Heap Reference Analysis Using Access Graphs

UDAY P. KHEDKER, AMITABHA SANYAL and AMEY KARKARE
Department of Computer Science & Engg., IIT Bombay.

Despite significant progress in the theory and practice of program analysis, analysing properties of heap data has not reached the same level of maturity as the analysis of static and stack data. The spatial and temporal structure of stack and static data is well understood while that of heap data seems arbitrary and is unbounded. We devise bounded representations which summarize properties of the heap data. This summarization is based on the structure of the program which manipulates the heap. The resulting summary representations are certain kinds of graphs called *access graphs*. The boundedness of these representations and the monotonicity of the operations to manipulate them make it possible to compute them through data flow analysis.

An important application which benefits from heap reference analysis is garbage collection, where currently liveness is conservatively approximated by reachability from program variables. As a consequence, current garbage collectors leave a lot of garbage uncollected, a fact which has been confirmed by several empirical studies. We propose the first ever end-to-end static analysis to distinguish live objects from reachable objects. We use this information to make dead objects unreachable by modifying the program. This application is interesting because it requires discovering data flow information representing complex semantics. In particular, we discover four properties of heap data: liveness, aliasing, availability, and anticipability. Together, they cover all combinations of directions of analysis (i.e. forward and backward) and confluence of information (i.e. union and intersection). Our analysis can also be used for plugging memory leaks in C/C++ languages.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*Memory management (garbage collection)*; *Optimization*; F.3.2 [**Logics and Meanings Of Programs**]: Semantics of Programming Languages—*Program analysis*

General Terms: Algorithms, Languages, Theory

Additional Key Words and Phrases: Aliasing, Data Flow Analysis, Heap References, Liveness

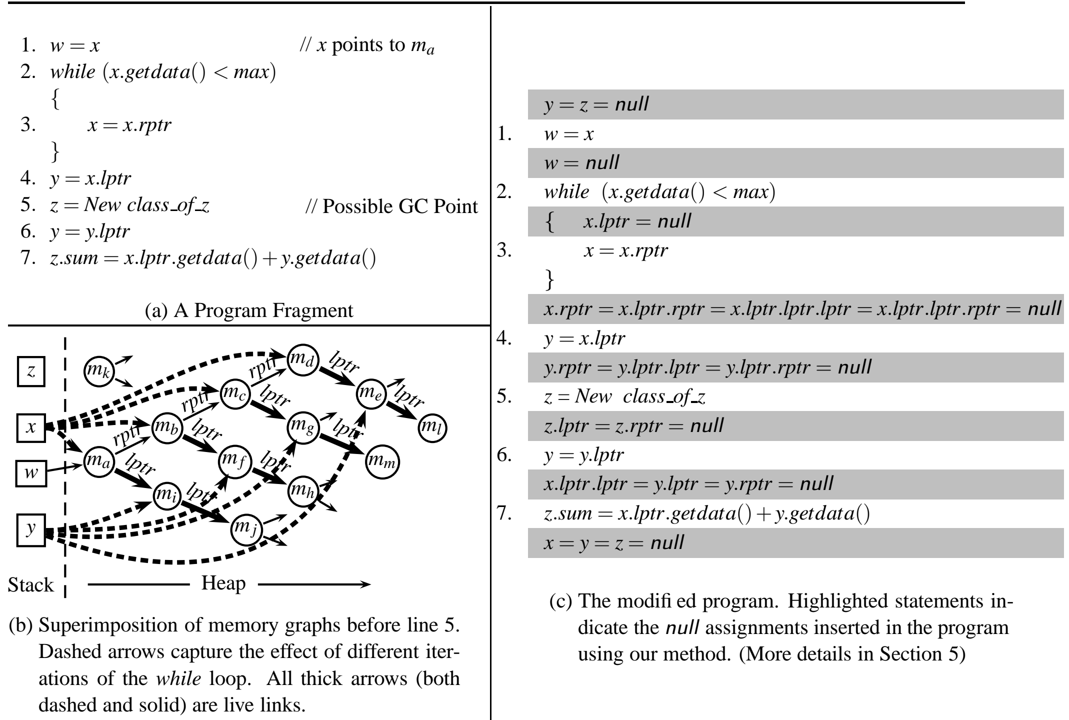
1. INTRODUCTION

Conceptually, data in a program is allocated in either the static data area, stack, or heap. Despite significant progress in the theory and practice of program analysis, analysing the properties of heap data has not reached the same level of maturity as the analysis of static and stack data. Section 1.2 investigates possible reasons.

In order to facilitate a systematic analysis, We devise bounded representations which summarize properties of the heap data. This summarization is based on the structure of the program which manipulates the heap. The resulting summary representations are certain kinds of graphs, called access graphs which are obtained through data flow analysis. We believe that our technique of summarization is general enough to be also used in contexts other than heap reference analysis.

1.1 Improving Garbage Collection through Heap Reference Analysis

An important application which benefits from heap reference analysis is garbage collection, where liveness of heap data is conservatively approximated by reachability. This amounts to approximating the future of an execution with its past. Since current garbage collectors



(a) A Program Fragment

(b) Superimposition of memory graphs before line 5. Dashed arrows capture the effect of different iterations of the *while* loop. All thick arrows (both dashed and solid) are live links.

(c) The modifi ed program. Highlighted statements indicate the *null* assignments inserted in the program using our method. (More details in Section 5)

Fig. 1. A motivating example.

cannot distinguish live data from data that is reachable but not live, they leave a lot of garbage uncollected. This has been confirmed by empirical studies [Hirzel et al. 2002; Hirzel et al. 2002; Shaham et al. 2001b; 2001a; 2002] which show that a large number (24% to 76%) of heap objects which are reachable at a program point are actually not accessed beyond that point. In order to collect such objects, we perform static analyses to make dead objects unreachable by setting appropriate references to *null*. The idea that doing so would facilitate better garbage collection is well known as "Cedar Mesa Folk Wisdom" [Gadbois et al.]. The empirical attempts at achieving this have been [Shaham et al. 2001a; 2002].

Garbage collection is an interesting application for us because it requires discovering data flow information representing complex semantics. In particular, we need to discover four properties of heap references: liveness, aliasing, availability, and anticipability. Liveness captures references that may be used beyond the program point under consideration. Only the references that are not live can be considered for *null* assignments. Safety of *null* assignments further requires (a) discovering all possible ways of accessing a given heap memory cell (aliasing), and (b) ensuring that the reference being nullified is accessible (availability and anticipability).

For simplicity of exposition, we present our method in a setting similar to Java. Extensions required for handling C/C++ model of heap usage are easy and are explained in Appendix C.

We assume that root variable references are on the stack and the actual objects cor-

responding to the root variables are in the heap. In the rest of the paper we ignore non-reference variables. We view the heap at a program point as a directed graph called *memory graph*. Root variables form the entry nodes of a memory graph. Other nodes in the graph correspond to objects on the heap and edges correspond to references. The out-edges of entry nodes are labeled by root variable names while out-edges of other nodes are labeled by field names. The edges in the memory graph are called *links*.

Example 1.1. Figure 1 shows a program fragment and its memory graphs before line 5. Depending upon the number of times the *while* loop is executed x points to m_a, m_b, m_c etc. Correspondingly, y points to m_i, m_f, m_g etc. The call to *New* on line 5 may require garbage collection. A conventional copying collector will preserve all nodes except m_k. However, only a few of them are used beyond line 5.

The modified program is an evidence of the strength of our approach. It makes the unused nodes unreachable by nullifying relevant links. The modifications in the program are general enough to nullify appropriate links for any number of iterations of the loop. Observe that a *null* assignment has also been inserted within the loop body thereby making some memory unreachable in each iteration of the loop. □

After such modifications, a garbage collector will collect a lot more garbage. Further, since copying collectors process only live data, garbage collection by such collectors will be faster. Both these facts are corroborated by our empirical measurements (Section 8).

In the context of C/C++, instead of setting the references to *null*, allocated memory will have to be explicitly deallocated after checking that no alias is live.

1.2 Difficulties in Analysing Heap Data

A program accesses data through expressions which have l-values and hence are called *access expressions*. They can be scalar variables such as x, or may involve an array access such as $a[2*i]$, or can be a reference expression such as $x.l.r$.

An important question that any program analysis has to answer is: *Can an access expression* α_1 *at program point* p_1 *have the same l-value as* α_2 *at program point* p_2*?* Note that the access expressions or program points could be identical. The precision of the analysis depends on the precision of the answer to the above question.

When the access expressions are simple and correspond to scalar data, answering the above question is often easy because, the mapping of access expressions to l-values remains fixed in a given scope throughout the execution of a program. However in the case of array or reference expressions, the mapping between an access expression and its l-value is likely to change during execution. From now on, we shall limit our attention to reference expressions, since these are the expressions that are primarily used to access the heap. Observe that manipulation of the heap is nothing but changing the mapping between reference expressions and their l-values. For example, in Figure 1, access expression *x.lptr* refers to m_i when the execution reaches line number 2 and may refer to m_i, m_f, m_g, or m_e at line 4.

This implies that, subject to type compatibility, any access expression can correspond to any heap data, making it difficult to answer the question mentioned above. The problem is compounded because the program may contain loops implying that the same access expression appearing at the same program point may refer to different l-values at different points of time. Besides, the heap data may contain cycles, causing an infinite number of access expressions to refer to the same l-value. All these make analysis of programs involving heaps difficult.

1.3 Contributions of This Paper

The contributions of this paper fall in the following two categories

—*Contributions in Data Flow Analysis.* We present a novel data flow framework in which the data flow values represent complex semantics going beyond the traditional bit-vectors/tuples. An interesting aspect of our method is the way we obtain bounded representations of the properties by using the structure of the program which manipulates the heap. As a consequence of this summarization, the values of data flow information constitute a complete lattice with finite height. Further, we have carefully identified a set of monotonic operations to manipulate this data flow information. Hence, the standard results of data flow analysis can be extended to heap reference analysis. Due to the generality of this approach, it can be applied to other analyses as well.

—*Contributions in Heap Data Analysis.* We propose the first ever end-to-end solution (in the intraprocedural context) for statically discovering heap references which can be made *null* to improve garbage collection. The only approach which comes close to our approach is the *heap safety automaton* based approach [Shaham et al. 2003]. However, our approach is superior to their approach in terms of completeness, effectiveness, and efficiency (details in Section 9.3).

The concept which unifies the contributions is the summarization of heap properties which uses the fact that *the heap manipulations consist of repeating patterns which bear a close resemblance to the program structure.* Our approach to summarization is more natural and more precise than other approaches because it does not depend on an a-priori bound [Jones and Muchnick 1979; 1982; Larus and Hilfinger 1988; Chase et al. 1990].

1.4 Organisation of the paper

The rest of the paper is organized as follows. Section 2 defines the relevant properties of heap references in terms of sets of access paths. Section 3 defines access graphs, which are finite representations of sets of access paths. Section 4 defines the data flow analyses while Section 5 explains how *null* assignments are inserted. Section 6 discusses convergence and complexity issues. Section 7 shows the soundness of our approach. Section 8 presents empirical results. Section 9 reviews related work. Appendix A presents the extensions for handling cycles in heap. Appendix B discusses distributivity properties of our analyses.

2. HEAP REFERENCES AND THEIR PROPERTIES

Our method discovers live links at each program point, i.e., links which may be used in the program beyond the point under consideration. Links which are not live can be set to *null*. In order to discover liveness and other properties, we need a way of naming links in the memory graph. We do this using access paths.

2.1 Access Paths

An *access path* is a root variable name followed by a sequence of zero or more field names and is denoted by $\rho_x \equiv x \rightarrow f_1 \rightarrow f_2 \rightarrow \cdots \rightarrow f_k$. Since an access path represents a path in a memory graph, it can be used for naming links and nodes. An access path consisting of just a root variable name is called a *simple* access path; it represents a path consisting of a single link corresponding to the root variable. $\mathcal{E}$ denotes an empty access path.

The last field name in an access path ρ is called its *frontier* and is denoted by $\mathit{Frontier}(\rho)$. The frontier of a simple access path is the root variable name. The access path correspond-

ing to the longest sequence of names in ρ excluding its frontier is called its *base* and is denoted by $\mathit{Base}(\rho)$. Base of a simple access path is $\mathcal{E}$. The object reached by traversing an access path ρ is called the *target* of the access path and is denoted by $\mathit{Target}(\rho)$. When we use an access path ρ to refer to a link in a memory graph, it denotes the last link in ρ, i.e. the link corresponding to $\mathit{Frontier}(\rho)$.

Example 2.1. As explained earlier, Figure 1(b) is the superimposition of memory graphs that can result before line 5 for different executions of the program. For $\rho_x \equiv x \rightarrowtriangle lptr \rightarrowtriangle lptr$, depending on whether the *while* loop is executed 0, 1, 2, or 3 times, $\mathit{Target}(\rho_x)$ denotes nodes m_j, m_h, m_m, or m_l. $\mathit{Frontier}(\rho_x)$ denotes one of the links $m_i \rightarrow m_j$, $m_f \rightarrow m_h$, $m_g \rightarrow m_m$ or $m_e \rightarrow m_l$. $\mathit{Base}(\rho_x)$ represents the following paths in the heap memory: $x \rightarrow m_a \rightarrow m_i$, $x \rightarrow m_b \rightarrow m_f$, $x \rightarrow m_c \rightarrow m_g$ or $x \rightarrow m_d \rightarrow m_e$. □

In the rest of the paper, α denotes an access expression, ρ denotes an access path and σ denotes a (possibly empty) sequence of field names separated by $\rightarrowtriangle$. Let the access expression α_x be $x.f_1.f_2 \ldots f_n$. Then, the corresponding access path ρ_x is $x \rightarrowtriangle f_1 \rightarrowtriangle f_2 \ldots f_n$. When the root variable name is not required, we drop the subscripts from α_x and ρ_x.

2.2 Program Flow Graph

In this paper we restrict ourselves to intraprocedural analysis only. Besides, the current version of analysis does not cover programs containing arrays and threads. For the purpose of analysis, exception handling can be modeled by explicating possible control flows. We assume that the program flow graph has a unique *Entry* and a unique *Exit* node. Further, each statement forms a basic block[1] and falls in one of the following categories:

—*Assignment Statements*. These are assignments to references and are denoted by $\alpha_x = \alpha_y$. Only these statements can modify the structure of the heap.

—*Use Statements*. These statements use heap references to access heap data but do not modify heap references. For the purpose of analysis, these statements are abstracted as lists of expressions α_y or $\alpha_y.d$ where α_y is an access expression and d is a non-reference.

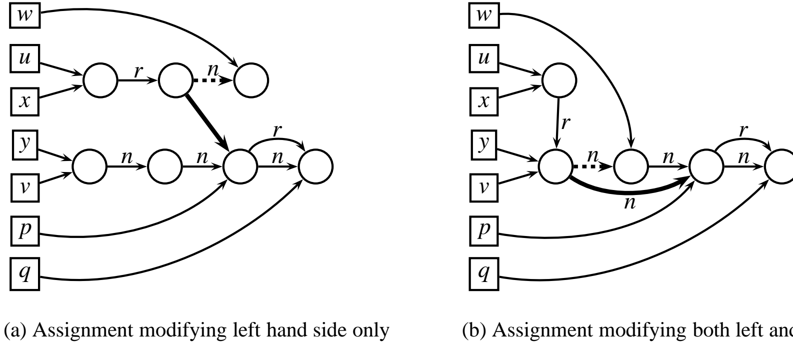
—*Other Statements*. These statements include all statements which do not refer to the heap. We ignore these statements since they do not influence heap reference analysis.

A *path* ψ in a program flow graph is a sequence of program points $p_1 \rightarrow p_2 \rightarrow \ldots \rightarrow p_k$ such that for $1 \le i \le k$ there is a direct transfer of control from p_i to p_{i+1} (i.e. p_i and p_{i+1} are adjacent program points). For $\psi = p_1 \rightarrow \ldots \rightarrow p_k$, $\psi' = \psi \bullet p_k \rightarrow p_{k+1}$ denotes that ψ is the prefix of ψ' ($\psi' = p_1 \rightarrow \ldots \rightarrow p_k \rightarrow p_{k+1}$), and $\psi'' = p_0 \rightarrow p_1 \bullet \psi$ denotes that ψ is the suffix of ψ'' ($\psi'' = p_0 \rightarrow p_1 \rightarrow \ldots \rightarrow p_k$).

We assume that the conditions that alter flow of control are made up only of simple variables. If not, the offending reference expression is assigned to a fresh simple variable before the condition and is replaced by the fresh variable in the condition. When we talk about the execution path, we shall refer to the execution of the program derived by retaining all assignment and use statements and ignoring the condition checks in the path.

For simplicity of exposition, we present the analyses assuming that there are no cycles in the heap. Handling cycles is easy and the required extensions are discussed in Appendix A. These extensions have been incorporated in our prototype implementation (Section 8).

[1]This is only for simplicity of exposition. As explained in Section 6.2, multiple statements can be grouped together in a larger basic block.



(a) Assignment modifying left hand side only

(b) Assignment modifying both left and right hand sides

Fig. 2. Effect of assignment $x.r.n = y.n.n$. The dotted and the thick arrows represents the links before and after the assignment.

2.3 Aliasing of Access Paths

Two access paths ρ_x and ρ_y are *aliased* at a program point p if $\mathit{Target}(\rho_x)$ is same as $\mathit{Target}(\rho_y)$ at p during some execution of the program. They are *link-aliased* if their frontiers represent the same link; they are *node-aliased* if they are aliased but their frontiers do not represent the same link. Link-aliases can be derived from node-aliases (or other link-aliases) by adding the same field names to aliased access paths.

We compute *flow-sensitive* aliases i.e. the aliases at a program point depend on the statements along control flow paths reaching the point. Two access paths are *must-aliased* at p if they are aliased along every control flow path reaching p; they are *may-aliased* if they are aliased along some control flow path reaching p. As an example, in Figure 1, $x \rightarrowtriangle lptr$ and y are must-node-aliases, $x \rightarrowtriangle lptr \rightarrowtriangle lptr$ and $y \rightarrowtriangle lptr$ are must-link-aliases, and w and x are node-aliases at line 5. Alias relations have the following properties:

	Node-aliasing	Link-aliasing
May-aliasing	Symmetry	Symmetry and Reflexivity
Must-aliasing	Symmetry and Transitivity	Symmetry, Reflexivity, and Transitivity

Note that since node-aliases cannot share frontier, they are not reflexive.

In the sequel, by alias we shall mean may-alias; a must-alias will be mentioned explicitly.

Example 2.2. Figure 2 shows the effect of assignment $x.r.n = y.n.n$ on the memory graph. We record node-aliases only and denote the aliasing between ρ_x and ρ_y by $\langle \rho_x, \rho_y \rangle$. The aliases after the assignment in Figure 2(a) are computed as follows:

(1) Since this assignment changes $\mathit{Frontier}(x \rightarrowtriangle r \rightarrowtriangle n)$, any alias involving an access path with a prefix that is must-link-aliased to $x \rightarrowtriangle r \rightarrowtriangle n$ is killed. Thus, assuming that $\langle x, u \rangle$ is a must-alias, the alias pairs $\langle x \rightarrowtriangle r \rightarrowtriangle n, w \rangle$ and $\langle u \rightarrowtriangle r \rightarrowtriangle n, w \rangle$ are both killed.

(2) As a direct effect of the assignment, all link-aliases of $x \rightarrowtriangle r \rightarrowtriangle n$ are aliased with all link-aliases of $y \rightarrowtriangle n \rightarrowtriangle n$. This generates the aliases $\langle x \rightarrowtriangle r \rightarrowtriangle n, y \rightarrowtriangle n \rightarrowtriangle n \rangle$, $\langle u \rightarrowtriangle r \rightarrowtriangle n, y \rightarrowtriangle n \rightarrowtriangle n \rangle$, $\langle x \rightarrowtriangle r \rightarrowtriangle n, v \rightarrowtriangle n \rightarrowtriangle n \rangle$ and $\langle u \rightarrowtriangle r \rightarrowtriangle n, v \rightarrowtriangle n \rightarrowtriangle n \rangle$.

(3) Any alias involving $y \rightarrowtriangle n \rightarrowtriangle n \rightarrowtriangle \sigma$ will generate aliases formed by replacing $y \rightarrowtriangle n \rightarrowtriangle n$ by $x \rightarrowtriangle r \rightarrowtriangle n$ or any of its link-aliases. Thus the existing alias $\langle y \rightarrowtriangle n \rightarrowtriangle n \rightarrowtriangle n,\ q \rangle$ will generate aliases $\langle x \rightarrowtriangle r \rightarrowtriangle n \rightarrowtriangle n,\ q \rangle$ and $\langle u \rightarrowtriangle r \rightarrowtriangle n \rightarrowtriangle n,\ q \rangle$. On the other hand, the alias

$\langle y{\rightarrow}n{\rightarrow}n{\rightarrow}n,\ y{\rightarrow}n{\rightarrow}n{\rightarrow}r\rangle$, which involves the same root variable unlike the previous alias, gives rise to the following aliases involving x:
—$\langle x{\rightarrow}r{\rightarrow}n{\rightarrow}n,\ y{\rightarrow}n{\rightarrow}n{\rightarrow}r\rangle$ (replacing the $y{\rightarrow}n{\rightarrow}n$ in the first component),
—$\langle y{\rightarrow}n{\rightarrow}n{\rightarrow}n,\ x{\rightarrow}r{\rightarrow}n{\rightarrow}r\rangle$ (replacing $y{\rightarrow}n{\rightarrow}n$ in the second component), and
—$\langle x{\rightarrow}r{\rightarrow}n{\rightarrow}n,\ x{\rightarrow}r{\rightarrow}n{\rightarrow}r\rangle$ (replacing $y{\rightarrow}n{\rightarrow}n$ in both components).
There will be a similar set of aliases involving u instead of x (and v instead of y).

The aliases for the assignment in Figure 2(b) are computed much in the same way except that alias pairs involving $y{\rightarrow}n{\rightarrow}n$ (and its must-link-aliases) will not be generated. □

We now define node-alias relations by generalizing the above observations. Since aliasing is influenced by assignments alone, we ignore other statements.

Definition 2.3. **May-Node-Aliases**. Let AP_p denote set of node-alias pairs at program point p. We define AP_p as follows:

(1) $AP_{Entry} = \emptyset$, where *Entry* denotes the program entry.

(2) If p is not *Entry*, then consider a control flow path $\psi = Entry \rightarrow \ldots \rightarrow p' \rightarrow p$. Assume that in this path, p follows an assignment $\alpha_x = \alpha_y$ and p' is the program point immediately before the assignment. Define *LhsPaths* and *RhsPaths* as the set of access paths which are link-aliased to ρ_x and ρ_y respectively, at p'. Define *KillPaths* as the set of access paths which have a prefix which is must-link-aliased to ρ_x at p'. Then,
   (a) Define AP_{Kill} as alias pairs where either one or both components are in *KillPaths*.
   (b) Define AP_{Direct} as $LhsPaths \times (RhsPaths - KillPaths)$. If α_y is *New* ... or *null*, define AP_{Direct} as $\emptyset$.
   (c) Define $AP_{Transfer}$ as union of the set of alias pairs
      i. $\langle \rho_z,\ \rho_u{\rightarrow}\sigma\rangle$ such that $\rho_u \in LhsPaths$, $\rho_z \notin KillPaths$, and $\langle \rho_z,\ \rho_y{\rightarrow}\sigma\rangle \in AP_{p'}$.
      ii. $\langle \rho_u{\rightarrow}\sigma_1, \rho_v{\rightarrow}\sigma_2\rangle$ such that $\rho_u, \rho_v \in LhsPaths$, and $\langle \rho_y{\rightarrow}\sigma_1,\ \rho_y{\rightarrow}\sigma_2\rangle \in AP_{p'}$. Note that, ρ_u and ρ_v can also be same.
      If α_y is *New* ... or *null*, define $AP_{Transfer}$ as $\emptyset$.

   For a given set X of node alias pairs, define

$$AF_\psi(X) \;=\; \begin{cases} X & \psi \text{ is empty} \\ (AF_{\psi'}(X) - AP_{Kill}) \cup AP_{Direct} \cup AP_{Transfer} & \text{otherwise}^{2} \end{cases} \qquad (1)$$
$$\text{where } \psi \;=\; \psi' \bullet p' \rightarrow p$$

   AP_p is defined as the union of alias sets $AF_\psi(\emptyset)$ along all control flow paths ψ from *Entry* to p, closed under symmetry. □

2.4 Liveness of Access Paths

A link l is *live* at a program point p if it is used in some control flow path starting from p. Note that l may be used in two different ways. It may be dereferenced to access an object or tested for comparison. An erroneous nullification of l would affect the two uses in different ways: Dereferencing l would result in an exception being raised whereas testing l for comparison may alter the result of condition and thereby the execution path.

Figure 1(b) shows links that are live before line 5 by thick arrows. For a link l to be live, there must be at least one access path from some root variable to l such that every link in this path is live. This is the path that is actually traversed while using l.

[2]Note that the values of AP_{Kill}, AP_{Direct} and $AP_{Transfer}$ depend upon information at p', i.e. $AF_{\psi'}(X)$.

An access path is defined to be *live* at p if the link corresponding to its frontier is possibly live along some path starting at p. We distinguish between the following:

—*Explicit liveness*. An access path is *explicitly* live at p if the liveness of its frontier depends solely on the execution of control flow paths from p to Exit. A formal specification of explicit liveness is provided in Definition 2.6.

—*Implicit liveness*. An access path is *implicitly* live at p if the liveness of its frontier depends not only on the execution of paths from p to Exit but also on the execution of paths from Entry to p due to aliasing. Implicitly live access paths are discovered by computing may-link-aliases of explicitly live access path. They can be viewed as alternative names for the links represented by the frontiers of explicitly live access paths.

Example 2.4. If the body of the *while* loop in Figure 1(a) is not executed even once, $\mathit{Target}(y) = m_i$ at line 5 and the link $m_i \rightarrow m_j$ is live at line 5 because it is used in line 6. The access path along which this link is used is $y{\rightarrowtriangle}lptr$, and therefore both the access paths y and $y{\rightarrowtriangle}lptr$ are explicitly live. Access path $w{\rightarrowtriangle}lptr{\rightarrowtriangle}lptr$ is implicitly live. Note that none of its proper prefixes is live at line 5. If the assignment $w = x$ were conditional, then depending upon whether it is executed, $w{\rightarrowtriangle}lptr{\rightarrowtriangle}lptr$ may or may not be live at line 5. However, the liveness of y or $y{\rightarrowtriangle}lptr$ does not depend on the past execution. □

Safety of null assignments requires that the access paths which are either explicitly or implicitly live are excluded from nullification.

Example 2.5. The assignments in Figure 2 have the following effect on liveness.

—As in the case of alias analysis, any access path which has a prefix that is must-link-aliased to $x{\rightarrowtriangle}r{\rightarrowtriangle}n$ will be killed. Any access path live after the assignment and not killed by it is also live before the assignment.

—All prefixes of $x{\rightarrowtriangle}r$ and $y{\rightarrowtriangle}n$ are explicitly live before the assignment.

—If $x{\rightarrowtriangle}r{\rightarrowtriangle}n{\rightarrowtriangle}\sigma$ is live after the assignment, then $y{\rightarrowtriangle}n{\rightarrowtriangle}n{\rightarrowtriangle}\sigma$ will be live before the assignment. For example, if $x{\rightarrowtriangle}r{\rightarrowtriangle}n{\rightarrowtriangle}n$ is live after the assignment, then $y{\rightarrowtriangle}n{\rightarrowtriangle}n{\rightarrowtriangle}n$ will be live before the assignment. The sequence of field names σ is viewed as being *transferred* from $x{\rightarrowtriangle}r{\rightarrowtriangle}n$ to $y{\rightarrowtriangle}n{\rightarrowtriangle}n$. □

We now define liveness by generalizing the above observations.

Definition 2.6. **Explicit Liveness**. The set of explicitly live access paths at a program point p, denoted as ELP_p is defined as follows.

(1) $ELP_{\mathit{Exit}} = \emptyset$, where Exit denotes the program exit.

(2) If p is not Exit, then consider a control flow path $\psi = p \rightarrow p' \rightarrow \ldots \rightarrow \mathit{Exit}$ such that p is followed by statement s and the program point immediately following s is p'.
  (a) If s is an assignment $\alpha_x = \alpha_y$, then define
    i. L_{Kill} as the set of all access paths that have a prefix which is must-link-aliased to ρ_x at p'. [3]
    ii. L_{Direct} as the union of the sets of all prefixes of $\mathit{Base}(\rho_x)$ and all prefixes of $\mathit{Base}(\rho_y)$. If α_y is $\mathit{New}\ldots$ or null, L_{Direct} is the set of all prefixes of $\mathit{Base}(\rho_x)$.

[3]Note that aliasing information, which depends on the path from Entry to p, is used only for killing explicit liveness. Generation of explicitly live paths depends only on the path from p to Exit.

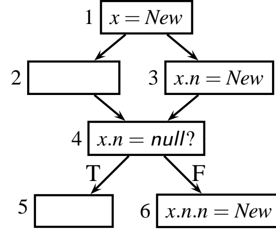


Fig. 3. Availability of Access Paths

iii. $L_{Transfer}$ as the set of all access paths $\rho_y \rightarrowtriangle \sigma$ where $\rho_x \rightarrowtriangle \sigma$ is in $ELP_{p'}$. Note that σ may be empty. If α_y is *New* ... or *null*, $L_{Transfer}$ is $\emptyset$.

(b) If s is a use $\alpha_y.d$ or α_y, then define L_{Direct} as the set of all prefixes of ρ_y and $L_{Kill} = L_{Transfer} = \emptyset$.

For a given set X of access paths, define

$$EF_\psi(X) = \begin{cases} X & \psi \text{ is empty} \\ (EF_{\psi'}(X) - L_{Kill}) \cup L_{Direct} \cup L_{Transfer} & \text{otherwise} \end{cases} \quad (2)$$

$$\text{where } \psi = p \rightarrow p' \bullet \psi'$$

ELP_p is defined as the union of access path sets $EF_\psi(\emptyset)$ along all control flow paths ψ from p to *Exit*. □

The definitions of L_{Kill}, L_{Direct}, and $L_{Transfer}$ ensure that the ELP_p is a prefix-closed set.

Example 2.7. In Figure 1, it cannot be statically determined which link is represented by access expression *x.lptr* at line 4. Depending upon the number of iterations of the *while* loop, it may be any of the links represented by thick arrows. Thus at line 1, we have to assume that all access paths $\{x \rightarrowtriangle lptr \rightarrowtriangle lptr, x \rightarrowtriangle rptr \rightarrowtriangle lptr \rightarrowtriangle lptr, x \rightarrowtriangle rptr \rightarrowtriangle rptr \rightarrowtriangle lptr \rightarrowtriangle lptr, \ldots\}$ are explicitly live. □

In general, an infinite number of access paths with unbounded lengths may be live before a loop. Clearly, performing data flow analysis for access paths requires a suitable finite representation. Section 3 defines access graphs for the purpose.

2.5 Availability and Anticipability of Access Paths

Liveness alone is not enough to decide whether an assignment $\alpha_x = null$ can be safely inserted at p. We have to additionally ensure that dereferencing links during execution of $\alpha_x = null$ does not cause an exception.

Example 2.8. In Figure 3, access path $x \rightarrowtriangle n \rightarrowtriangle n$ is not live in block 2. However, it cannot be set to *null* since the object pointed to by $x \rightarrowtriangle n$ does not exist in memory when the execution reaches block 2. Therefore, insertion of $x.n.n = null$ in block 2 will raise an exception at run-time. □

The above example shows that safety of inserting an assignment $\alpha_x = null$ at a program point p requires that whenever control reaches p, every prefix of $Base(\rho_x)$ has a non-*null* l-value. Such an access path is said to be *accessible* at p. To ensure accessibility, we assume that the program being analyzed is correct and every use of an access expression appearing in the program can be dereferenced. In particular,

—We define an access path ρ_x to be *available* at a program point p, if along every path reaching p, there exists a program point p' such that $\mathit{Frontier}(\rho_x)$ is either dereferenced or assigned a non-*null* l-value at p' and is not made *null* between p' to p.

—We define an access path ρ_x to be *anticipable* at a program point p, if along every path starting from p, $\mathit{Frontier}(\rho_x)$ is dereferenced before being assigned.

In either case, testing $\mathit{Frontier}(\rho_x)$ for comparison is excluded. Clearly, an access path ρ_x is accessible at p if all of its prefixes are either available or anticipable at p.

Example 2.9. For the example in Figure 2, we show the effect of the an assignment on available access paths.

—Any access path which has a prefix that is link-aliased to $x \rightarrow r \rightarrow n$ ceases to be available after the assignment, unless made available by the right hand side (see below).

—The access paths x and $x \rightarrow r$ becomes available after the assignment.

—The access paths y and $y \rightarrow n$ also become available after the assignment.

—If an access path $y \rightarrow n \rightarrow n \rightarrow \sigma$ is available before the assignment, then the access path $x \rightarrow r \rightarrow n \rightarrow \sigma$ becomes available after the assignment. As examples, if $y \rightarrow n \rightarrow n$ and $y \rightarrow n \rightarrow n \rightarrow n$ were available before the assignment, then $x \rightarrow r \rightarrow n$ and $x \rightarrow r \rightarrow n \rightarrow n$ become available after the assignment. Note that σ may be empty.

—Availability is closed under must-link-aliasing. For the assignment in Figure 2(b), this makes y and v available after the assignment. □

Similar observations can be made about anticipability of access paths. We generalize the above observations to define availability of access paths.

Definition 2.10. **Availability**. Let AV_p denote the set of available access paths at program point p.

(1) $AV_{\mathit{Entry}} = \emptyset$, where *Entry* denotes the program entry.

(2) If p is not *Entry*, then consider a control flow path $\psi = \mathit{Entry} \rightarrow \ldots \rightarrow p' \rightarrow p$. Assume that in this path, p follows statement s and the program point before s is p'.
  (a) If s is an assignment $\alpha_x = \alpha_y$, then define
    i. Av_{Kill} as the set of access paths with a prefix link-aliased to ρ_x at p'.
    ii. Av_{Direct} as
      —If α_y is *New* ... then Av_{Direct} is the set of all prefixes of ρ_x.
      —If α_y is *null*, Av_{Direct} is the set of all prefixes of $\mathit{Base}(\rho_x)$.
      —Otherwise, Av_{Direct} is the union of
        . the set of all prefixes of $\mathit{Base}(\rho_x)$, and
        . the set of those prefixes of $\mathit{Base}(\rho_y)$ which are not in Av_{Kill}.
    iii. $Av_{Transfer}$ as the set of access paths $\rho_x \rightarrow \sigma$, where $\rho_y \rightarrow \sigma$ is in $AV_{p'}$. Note that σ may be empty. If α_y is *null* or *New* ..., then $Av_{Transfer}$ is $\emptyset$.
  (b) If s is a use statement $\alpha_y.d$, then define Av_{Direct} as the set of all prefixes of ρ_y. Otherwise, if s is a use statement α_y, then define Av_{Direct} as the set of all prefixes of $\mathit{Base}(\rho_y)$. Define $Av_{Kill} = Av_{Transfer} = \emptyset$.

  For a given set X of access paths, define

$$AvF_\psi(X) = \begin{cases} X & \psi \text{ is empty} \\ (AvF_{\psi'}(X) - Av_{Kill}) \cup Av_{Direct} \cup Av_{Transfer} & \text{otherwise} \end{cases} \quad (3)$$

  where $\psi = \psi' \bullet p' \rightarrow p$

AV_p is defined as the must-aliases of the intersection of access path sets $AvF_\psi(\emptyset)$ along all control flow paths ψ from *Entry* to p. □

In a similar manner, we define anticipability of access paths.

Definition 2.11. **Anticipability**. Let AN_p denote the set of anticipable access paths at program point p.

(1) $AN_{Exit} = \emptyset$.

(2) If p is not *Exit*, then consider a control flow path $\psi = p \rightarrow p' \rightarrow \ldots \rightarrow Exit$. Assume that in this path, p is followed by the statement s and the program point following s is p'.
   (a) If s is an assignment $\alpha_x = \alpha_y$, then define
      i. An_{Kill} as the set of access paths with a prefix link-aliased to ρ_x at p'.
      ii. An_{Direct} as the union of sets of all prefixes of $Base(\rho_x)$ and all prefixes of $Base(\rho_y)$. If α_y is *New* ... or *null*, An_{Direct} is the set of all prefixes of $Base(\rho_x)$.
      iii. $An_{Transfer}$ as the set of access paths $\rho_y \rightarrow \sigma$, where $\rho_x \rightarrow \sigma$ is in $AN_{p'}$. Note that σ may be empty. If α_y is *New* ... or *null*, $An_{Transfer}$ is $\emptyset$.
   (b) If s is a use statement $\alpha_y.d$, then define An_{Direct} as the set of all prefixes of ρ_y. Otherwise, if s is a use statement α_y, then define An_{Direct} as the set of all prefixes of $Base(\rho_y)$. Define $An_{Kill} = An_{Transfer} = \emptyset$.

   For a given set X of access paths, define

$$AnF_\psi(X) = \begin{cases} X & \psi \text{ is empty} \\ (AnF_{\psi'}(X) - An_{Kill}) \cup An_{Direct} \cup An_{Transfer} & \text{otherwise} \end{cases} \quad (4)$$
$$\text{where } \psi = p \rightarrow p' \bullet \psi'$$

   AN_p is defined as the must-aliases of the intersection of access path sets $AnF_\psi(\emptyset)$ along all control flow paths ψ from p to *Exit*. □

Both AV_p and AN_p are prefix-closed.

2.6 Nullability of Access Paths

An access path ρ_x is *nullable* at a program point p if the assignment $\alpha_x = null$ can be inserted at p without affecting the semantics of the program in any way. As observed in Example 2.8, safety of inserting $\alpha_x = null$ at p requires that (a) ρ_x should not be live at p and (b) every prefix of $Base(\rho_x)$ should be accessible. Further, from considerations of efficiency, inserting $\alpha_x = null$ at p is redundant, if (a) a proper prefix of ρ_x is nullable at p, or (b) link corresponding to $Frontier(\rho_x)$ has already been nullified before p.

The candidate access paths for *null* assignment at program point p are created using the notion of accessibility as follows: All prefixes of accessible paths at p are extended by the relevant field names and the paths which are live at p are excluded. Additional criteria capturing profitability is used for the final decision (section 5).

Example 2.12. Consider line 3 in the program in Figure 1(a). It is easy to see that the access path x is both available and anticipable just before line 3. We extend x and observe that $x \rightarrow rptr$ is live and cannot be nullified. However, $x \rightarrow lptr$ can be set to *null*. □

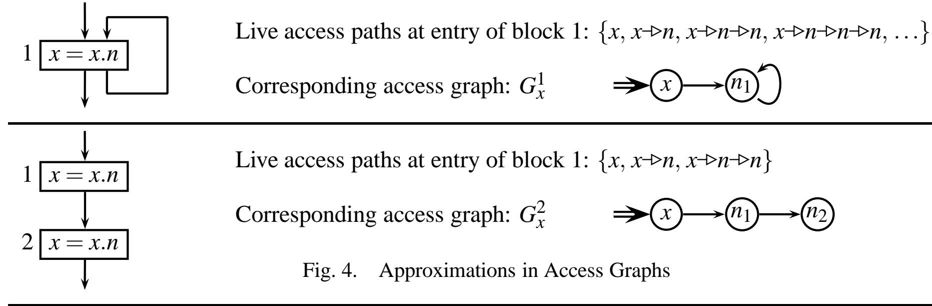


Fig. 4. Approximations in Access Graphs

3. REPRESENTING SETS OF ACCESS PATHS BY ACCESS GRAPHS

In the presence of loops, the set of access paths may be infinite and the lengths of access paths may be unbounded. If the algorithm for analysis tries to compute sets of access paths directly, termination cannot be guaranteed. We solve this problem by devising a bounded representation to summarize sets of access paths.

3.1 Defining Access Graphs

An *access graph*, denoted by G_v, is a directed graph used for representing a set of access paths starting from a root variable v.[4] It is a 4-tuple $\langle n_0, N_F, N_I, E\rangle$ where N_F is the set of *final* nodes, N_I is the set of *intermediate* nodes, $n_0 \in (N_F \cup N_I)$ is the entry node with no in-edges and E is the set of edges. Every path starting with the entry node of an access graph and reaching a final node represents an access path contained in the access graph. Except for the *empty graph* $\mathcal{E}_G$ which has no nodes or edges, every access graph has at least one final node, a path from the entry node to all other nodes and a path from every intermediate node to a final node. Note that, *$\mathcal{E}_G$ does not accept* any access path, as there are no final nodes in it.

The entry node of an access graphs is labeled with the name of the root variable while the non-entry nodes are labeled with a unique label created as follows: If a field name f is referenced in basic block b, we create an access graph node with a label $\langle f, b, i\rangle$ where i is the instance number used for distinguishing multiple occurrences of the field name f in block b. Note that this implies that the nodes with the same label are treated as identical. Often, i is 0 and in such a case we denote the label $\langle f, b, 0\rangle$ by f_b for brevity.

A node in the access graph represents one or more links in the memory graph. Additionally, during analysis, it represents a state of access graph construction (explained in Section 3.2). An edge $f_n \rightarrow g_m$ in an access graph at program point p indicates that a link corresponding to field f dereferenced in block n may be used to dereference a link corresponding to field g in block m. For liveness analysis, this dependence occurs on some path starting at p. For alias analysis, this dependence occurs on some path reaching p.

Pictorially, the entry node of an access graph is indicated by an incoming double arrow, final nodes are indicated by solid circles while dotted circles indicate intermediate nodes. The access graph $\Rightarrow (y) \rightarrow (l_1)$ represents two access paths y and $y{\rightarrow}l$, while the access graph $\Rightarrow (w) \rightarrow (r_2) \rightarrow (l_3)$ represents the access path $w{\rightarrow}r{\rightarrow}l$ only.

[4]Where the root variable name is not required, we drop the subscript.

3.2 Summarization

Recall that a link is live at a program point p if it is used along some control flow path from p to *Exit*. Since different access paths may be live along different control flow paths and there may be infinitely many control flow paths in the case of a loop following p, there may be infinitely many access paths which are live at p. Hence, the lengths of access paths will be unbounded. In such a case summarization is required.

Summarization is achieved by merging appropriate nodes in access graphs, retaining all in and out edges of merged nodes. We explain merging with the help of Figure 4:

—Node n_1 in access graph G_x^1 indicates references of n at *different execution instances of the same* program point. Every time this program point is visited during analysis, the same state is reached in that the pattern of references after n_1 is repeated. Thus all occurrences of n_1 are merged into a single state. This creates a cycle which captures the repeating pattern of references.

—In G_x^2, nodes n_1 and n_2 indicate referencing n at *different* program points. Since the references made after these program points may be different, n_1 and n_2 are not merged.

Summarization captures the pattern of heap traversal in the most straightforward way. Traversing a path in the heap requires the presence of reference assignments $\alpha_x = \alpha_y$ such that ρ_x is a proper prefix of some link-alias of ρ_y. Assignments in Figure 2(b) and Figure 4 are examples of such assignments. The structure of the flow of control between such assignments in a program determines the pattern of heap traversal. Summarization captures this pattern without the need of control flow analysis and the resulting structure is reflected in the access graphs as can be seen in Figure 4. More examples of the resemblance of program structure and access graph structure can be seen in the access graphs in Figure 8.

3.3 Operations on Access Graphs

Section 2 defined heap properties by applying some operations on access paths. Here we define corresponding operations on access graphs. All these operations are pure functions in that they do not modify their arguments. Unless specified otherwise, the binary operations are applied only to access graphs having same root variable. The auxiliary operations and associated notations are:

—$\mathit{Root}(\rho)$ denotes the root variable of access path ρ, while $\mathit{Root}(G)$ denotes the root variable of access graph G.

—$\mathit{Field}(n)$ for a node n denotes the field name component of the label of n.

—$\mathit{GAll}(\rho)$ and $\mathit{GOnly}(\rho)$ construct access graphs corresponding to ρ. $\mathit{GAll}(\rho)$ constructs an access graph containing all prefixes of ρ, whereas $\mathit{GOnly}(\rho)$ constructs an access graph containing only ρ. In either case the resulting graph is linear except that $\mathit{GAll}(\rho)$ marks all nodes as final nodes whereas $\mathit{GOnly}(\rho)$ marks only the last node as final node. Both of them use the current basic block number and the field names to create appropriate labels for nodes. The instance number depends on the number of occurrences of a field name in the block.

—$\mathit{CleanUp}(G)$ deletes the nodes which are not reachable from the entry node or which do not have a path to a final node.

—$\mathit{CFN}(G, G')$ computes the set of nodes of G which correspond to the final nodes of G'. To compute $\mathit{CFN}(G, G')$, we define $\mathit{CN}(G, G')$, the set of pairs of *corresponding nodes*.

Let $G = \langle n_0, N_F, N_I, E\rangle$ and $G' = \langle n'_0, N'_F, N'_I, E'\rangle$. A node $n \in N_F \cup N_I$ in G corresponds to a node $n' \in N'_F \cup N'_I$ in G' if there there exists an access path ρ which starting at n_0 in G, reaches n, and starting at n'_0 in G', reaches n'. Formally,

$$\mathit{CN}(G,G') = \begin{cases} \emptyset & \mathit{Root}(G) \neq \mathit{Root}(G') \\ \mathit{FIX}(\{\langle n_0, n'_0\rangle\} \cup \{\langle n_j, n'_j\rangle \mid \mathit{Field}(n_j) = \mathit{Field}(n'_j), & \text{otherwise}^5 \\ \quad n_i \rightarrow n_j \in E, n'_i \rightarrow n'_j \in E', & \\ \quad \langle n_i, n'_i\rangle \in \mathit{CN}(G,G')\}) & \end{cases}$$

where FIX computes the least fixed point of its argument.

$$\mathit{CFN}(G,G') = \{n \mid \langle n, n'\rangle \in \mathit{CN}(G,G'), n' \in N'_F\}$$

Let $G = \langle n_0, N_F, N_I, E\rangle$ and $G' = \langle n_0, N'_F, N'_I, E'\rangle$ be access graphs (having the same entry node). The main operations of interest are defined below and are illustrated in Figure 5.

(1) *Union* ($\uplus$). $G \uplus G'$ combines access graphs G and G' such that any access path contained in G or G' is contained in the resulting graph.

$$G \uplus G' = \left\langle n_0, N_F \cup N'_F, (N_I \cup N'_I) - (N_F \cup N'_F), E \cup E' \right\rangle$$

Because of associativity, $\uplus$ can be generalized to arbitrary number of arguments in an obvious manner.

(2) *Path Removal* ($\ominus$). The operation $G \ominus \rho$ removes those access paths in G which have ρ as a prefix.

$$G \ominus \rho = \begin{cases} G & \rho = \mathcal{E} \text{ or } \mathit{Root}(\rho) \neq \mathit{Root}(G) \\ \mathcal{E}_G & \rho \text{ is a simple access path} \\ \mathit{CleanUp}(\langle n_0, N_F, N_I, E - E_{del}\rangle) & \mathit{otherwise} \end{cases}$$

where

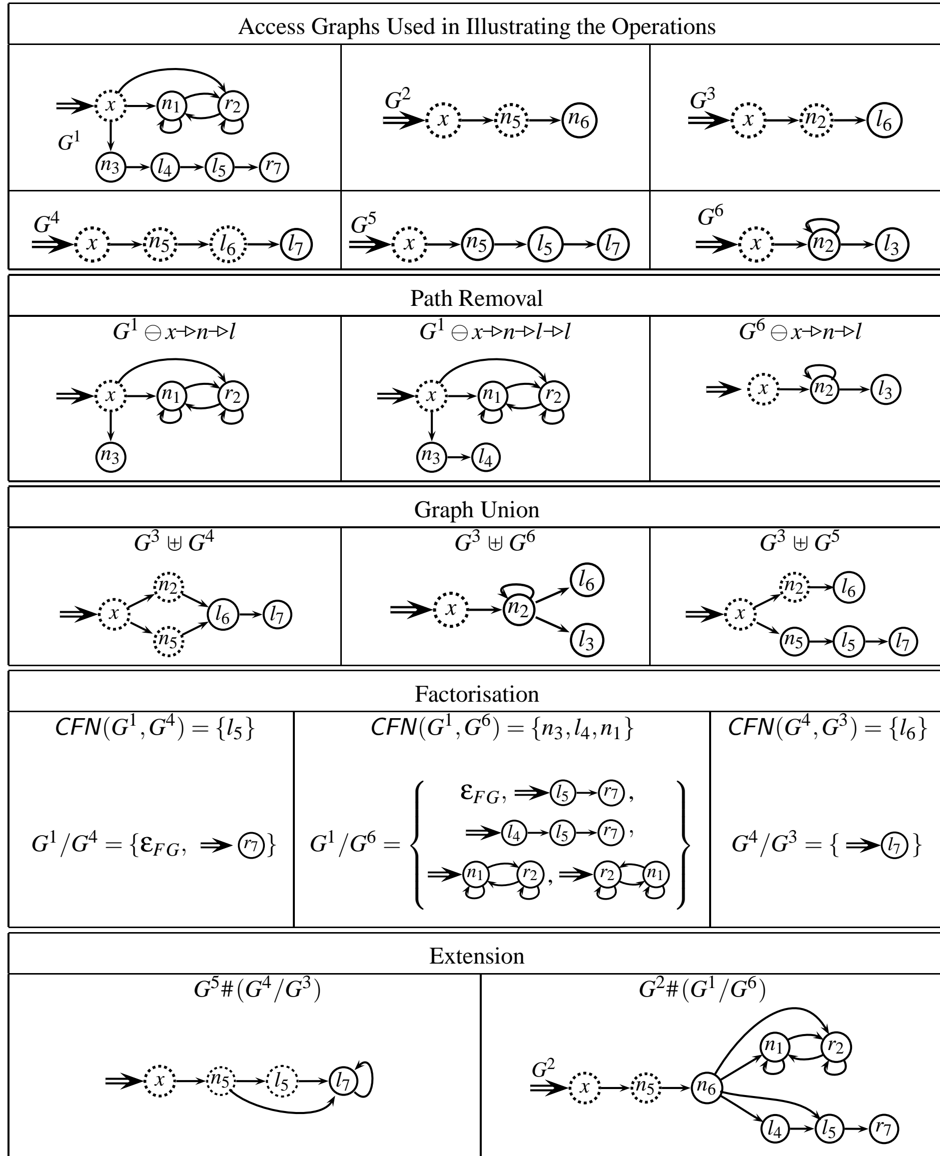
$$E_{del} = \{n_i \rightarrow n_j \mid n_i \rightarrow n_j \in E, n_i \in \mathit{CFN}(G, G^B), \mathit{Field}(n_j) = \mathit{Frontier}(\rho), G^B = \mathit{GOnly}(\mathit{Base}(\rho)), \mathit{UniqueAccessPath?}(G, n_i)\}$$

$\mathit{UniqueAccessPath?}(G, n)$ returns true if in G, all paths from the entry node to node n represent the same access path. Note that path removal is conservative in that some paths having ρ as prefix may not be removed. Since an access graph edge may be contained in more than one access paths, we have to ensure that access paths which do not have ρ as prefix are not erroneously deleted.

(3) *Factorisation* (/). Recall that the *Transfer* term in Definitions 2.3, 2.6, 2.10, and 2.11 requires extracting suffixes of access paths and attaching them to some other access paths. The corresponding operations on access graphs are performed using factorisation and extension.

Given a node $n' \in (N_F \cup N_I - \{n_0\})$, let the *Factorised Subgraph* of an access graph G, denoted by $FG(G, n')$, be the subgraph of G reachable from n'. G/G' computes a

[5] Note that $n_0 = n'_0$ in this case.



The path removal $G^6 \ominus x \rightarrow n \rightarrow l$ does not delete edge $n_2 \rightarrow l_3$, because deleting the edge will also remove access paths $x \rightarrow n \rightarrow n \rightarrow l, x \rightarrow n \rightarrow n \rightarrow n \rightarrow l, \ldots, x \rightarrow n \rightarrow \cdots \rightarrow n \rightarrow l$ in G^6. G^4/G^3 does not contain ε_{FG} because only node in $\mathit{CFN}(G^4, G^3)$, i.e. l_6, is not a fi nal node in G^4. $G^5\#(G^4/G^3)$ introduces a loop over l_7, because an edge has to be created from l_7 in G^5 to l_7 in G^4/G^3.

Fig. 5. Examples of operations on access graphs.

Operation	Access Graphs	Access Paths
Union	$G_3 = G_1 \uplus G_2$	$P_{G_3} \supseteq P_{G_1} \cup P_{G_2}$
Path Removal	$G_2 = G_1 \ominus \rho$	$P_{G_2} \supseteq P_{G_1} - \{\rho \rightarrow \sigma \mid \rho \rightarrow \sigma \in P_{G_1}\}$
Factorisation	$S^F = G_1 / G_2$	$P_{S^F} = \{\sigma \mid \rho' \rightarrow \sigma \in P_{G_1}, \rho' \in P_{G_2}\}$
Extension	$G_2 = G_1 \# S^F$	$P_{G_2} \supseteq \{\rho \rightarrow \sigma \mid \rho \in P_{G_1}, \sigma \in P_{S^F}\}$

Fig. 6. Safety of Access Graph Operations. P_{G_i} is the set of access paths in access graph G_i. P_{S^F} is the set of sequences of field names in S^F.

set of factorised subgraphs of G with respect to access paths in G'.

$$G/G' = \begin{cases} \emptyset & C = \emptyset \\ \{FG(G, n_j) \mid n_i \rightarrow n_j \in E, n_i \in C\} & C \cap N_F = \emptyset \\ \{FG(G, n_j) \mid n_i \rightarrow n_j \in E, n_i \in C\} \cup \{\varepsilon_{FG}\} & C \cap N_F \neq \emptyset \end{cases}$$

where $C = \mathit{CFN}(G, G')$ and ε_{FG} is a special factorised subgraph which corresponds to empty suffix.

Note that a factorised subgraph is similar to an access graph except that (a) its entry node does not correspond to a root variable but to a field name and (b) the entry node can have incoming edges. Also note that the special factorised subgraph ε_{FG} accepts only empty string.

(4) *Extension*. This operation is defined only for a non-empty access graph.

(a) *Extension with a factorised subgraph* ($\cdot$). $G \cdot FG$ appends the sequences of field names in factorised subgraph $FG = \langle n', N_F^{FG}, N_I^{FG}, E^{FG} \rangle$ to the access paths in G.

$$G \cdot \varepsilon_{FG} = G$$
$$G \cdot FG = \langle n_0, N_F^{FG}, (N_I \cup N_F \cup N_I^{FG}) - N_F^{FG}, E \cup E^{FG} \cup E_{new} \rangle$$

where $E_{new} = \{n_i \rightarrow n' \mid n_i \in N_F\}$.

(b) *Extension with a set of factorised subgraphs* (#). $G \# S^{FG}$ extends access graph G with every factorised subgraph in S^{FG}.

$$G \# \emptyset = \mathcal{E}_G$$
$$G \# S^{FG} = \biguplus_{FG \in S^{FG}} (G \cdot FG)$$

3.4 Safety of Access Graph Operations

Since access graphs are not exact representations of sets of access paths, the safety of approximations needs to be defined explicitly. The safety properties defined in Figure 6 have been proved [Iyer 2005] using the PVS theorem prover[6].

4. DATA FLOW ANALYSIS FOR HEAP REFERENCES

In this section we define data flow analyses for capturing the properties of aliasing, liveness, availability, and anticipability of heap references. The data flow equations approximate the specifications in Section 2. *BoundaryInfo* denotes a safe approximation of interprocedural

[6]Available from `http://pvs.csl.sri.com`.

information. We do not perform must-alias analysis and conservatively assume that every access path is must-link-aliased only to itself.

4.1 Alias Analysis

The data flow equations for alias analysis compute the may-node-alias relations defined in Section 2.3 using access graphs. The alias information is stored in the form of a set of access graph pairs. A pair $\langle G^i, G^j \rangle$ indicates that all access paths in G^i are aliased to all access paths in G^j. Though the alias relation represented as access graph pairs is symmetric, we explicitly store only one of the pairs $\langle G^i, G^j \rangle$ and $\langle G^j, G^i \rangle$. A pair $\langle G^i, G^j \rangle$ is removed from the set of alias pairs if G^i or G^j is $\mathcal{E}_G$.

$\mathbb{A}\mathsf{In}(i)$ and $\mathbb{A}\mathsf{Out}(i)$ denote the set of may-node-aliases before and after the statement i. Their initial values are $\emptyset$.

$$\mathbb{A}\mathsf{In}(i) \;=\; \begin{cases} \mathit{BoundaryInfo} & i = \mathit{Entry} \\ \bigcup\limits_{p \in pred(i)} \mathbb{A}\mathsf{Out}(p) & \text{otherwise} \end{cases} \tag{5}$$

$$\mathbb{A}\mathsf{Out}(i) \;=\; (\mathbb{A}\mathsf{In}(i) \,-\, \mathbb{A}\mathsf{Kill}(i)) \,\cup\, \mathbb{A}\mathsf{Gen}(i) \tag{6}$$

Since all our analyses require link-aliases, we derive them from node-aliases. Given a set of AS of node-aliases, $\mathsf{Ln}\mathbb{A}(G_v, AS)$ computes a set of graphs representing link-aliases of an access graph G_v. All link-aliases of every access path in G_v are contained in the resulting access graphs.

$$\mathsf{Ln}\mathbb{A}(G_v, AS) \;=\; \{G_v\} \cup \left\{ G_u \mid G_u = G'_u \# (G_v / G'_v - \{\varepsilon_{FG}\}), \langle G'_v, G'_u \rangle \in AS \right\} \tag{7}$$

Note that link-alias computation should add at least one link to node-aliases and hence should exclude empty suffixes from extension.

We now define the flow functions for a statement i. Since use statements do not modify heap references, both $\mathbb{A}\mathsf{Gen}(i)$ and $\mathbb{A}\mathsf{Kill}(i)$ are $\emptyset$. Thus, $\mathbb{A}\mathsf{Out}(i) = \mathbb{A}\mathsf{In}(i)$ for such statements. For an assignment $\alpha_x = \alpha_y$, access graphs capturing the sets of access paths *LhsPaths* and *RhsPaths* (Definition 2.3) are:

$$\begin{aligned} \mathit{LhsGraphs} &= \mathsf{Ln}\mathbb{A}(\mathit{GOnly}(\rho_x), \mathbb{A}\mathsf{In}(i)) \\ \mathit{RhsGraphs} &= \mathsf{Ln}\mathbb{A}(\mathit{GOnly}(\rho_y), \mathbb{A}\mathsf{In}(i)) \end{aligned}$$

KillPaths defined by the specifications includes all paths which have a prefix which is must-link-aliased to ρ_x. Since we do not compute must-aliases, we conservatively assume that an access path is must-aliased only to itself. Hence we approximate *KillPaths* to contain all access paths which have ρ_x as a prefix. In order to remove such paths, the path removal operation uses ρ_x. Note that the absence of explicit must-alias information and the path removal operation both introduce (safe) approximations. Effectively, we kill fewer paths than are required by the specifications.

The assignment $\alpha_x = \alpha_y$ kills the aliases involving the access path which contain the link corresponding to $\mathit{Frontier}(\rho_x)$. Instead of computing $\mathbb{A}\mathsf{Kill}(i)$ explicitly, we calculate $\mathbb{A}\mathsf{In}(i) - \mathbb{A}\mathsf{Kill}(i)$ directly as:

$$\left\{ \left\langle G \ominus \rho_x, G' \ominus \rho_x \right\rangle \mid \langle G, G' \rangle \in \mathbb{A}\mathsf{In}(i) \right\}$$

$\mathbb{A}\mathsf{Gen}(i)$ for the assignment is defined as:

$$\mathbb{A}\mathsf{Gen}(i) \;=\; \begin{cases} \emptyset & \rho_y \text{ is } \mathit{null} \text{ or } \mathit{New} \ldots \\ \mathit{ADirect}(i) \cup \mathit{ATransfer}(i), & \text{otherwise} \end{cases}$$

i	$\mathbb{A}\mathsf{In}(i)$	$\mathbb{A}\mathsf{Out}(i)$
1	$\emptyset$	$\{\langle \to x, \to w \rangle\}$
2	$\{\langle \to x, \to w \to r_3 \circlearrowleft \rangle\}$	$\{\langle \to x, \to w \to r_3 \circlearrowleft \rangle\}$
3	$\{\langle \to x, \to w \to r_3 \circlearrowleft \rangle\}$	$\{\langle \to x, \to w \to r_3 \circlearrowleft \rangle\}$
4	$\{\langle \to x, \to w \to r_3 \circlearrowleft \rangle\}$	$\{\langle \to x, \to w \to r_3 \circlearrowleft \rangle, \langle \to y, \to x \to l_4 \rangle, \langle \to y, \to w \to r_3 \circlearrowleft \to l_4 \rangle\}$
5	$\{\langle \to x, \to w \to r_3 \circlearrowleft \rangle, \langle \to y, \to x \to l_4 \rangle, \langle \to y, \to w \to r_3 \circlearrowleft \to l_4 \rangle\}$	$\{\langle \to x, \to w \to r_3 \circlearrowleft \rangle, \langle \to y, \to x \to l_4 \rangle, \langle \to y, \to w \to r_3 \circlearrowleft \to l_4 \rangle\}$
6	$\{\langle \to x, \to w \to r_3 \circlearrowleft \rangle, \langle \to y, \to x \to l_4 \rangle, \langle \to y, \to w \to r_3 \circlearrowleft \to l_4 \rangle\}$	$\{\langle \to x, \to w \to r_3 \circlearrowleft \rangle, \langle \to y, \to x \to l_4 \to l_6 \rangle, \langle \to y, \to w \to r_3 \circlearrowleft \to l_4 \to l_6 \rangle\}$
7	$\{\langle \to x, \to w \to r_3 \circlearrowleft \rangle, \langle \to y, \to x \to l_4 \to l_6 \rangle, \langle \to y, \to w \to r_3 \circlearrowleft \to l_4 \to l_6 \rangle\}$	$\{\langle \to x, \to w \to r_3 \circlearrowleft \rangle, \langle \to y, \to x \to l_4 \to l_6 \rangle, \langle \to y, \to w \to r_3 \circlearrowleft \to l_4 \to l_6 \rangle\}$

Field names *lptr* and *rptr* have been abbreviated by *l* and *r*. For convenience, multiple pairs have been merged into a single pair where possible, e.g. alias pair $\langle \to x, \to w \to r_3 \circlearrowleft \rangle$ implies three alias pairs $\langle \to x, \to w \rangle$, $\langle \to x, \to w \to r_3 \rangle$, and $\langle \to x, \to w \to r_3 \circlearrowleft \rangle$.

Fig. 7. Aliases for the program in Figure 1.

where

$$\begin{aligned}
\mathit{ADirect}(i) &= \mathit{LhsGraphs} \times \{G \ominus \rho_x \mid G \in \mathit{RhsGraphs}\} \\
\mathit{ATransfer}(i) &= \mathit{ATransfer}_1(i) \cup \mathit{ATransfer}_2(i) \\
\mathit{ATransfer}_1(i) &= \{\langle G_z \ominus \rho_x, G_u \# (G_y / \mathit{GOnly}(\rho_y))\rangle \mid \\
&\qquad \langle G_z, G_y\rangle \in \mathbb{A}\mathsf{In}(i), G_u \in \mathit{LhsGraphs}\} \\
\mathit{ATransfer}_2(i) &= \{\langle G_u \# (G^1_y / \mathit{GOnly}(\rho_y)), G_v \# (G^2_y / \mathit{GOnly}(\rho_y))\rangle \mid \langle G^1_y, G^2_y\rangle \in \mathbb{A}\mathsf{In}(i), \\
&\qquad G_u \in \mathit{LhsGraphs}, G_v \in \mathit{LhsGraphs}\}
\end{aligned}$$

Example 4.1. Figure 7 lists the alias information for the program in Figure 1. We have shown only the final result. The alias pair $\langle \to x, \to w \to r_3 \circlearrowleft \rangle$ in $\mathbb{A}\mathsf{In}(3)$ represents an infinite number of aliases $\langle x, w \rhd rptr\rangle$, $\langle x, w \rhd rptr \rhd rptr\rangle$, $\langle x, w \rhd rptr \rhd rptr \rhd \ldots\rangle$. created in different execution instances of line 3. Alias $\langle x, w\rangle$ is created at line 1 is represented by

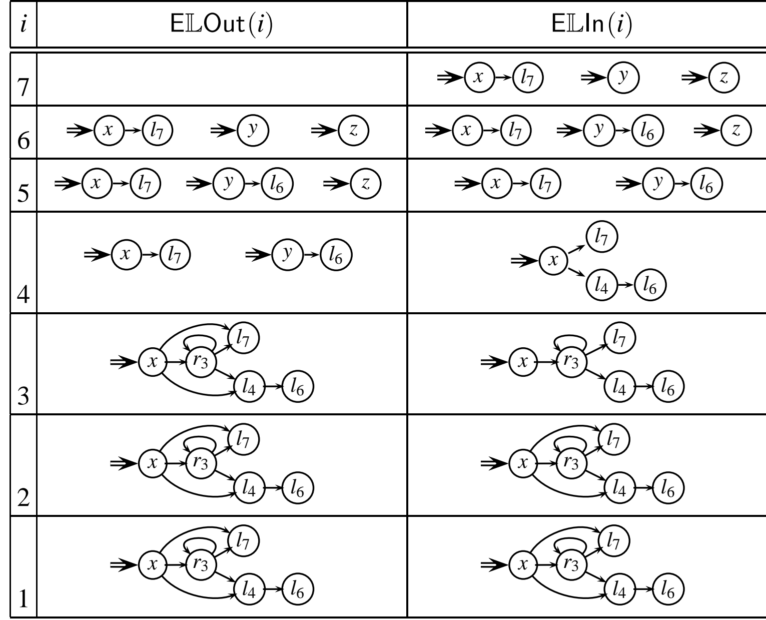


Fig. 8. Explicit liveness for the program in Figure 1.

the pair $\langle \rightarrow\!(x), \rightarrow\!(w) \rangle$. □

4.2 Liveness Analysis

We perform data flow analysis to discover explicitly live access graphs. For a given root variable v, $\mathsf{E}\mathbb{L}\mathsf{In}_v(i)$ and $\mathsf{E}\mathbb{L}\mathsf{Out}_v(i)$ denote the access graphs representing explicitly live access paths at the entry and exit of basic block i. We use $\mathcal{E}_G$ as the initial value for $\mathsf{E}\mathbb{L}\mathsf{In}_v(i)/\mathsf{E}\mathbb{L}\mathsf{Out}_v(i)$.

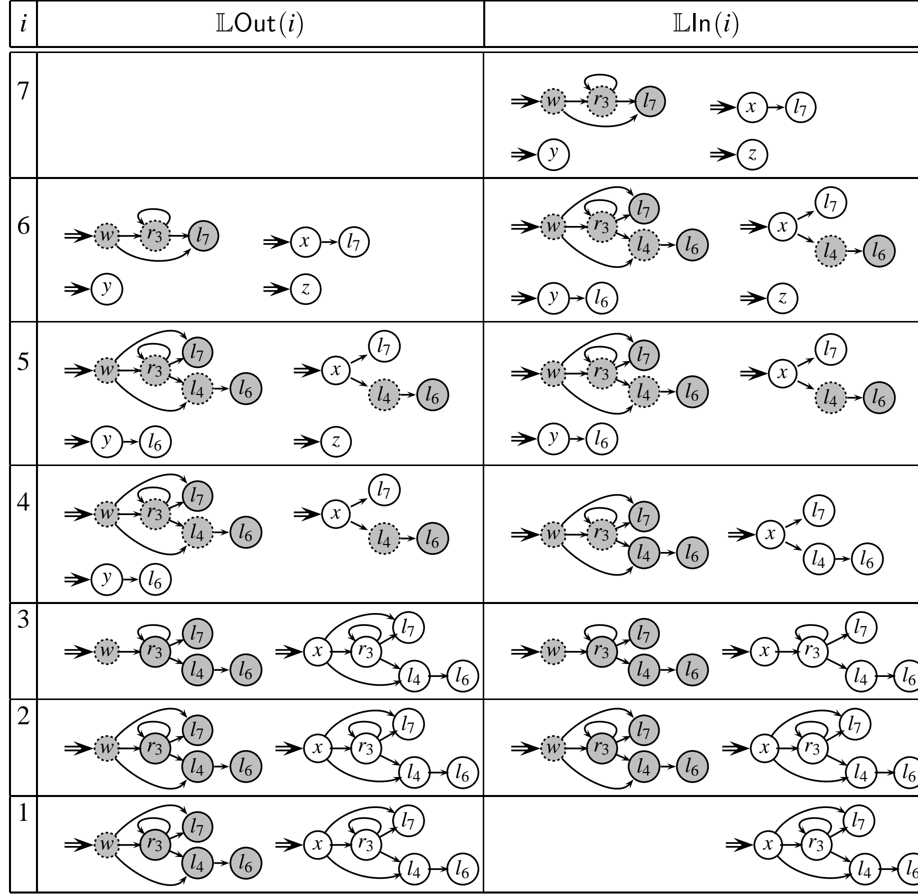
$$\begin{aligned}
\mathsf{E}\mathbb{L}\mathsf{In}_v(i) &= (\mathsf{E}\mathbb{L}\mathsf{Out}_v(i) \ominus \mathsf{E}\mathbb{L}\mathsf{KillPath}_v(i)) \uplus \mathsf{E}\mathbb{L}\mathsf{Gen}_v(i) \\
\mathsf{E}\mathbb{L}\mathsf{Out}_v(i) &= \begin{cases} \mathit{BoundaryInfo} & i = \mathit{Exit} \\ \biguplus_{s \in succ(i)} \mathsf{E}\mathbb{L}\mathsf{In}_v(s) & \text{otherwise} \end{cases}
\end{aligned}$$

where

$$\mathsf{E}\mathbb{L}\mathsf{Gen}_v(i) = \mathit{LDirect}_v(i) \uplus \mathit{LTransfer}_v(i)$$

We define $\mathsf{E}\mathbb{L}\mathsf{KillPath}_v(i)$, $\mathit{LDirect}_v(i)$, and $\mathit{LTransfer}_v(i)$ depending upon the statement.

(1) *Assignment statement* $\alpha_x = \alpha_y$. Apart from defining the desired terms for x and y, we



Gray nodes are included by link-alias computation. Besides, intermediate nodes get included in the access graphs.

Fig. 9. Liveness access graphs including implicit liveness information for the program in Figure 1.

also need to define them for any other variable z.

$$
\begin{aligned}
\mathit{LDirect}_x(i) &= \mathit{GAll}(\mathit{Base}(\rho_x)) \\
\mathit{LDirect}_y(i) &= \begin{cases} \mathcal{E}_G & \alpha_y \text{ is } \mathit{New} \ldots \text{ or } \mathit{null} \\ \mathit{GAll}(\mathit{Base}(\rho_y)) & \text{otherwise} \end{cases} \\
\mathit{LDirect}_z(i) &= \mathcal{E}_G, \text{for any variable } z \text{ other than } x \text{ and } y \\
\mathit{LTransfer}_y(i) &= \begin{cases} \mathcal{E}_G & \alpha_y \text{ is } \mathit{New} \text{ or } \mathit{null} \\ \mathit{GOnly}(\rho_y) \# (\mathsf{E}\mathbb{L}\mathsf{Out}_x(i) / \mathit{GOnly}(\rho_x)) & \text{otherwise} \end{cases} \\
\mathit{LTransfer}_z(i) &= \mathcal{E}_G, \text{ for any variable } z \text{ other than } y \\
\mathsf{E}\mathbb{L}\mathsf{KillPath}_x(i) &= \rho_x \\
\mathsf{E}\mathbb{L}\mathsf{KillPath}_z(i) &= \mathcal{E}, \text{ for any variable } z \text{ other than } x
\end{aligned}
$$

For the same reasons as in alias analysis, we may kill fewer access paths than are required by the specifications.

(2) *Use Statements*

$$\begin{aligned}
LDirect_y(i) &= \biguplus GAll(\rho_y) \text{ for every } \alpha_y \text{ or } \alpha_y.d \text{ used in } i \\
LDirect_z(i) &= \mathcal{E}_G \text{ for any variable } z \text{ other than } y \\
LTransfer_v(i) &= \mathcal{E}_G, \text{ for every variable } v \\
\mathsf{E}\mathbb{L}\mathsf{KillPath}_v(i) &= \mathcal{E}, \text{ for every variable } v
\end{aligned}$$

Once explicitly live access graphs are computed, implicitly live access graphs at a given program point can be discovered by computing may-link-aliases of explicitly live access graphs at that point. Mathematically,

$$\begin{aligned}
\mathbb{L}\mathsf{In}_v(i) &= \biguplus \{G_v \mid G_v \in \mathsf{Ln}\mathbb{A}(\mathsf{E}\mathbb{L}\mathsf{In}_u(i), \mathbb{A}\mathsf{In}(i)) \text{ for some variable } u\} \\
\mathbb{L}\mathsf{Out}_v(i) &= \biguplus \{G_v \mid G_v \in \mathsf{Ln}\mathbb{A}(\mathsf{E}\mathbb{L}\mathsf{Out}_u(i), \mathbb{A}\mathsf{Out}(i)) \text{ for some variable } u\}
\end{aligned}$$

Example 4.2. Figure 8 lists the explicit liveness information, while Figure 9 gives complete liveness information for program in Figure 1. □

4.3 Availability and Anticipability Analyses

Availability and Anticipability are *all (control-flow) paths* properties in that the desired property must hold along every path reaching/leaving the program point under consideration. Thus these analyses identify access paths which are common to all control flow paths *including acyclic control flow paths*. Since acyclic control flow paths can generate only acyclic[7] and hence finite access paths, anticipability and availability analyses deal with a finite number of access paths and summarization is not required.

Thus there is no need to use access graphs for availability and anticipability analyses. The data flow analysis can be performed using a set of access paths because the access paths are bounded and the sets would be finite. Besides, the prefix-closed property of these sets facilitates efficient representation. The data flow equations are exactly same as the formal specifications of these analyses (Definitions 2.10 and 2.11). $\mathbb{A}\mathsf{vIn}(i)$ and $\mathbb{A}\mathsf{vOut}(i)$ denote the set of available access paths before and after the statement i, while $\mathbb{A}\mathsf{nIn}_((i)$ and $\mathbb{A}\mathsf{nOut}_((i)$ denote the set of anticipable access paths before and after the statement i. We use the universal set of access paths as the initial value for all blocks other than *Entry* for availability analysis and *Exit* for anticipability analysis.

Example 4.3. Figure 10 gives the availability and anticipability information for program in Figure 1. □

5. *NULL* ASSIGNMENT INSERTION

We now explain how the analyses described in preceding sections can be used to insert appropriate *null* assignments to nullify dead links. The inserted assignments should be safe and profitable as defined below.

[7] In the presence of cycles in heap, cycles in access graphs do not represent summarization (Appendix A).

i	$\mathbb{A}\mathsf{vIn}(i)$	$\mathbb{A}\mathsf{vOut}(i)$	$\mathbb{A}\mathsf{nIn}(i)$	$\mathbb{A}\mathsf{nOut}(i)$
1	$\emptyset$	$\emptyset$	$\{x\}$	$\{x\}$
2	$\emptyset$	$\{x\}$	$\{x\}$	$\{x\}$
3	$\{x\}$	$\emptyset$	$\{x, x{\rightarrow}r\}$	$\{x\}$
4	$\{x\}$	$\{x\}$	$\{x, x{\rightarrow}l, x{\rightarrow}l{\rightarrow}l\}$	$\{x, x{\rightarrow}l, y, y{\rightarrow}l\}$
5	$\{x\}$	$\{x, z\}$	$\{x, x{\rightarrow}l, y, y{\rightarrow}l\}$	$\{x, x{\rightarrow}l, y, y{\rightarrow}l, z\}$
6	$\{x, z\}$	$\{x, z\}$	$\{x, x{\rightarrow}l, y, y{\rightarrow}l, z\}$	$\{x, x{\rightarrow}l, y, z\}$
7	$\{x, z\}$	$\{x, x{\rightarrow}l, y, z\}$	$\{x, x{\rightarrow}l, y, z\}$	$\emptyset$

Fig. 10. Availability and anticipability for the program in Figure 1.

Definition 5.1. **Safety**. It is safe to insert an assignment $\alpha = \mathit{null}$ at a program point p if and only if ρ is not live at p and $\mathit{Base}(\rho)$ can be dereferenced without raising an exception.

An access path ρ is *nullable* at a program point p if and only if it is safe to insert assignment $\alpha = \mathit{null}$ at p.

Definition 5.2. **Profitability**. It is profitable to insert an assignment $\alpha = \mathit{null}$ at a program point p if and only if no proper prefix of ρ is nullable at p and the link corresponding to $\mathit{Frontier}(\rho)$ is not made *null* before execution reaches p.

Note that profitability definition is strict in that every control flow path may nullify a particular link only once. Redundant *null* assignments on any path are prohibited. Since control flow paths have common segments, a *null* assignment may be redundant along one path but not along some other path. Such *null* assignments will be deemed unprofitable by Definition 5.2. Our algorithm may not be able to avoid all redundant assignments.

Example 5.3. We illustrate some situations of safety and profitability for the program in Figure 1.

— Access path $x{\rightarrow}lptr{\rightarrow}lptr$ is not nullable at the entry of 6. This is because $x{\rightarrow}lptr{\rightarrow}lptr$ is implicitly live, due to the use of $y{\rightarrow}lptr$ in 6. Hence it is not safe to insert $x.lptr.lptr = \mathit{null}$ at the entry of 6.

— Access path $x{\rightarrow}rptr$ is nullable at the entry of 4, and continues to be so on the path from the entry of 4 to the entry of 7. The assignment $x.rptr = \mathit{null}$ is profitable only at the entry of 4. □

Section 5.1 describes the criteria for deciding whether a give path ρ should be considered for a *null* assignment at a program point p. Section 5.2 describes how we create the set of candidate access paths.

5.1 Computing Safety and Profitability

To find out if ρ can be nullified at p, we compute two predicates: *Nullable* and *Nullify*. $\mathit{Nullable}(\rho, p)$ captures the safety property—it is true if insertion of assignment $\alpha = \mathit{null}$ at program point p is safe.

$$\mathit{Nullable}(\rho, p) \;=\; \rho \notin \mathit{Live}(p) \wedge \mathit{Base}(\rho) \in \mathit{Available}(p) \cup \mathit{Anticipable}(p) \qquad (8)$$

where *Live*(p), *Available*(p), and *Anticipable*(p) denote set of live paths, set of available paths and set of anticipable paths respectively at program point p [8].

Nullify(ρ, p) captures the profitability property—it is true if insertion of assignment $\alpha = \mathit{null}$ at program point p is profitable. To compute *Nullify*, we note that it is most profitable to set a link to *null* as soon as it ceases to be live. That is, the *null* assignment for a memory link should be as close to the entry as possible, taking safety into account. Therefore, *Nullify* predicate at a point has to take into account the possibility of *null* assignment insertion at previous point(s). For a statement i in the program, let In_i and Out_i denote respectively the program point immediately before and after i. Then,

$$\begin{aligned} \mathit{Nullify}(\rho, \mathsf{Out}_i) &= \mathit{Nullable}(\rho, \mathsf{Out}_i) \wedge \neg \mathit{Nullable}(\mathit{Base}(\rho), \mathsf{Out}_i) \\ &\quad \wedge (\neg \mathit{Nullable}(\rho, \mathsf{In}_i) \vee \neg\, \mathit{Transp}(\rho, i)) \qquad (9) \\ \mathit{Nullify}(\rho, \mathsf{In}_i) &= \mathit{Nullable}(\rho, \mathsf{In}_i) \wedge \neg \mathit{Nullable}(\mathit{Base}(\rho), \mathsf{In}_i) \\ &\quad \wedge \rho \neq \mathit{lhs}(i) \wedge (\neg \bigwedge_{j \in \mathit{pred}(i)} \mathit{Nullable}(\rho, \mathsf{Out}_j)) \qquad (10) \end{aligned}$$

where, $\mathit{Transp}(\rho, i)$ denotes that ρ is transparent with respect to statement i, i.e. no prefix of ρ is may-link-aliased to the access path corresponding to the lhs of statement i at In_i. $\mathit{lhs}(i)$ denotes the access path corresponding to the lhs access expression of assignment in statement i. $\mathit{pred}(i)$ is the set of predecessors of statement i in the program.

We insert assignment $\alpha = \mathit{null}$ at program point p if *Nullify*(ρ, p) is true.

5.2 Computing Candidate Access Paths for *null* Insertion

The method described above only checks whether a given access path ρ can be nullified at a given program point p. We can generate the *candidate* set of access paths for *null* insertion at p as follows: For any candidate access path ρ, $\mathit{Base}(\rho)$ must either be available or anticipable at p. Additionally, all simple access paths are also candidates for *null* insertions. Therefore,

$$\begin{aligned} \mathit{Candidates}(p) &= \{\rho \rightarrow f \mid \rho \in \mathit{Available}(p) \cup \mathit{Anticipable}(p), f \in \mathit{OutField}(\rho, p)\} \\ &\quad \cup \{\rho \mid \rho \text{ is a simple access path }\} \qquad (11) \end{aligned}$$

Where $\mathit{OutField}(\rho, p)$ is the set of fields which can be used to extend access path ρ at p. It can be obtained easily from the type information of the object $\mathit{Target}(\rho)$ at p.

Note that all the information required for Equations 8, 9, 10 and 11 is obtained from the result of data flow analyses described in preceding sections. Type information of objects required by Equation 11 can be obtained from the front end of compiler. We use liveness graphs (Section 4.2) at program point p to find out if an access path is live. *Transp* uses may alias information as computed in terms of pairs of access graph (Section 4.1).

Example 5.4. Figure 11 lists a trace of the null insertion algorithm for the program in Figure 1. □

5.3 Reducing Redundant *null* Insertions

Consider a program with an assignment statement $i : \alpha_x = \alpha_y$. Assume a situation where, for some nonempty suffix σ, both $\mathit{Nullify}(\rho_y \rightarrow \sigma, \mathsf{In}_i)$ and $\mathit{Nullify}(\rho_x \rightarrow \sigma, \mathsf{Out}_i)$ are true. In

[8] Because availability and anticipability properties are prefix closed, $\mathit{Base}(\rho) \in \mathit{Available}(p) \cup \mathit{Anticipable}(p)$ guarantees that all proper prefixes of ρ are either available or anticipable.

i	p	$Candidates(p)$	$lhs(i)$	$\{\rho \mid \neg Transp(\rho,i)\}$	$\{\rho \mid Nullable(\rho,p)\}$	$\{\rho \mid Nullify(\rho,p)\}$
1	In_1	$\{w,x,y,z,x{\rightarrow}l,x{\rightarrow}r\}$	w		$\{w,y,z\}$	$\{y,z\}$
	Out_1	$\{w,x,y,z,x{\rightarrow}l,x{\rightarrow}r\}$		$\{w\}$	$\{w,y,z\}$	$\{w\}$
2	In_2	$\{w,x,y,z,x{\rightarrow}l,x{\rightarrow}r\}$	-		$\{w,y,z\}$	$\emptyset$
	Out_2	$\{w,x,y,z,x{\rightarrow}l,x{\rightarrow}r\}$		$\emptyset$	$\{w,y,z\}$	$\emptyset$
3	In_3	$\{w,x,y,z,x{\rightarrow}l,x{\rightarrow}r,$ $x{\rightarrow}r{\rightarrow}l,x{\rightarrow}r{\rightarrow}r\}$	x		$\{w,y,z,x{\rightarrow}l\}$	$\{x{\rightarrow}l\}$
	Out_3	$\{w,x,y,z,x{\rightarrow}l,x{\rightarrow}r\}$		$\{x,x{\rightarrow}l,$ $x{\rightarrow}r\}$	$\{w,y,z\}$	$\emptyset$
4	In_4	$\{w,x,y,z,x{\rightarrow}l,x{\rightarrow}r,$ $x{\rightarrow}l{\rightarrow}l,x{\rightarrow}l{\rightarrow}r,$ $x{\rightarrow}l{\rightarrow}l{\rightarrow}l,$ $x{\rightarrow}l{\rightarrow}l{\rightarrow}r\}$	y		$\{w,y,z,x{\rightarrow}r,$ $x{\rightarrow}l{\rightarrow}r,$ $x{\rightarrow}l{\rightarrow}l{\rightarrow}l,$ $x{\rightarrow}l{\rightarrow}l{\rightarrow}r\}$	$\{x{\rightarrow}r,$ $x{\rightarrow}l{\rightarrow}r,$ $x{\rightarrow}l{\rightarrow}l{\rightarrow}l,$ $x{\rightarrow}l{\rightarrow}l{\rightarrow}r\}$
	Out_4	$\{w,x,y,z,x{\rightarrow}l,x{\rightarrow}r,$ $y{\rightarrow}l,y{\rightarrow}r,$ $x{\rightarrow}l{\rightarrow}l,x{\rightarrow}l{\rightarrow}r,$ $y{\rightarrow}l{\rightarrow}l,y{\rightarrow}l{\rightarrow}r\}$		$\{y,y{\rightarrow}l,$ $y{\rightarrow}r,$ $y{\rightarrow}l{\rightarrow}l,$ $y{\rightarrow}l{\rightarrow}r\}$	$\{w,z,x{\rightarrow}r,y{\rightarrow}r,$ $x{\rightarrow}l{\rightarrow}r,$ $y{\rightarrow}l{\rightarrow}l,y{\rightarrow}l{\rightarrow}r\}$	$\{y{\rightarrow}r,$ $y{\rightarrow}l{\rightarrow}l,$ $y{\rightarrow}l{\rightarrow}r\}$
5	In_5	$\{w,x,y,z,x{\rightarrow}l,x{\rightarrow}r,$ $y{\rightarrow}l,y{\rightarrow}r,$ $x{\rightarrow}l{\rightarrow}l,x{\rightarrow}l{\rightarrow}r,$ $y{\rightarrow}l{\rightarrow}l,y{\rightarrow}l{\rightarrow}r\}$	z		$\{w,z,x{\rightarrow}r,y{\rightarrow}r,$ $x{\rightarrow}l{\rightarrow}r,$ $y{\rightarrow}l{\rightarrow}l,y{\rightarrow}l{\rightarrow}r\}$	$\emptyset$
	Out_5	$\{w,x,y,z,x{\rightarrow}l,x{\rightarrow}r,$ $y{\rightarrow}l,y{\rightarrow}r,z{\rightarrow}l,z{\rightarrow}r,$ $x{\rightarrow}l{\rightarrow}l,x{\rightarrow}l{\rightarrow}r,$ $y{\rightarrow}l{\rightarrow}l,y{\rightarrow}l{\rightarrow}r\}$		$\{z\}$	$\{w,x{\rightarrow}r,y{\rightarrow}r,$ $z{\rightarrow}l,z{\rightarrow}r,x{\rightarrow}l{\rightarrow}r,$ $y{\rightarrow}l{\rightarrow}l,y{\rightarrow}l{\rightarrow}r\}$	$\{z{\rightarrow}l,z{\rightarrow}r\}$
6	In_6	$\{w,x,y,z,x{\rightarrow}l,x{\rightarrow}r,$ $y{\rightarrow}l,y{\rightarrow}r,z{\rightarrow}l,z{\rightarrow}r,$ $x{\rightarrow}l{\rightarrow}l,x{\rightarrow}l{\rightarrow}r,$ $y{\rightarrow}l{\rightarrow}l,y{\rightarrow}l{\rightarrow}r\}$	y		$\{w,x{\rightarrow}r,y{\rightarrow}r,$ $z{\rightarrow}l,z{\rightarrow}r,x{\rightarrow}l{\rightarrow}r,$ $y{\rightarrow}l{\rightarrow}l,y{\rightarrow}l{\rightarrow}r\}$	$\emptyset$
	Out_6	$\{w,x,y,z,x{\rightarrow}l,x{\rightarrow}r,$ $y{\rightarrow}l,y{\rightarrow}r,z{\rightarrow}l,z{\rightarrow}r,$ $x{\rightarrow}l{\rightarrow}l,x{\rightarrow}l{\rightarrow}r\}$		$\{y,y{\rightarrow}l,$ $y{\rightarrow}r\}$	$\{w,x{\rightarrow}r,y{\rightarrow}l,$ $y{\rightarrow}r,z{\rightarrow}l,z{\rightarrow}r,$ $x{\rightarrow}l{\rightarrow}l,x{\rightarrow}l{\rightarrow}r\}$	$\{y{\rightarrow}l,y{\rightarrow}r,$ $x{\rightarrow}l{\rightarrow}l\}$
7	In_7	$\{w,x,y,z,x{\rightarrow}l,x{\rightarrow}r,$ $y{\rightarrow}l,y{\rightarrow}r,z{\rightarrow}l,z{\rightarrow}r,$ $x{\rightarrow}l{\rightarrow}l,x{\rightarrow}l{\rightarrow}r\}$	-		$\{w,x{\rightarrow}r,y{\rightarrow}l,$ $y{\rightarrow}r,z{\rightarrow}l,z{\rightarrow}r,$ $x{\rightarrow}l{\rightarrow}l,x{\rightarrow}l{\rightarrow}r\}$	$\emptyset$
	Out_7	$\{w,x,y,z,x{\rightarrow}l,x{\rightarrow}r,$ $y{\rightarrow}l,y{\rightarrow}r,z{\rightarrow}l,z{\rightarrow}r,$ $x{\rightarrow}l{\rightarrow}l,x{\rightarrow}l{\rightarrow}r\}$		$\emptyset$	$\{w,x,y,z,x{\rightarrow}l,x{\rightarrow}r,$ $y{\rightarrow}l,y{\rightarrow}r,z{\rightarrow}l,z{\rightarrow}r,$ $x{\rightarrow}l{\rightarrow}l,x{\rightarrow}l{\rightarrow}r\}$	$\{x,y,z\}$

Fig. 11. Null insertion for the program in Figure 1.

that case, we will be inserting $\alpha_y.\sigma = null$ at In_i and $\alpha_x.\sigma = null$ at Out_i. Clearly, the latter *null* assignment is redundant in this case and can be avoided by checking if $\rho_y \triangleright \sigma$ is nullable at In_i.

If must-alias analysis is performed then redundant assignments can be reduced further. Recall from Section 2.3 that must-link-alias relation is symmetric, reflexive, and transitive and hence an equivalence relation. Therefore, the set of candidate paths at a program point can be divided into equivalence classes based on must-link-alias relation. Redundant *null* assignments can be reduced by nullifying atmost one access path in any equivalence class.

6. CONVERGENCE OF HEAP REFERENCE ANALYSIS

The *null* assignment insertion algorithm makes a single traversal over the control flow graph. We show the termination of alias and liveness analysis using the properties of access graph operations. Termination of availability and anticipability can be shown by similar arguments over finite sets of bounded access paths.

6.1 Monotonicity

For a program there are a finite number of basic blocks, a finite number of fields for any root variable, and a finite number of field names in any access expression. Hence the number of access graphs for a program is finite. Further, the number of nodes and hence the size of each access graph, is bounded by the number of labels which can be created for a program.

Access graphs for a variable x form a complete lattice with a partial order $\sqsubseteq_G$ induced by $\uplus$. Note that $\uplus$ is commutative, idempotent, and associative. Let $G = \langle x, N_F, N_I, E\rangle$ and $G' = \langle x, N'_F, N'_I, E'\rangle$. The partial order $\sqsubseteq_G$ is defined as

$$G \sqsubseteq_G G' \Leftrightarrow \left(N'_F \subseteq N_F\right) \wedge \left(N'_I \subseteq (N_F \cup N_I)\right) \wedge \left(E' \subseteq E\right) \qquad (12)$$

Clearly, $G \sqsubseteq_G G'$ implies that G contains all access paths of G'. We extend $\sqsubseteq_G$ to a set of access graphs as follows:

$$S_1 \sqsubseteq_S S_2 \Leftrightarrow \forall G_2 \in S_2, \exists G_1 \in S_1 \text{ s.t. } G_1 \sqsubseteq_G G_2$$

It is easy to verify that $\sqsubseteq_G$ is reflexive, transitive, and antisymmetric. For a given variable x, the access graph $\mathcal{E}_G$ forms the $\top$ element of the lattice while the $\perp$ element is a greatest lower bound of all access graphs.

The partial order over access graphs and their sets can be carried over unaltered to factorised subgraphs ($\sqsubseteq_{FG}$) and their sets ($\sqsubseteq_{FS}$), with the added condition that ε_{FG} is incomparable to any other non empty factorized graph, i.e. for any non empty factorized subgraph $F \neq \varepsilon_{FG}$, we have $F \not\sqsubseteq_{FG} \varepsilon_{FG}$ and $\varepsilon_{FG} \not\sqsubseteq_{FG} F$.

Access graph operations are monotonic as described in Figure 12. Path removal and factorisation are monotonic in the first argument but not in the second argument. However, we show that in each context where they are used, the resulting functions are monotonic:

(1) Path removal is used only for an assignment $\alpha_x = \alpha_y$. Its second argument is ρ_x for both alias and liveness analysis. Since ρ_x is constant for any assignment statement $\alpha_x = \alpha_y$, the resulting flow functions are monotonic.

(2) Factorisation is used in the following situations:

  (a) *Link-alias computation*. Since the first argument of link-alias computation is also the first argument of factorisation, link-alias computation is monotonic in the first

Operation	Monotonicity
Union	$G_1 \sqsubseteq_G G'_1 \wedge G_2 \sqsubseteq_G G'_2 \;\Rightarrow\; G_1 \uplus G_2 \sqsubseteq_G G'_1 \uplus G'_2$
Path Removal	$G \sqsubseteq_G G' \;\Rightarrow\; G \ominus \rho \sqsubseteq_G G' \ominus \rho$
Factorisation	$G \sqsubseteq_G G' \;\Rightarrow\; G/G'' \sqsubseteq_{FS} G'/G''$
Extension	$S_1^F \sqsubseteq_S S_2^F \wedge G_1 \sqsubseteq_G G_2 \;\Rightarrow\; G_1 \# S_1^F \sqsubseteq_G G_2 \# S_2^F$
Link-Alias Computation	$G_1 \sqsubseteq_G G_2 \wedge AS_1 \supseteq AS_2 \;\Rightarrow\; \mathsf{Ln}\mathbb{A}(G_1, AS_1) \sqsubseteq_S \mathsf{Ln}\mathbb{A}(G_2, AS_2)$

Fig. 12. Monotonicity of Access Graph Operations

.

argument. The second argument of link-alias computation is used to derive the second argument of factorisation. However, it is easy to verify that

$$\begin{aligned} AS_1 \supseteq AS_2 \;&\Rightarrow\; \mathsf{Ln}\mathbb{A}(G_v, AS_1) = \mathsf{Ln}\mathbb{A}(G_v, AS_2) \cup \mathsf{Ln}\mathbb{A}(G_v, AS_1 - AS_2) \\ &\Rightarrow\; \mathsf{Ln}\mathbb{A}(G_v, AS_1) \sqsubseteq_S \mathsf{Ln}\mathbb{A}(G_v, AS_2) \end{aligned}$$

Thus link-alias computation is monotonic in both arguments.

(b) *Alias analysis*. Factorisation is used for the flow function corresponding to an assignment $\alpha_x = \alpha_y$ and its second argument is $GOnly(\rho_y)$. Since $GOnly(\rho_y)$ is constant for any assignment statement $\alpha_x = \alpha_y$, the resulting flow functions are monotonic.

(c) *Liveness analysis*. Factorisation is used for the flow function corresponding to an assignment $\alpha_x = \alpha_y$ and its second argument is $GOnly(\rho_x)$. Since $GOnly(\rho_x)$ is constant for any assignment statement $\alpha_x = \alpha_y$, the resulting flow functions are monotonic.

Thus we conclude that all flow functions are monotonic. Since lattices are finite, termination of heap reference analysis follows.

6.2 Complexity

This section discusses the issues which influence the complexity and efficiency of performing heap reference analysis. Empirical measurements which corroborate the observations made in this section are presented in Section 8.

The data flow frameworks defined in this paper are not *separable* [Khedker 2002] because the data flow information of a variable depends on the data flow information of other variables. Thus the number of iterations over control flow graph is not bounded by the depth of the graph [Aho et al. 1986; Hecht 1977; Khedker 2002] but would also depend on the number of root variables which depend on each other.

Although we consider each statement to be a basic block, our control flow graphs retain only statements involving references. A further reduction in the size of control flow graphs follows from the fact that successive use statements need not be kept separate and can be grouped together into a block which ends on a reference assignment.

The amount of work done in each iteration is not fixed but depends on the size of access graphs. Of all operations performed in an iteration, only $CFN(G, G')$ is costly. Conversion to deterministic access graphs is also a costly operations but is performed for a single pass during *null* assignment insertion. In practice, the access graphs are quite small because of

the following reason: Recall that edges in access graphs capture dependence of a reference made at one program point on some other reference made at another point (Section 3.1). In real programs, starting from a root variable reference, chains of such dependences are quite small in size. This is corroborated by Figure 14 which provides the empirical data for the access graphs in our examples. The average number of nodes in these access graphs is less than 7 while the average number of edges is less than 12. Hence the complexities of access graph operations is not a matter of concern.

7. SAFETY OF *NULL* ASSIGNMENT INSERTION

We have to prove that the *null* assignments inserted by our algorithm (Section 5) in a program are safe in that they do not alter the result of executing the program. We do this by showing that (a) an inserted statement itself does not raise an exception, and (b) an inserted statement does not affect any other statement, both original and inserted.

We use the subscripts b and a for a program point p to denote "before" and "after" in an execution order. Further, the corresponding program points in the original and modified program are distinguished by the superscript o and m. The correspondence is defined as follows: If p^m is immediately before or after an inserted assignment $\alpha = \mathit{null}$, p^o is the point where the decision to insert the *null* assignment is taken. For any other p^m, there is an obvious p^o.

We first assert the soundness of availability, anticipability and alias analyses without proving them.

LEMMA 7.1. (Soundness of Availability Analysis). *Let AV_{p_a} be the set of access paths available at program point p_a. Let $\rho \in AV_{p_a}$. Then along every path reaching p_a, there exists a program point p_b, such that the link represented by $\mathit{Frontier}(\rho)$ is either dereferenced or assigned a non-null l-value at p_b and is not made null between p_b and p_a.*

LEMMA 7.2. (Soundness of Anticipability Analysis). *Let AN_p be the set of access paths anticipable at program point p. Let $\rho \in AN_p$. Then along every path starting from p, the link represented by $\mathit{Frontier}(\rho)$ is dereferenced before being assigned.*

For semantically valid input programs (i.e. programs which do not generate exceptions), Lemma 7.1 and Lemma 7.2 guarantee that if ρ is available or anticipable at p, $\mathit{Target}(\rho)$ can be dereferenced at p.

LEMMA 7.3. (Soundness of Alias Analysis). *Let $\mathit{Frontier}(\rho_x)$ represents the same link as $\mathit{Frontier}(\rho_y)$ at a program point p during some execution of the program. Then link-alias computation of ρ_x at p would discover ρ_y to be link-aliased to ρ_x.*

For the main claim, we relate the access paths at p_a to the access paths at p_b by incorporating the effect of intervening statements only, regardless of the statements executed before p_b. In some execution of a program, let ρ be the access path of interest at p_a and the sequence of statements between p_b and p_a be s. Then $T(s,\rho)$ represents the access path at p_b which, if non-$\mathcal{E}$, can be used to access the link represented by $\mathit{Frontier}(\rho)$. $T(s,\rho)$

captures the transitive effect of backward transfers of ρ through s. T is defined as follows:

$$T(s,\rho) = \begin{cases} \rho & s \text{ is a use statement} \\ \rho & s \text{ is } \alpha_x = \ldots \text{ and } \rho_x \text{ is not a prefix of } \rho \\ \mathcal{E} & s \text{ is } \alpha_x = \mathit{New} \text{ and } \rho = \rho_x \rightarrow\!\triangleright \sigma \\ \mathcal{E} & s \text{ is } \alpha_x = \mathsf{null} \text{ and } \rho = \rho_x \rightarrow\!\triangleright \sigma \\ \rho_y \rightarrow\!\triangleright \sigma & s \text{ is } \alpha_x = \alpha_y \text{ and } \rho = \rho_x \rightarrow\!\triangleright \sigma \\ T(s_1, T(s_2,\rho)) & s \text{ is a sequence } s_1; s_2 \end{cases}$$

LEMMA 7.4. (Liveness Propagation). *Let ρ^a be in some explicit liveness graph at p_a. Let the sequence of statements between p_b to p_a be s. Then, if $T(s,\rho^a) = \rho^b$ and ρ^b is not $\mathcal{E}$, then ρ^b is in some explicit liveness graph at p_b.*

PROOF. The proof is by structural induction on s. Since ρ^b is non-$\mathcal{E}$, the base cases are:

(1) s is a use statement. In this case $\rho^b = \rho^a$.

(2) s is an assignment $\alpha_x = \ldots$ such that ρ_x is not a prefix of ρ^a. Here also $\rho^b = \rho^a$.

(3) s is an assignment $\alpha_x = \alpha_y$ such that $\rho^a = \rho_x \rightarrow\!\triangleright \sigma$. In this case $\rho^b = \rho_y \rightarrow\!\triangleright \sigma$.

For (1) and (2), since ρ^a is not in $\mathsf{E\mathbb{L}KillPath}$, ρ^b is in some explicit liveness graph at p_b. For (3), from Equation (8), ρ^b is in some explicit liveness graph at p_b.

For the inductive step, assume that the lemma holds for s_1 and s_2. From the definition of T, there exists a non-$\mathcal{E}$ ρ^i at the intermediate point p_i between s_1 and s_2, such that $\rho^i = T(s_2,\rho^a)$ and $\rho^b = T(s_1,\rho^i)$. Since ρ^a is in some explicit liveness graph at p_a, by the induction hypothesis, ρ^i must be in some explicit liveness graph at p_i. Further, by the induction hypothesis, ρ^b must be in some explicit liveness graph at p_b. □

LEMMA 7.5. *Every access path which is in some explicit liveness graph at p_b^m is also in some explicit liveness graph at p_b^o.*

PROOF. If an extra explicitly live access path is is introduced in the modified program, it could be only because of an inserted assignment $\alpha = \mathsf{null}$ at some p_a^m. The only access paths which this statement can add to an explicit liveness graph are the paths corresponding the proper prefixes of α. However, the algorithm selects α for nullification only if the access paths corresponding to all its proper prefixes are in some explicit liveness graph. Therefore every access path which is in some explicit liveness graph at p_a^m is also in some explicit liveness graph at p_a^o. The same relation would hold at p_b^m and p_b^o. □

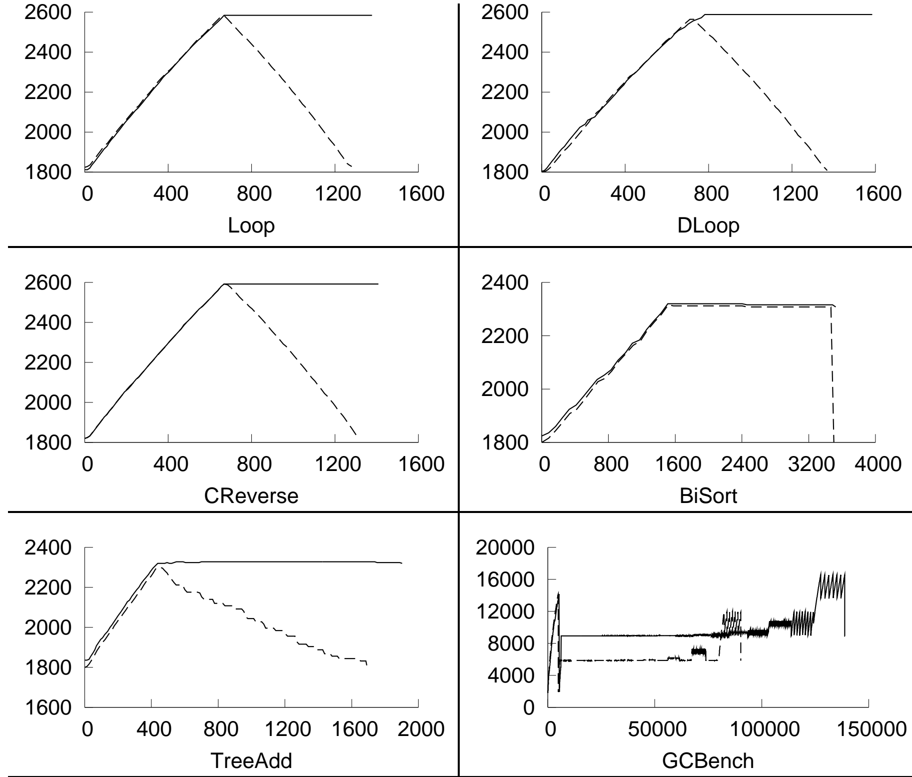
THEOREM 7.1. (Safety of *null* insertion). *Let the assignment $\alpha^b = \mathsf{null}$ be inserted by the algorithm immediately before p_b^m. Then:*

(*1*) *Execution of $\alpha^b = \mathsf{null}$ does not raise any exception due to dereferencing.*

(*2*) *Let α^a be used immediately after p_a^m (in an original statement or an inserted null assignment). Then, execution of $\alpha^b = \mathsf{null}$ cannot nullify any link used in α^a.*

PROOF. We prove the two parts separately.

(1) If α^b is a root variable, then the execution of $\alpha^b = \mathsf{null}$ cannot raise an exception. When α^b is not a root variable, from the null assignment algorithm, every proper prefix ρ' of ρ^b is either anticipable or available. From the soundness of both these analyses, $\mathit{Target}(\rho')$ exists and the execution of $\alpha^b = \mathsf{null}$ cannot raise an exception.



X axis indicates measurement instants in milliseconds. Y axis indicates heap usage in KB. Solid line represents memory required for original program while dashed line represents memory for the modified program. Observe that the modified program executed faster than the original program in each case.

Fig. 13. Temporal plots of memory usages.

(2) We prove this by contradiction. Let s denote the sequence of statements between p_b^m and p_a^m. Assume that $\alpha^b = \mathit{null}$ nullifies a link used in α^a. This is possible only if there exists a prefix ρ' of ρ^a such that $T(s,\rho')$ shares its frontier with ρ^b at p_b^m. Since *null* assignments do not generate any new aliases, this sharing must also hold at p_b^o. Since α^a is used at p_a^m, ρ' is in some explicit liveness graph at p_a^m. By Lemma 7.4, $T(s,\rho')$ must be in some explicit liveness graph at p_b^m and hence at p_b^o (Lemma 7.5). Since $\mathit{Frontier}(T(s,\rho'))$ is same as $\mathit{Frontier}(\rho^b)$ at p_b^o, ρ^b must be discovered to be link-aliased to $\mathit{Frontier}(T(s,\rho'))$ at p_b^o (Lemma 7.3). Hence ρ^b is in some liveness graph at p_b^o. Thus a decision to insert $\alpha^b = \mathit{null}$ cannot be taken at p_b^o.

□

8. EMPIRICAL MEASUREMENTS

In order to show the effectiveness of heap reference analysis, we have implemented a prototype heap reference analyzer.

Program Name	Analysis		Access Graphs					#*null*	Execution		
		Time		Nodes		Edges			Time (sec)		%
	#Iter	(sec)	#G	Avg	Max	Avg	Max		Orig.	Mod.	Gain
`Loop`	5	0.082	172	1.13	2	0.78	2	9	1.503	1.388	8.3
`DLoop`	5	1.290	332	2.74	4	5.80	10	11	1.594	1.470	8.4
`CReverse`	5	0.199	242	1.41	4	1.10	6	8	1.512	1.414	6.9
`BiSort`	6	0.083	63	2.16	3	3.81	6	5	3.664	3.646	0.5
`TreeAdd`	6	0.255	132	2.84	7	4.87	14	7	1.976	1.772	11.5
`GCBench`	6	0.247	136	2.73	7	4.63	14	7	132.99	88.86	49.7

—#Iter is the maximum number of iterations taken by any analysis.
—Analysis Time is the total time taken by all analyses.
—#G is total number of access graphs created by alias analysis and liveness analysis.
—Max nodes (edges) is the maximum over number of nodes (edges) in all access graphs.
—Avg nodes (edges) is the average number of nodes (edges) over all access graphs.
—#*null* is the number of inserted *null* assignments.

Fig. 14. Preliminary Measurements for Original and Modified Programs.

8.1 Experimentation Methodology

The prototype has been implemented in XSB-Prolog[9]. The measurements were made on a 800 MHz Pentium III machine with 128 MB memory running Fedora Core release 2. The benchmarks used were `Loop`, `DLoop`, `CReverse`, `BiSort`, `TreeAdd` and `GCBench`. Three of these (`Loop`, `DLoop` and `CReverse`) are similar to those in [Shaham et al. 2003]. `Loop` creates a singly linked list and traverses it, `DLoop` is doubly linked list variation of the same program, `CReverse` reverses a singly linked list. `BiSort` and `TreeAdd` are taken from Java version of Olden benchmark suite [Carlisle 1996]. `GCBench` is taken from [Boehm]. For `TreeAdd`, we have analyzed `addtree` function only, and for `GCBench`, we have analyzed `Populate` function only.

The Java programs were manually translated to Prolog representations. Since our analysis is currently restricted to intraprocedural level, for the purpose of analysis, the original Java programs were approximated in the Prolog representations in the following manner: Calls to non-recursive functions were inlined and calls to recursive functions were replaced by iterative constructs which approximated the liveness property of heap manipulations in the function bodies. The result of the analysis was used to manually insert *null* assignments in the original Java programs to create modified Java programs.

The prototype implementation, along with the test programs (with their original, modified, and Prolog versions) are available at [Karkare 2005].

8.2 Measurements and Observations

Our experiments were directed at measuring:

(1) *The efficiency of Analysis*. We measured the total time required, number of iterations of round robin analyses, and the number and sizes of access graphs.

(2) *The effectiveness of null assignment insertions*. The programs were made to create huge data structures. Memory usage was measured by explicit calls to garbage collector in both modified and original Java programs at specific probing points such as

[9]Available from `http://xsb.sourceforge.net`.

call sites, call returns, loop begins and loop ends. The overall execution time for the original and the modified programs was also measured.

The results of our experiments are shown in Figure 13 and Figure 14. As can be seen from Figure 13, nullification of links helped the garbage collector to collect a lot more garbage, thereby reducing the allocated heap memory. In case of BiSort, however, the links were last used within a recursive procedure which was called multiple times. Hence, safety criteria prevented *null* assignment insertion within the called procedure. Our analysis could only nullify the root of the data structure at the end of the program. Thus the memory was released only at the end of the program.

As can be seen from Figure 14, modified programs executed faster. In general, a reduction in execution time can be attributed to the following two factors: (a) a decrease in the number of calls to garbage collector and (b) reduction in the time taken for garbage collection in each call. The former is possible because of availability of a larger amount of free memory, the latter is possible because lesser reachable memory needs to be copied.[10] In our experiments, factor (a) above was absent because the number of (explicit) calls to garbage collector were kept same. `GCBench` showed a large improvement in execution time after *null* assignment insertion. This is because `GCBench` creates large trees in heap, which are not used in the program after creation and our implementation was able to nullify left and right subtrees of these trees immediately after their creation. This also reduced the high water mark of the heap memory requirement.

As explained in Section 6.2, sizes of the access graphs (average number of nodes and edges) is small. This can be verified from Figure 14. The analysis of `DLoop` creates a large number of access graphs because of the presence of cycles in heap. In such a case, a large number of alias pairs are generated, many of which are redundant. Though it is possible to reduce analysis time by eliminating redundant alias pairs, our prototype does not do so for the sake of simplicity.

Our technique and prototype implementation compares well with the technique and results described in [Shaham et al. 2003]. The implementation described in [Shaham et al. 2003] runs on a 900 MHz P-III with 512 MB RAM running Windows 2000. It takes 1.76 seconds, 2.68 seconds and 4.79 seconds respectively for `Loop`, `DLoop` and `CReverse` for *null* assignment insertion. Time required by our prototype implementation for the above mentioned programs is given in Figure 14. Our implementation automatically computes the program points for *null* insertion whereas their method cannot do so (Section 9.3). Our prototype performs much better in all cases. We expect a C++ implementation to be much more efficient than XSB-Prolog prototype.

9. RELATED WORK

The properties of heap which have been explored in past are listed below (more details can be found in [Sagiv et al. 2002b]). All of these properties (except the last) are specific to a program point under consideration.

(1) Properties of access expressions.
   (a) *Nullity.* Does an access expression evaluate to *null* address?
   (b) *Aliasing.* Do two access expressions evaluate to the same address?
(2) Properties of heap cells.

[10] This happens because Java Virtual Machine uses a copying garbage collector.

   (a) *Sharing.* Is a heap cell a part of two data structures?
   (b) *Reachability.* Is a heap cell accessible? Through a particular access expression?
   (c) *Cyclicity.* Is a heap cell part of a cycle?
   (d) *Liveness.* Is a heap cell accessed beyond the current program point?

(3) Properties of heap data structures.
   (a) *Disjointness.* Do two data structure have a common heap cell?
   (b) *Shape.* Is a data structure a tree, a DAG, a graph, a cyclic list etc?

(4) Properties of procedures manipulating heap.
   (a) *Memory leaks.* Does a procedure leave behind unreachable heap cells?

Most of these properties are related to each other; they basically differ in the domain for which the question is being asked. In the rest of this section, we review the related work in the two main properties of interest: aliasing and liveness. We are not aware of past work in availability and anticipability analysis of heap references.

9.1 Alias Analysis of Heap Data

There is a plethora of literature on alias analysis of heap data. The distinguishing features of various investigations have been:

(1) *Applications of alias analysis.* Alias analysis increases the precision of information for various optimizing transformations. Apart from the traditional optimizations, it also helps in parallelization [Larus 1989], compile time garbage collection, improved register allocation, improved code generation and debugging etc. [Wilson 1997].

(2) *Type of aliases discovered.* Various categories of aliases which have been explored are: may-aliases or must-aliases, flow-sensitive aliases or flow-insensitive aliases. In the case of interprocedural analysis aliases could be context-sensitive or context-insensitive [Larus and Hilfinger 1988; Choi et al. 1993; Emami et al. 1994; Muchnick 1997; Hind and Pioli 1998; Hind et al. 1999].

(3) *Aliasing models.* In Java, pointers (references) can point only to the heap data and not to stack data. Further, pointer dereferencing is implicit and pointer arithmetic and explicit type casting is not allowed. In C/C++, pointers can point to stack as well as heap data; pointer arithmetic is allowed and pointer and non-pointer data can be mixed.

(4) *Representations and summarization heuristics used.* Various representations for storing alias information include: Graph Representation [Larus and Hilfinger 1988], Compact Representation, Explicit Representation [Choi et al. 1993; Hind et al. 1999] and Points-to Abstraction [Ghiya 1992; Emami et al. 1994]. Some techniques used for approximating alias information are: k-limiting [Jones and Muchnick 1979], s-l limiting [Larus and Hilfinger 1988], and Storage Shape Graph (SSG) [Jones and Muchnick 1982; Chase et al. 1990]. None of them capture the pattern of heap manipulations as directly as access graphs because
   —their primary objective is to identify the shape of the data, rather than the underlying heap manipulations which create the shape, or
   —they use only memory allocation points in the program for summarization and other address assignments are considered only implicitly.

   Further, most of them either do not seem to handle cycles in heap or deal with them in an ad hoc manner.

Shape analysis [Sagiv et al. 1999; 2002a; 2002b] is a general method of creating suitable abstractions (called Shape Graphs) of heap memory with respect to the relevant properties. Program execution is then modeled as operations on shape graphs. It seems to be emerging as the most promising technique for analyzing heap data.

9.2 Liveness Analysis

As noted earlier little or no success has been achieved in analysing liveness of heap objects; most of the reported literature does not address liveness of individual objects. Since the precision of a garbage collector depends on its ability to distinguish between reachable heap objects and live heap objects, even state of art garbage collectors leave a significant amount of garbage uncollected [Agesen et al. 1998; Shaham et al. 2001b; 2001a; 2002]. All reported attempts to incorporate liveness in garbage collection have been quite approximate. The known approaches have been:

(1) *Liveness of root variables.* A popular approach (which has also been used in some state of art garbage collectors) involves identifying liveness of root variable on the stack. All heap objects reachable from the live root variables are considered live [Agesen et al. 1998].

(2) *Imposing stack discipline on heap objects.* These approaches try to change the statically unpredictable lifetimes of heap objects into predictable lifetimes similar to stack data. They can be further classified as

- *Allocating objects on call stack.* These approach try to detect which objects can be allocated on stack frame so that they are automatically deallocated without the need of traditional garbage collection. A profile based approach which tracks the last use of an object is reported in [McDowell 1998], while a static analysis based approach is reported in [Reid et al. 1999].

  Some approaches ask a converse question: which objects are unstackable (i.e. their lifetimes outlive the procedure which created it)? They use abstract interpretation and perform *escape analysis* to discover objects which *escape* a procedure[Blanchet 1999; 2003; Choi et al. 1999]. All other objects are allocated on stack.
- *Associating objects with call stack* [Cannarozzi et al. 2000]. This approach identifies the stackability. The objects are allocated in the heap but are associated with a stack frame and the runtime support is modified to deallocate these (heap) objects when the associated stack frame is popped.
- *Allocating objects on separate stack.* This approach uses a static analysis called *region inference* [Tofte and Birkedal 1998; Hallenberg et al. 2002] to identify *regions* which are storages for objects. These regions are allocated on a separate region stack.

All these approaches require modifying the runtime support for the programs.

(3) The *Heap Safety Automaton* based approach [Shaham et al. 2003] is a recently reported work which comes closest to our approach since it tries to determine if a reference can be made *null*. We discuss this approach in the next section.

9.3 Heap Safety Automaton Based Approach

This approach models safety of inserting a null statement at a given point by an automaton. A shape graph based abstraction of the program is then model-checked against the heap

safety automaton. Additionally, they also consider freeing the object; our approach can be easily extended to include freeing.

The fundamental differences between the two approaches are

—Their method answers the following question: Given an access expression and a program point, can the access expression be set to *null* immediately after that program point? However, they leave a very important question unanswered: Which access expressions should we consider and at which point in the program? It is impractical to use their method to ask this question for every pair of access expression and program point. Our method answers both the questions by finding out appropriate access expressions and program points.

—We insert *null* assignments at the earliest possible point. The effectiveness of any method to improve garbage collection depends crucially on this aspect. Their method does not address this issue directly.

—As noted in Section 8, their method is inefficient in practice. For a simple Java program containing 11 lines of executable statements, it takes over 1.37 MB of storage and takes 1.76 seconds for answering the question: Can the variable y be set to *null* after line 10?

Hence our approach is superior to their approach in terms of completeness, effectiveness, and efficiency.

10. CONCLUSIONS AND FURTHER WORK

Two fundamental challenges in analysing heap data are that the temporal and spatial structures of heap data seem arbitrary and are unbounded. The apparent arbitrariness arises due to the fact that the mapping between access expressions and l-values varies dynamically.

We create an abstract representation of heap in terms of sets of access paths. Further, a bounded representation, called access graphs, is used for summarizing sets of access paths. Summarization is based on the fact that the heap can be viewed as consisting of repeating patterns which bear a close resemblance to the program structure. Access graphs capture this fact directly by tagging program points to access graph nodes. Unlike [Horwitz et al. 1989; Chase et al. 1990; Choi et al. 1993; Wilson and Lam 1995; Hind et al. 1999] where only memory allocation points are remembered, we remember all program points where references are used. This allows us to combine data flow information arising out of the same program point, resulting in bounded representations of heap data. These representations are simpler, more precise, and more natural than the traditional representations.

The dynamically varying mapping between access expressions and l-values is handled by abstracting out regions in the heap which can possibly be accessed by a program. These regions are represented by sets of access paths and access graphs which are manipulated using a carefully chosen set of operations. The computation of access graphs and access paths using data flow analysis is possible because of their finiteness and the monotonicity of the chosen operations. We define data flow analyses for aliasing, liveness, availability and anticipability of heap references. Aliasing and liveness analyses are any path problems, hence they involve unbounded information requiring access graphs as data flow values. Availability and anticipability analyses are all paths problems, hence they involve bounded information which is represented by finite sets of access paths.

An immediate application of these analyses is a technique to improve garbage collection. This technique works by identifying objects which are dead and rendering them unreachable by setting them to null as early as possible. Though this idea was previously known to

yield benefits [Gadbois et al.], nullification of dead objects was based on profiling [Shaham et al. 2001a; 2002]. Our method, instead, is based on static analysis.

We intend to pursue future work in the following two directions:

(1) *Heap reference analysis*. We find some scope of improvements and a need of some important extensions.
 —*Improvements*. We would like to minimize the generation of redundant aliases in presence of cycles in heap. Besides, we would like to devise a better *null* insertion algorithm which ensures that if ρ and ρ' are link-aliased and nullable, only one of them is used for *null* assignment.
 —*Extensions.* We would like to perform this analysis at interprocedural level and also analyze array fragments. We would also like to implement this approach for C/C++ and use it for plugging memory leaks statically.

(2) *Applying the summarization heuristic to other analyses*. Our initial explorations indicate that a similar approach would be useful for extending static inferencing of flow-sensitive monomorphic types [Khedker et al. 2003] to include polymorphic types. This is possible because polymorphic types represent an infinite set of types and hence discovering them requires summarizing unbounded information.

ACKNOWLEDGMENTS

Several people have contributed to this work. We would particularly like to thank Neha Adkar, C Arunkumar, Mugdha Jain, and Reena Kharat. Neha's work was supported by Tata Infotech Ltd. A prototype implementation of this work was partially supported by Synopsys India Ltd. Amey Karkare has been supported by Infosys Fellowship.

REFERENCES

AGESEN, O., DETLEFS, D., AND MOSS, J. E. 1998. Garbage collection and local variable type-precision and liveness in Java virtual machines. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*. ACM Press, New York, NY, USA, 269–279.

AHO, A. V., SETHI, R., AND ULLMAN, J. D. 1986. *Compilers – Principles, Techniques, and Tools*. Addison-Wesley.

BLANCHET, B. 1999. Escape analysis for object-oriented languages: application to Java. In *OOPSLA '99: Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. ACM Press, New York, NY, USA, 20–34.

BLANCHET, B. 2003. Escape analysis for Java^{TM}: Theory and practice. *ACM Transactions on Programming Languages and Systems 25,* 6, 713–775.

BOEHM, H. An artifi cial garbage collection benchmark. `http://www.hpl.hp.com/personal/Hans_Boehm/gc/gc_bench.html`.

CANNAROZZI, D. J., PLEZBERT, M. P., AND CYTRON, R. K. 2000. Contaminated garbage collection. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*. ACM Press, New York, NY, USA, 264–273.

CARLISLE, M. C. 1996. Olden: Parallelizing programs with dynamic data structures on distributed-memory machines. Ph.D. thesis, Princeton University.

CHASE, D. R., WEGMAN, M., AND ZADECK, F. K. 1990. Analysis of pointers and structures. In *PLDI '90: Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*. ACM Press, New York, NY, USA, 296–310.

CHENG, B.-C. AND HWU, W.-M. W. 2000. Modular interprocedural pointer analysis using access paths: design, implementation, and evaluation. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*. ACM Press, New York, NY, USA, 57–69.

CHOI, J.-D., BURKE, M., AND CARINI, P. 1993. Effi cient fbw-sensitive interprocedural computation of pointer-induced aliases and side effects. In *POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM Press, New York, NY, USA, 232–245.

CHOI, J.-D., GUPTA, M., SERRANO, M., SREEDHAR, V. C., AND MIDKIFF, S. 1999. Escape analysis for Java. In *OOPSLA '99: Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. ACM Press, New York, NY, USA, 1–19.

EMAMI, M., GHIYA, R., AND HENDREN, L. J. 1994. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*. ACM Press, New York, NY, USA, 242–256.

GADBOIS, D., FITERMAN, C., CHASE, D., SHAPIRO, M., NILSEN, K., HAAHR, P., BARNES, N., AND PIRINEN, P. P. The GC FAQ. `http://www.iecc.com/gclist/GC-faq.html`.

GHIYA, R. 1992. Interprocedural analysis in the presence of function pointers. Tech. rep., ACAPS Technical Memo 62. School of Computer Science, McGill University. December.

HALLENBERG, N., ELSMAN, M., AND TOFTE, M. 2002. Combining region inference and garbage collection. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*. ACM Press, New York, NY, USA, 141–152.

HECHT, M. S. 1977. *Flow Analysis of Computer Programs*. Elsevier North-Holland Inc.

HIND, M., BURKE, M., CARINI, P., AND CHOI, J.-D. 1999. Interprocedural pointer alias analysis. *ACM Transactions on Programming Languages and Systems 21,* 4, 848–894.

HIND, M. AND PIOLI, A. 1998. Assessing the effects of fbw-sensitivity on pointer alias analyses. In *SAS '98: Proceedings of the 5th International Symposium on Static Analysis*. Springer-Verlag, London, UK, 57–81.

HIRZEL, M., DIWAN, A., AND HENKEL, J. 2002. On the usefulness of type and liveness accuracy for garbage collection and leak detection. *ACM Transactions on Programming Languages and Systems 24,* 6, 593–624.

HIRZEL, M., HENKEL, J., DIWAN, A., AND HIND, M. 2002. Understanding the connectivity of heap objects. In *ISMM '02: Proceedings of the 3rd international symposium on Memory management*. ACM Press, New York, NY, USA, 36–49.

HORWITZ, S., PFEIFFER, P., AND REPS, T. 1989. Dependence analysis for pointer variables. In *PLDI '89: Proceedings of the ACM SIGPLAN 1989 Conference on Programming language design and implementation*. ACM Press, New York, NY, USA, 28–40.

IYER, P. C. 2005. PVS based proofs of safety properties of access graph operations. `http://www.cse.iitb.ac.in/~uday/hraResources/AGSafety.html`.

JONES, N. D. AND MUCHNICK, S. S. 1979. Flow analysis and optimization of lisp-like structures. In *POPL '79: Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. ACM Press, New York, NY, USA, 244–256.

JONES, N. D. AND MUCHNICK, S. S. 1982. A fexible approach to interprocedural data fbw analysis and programs with recursive data structures. In *POPL '82: Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM Press, New York, NY, USA, 66–74.

KARKARE, A. 2005. XSB-Prolog based prototype implementation of heap reference analysis. `http://www.cse.iitb.ac.in/~uday/hraResources/hraPrototpye.html`.

KHEDKER, U. P. 2002. Data fbw analysis. In *Compiler Design Handbook: Optimizations and Machine Code Generation*, Y. N. Srikant and P. Shankar, Eds. CRC Press, Inc., Boca Raton, FL, USA.

KHEDKER, U. P., DHAMDHERE, D. M., AND MYCROFT, A. 2003. Bidirectional data fbw analysis for type inferencing. *Computer Languages, Systems and Structures 29,* 1-2, 15–44.

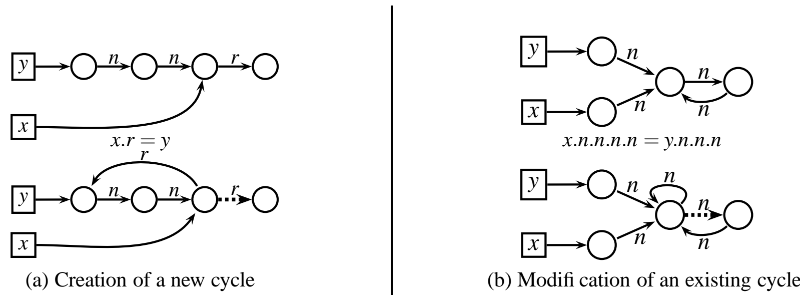
LARUS, J. R. 1989. Restructuring symbolic programs for concurrent execution on multiprocessors. Ph.D. thesis, University of California at Berkeley.

LARUS, J. R. AND HILFINGER, P. N. 1988. Detecting conflcts between structure accesses. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*. ACM Press, New York, NY, USA, 24–31.

MCDOWELL, C. E. 1998. Reducing garbage in java. *SIGPLAN Notices 33,* 9, 84–86.

MUCHNICK, S. S. 1997. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

REID, A., MCCORQUODALE, J., BAKER, J., HSIEH, W., AND ZACHARY, J. 1999. The need for predictable garbage collection. In *Proceedings of the ACM SIGPLAN Workshop on Compiler Support for System Software (WCSSS'99)*.



(a) Creation of a new cycle

(b) Modification of an existing cycle

Fig. 15. Memory graphs before and after assignment showing creation of cycles.

SAGIV, M., REPS, T., AND WILHELM, R. 1999. Parametric shape analysis via 3-valued logic. In *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM Press, New York, NY, USA, 105–118.

SAGIV, M., REPS, T., AND WILHELM, R. 2002a. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems 24,* 3, 217–298.

SAGIV, M., REPS, T., AND WILHELM, R. 2002b. Shape analysis and applications. In *Compiler Design Handbook: Optimizations and Machine Code Generation*, Y. N. Srikant and P. Shankar, Eds. CRC Press, Inc, Boca Raton, FL, USA.

SHAHAM, R., KOLODNER, E. K., AND SAGIV, M. 2001a. Heap profiling for space-efficient java. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*. ACM Press, New York, NY, USA, 104–113.

SHAHAM, R., KOLODNER, E. K., AND SAGIV, M. 2001b. On effectiveness of GC in Java. *SIGPLAN Notices 36,* 1, 12–17.

SHAHAM, R., KOLODNER, E. K., AND SAGIV, M. 2002. Estimating the impact of heap liveness information on space consumption in Java. In *ISMM '02: Proceedings of the 3rd international symposium on Memory management*. ACM Press, New York, NY, USA, 64–75.

SHAHAM, R., YAHAV, E., KOLODNERE, E. K., AND SAGIV, S. 2003. Establishing local temporal heap safety properties with applications to compile-time memory management. In *SAS '03: Proceedings of the 10th International Symposium on Static Analysis*. Springer-Verlag, London, UK, 483–503.

TOFTE, M. AND BIRKEDAL, L. 1998. A region inference algorithm. *ACM Transactions on Programming Languages and Systems 20,* 4, 724–767.

WILSON, R. P. 1997. Efficient, context-sensitive pointer analysis for C programs. Ph.D. thesis, Stanford University.

WILSON, R. P. AND LAM, M. S. 1995. Efficient, context-sensitive pointer analysis for C programs. In *PLDI '95: Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*. ACM Press, New York, NY, USA, 1–12.

YONG, S. H., HORWITZ, S., AND REPS, T. 1999. Pointer analysis for programs with structures and casting. In *PLDI '99: Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*. ACM Press, New York, NY, USA, 91–103.

A. HANDLING CYCLIC DATA STRUCTURES

The specifications in Section 2 and the resulting data flow equations in Section 4 do not capture the effect of cycles in heap. This section defines the extensions required to handle them. These have been incorporated in our prototype implementation (Section 8).

Example A.1. We illustrate the issues through Figure 15.

—*Creation of infinite pairs of node-aliases*. In Figure 15(a), alias $\langle x, y \rightarrow n \rightarrow n \rangle$ exists before the assignment. Following the specifications in Section 2 the assignment creates

```
1.  function GraphCycleClosure(AG, ⟨G_x, G'_x⟩)
2.  /* G'_x contains some prefix of G_x and AG is the set of access graph pairs */
3.  {   NewAG = ∅
4.      G_x^pre = G_x#(G_x/G'_x)
5.      for each pair ⟨G, G'⟩ ∈ AG
6.      {   G^cyc = G_x^pre#(G/G'_x)
7.          G'^cyc = G_x^pre#(G'/G'_x)
8.          if ((G^cyc ≠ E_G) && (G'^cyc ≠ E_G)) then
9.              NewAG = NewAG ∪ ⟨G^cyc, G'^cyc⟩
10.     }
11.     return AG ∪ NewAG
12. }
```

Fig. 16. Generating infinitely many alias pairs in the presence of cycles in heap.

the following new node-aliases: $\langle x \to r, y\rangle$ and $\langle y, y \to n \to n \to r\rangle$. Since y and $y \to n \to n \to r$ have the same l-value, it follows that all access paths y, $y \to n \to n \to r$, $y \to n \to n \to n \to r \to n \to r$, $\ldots y \to n \to n \to r \cdots \to n \to n \to r$ have the same l-value. The corresponding aliases must be detected to ensure that live access paths are not missed.

—*Modification of cyclic access paths in aliases*. A cycle may exist before the assignment and may get folded into a shorter cycle due to modification of some access path in $\mathit{Base}(\rho_x)$. As illustrated in Figure 15(b), the assignment modifies $\mathit{Frontier}(x \to n \to n)$ apart from $\mathit{Frontier}(x \to n \to n \to n \to n)$. As a result, shorter aliases like $\langle x \to n, y \to n \to n\rangle$, $\langle x \to n \to n, y \to n\rangle$, $\langle x \to n \to n, x \to n\rangle$, and $\langle y \to n \to n, y \to n\rangle$ and cycles involving them also are created and should be detected. □

The creation of cycles is characterized by the presence of alias $\langle \mathit{Base}(\rho_x), \rho_y \to \sigma\rangle$ before an assignment $\alpha_x = \alpha_y$. In general, after a cycle-creating assignment, node-aliases of the kind $\langle \rho_v \to \sigma, \rho_v\rangle$ are created. We compute their transitive closures to create infinite node-aliases. All pairs in the closure can be generated based on the following intuition:

> For a node-alias $\langle \rho_v \to \sigma, \rho_v\rangle$, since $\rho_v \to \sigma$ and ρ_v have the same l-value, we view them as having reached the same state of access path construction as the state reached after suffixing any number of σ's to ρ_v.

We capture this using the operation of extension in function *GraphCycleClosure* (Figure 16) which is invoked whenever node-aliases at a program point are computed.

Creation of cycles does not require change in liveness analysis because link-alias computation provides a safe approximation of live access paths. In case of availability and anticipability analyses, a proper prefix of left hand side of assignment may also get modified due to presence of cycle, and hence information generated at an assignment should not report such prefixes as available/anticipable. This is a conservative approximation.

B. NON-DISTRIBUTIVITY IN HEAP REFERENCE ANALYSIS

Alias and liveness analyses defined in this paper are not distributive whereas availability and anticipability analyses are distributive.

Example B.1. Figure 17 shows an example of non-distributivity of aliasing. Since every may-node-alias of $y \to n$ should be may-node-aliased to every may-node-alias of z, our

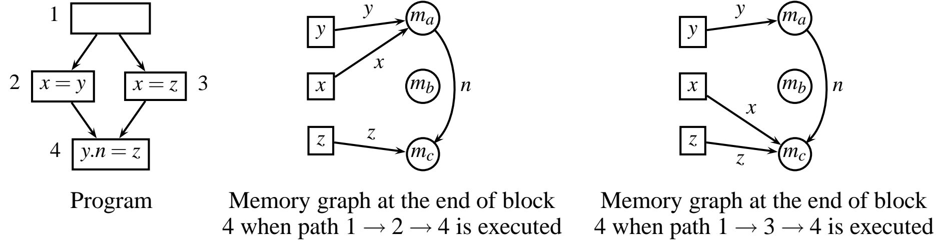


Fig. 17. Non-distributivity of alias analysis. $x \rightarrow n$ is not aliased to x along any execution path in the program.

analysis concludes that x is aliased to $x \rightarrow n$. However, it can be verified from the memory graphs that this is not possible. Let f_4 denote the flow function of block 4. Then, it is easy to see that

$$f_4(\mathbb{A}\mathsf{Out}(2) \cup \mathbb{A}\mathsf{Out}(4)) \supset f_4(\mathbb{A}\mathsf{Out}(2)) \cup f_4(\mathbb{A}\mathsf{Out}(4))$$

because of the spurious alias $\langle x, x \rightarrow n \rangle$. The spurious node-alias $\langle x, x \rightarrow n \rangle$ generates further spurious aliases due the cycle closure. □

As Example B.1 illustrates, aliasing is non-distributive because though the confluence ($\cup$) is an exact operation, the flow functions are not exact since they work on a combination of elements in the input set. The flow functions in liveness analysis are exact because they work on individual elements in the set of the paths represented by the access graph. However, it is non-distributive because of the approximation introduced by the $\uplus$ operation. $G_1 \uplus G_2$ may contain access paths which are neither in G_1 nor in G_2.

Example B.2. Figure 18 illustrates the non-distributivity of liveness analysis. Liveness graphs associated with the entry each block is shown in shaded boxes. Let f_1 denote the flow function which computes x-rooted liveness graphs at the entry of block 1. Neither $\mathsf{E}\mathbb{L}\mathsf{In}_x(2)$ nor $\mathsf{E}\mathbb{L}\mathsf{In}_x(4)$ contains the access path $x \rightarrow r \rightarrow n \rightarrow r$ but their union contains it. It is easy to see that

$$f_1(\mathsf{E}\mathbb{L}\mathsf{In}_x(2) \uplus \mathsf{E}\mathbb{L}\mathsf{In}_x(4)) \sqsubseteq_G f_1(\mathsf{E}\mathbb{L}\mathsf{In}_x(2)) \uplus f_1(\mathsf{E}\mathbb{L}\mathsf{In}_x(4))$$

□

The confluence operation used in availability and anticipability analyses use $\cap$ operation which is exact. Further the flow functions are also exact in the sense that they work on each element of the set independently rather than on a combination of elements. Hence availability and anticipability analyses are distributive.

C. ACCESS GRAPHS FOR C++ TYPE OF LANGUAGES

In order to extend the concept of access graphs to C++ type of languages, we need to take care of two major differences in C++ and Java memory model:

(1) Unlike Java, C++ has explicit pointers. Field of a structure (`struct` or `class`) can be accessed in two different ways in C++: (a) using pointer dereferencing ($*.$), e.g. $(*x).lptr$[11] or (b) using simple dereferencing (.) , e.g. $y.rptr$. We need to differentiate between the two ways.

[11]This is equivalently written as $x\text{->}lptr$.

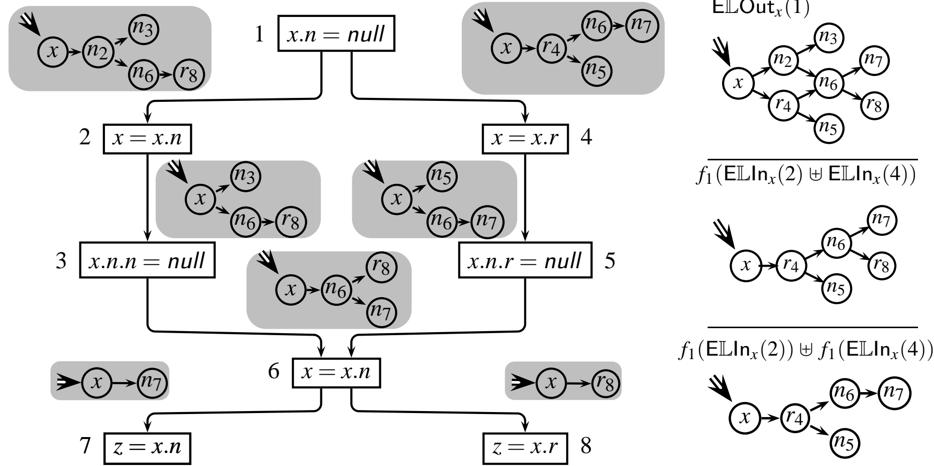


Fig. 18. Non-distributivity of liveness analysis. Access path $x \rightarrow r \rightarrow n \rightarrow r$ is a spurious access path which does not get killed by the assignment in block 1.

(2) In C++, root variables are on stack, same as that in Java. However, C++, through the use of addressof (&) operator, allows a pointer/reference to point to root variables on stack. Java does not allow a reference to point to stack. Root nodes in access graphs, following Java model, do not have an incoming edge by definition. Therefore, it is not possible to use access graphs directly to represent memory links in C++. We need to create a view of C++ memory model such that this view follows Java model.

We create access graphs for C++ memory model as follows:

(1) We treat dereference of a pointer as a field reference, i.e., $*$ is considered as a field. For example, an access expression $(*x).lptr$ is viewed as $x.*.lptr$, and corresponding access path is $x \rightarrow * \rightarrow lptr$. The access path for $x.lptr$ is $x \rightarrow lptr$.

(2) Even though a pointer can point to a variable x, it is not possible to point to $\&x$. In Java we partitioned memory as stack and heap, and had root variables of access graphs correspond to stack variables. In C++, we partition the memory as *address of variables* and rest of the memory (stack and heap together). We make the roots of access graphs correspond to addresses of variables. A root variable y is represented as $*\&y$. Thus, $\rightarrow \&y \rightarrow *_1 \rightarrow l_2$ represents access paths $\&y$ and $\&y \rightarrow *$ and $\&y \rightarrow * \rightarrow l$, which correspond to access expressions $\&y$, y and $y.l$ respectively.

Handling pointer arithmetic and type casting in C++ is orthogonal to above discussion, and requires techniques similar to [Yong et al. 1999; Cheng and Hwu 2000] to be used.