D}^2_{\mathbf{append}} = \{\varepsilon\} \cup \mathsf{D}^2_{\mathbf{append}}\bar{\mathbf{1}}$$

Note that these equations can be viewed as CFGs, with all but $\mathsf{D}^1_{\mathbf{append}}$ being regular, and that any solution of I^i_f and D^i_f yields a solution of LF_f.

4.2 Generating liveness equations L for function bodies

We now calculate a 0-CFA-style summary liveness for each GC point of a program. There are two parts to this. First, for each function f, we determine a *summary demand* σ_f over-approximating any demand σ passed to f. Such demands are caused by calls to f occurring at call sites. We introduce the notation $\delta_f(\pi, g)$ for the contribution to σ_f caused a call site π occurring in function g. So, suppose function g contains a call site π to f, say π: (**let** $x \leftarrow (f\, y_1 \ldots y_n)$ **in** e). Under the assumption that the demand on g is σ_g, the liveness at e is $\mathsf{L} = \mathcal{L}(e, \sigma_g, \mathsf{LF})$, and the **let** case of Figure 4 tells us this call site contributes $\mathsf{L}(x)$ to the demand σ_f placed on f; hence $\delta_f(\pi, g)$ is simply $\mathsf{L}(x)$.

Now, supposing the k call sites to function f are π^1 (in function g^1) $\ldots\pi^k$ (in function g^k), then the over-approximation requirement on σ_f is achieved by taking $\sigma_f = \delta_f(\pi^1, g^1) \cup \cdots \cup \delta_f(\pi^k, g^k)$.

The expression $e_{\mathbf{main}}$ is a special case; we assume it may be called externally with demand $\sigma_{\mathbf{main}} = \{\mathbf{0}, \mathbf{1}\}^*$ (denoted σ_{all}). This is because any part of its value may be used by the environment—for printing the result, for instance.

For the running example, **append** has calls from **main** at π_9 and a recursive call at π_5. So $\sigma_{\mathbf{append}} = \delta_{\mathbf{append}}(\pi_9, \mathbf{main}) \cup \delta_{\mathbf{append}}(\pi_5, \mathbf{append})$. Calculating the $\delta_{\mathbf{append}}(\pi, g)$ for the two call sites, and substituting gives:

$$\sigma_{\mathbf{append}} = (\{\varepsilon, \mathbf{1}\} \cup \mathbf{10}\sigma_{all}) \;\cup\; \bar{\mathbf{1}}\sigma_{\mathbf{append}}$$

Second, for each function f (possibly **main**) we need the liveness at each contained GC point π. Given σ_f calculated above, this is simply $\mathcal{L}(\pi, \sigma_f, \mathsf{LF})$. For the running example, containing GC points π_6, π_7 in **append** and π_9, π_{10} in $e_{\mathbf{main}}$, this gives (recall Equation (1) above states $\mathsf{LF}^i_f(\sigma) = \mathsf{I}^i_f \cup \mathsf{D}^i_f.\sigma$):

$$\mathsf{L}^{\mathtt{l1}}_6 = \{\varepsilon\} \cup \mathbf{0}\bar{\mathbf{0}}\sigma_{\mathbf{append}} \qquad \mathsf{L}^{\mathtt{rec}}_6 = \bar{\mathbf{1}}\sigma_{\mathbf{append}}$$
$$\mathsf{L}^{\mathtt{hd}}_7 = \bar{\mathbf{0}}\sigma_{\mathbf{append}} \qquad \mathsf{L}^{\mathtt{rec}}_7 = \bar{\mathbf{1}}\sigma_{\mathbf{append}}$$

$$\mathsf{L}^{\mathtt{y}}_9 = \mathsf{LF}^1_{\mathbf{append}}(\{\varepsilon, \mathbf{1}\} \cup \mathbf{10}\sigma_{all}) \qquad \mathsf{L}^{\mathtt{z}}_9 = \mathsf{LF}^2_{\mathbf{append}}(\{\varepsilon, \mathbf{1}\} \cup \mathbf{10}\sigma_{all})$$
$$\mathsf{L}^{\mathtt{w}}_{10} = \{\varepsilon, \mathbf{1}\} \cup \mathbf{10}\sigma_{all}$$

In summary, the equations generated during liveness analysis are:

1. For each function f, equations defining l_f^i and 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.

4.3 Solving liveness equations—the grammar interpretation

The liveness equations above (of the form $X = \ldots$) can now be re-interpreted as a context-free grammar (CFG) on the alphabet $\{\mathbf{0},\mathbf{1},\bar{\mathbf{0}},\bar{\mathbf{1}}\}$. We use $\langle \mathrm{X}\rangle$ to denote the corresponding non-terminal which then appears in a production $\langle \mathrm{X}\rangle \to \ldots$. 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 $\langle \mathsf{L}_9^{\mathsf{y}}\rangle$ comprises the following productions:

$$\begin{aligned}
\langle \mathsf{L}_9^{\mathsf{y}}\rangle &\to \langle \mathsf{l}_{\mathbf{append}}^1\rangle \mid \langle \mathsf{D}_{\mathbf{append}}^1\rangle(\varepsilon \mid \mathbf{1} \mid \mathbf{10}\langle \sigma_{all}\rangle)\\
\langle \mathsf{l}_{\mathbf{append}}^1\rangle &\to \varepsilon \mid \mathbf{1}\langle \mathsf{l}_{\mathbf{append}}^1\rangle\\
\langle \mathsf{D}_{\mathbf{append}}^1\rangle &\to \mathbf{0}\bar{\mathbf{0}} \mid \mathbf{1}\langle \mathsf{D}_{\mathbf{append}}^1\rangle\bar{\mathbf{1}}\\
\langle \sigma_{all}\rangle &\to \varepsilon \mid \mathbf{0}\langle \sigma_{all}\rangle \mid \mathbf{1}\langle \sigma_{all}\rangle
\end{aligned}$$

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, recall from our earlier discussion that the decision problem that we are interested in during garbage collection is:

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

We could convert the rules defining $\hookrightarrow$ into productions and add them to the grammar. However, this results in an *unrestricted grammar* [11], and the membership problem for such grammars is undecidable. We circumvent the problem by over-approximating the CFG generated by the analysis to *strongly regular* CFGs which have easy translations to non-deterministic finite state automata (NFA). The NFAs are then simplified on the basis of the $\hookrightarrow$ rules to enable checking of membership of forward access paths. The resulting NFAs are finally converted to DFAs for use during garbage collection.

Approximating CFGs using NFAs. We use the algorithm by Mohri and Nederhof [10] to approximate a CFG to a *strongly regular* grammar. The transformation has the property that if L is a non-terminal in the grammar G and G' is the grammar after the Mohri-Nederhof transformation, then $\mathscr{L}_G(\mathsf{L}) \subseteq \mathscr{L}_{G'}(\mathsf{L})$. This is required for the approximation to be safe with respect to liveness.

We exemplify the Mohri-Nederhof transformation on the $\langle \mathsf{L}_9^{\mathsf{y}}\rangle$ grammar above. We pick the only production that is affected by the transformation—the production for $\mathsf{D}_{\mathbf{append}}^1$. The production for $\mathsf{l}_{\mathbf{append}}^1$, while recursive, is already in strongly regular form and is therefore unaffected by the transformation.

$$\begin{aligned}
\langle \mathsf{D}_{\mathbf{append}}^1\rangle &\to \mathbf{0}\bar{\mathbf{0}}\langle \mathsf{D}_{\mathbf{append}}^1\rangle' \mid \mathbf{1}\langle \mathsf{D}_{\mathbf{append}}^1\rangle\\
\langle \mathsf{D}_{\mathbf{append}}^1\rangle' &\to \bar{\mathbf{1}}\langle \mathsf{D}_{\mathbf{append}}^1\rangle' \mid \varepsilon
\end{aligned}$$

```
Input: NFA $\overline{N}$ with underlying alphabet $\{\mathbf{0},\mathbf{1},\bar{\mathbf{0}},\bar{\mathbf{1}}\}$
Output: NFA $N$ with underlying alphabet $\{\mathbf{0},\mathbf{1}\}$ such that $\mathscr{L}(\overline{N}) \overset{*}{\hookrightarrow} \mathscr{L}(N)$.
Steps:
  $i \leftarrow 0$
  $N_0 \leftarrow$ Equivalent NFA of $\overline{N}$ without ε-moves [11]
  repeat
    $N'_{i+1} \leftarrow N_i$
    for all states $q$ in $N_i$ such that $q$ has an incoming edge from $q'$ with label $\bar{\mathbf{0}}$ and outgoing
    edge to $q''$ with label $\mathbf{0}$ do
      add an edge in $N'_{i+1}$ from $q'$ to $q''$ with label ε. {bypass $\bar{\mathbf{0}}\mathbf{0}$ using ε}
    end for
    for all states $q$ in $N_i$ such that $q$ has an incoming edge from $q'$ with label $\bar{\mathbf{1}}$ and outgoing
    edge to $q''$ with label $\mathbf{1}$ do
      add an edge in $N'_{i+1}$ from $q'$ to $q''$ with label ε. {bypass $\bar{\mathbf{1}}\mathbf{1}$ using ε}
    end for
    $N_{i+1} \leftarrow$ Equivalent NFA of $N'_{i+1}$ without ε-moves
    $i \leftarrow i+1$
  until $(N_i = N_{i-1})$
  $N \leftarrow N_i$
```

Fig. 6. Algorithm for transforming an NFA to accept forward paths only.

The languages generated for $\langle \mathsf{D}^1_{\textbf{append}} \rangle$ in the original grammar and the new grammar are $\mathbf{1}^i\mathbf{0}\bar{\mathbf{0}}\bar{\mathbf{1}}^i$ and $\mathbf{1}^*\mathbf{0}\bar{\mathbf{0}}\bar{\mathbf{1}}^*$, showing a loss of precision.

Transforming NFAs to accept forward paths: The strongly regular CFGs obtained after the Mohri-Nederhof transformation are first converted into NFAs. The algorithm described in Figure 6 converts an NFA $\overline{N}$ to a NFA N such that $\mathscr{L}(\overline{N}) \overset{*}{\hookrightarrow} \mathscr{L}(N)$, where N accepts forward paths only. Thus N can be used to check membership of forward paths.

The algorithm repeatedly introduces ε edges to bypass a pair of consecutive edges labelled $\bar{\mathbf{0}}\mathbf{0}$ or $\bar{\mathbf{1}}\mathbf{1}$. The process is continued until a fixed point is reached. When the fixed point is reached, the resulting NFA contains all possible reductions corresponding to all the paths in the original NFA. The proofs of the termination and correctness of the algorithm are given in our earlier paper [8].

We illustrate the algorithm in Figure 6 by constructing the automaton for $\langle \mathsf{L}^{\mathtt{y}}_9 \rangle$. Figure 7 (c) shows the automaton for $\langle \mathsf{L}^{\mathtt{y}}_9 \rangle$ constructed by composing the automata for $\langle \mathsf{I}^1_{\textbf{append}} \rangle$, $\langle \mathsf{D}^1_{\textbf{append}} \rangle$ and $(\varepsilon \mid \mathbf{1} \mid \mathbf{10}\sigma_{all})$. After ε removal we get (d). We add an ε edge from q_2 to q_3 bypassing the $\bar{\mathbf{1}}\mathbf{1}$ pair from q_2 to q_2 and then to q_3. This is shown in (e). The ε edge is removed in (f) and a second ε is added to the automaton bypassing the $\bar{\mathbf{0}}\mathbf{0}$ pair from q_1 to q_2. Removing this ε edge gives the automaton shown in (g). Restricting this automaton to forward edges only, we get the final automaton shown in (h). This automaton recognises $\mathbf{1}^* \mid \mathbf{1}^*\mathbf{0}\sigma_{all}$, showing that the entire list $\mathtt{y}$, including its elements, is live at π_9. Also note that the language accepted by the final automaton satisfies the prefix-closed property.

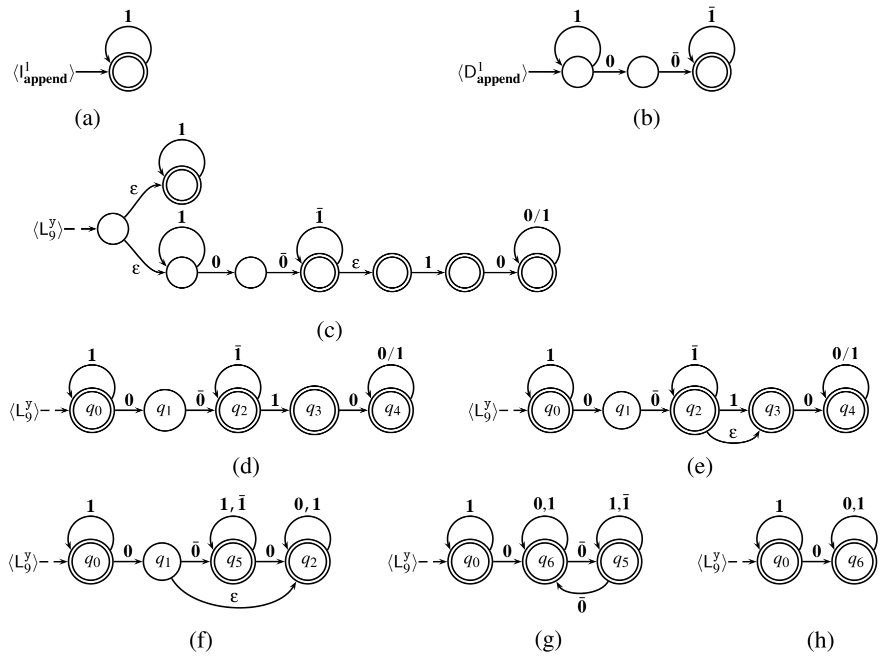


Fig. 7. Automata for the example program

5 Prototype and evaluation

To demonstrate the effectiveness of liveness-based garbage collection, we have built a prototype consisting of an interpreter for our language, a liveness analyser and a copying collector that can optionally use the results of liveness analysis for marking instead of reachability. When the collector uses liveness for marking, we call it a *liveness-based collector* (LGC), else we use the term *reachability-based collector* (RGC). The collector is neither incremental nor generational. As a consequence, any cell that becomes unreachable or dead is assuredly collected in the next round of garbage collection.

When LGC is invoked (by a call to **cons**) the activation records on the stack all correspond to functions suspended at GC points, and by construction at each GC point we have a DFA specifying liveness of each local variable in the activation record. As usual such local variables form the root set for garbage collection.

Let dfa^x_π denote the DFA for the variable and program point pair (x, π). We write $\mathsf{initial}(\mathsf{dfa}^x_\pi)$ for the initial state of dfa^x_π. Considering a DFA as a table, $\mathsf{dfa}^x_\pi(q, sym)$ returns the next state for the state q and the symbol *sym*, where *sym* is **0** or **1**. We shall also write $\mathsf{dfa}^x_\pi(q, sym)?$ for a predicate indicating whether there is a transition from q on *sym*. The LGC action to chase the root variable x at π can be described as follows: If $\mathscr{L}(\mathsf{dfa}^x_\pi)$ is empty, then nothing needs to be done. Otherwise we call $\mathsf{copy}(\mathsf{dfa}^x_\pi, \mathsf{initial}(\mathsf{dfa}^x_\pi), x)$ in Figure 8 and assign the returned pointer to x. The function $\mathsf{move\_to\_tospace}(x)$ copies the value of x in the other semi-space and returns the new

```
function copy(dfa, $q$, $x$)
    let $y \leftarrow$ move_to_tospace($x$)
    if $x$.tag $\neq$ cons then skip
    else if dfa($q$, 0)? then $y$.car = copy(dfa, dfa($q$, 0), $x$.car)
        if dfa($q$, 1)? then $y$.cdr = copy(dfa, dfa($q$, 1), $x$.cdr)
    return $y$
```

Fig. 8. Function for copying a root set variable.

address. It hides details such as returning the forwarding pointer if the value of x is already copied, and creating the forwarding pointer otherwise.

The graphs in Figure 9[9] show the number of cells in the heap over time for RGC and LGC—here time is measured in terms of the number of **cons** cells allocated. In addition, they also show the number of reachable cells and the number of cells that are actually live (this is statically approximated by our liveness analysis). Since the programs have different memory requirements, we have tuned the size of heap for each program to ensure a reasonable number of collections. An invocation of RGC decreases the number of cells in heap until it touches the curve of reachable cells. An invocation of LGC decreases the number of heap cells to no lower than the curve of live cells.

To construct the reachable and live curves, we record for every cell its creation time (`Create_time`), its last use time (`Use_time`), and the earliest time when the cell becomes unreachable and can be garbage collected (`Collection_time`). For accurate recording of `Collection_time`, we force frequent invocations of a reachability-based collector in a separate run. A cell is live at time T if `Create_time` $\leq$ T $\leq$ `Use_time`. If `Create_time` $\leq$ T $\leq$ `Collection_time`, it is reachable.

The benchmark programs are drawn from the `no-fib` suite and other sources and have been manually converted to ANF. All graphs except `fibheap` show strictly fewer garbage collector invocations for LGC; `fibheap` is an exception in that the number of reachable cells first grows steadily until it almost fills the heap. This triggers garbage collections in both LGC and RGC. The number of reachable cells then drops steeply to a low level and remains low resulting in no further garbage collections. The graphs also show the precision of our liveness analysis. For all programs except `nperm` and `lambda`, LGC manages to collect a good portion of the cells that are not live.

5.1 Results

The increased effectiveness of LGC over RGC is also shown in the tables in Figure 10. The first table provides statistics regarding the analysis itself. The number of states and the analysis times are within tolerable limits. Precision of analysis refers to the percentage of dead cells that is collected by LGC, averaged over all invocations. The second table shows garbage collection statistics for RGC and LGC. LGC collects larger garbage per invocation, drags cells for lesser time and requires a smaller heap size (*MinHeap*) for program to run in comparison with RGC.

[9] For better clarity, visit `http://www.cse.iitk.ac.in/users/karkare/fhra` for coloured versions of the graphs.

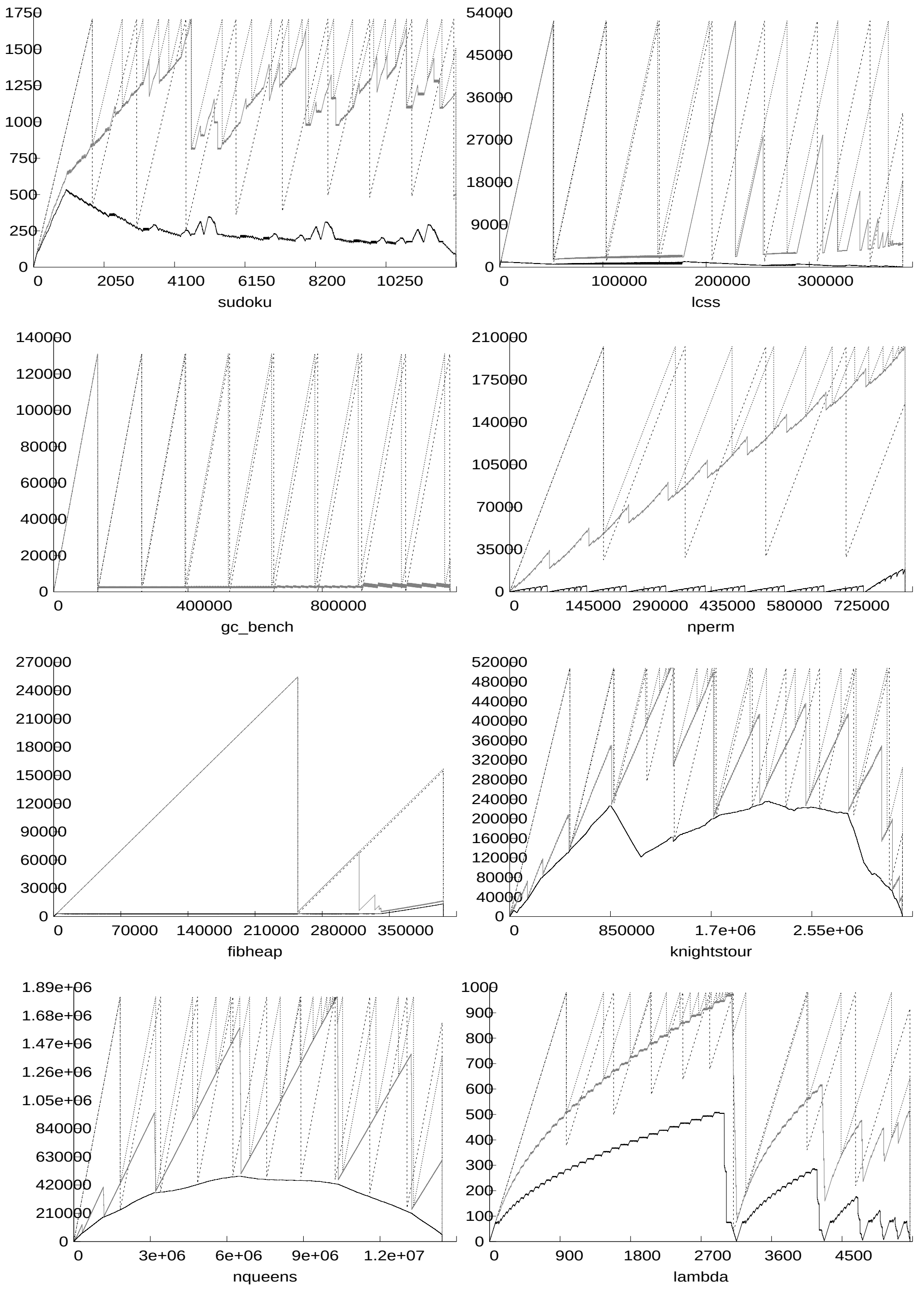


Fig. 9. Memory usage of programs. The dotted-grey and the dashed-black curves indicate the number of **cons** cells in the active semi-space for RGC and LGC respectively. The solid-grey curve represents the number of reachable cells and the solid-black curve represents the number of cells that are actually live (of which liveness analysis does a static approximation). x-axis is the time measured in number of **cons** cells allocated. y-axis is the number of **cons** cells.

Program	sudoku	lcss	gc_bench	nperm	fibheap	knightstour	treejoin	nqueens	lambda
Time (msec)	120.95	2.19	0.32	1.16	2.4	3.05	2.61	0.71	20.51
DFA size	4251	726	258	526	675	922	737	241	732
Precision(%)	87.5	98.8	99.9	87.1	100	94.3	99.6	98.8	83.8

(a)

	# Collected cells per GC		# Touched cells per GC		#GCs		MinHeap (#cells)		Avg. Drag (#cells)		GC time (sec)	
Program	RGC	LGC	RGC	LGC	RGC	LGC	RGC	LGC	RGC	LGC	RGC	LGC
sudoku	490	1306	1568	774	22	9	1704	589	858	146	.028	.122
lcss	46522	51101	6216	1363	8	7	52301	1701	5147	588	.045	.144
gc_bench	129179	131067	1894	4	9	9	131071	6	16970	4	.086	.075
nperm	47586	174478	201585	60882	14	4	202597	37507	171878	76618	1.406	.9
fibheap	249502	251525	5555	2997	1	1	254520	13558	78720	0	.006	.014
knightstour	2593	314564	907502	319299	1161	10	508225	307092	206729	82112	464.902	14.124
treejoin	288666	519943	297570	5547	2	1	525488	7150	212653	1954	.356	.217
nqueens	283822	1423226	2133001	584143	46	9	1819579	501093	521826	39465	70.314	24.811
lambda	205	556	2072	90345	23	8	966	721	303	95	.093	2.49

(b)

Fig. 10. Experimental results comparing RGC and LGC. Table (a) gives data related to liveness analysis, and (b) gives garbage collection data.

There are a couple of issues of concern. The garbage collection time is larger in the case of LGC for some programs. The reason is that the cost of consulting the liveness DFA may outweigh the combined benefits of fewer garbage collections and fewer markings per garbage collection. The other issue is illustrated by the program `lambda`. As can be seen from the table in Figure 10, the number of touched cells[10] in this example is much higher for LGC. This increase is due to excessive sharing among heap nodes in this program. Note that a node re-visited because of sharing is not explored any further during RGC. However, this curtailment cannot happen in LGC because of the possibility that the node, re-visited in a different liveness state, may mark a set of cells different from the earlier visit.

6 Collecting more garbage can never slow things down

Since garbage collection is effectively asynchronous to the allocator thread, one might worry as to how *robust* our measurements are. For example, while LGC would, in general, collect more garbage than RGC in the same heap state, might LGC do a larger number of collections for some programs? We prove below that this cannot happen. This result applies to classical mark-and-sweep and copying garbage collectors and, we believe, also to generational collectors.

Lemma 1. *For the same mutator, a liveness-based collector can never do more garbage collections than a reachability-based collector.*

Proof. Assume, as before, that time is measured in terms of the number of **cons** cells allocated. Now run two copies of the mutator, one with RGC and one with LGC, in

[10] These are the cells visited during the marking phase, often more than once due to sharing.

parallel. Memory allocations by **cons** happen simultaneously, but the times of garbage collections diverge.

To prove the lemma, it is enough to show the truth of the following statement: After every LGC invocation, the count of LGC invocations is no greater than RGC invocations. The base case holds since the first invocations of both GCs happen at the same time. Assume the statement to be true after n invocations of LGC. Since LGC copies a subset of reachable cells, its heap would contain no more cells than RGC heap at the end of the nth invocation. Thus either RGC is invoked next before LGC, or LGC and RGC are both invoked next at the same time. In either case, the statement holds after $n+1$ invocations of LGC.

7 Related Work

Previous attempts to increase the space efficiency of functional programs by additional reclamation of memory fall in two broad categories. In the first, the program itself is instrumented to manage reclamation and reallocation without the aid of the garbage collector. Such attempts include: sharing analysis based reallocation [12], deforestation techniques [13–15], methods based on linear logic [16] and region analysis [17]. Closer to our approach, there are methods that enable the garbage collector to collect more garbage [18, 6] by explicitly nullifying pointers that are not live. However, the nullification, done at compile time, requires sharing (alias) analysis. Our method, in contrast, does not require sharing because of the availability of the heap itself at run time. To the best of our knowledge, this is the first attempt at liveness-based marking of the heap during garbage collection.

8 Conclusions

We have defined a notion of liveness on structured data; this generalises classical liveness and strong liveness. We started with a general fully context-sensitive analysis which we proved correct with respect to a minefield semantics (this models the effect of garbage collection between every evaluation step).

To avoid scalability issues (and to avoid performing part of the liveness computation at run time) we defined an 0-CFA version of this liveness analysis in which demands for function f at all calling contexts are conflated into a single demand σ_f. This enabled us to treat the liveness equations symbolically obtaining context-free grammars for liveness at each GC point (calls to user functions and to **cons**). These were then converted to DFAs for run-time consultation by the garbage collector. Experiments confirm the precision of the analysis.

To obtain performance figures we compared a reachability-based garbage collector with a liveness-based collector. This showed a decrease in the number of GCs, more garbage collected per invocation. A significant benefit of LGC is that programs can run in smaller memory when compared to RGC. This is potentially useful in situations where memory is limited—as with embedded systems. For a majority of programs, the garbage collection times were reduced.

One issue we highlighted was that while fewer nodes were marked (and hence more garbage collected), sometimes **cons** cells could be visited and traversed multiple times with different sets of liveness paths to explore; this risks infinite looping if extended to languages with cyclic data structures. Avenues of further work include static analysis to avoid revisiting cells known to have been visited. One possibility is to record a representation of the liveness paths already visited from each **cons** cell; the classical mark bit indicates that all paths have been visited from the cell.

Acknowledgements We thank the anonymous referees for their helpful comments. Thanks are also due to Hemanshu Vadehra for a preliminary implementation of the prototype system. Amey Karkare was supported for this work by the DST/SERC fast track scheme for young scientists.

References

1. Röjemo, N., Runciman, C.: Lag, drag, void and use—heap profiling and space-efficient compilation revisited. In: ICFP. (1996)
2. Karkare, A., Sanyal, A., Khedker, U.: Effectiveness of garbage collection in MIT/GNU Scheme. `http://arxiv.org/abs/cs/0611093` (2006)
3. Shaham, R., Kolodner, E.K., Sagiv, M.: Estimating the impact of heap liveness information on space consumption in Java. In: ISMM. (2002)
4. Chatterjee, R., Ryder, B.G., Landi, W.A.: Relevant context inference. In: POPL. (1999)
5. Cherem, S., Rugina, R.: A practical escape and effect analysis for building lightweight method summaries. In: CC. (2007)
6. Lee, O., Yang, H., Yi, K.: Static insertion of safe and effective memory reuse commands into ML-like programs. Science of Computer Programming (2005)
7. Khedker, U.P., Sanyal, A., Karkare, A.: Heap reference analysis using access graphs. TOPLAS (2007)
8. Karkare, A., Khedker, U., Sanyal, A.: Liveness of heap data for functional programs. In: Heap Analysis and Verification Workshop (HAV). (2007) `http://research.microsoft.com/~jjb/papers/HAV_proceedings.pdf`.
9. Chakravarty, M.M.T., Keller, G., Zadarnowski, P.: A functional perspective on SSA optimisation algorithms. In: COCV. (2003)
10. Mohri, M., Nederhof, M.J.: Regular approximation of context-free grammars through transformation. In Junqua, J.C., van Noord, G., eds.: Robustness in Language and Speech Technology. Kluwer Academic Publishers (2000) 251–261
11. Hopcroft, J.E., Ullman, J.D.: Introduction To Automata Theory, Languages, And Computation. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (1990)
12. Jones, S.B., Metayer, D.L.: Compile-time garbage collection by sharing analysis. In: FPCA. (1989)
13. Wadler, P.: Deforestation: transforming programs to eliminate trees. In: ESOP. (1988)
14. Gill, A., Launchbury, J., Jones, S.L.P.: A short cut to deforestation. In: FPCA. (1993)
15. Chitil, O.: Type inference builds a short cut to deforestation. In: ICFP. (1999)
16. Hofmann, M.: A type system for bounded space and functional in-place update. In: ESOP. (2000)
17. Tofte, M., Birkedal, L.: A region inference algorithm. TOPLAS (1998)
18. Inoue, K., Seki, H., Yagi, H.: Analysis of functional programs to detect run-time garbage cells. TOPLAS (1988)