Symbolic Pointer Analysis for Detecting Memory Leaks

Bernhard Scholz
Institute for Computer Languages
Vienna University of Technology, Austria
scholz@complang.tuwien.ac.at

Johann Blieberger
Department of Computer-Aided Automation
Vienna University of Technology, Austria
blieb@auto.tuwien.ac.at

Thomas Fahringer
Institute for Softwaretechnology and Parallel Systems
University of Vienna, Austria
tf@par.univie.ac.at

ABSTRACT

It is well accepted that pointers are a common source of memory anomalies such as loosing references to dynamic records without deallocating them (also known as memory leaks). This paper presents a novel pointer analysis framework that detects memory leaks by statically analyzing the behavior of programs.

Our approach is based on symbolic evaluation of programs. Symbolic evaluation is an advanced static symbolic analysis that is centered around symbolic variable values, assumptions about and constraints between variable values, and control flow information (path conditions). As part of symbolic evaluation we introduce a new symbolic heap algebra for modeling heap operations. Predicates – defined over the program's input – are derived which allow to detect memory leaks. Our approach goes beyond previous work in the field of statically detecting memory leaks by considering also path conditions which increases the accuracy of our results, symbolically modeling heap data structures and heap operations. Examples are used to illustrate the effectiveness of our approach.

1. INTRODUCTION

Many important programming languages support explicit dynamic memory management by pointer manipulations. Supporting pointers considerably increases the expressiveness of such languages. However, it is widely agreed that programming with pointers is a tedious task and can cause a large number of memory errors. Such errors are particularly hard to detect as it is usually very difficult to reproduce them and to identify the source of the error in the program [1]. Clearly, it is of paramount importance to detect memory errors at compile time as compared to exhaustively testing programs at runtime. A static analysis tool that targets the detection of memory anomalies therefore greatly alleviates program development and testing.

Memory leaks cause severe problems in modern software development. A memory leak occurs if references to dynamic records are lost. A program that leaks memory consumes more memory than expected. E.g. for server applications that run for a long period of time memory leaks can be a potential threat. Applications may slow down (caused by memory swapping) and eventually may even crash due to lack of available memory. An important observation is that memory leaks cannot be eliminated by any subsequent operation in the program.

We present a novel static analysis for pointer programs based on symbolic evaluation that detects and localizes these kinds of memory leaks. Symbolic evaluation is a powerful data and control flow analysis that determines runtime properties of a given program without executing it. Symbolic expressions and recurrences are used to maintain the relationship between input data and resulting values. For symbolic evaluation a new data representation (*program context*) is employed to describe the semantics of program statements. Program contexts comprise symbolic variable values, assumptions about and constraints between variable values, and path conditions (the conditions under which the control flow reaches a program point). A symbolic heap algebra for describing the heap operations of a program – based on symbolic evaluation – is introduced. Recurrences over the heap algebra accurately express the dynamic behavior of heap operations inside of loops.

In order to detect memory leaks we introduce the notion of a *pointer graph*. We show that a memory leak occurs if and only if the graph is disconnected. Based on this observation we can derive a predicate that reports a memory leak.

Our method goes beyond previous work in the following points:

- Our analysis is more accurate due to highly sophisticated symbolic evaluation that models both data and control flow.
- We *statically* detect and localize memory leaks.
- We symbolically describe heap operation based on symbolic evaluation and a novel heap algebra.

Although we have examined our framework for a subset of Ada-like programs the underlying techniques are equally applicable to any similar imperative programming language.

The remainder of the article is organized as follows. In Section 2 we introduce the notions of symbolic evaluation. Section 3 presents a novel heap algebra for modeling a heap and shows how the heap operations are symbolically evaluated. In the next section we describe how to statically detect and localize memory leaks. In Section 5 we survey related work.

Finally, in Section 6 we give our conclusion and describe future work.

2. SYMBOLIC EVALUATION

A symbolic evaluation function [12; 13] $\mathcal{F}$ is a constructive description of the semantics of a program. Moreover, symbolic evaluation is not merely an arbitrary alternative semantic description of a program. As in the relationship between arithmetic and algebra the specific (arithmetic) computations dictated by the program operators are generalized and "delayed" using the appropriate formulas. The dynamic behavior is represented by *symbolic* equations. Symbolic evaluation satisfies a commutativity property.

$$\begin{tikzcd}(S_{\mathrm{conc}}[\![\mathfrak{p}]\!], \mathrm{i}) \arrow[r, "\text{(Symb. Evaluation)}"] \arrow[d, "\text{Set param. to i}"'] & (\mathcal{F}[\![\mathfrak{p}]\!], \mathrm{i}) \arrow[d, "\text{Substitute i into result}"] \\ S_{\mathrm{conc}}[\![\mathfrak{p}]\!]\mathrm{i} \arrow[r, "\text{(Conv. Execution)}"] & \mathcal{F}[\![\mathfrak{p}]\!]\mathrm{i}\end{tikzcd}$$

If a program $\mathfrak{p}$ is conventionally executed with the standard semantics $S_{\mathrm{conc}}[\![\mathfrak{p}]\!]$ over a given input i the result of the symbolically evaluated program $\mathcal{F}[\![\mathfrak{p}]\!]$ instantiated by i is the same. Clearly, symbolic evaluation can be seen as a compiler that translates a program into a different language. Here, we use as a target language *symbolic expressions* and *recurrences* to model the semantics of a program.

Symbolic evaluation of a program is done based on a novel representation called *program context*. Every statement is associated with a program context c that describes the variable values, assumptions regarding and constraints between variable values and a path condition. The path condition specifies whether the control flow reaches a given program point. In this paper we use an improved context representation [13]. Formally, a context c in the set of contexts C associated with a statement ℓ is defined by a triple $[s, t, p]$. A state s is a finite set of pairs $(V \times E)^n$ where V is the set of variables, n the number of variables in the program, and E is the set of symbolic expressions. A state condition $t \in E$ and a path condition $p \in E$ are symbolic expressions.

- The state s is described by a set of variable/symbolic expression pairs $\{v_1 = e_1, \ldots, v_n = e_n\}$ where v_i is a program variable and e_i a symbolic expression describing the symbolic value of v_i for $1 \leq i \leq n$. For all program variables v_i there exists exactly one pair $v_i = e_i$ in state s.
- The state condition contains constraints on variable values such as those implied by loops, variable declarations, and user assertions.
- Path condition is a predicate, which is true if and only if statement ℓ is reached.

Note that all components of a context — including state information — are expressed as symbolic expressions and recurrences. All program contexts are automatically generated at compile-time (see algorithm in [13]).

E.g. an unconditional sequence of statements ℓ_j $(1 \leq j \leq r)$ is symbolically evaluated by $[s_0, t_0, p_0]$ ℓ_1 $[s_1, t_1, p_1]$ $\ldots$ ℓ_r $[s_r, t_r, p_r]$. The initial context $[s_0, t_0, p_0]$ represents the context before ℓ_1 is executed and $[s_r, t_r, p_r]$ the context after ℓ_r. If ℓ_i in the sequence $\ldots$ $[s_i, t_i, p_i]$ ℓ_i $[s_{i+1}, t_{i+1}, p_{i+1}]$ $\ldots$ does not contain any side effects (implying a change of a variable value) then $s_i = s_{i+1}$.

Furthermore, a context $c = [s, t, p]$ is a logical assertion $\overline{c} = s \wedge t \wedge p$ where $\overline{c}$ is a predicate over the set of program variables and the program input which are free variables. If for all input values $\overline{c_{i-1}}$ holds before executing the statement ℓ_i then $\overline{c_i}$ is the strongest post condition [6] and the program variables are in a state satisfying $\overline{c_i}$ after executing ℓ_i.

Note that the algorithm introduced by the authors handle control flow (IFs, GOTOs, etc.). For further technical details we refer the reader to [12; 13]. In the following we apply this approach to evaluate dynamic data structures.

3. DYNAMIC DATA STRUCTURES

Dynamically allocated records are data items stored on the heap. Heap operations allocate and deallocate records and allow reading and writing values from/to record fields. In our symbolic evaluation framework the heap is described by a heap algebra, which is based on symbolic expressions and recurrences. The symbolic heap of a program point symbolically represents the current state of the heap and comprises all modifying operations performed on the heap. Moreover, in the semantic domain of symbolic evaluation we use *symbolic pointers* to refer to dynamic records. Therefore, we tag all dynamically created records with a unique symbolic number. In the symbolic domain `nil` is represented as $\perp$.

In order to ease the readability of this paper we impose the following restrictions:

- The language is strongly typed.
- We only allow (1) basic types, (2) records (no variants), and (3) pointer to records. We assume that all records have a *type declaration* <`t`>. The type declaration defines the *fields* of <`t`>. A field is uniquely identified by a *qualifier* <`q`>.
- Pointers are either allowed to point to dynamically allocated records or to `nil`.
- A pointer variable is assigned a value from other pointer variables, from a `new` operation or it is set to `nil`.
- There is no memory overlap — every field is stored in a unique storage location*.

The third and forth restriction are important to avoid alias effects of program variables [2] which goes beyond the scope of this paper. In the following we introduce heap algebra $\mathbb{H}$, which is a formal vehicle to symbolically describe the heap. Then, we give the notions of symbolic evaluation for heap operations. Finally, we conclude this section with examples.

3.1 Heap Algebra

In the semantic domain of symbolic evaluation the heap is a *symbolic chain* consisting of three functions **new**, **free** and **put**. Semantically, a function **new**(r, <`t`>) allocates a new element with symbolic pointer r and type <`t`>. A function **free**(r) denotes a free operation where dynamic record r is freed. The **put**(r, <`q`>, e) assigns a new symbolic value of e to dynamic field <`q`> of dynamic record r. Pointer r of pointer variable v is stored on the symbolic heap by **put**($\mathrm{v}, \perp, r$). Note that this information is kept in the state

*E.g. two dynamic records share the same storage location.

as well. Nevertheless, the additional heap function facilitates the detection of memory leaks. The following table informally describes the heap functions by relating them to the constructs of the language.

Heap operation	Language Example
$\mathbf{new}(r, \texttt{<t>})$	`new t;`
$\mathbf{free}(r)$	`delete r;`
$\mathbf{put}(r, \texttt{<q>}, e)$	`r.q := e;`
$\mathbf{put}(\mathrm{v}, \perp, r)$	`v := e;`

More formally, the heap is modeled as a heap algebra $\mathbb{H}$. The algebra is inductively defined as follows.

1. $\perp$ is in $\mathbb{H}$.
2. If $h \in \mathbb{H}$, $r \in E$ is a symbolic pointer, and $\texttt{<t>}$ is a type, then $h \oplus \mathbf{new}(r, \texttt{<t>}) \in \mathbb{H}$.
3. If $h \in \mathbb{H}$ and $r \in E$ is a symbolic pointer then $h \oplus \mathbf{free}(r) \in \mathbb{H}$.
4. If $h \in \mathbb{H}$, r is a symbolic pointer, $\texttt{<q>}$ is a qualifier, and e is a symbolic expression then $h \oplus \mathbf{put}(r, \texttt{<q>}, e) \in \mathbb{H}$.
5. If $h \in \mathbb{H}$, v is pointer variable, and r is a symbolic pointer, then $h \oplus \mathbf{put}(\mathrm{v}, \perp, r) \in \mathbb{H}$.

An element h in $\mathbb{H}$ can be written as $\oplus$-chain $\perp \bigoplus_{l=1}^{m} \mu_l$ where μ_l is a **new**, **free**, or **put** function. In the further text μ_l is also referred to as *heap functions*. The length of chain $|h|$ is the number of functions μ_l in chain h.
We introduce heap function $\mathbf{get}(h, r, \texttt{<q>})$ to access field $\texttt{<q>}$ from dynamic record r where h is a symbolically described heap and element of $\mathbb{H}$. The operator **get** seeks for the right-most function $\mathbf{put}(r, \texttt{<q>}, e)$ in h whose symbolic pointer and qualifier matches the arguments of operator **get**. If such a function $\mathbf{put}(r, \texttt{<q>}, e)$ exists the result of operator **get** is e; otherwise $\perp$.

$$\mathbf{get}\left(\perp \bigoplus_{l=1}^{m} \mu_l, r, \texttt{<q>}\right) = \begin{cases} e_l, & \text{if } \exists l = \max\{l \mid 1 \leq l \leq m \wedge \\ & \quad \mu_l = \mathbf{put}(r, \texttt{<q>}, e_l)\} \\ \perp, & \text{otherwise} \end{cases} \tag{1}$$

In general determining whether $\mu_l = \mathbf{put}(r, \texttt{<q>}, e_l)$ matches the arguments of operator **get** in a symbolic expression (or not) is undecidable. In practice a wide class of symbolic relations can be solved by our techniques for comparing symbolic expressions [10]. If our symbolic evaluation framework cannot prove that the result of **get** is $\texttt{<q>}$ or $\perp$ then **get** is not resolvable and remains unchanged in the symbolic expression. See Section 4 with respect to memory leaks.
For loops, we extend algebra $\mathbb{H}$ in order to model recurrences [12; 13]. A recurrence system over $\mathbb{H}$ consists of a boundary condition $h(0) = h$, $h \in \mathbb{H}$ and a recurrence relation

$$h(k+1) = h(k) \bigoplus_{l=1}^{m} \mu_l(k), \quad k \geq 0 \tag{2}$$

where $\mu_l(k)$ can be a function $\mathbf{new}\,(r_l(k), \texttt{<t>}_l)$, $\mathbf{free}\,(r_l(k))$ or $\mathbf{put}(r_l(k)\ , \texttt{<q>}_l, e_l(k))$. The boundary condition represents the symbolic value before entering the loop. For each iteration of a loop a symbolic value is computed by the recurrence relation. The index k determines the loop iteration. Note that the parameters of the heap functions are functions over index k. Clearly, every instance of the recurrence is an element of $\mathbb{H}$. Moreover, for chains the operator **get** is not sophisticated enough to cope with general read accesses of recursively defined heaps. For this purpose the operator **get** needs to be extended for recurrences such that heap operations inside of loops can be accessed, e. g. $\mathbf{get}(h(z), r, \texttt{<q>})$. The symbolic expression z is the number of loop iterations determined by the loop exit condition and r is the symbolic pointer of the dynamic record. Furthermore, the recurrence index k has an upper and lower bound of $0 \leq k \leq z$. To determine a possible function **put** where the accessed element is written, a *potential index set* $X_l(r, \texttt{<q>})$ of the function μ_l is computed.

$$\forall 1 \leq l \leq m : X_l(r, \texttt{<q>}) = \{k \mid \mu_l(k) = \mathbf{put}(r, \texttt{<q>}, e_l(k)) \wedge \\ 0 \leq k \leq z\} \tag{3}$$

$X_l(r, \texttt{<q>})$ represents all possible indices of k such that function $\mu_l(k)$ potentially describes the value of the read access. If the cardinality of X_l is zero then the corresponding heap function is irrelevant to find the value of the requested field of a dynamic record. If an index set $X_l(r, \texttt{<q>})$ has more than one element the field $\texttt{<q>}$ of record r is written more than once in different loop iterations. We are only interested in the right-most function of $h(z)$. Consequently, we choose the element with the largest index. The *supremum* $x_l(r, \texttt{<q>})$ of an index set $X_l(r, \texttt{<q>})$ is the largest index such that

$$\forall 1 \leq l \leq m : x_l(r, \texttt{<q>}) = \max X_l(r, \texttt{<q>})$$

Finally, we define the operator **get** for recurrences as follows,

$$\mathbf{get}\ (h(z), r, \texttt{<q>}) = \begin{cases} e_l(k), & \text{if } \exists 1 \leq l \leq m : x_l(r, \texttt{<q>}) = \\ & \quad \max_{1 \leq k \leq m} x_k(r, \texttt{<q>}) \\ \mathbf{get}(h(0), r, \texttt{<q>}), & \text{otherwise} \end{cases} \tag{4}$$

where $e_l(k)$ is the value of $\mu_l(k) = \mathbf{put}(r_l(k), \texttt{<q>}, e_l(k))$. The maximum of the supremum indices $x_l(r, \texttt{<q>})$ determines the symbolic value $e_l(x_l(r, \texttt{<q>}))$. If no supremum index exists then **get** returns the access to the value before the loop.

3.2 Symbolic Evaluation

In the following we relate the syntax of heap operations to the semantics for allocating dynamic records, heap function **put** for setting fields and heap function **free** for freeing dynamic records. For this purpose we use denotational semantics [24]. Figure 1 lists the symbolic evaluation rules of the heap operations. Nonterminal `<stmt>` denotes a heap operation, nonterminal `<ptr>` a pointer value, and nonterminal `<value>` a value of a basic type.
For symbolically evaluating a program symbolic counter $\mathfrak{c}$ is employed and it is incremented each time when a new dynamic record is allocated. The symbolic value of the counter is used to generate a label for the newly allocated record. The symbolic pointer of the new record is the value of $\mathfrak{c}$ after incrementing it. The modifications on the heap are symbolically described by a heap element $\mathfrak{h}$ of $\mathbb{H}$. For both, $\mathfrak{c}$ and $\mathfrak{h}$, we introduce two new pseudo variables in state s. In the first program context heap $\mathfrak{h}$ is set to $\perp$ and $\mathfrak{c}$ to 0.
In Figure 1 the denotational semantic rules (D1)-(D10) uses the functions eval and δ. The function $\mathrm{eval}(e, [s, t, p])$ determines the symbolic expression under the constraints of pro-

$$\mathcal{F} : \texttt{<stmt>} \to C \to C$$

(D1) $\mathcal{F}[\![\texttt{<ptr>}_1\texttt{.<q> := <ptr>}_2\texttt{;}]\!] =$
$\lambda[s,t,p] \in C.[\delta\,(s, \mathfrak{h} = \text{eval}(\mathfrak{h},[s,t,p]) \oplus \mathbf{put}(\mathcal{F}[\![\texttt{<ptr>}_1]\!]([s,t,p]), \texttt{<q>}, \mathcal{F}[\![\texttt{<ptr>}_2]\!]([s,t,p])))\,, t, p]$

(D2) $\mathcal{F}[\![\texttt{<ptr>.<q> := <value>;}]\!] =$
$\lambda[s,t,p] \in C.[\delta\,(s, \mathfrak{h} = \text{eval}(\mathfrak{h},[s,t,p]) \oplus \mathbf{put}(\mathcal{F}[\![\texttt{<ptr>}]\!]([s,t,p]), \texttt{<q>}, \text{eval}(\texttt{<value>},[s,t,p])))\,, t, p]$

(D3) $\mathcal{F}[\![\texttt{<ptr>.<q> := new <t>;}]\!] =$
$\lambda[s,t,p] \in C.[\lambda r \in E.\delta\,(s, \mathfrak{c} = r, \mathfrak{h} = \text{eval}(\mathfrak{h},[s,t,p]) \oplus \mathbf{new}(r, \texttt{<t>}) \oplus$
$\mathbf{put}(\mathcal{F}[\![\texttt{<ptr>}]\!]([s,t,p]), \texttt{<q>}, r))\,(\text{eval}(\mathfrak{c},[s,t,p])+1), t, p]$

(D4) $\mathcal{F}[\![\texttt{<var> := <ptr>;}]\!] =$
$\lambda[s,t,p] \in C.[\lambda r \in E.\delta\,(s, \texttt{<var>} = r, \mathfrak{h} = \text{eval}(\mathfrak{h},[s,t,p]) \oplus \mathbf{put}(\text{eval}(\texttt{<var>},[s,t,p]), \perp, r))$
$(\mathcal{F}[\![\texttt{<ptr>}]\!]([s,t,p])), t, p]$

(D5) $\mathcal{F}[\![\texttt{<var> := new <t>;}]\!] =$
$\lambda[s,t,p] \in C.[\lambda r \in E.\delta\,(s, \texttt{<var>} = r, \mathfrak{c} = r, \mathfrak{h} = \text{eval}(\mathfrak{h},[s,t,p]) \oplus \mathbf{new}(r, \texttt{<t>}) \oplus \mathbf{put}(\texttt{<var>}, \perp, r))$
$(\text{eval}(\mathfrak{c},[s,t,p])+1), t, p]$

(D6) $\mathcal{F}[\![\texttt{free <ptr>;}]\!] =$
$\lambda[s,t,p] \in C.[\delta\,(s, \mathfrak{h} = \text{eval}(\mathfrak{h},[s,t,p]) \oplus \mathbf{free}(\text{eval}(\texttt{<ptr>},[s,t,p])))\,, t, p]$

$$\mathcal{F} : \texttt{<ptr>} \to C \to E$$

(D7) $\mathcal{F}[\![\texttt{nil}]\!] = \lambda[s,t,p] \in C.\perp$

(D8) $\mathcal{F}[\![\texttt{<var>}]\!] = \lambda[s,t,p] \in C.\,\text{eval}(\texttt{<var>},[s,t,p])$

(D9) $\mathcal{F}[\![\texttt{<ptr>.<q>}]\!] = \lambda[s,t,p] \in C.\mathbf{get}(\text{eval}(\mathfrak{h},[s,t,p]), \mathcal{F}[\![\texttt{<ptr>}]\!]([s,t,p]), \texttt{<q>})$

(D10) $\text{eval}(\texttt{<ptr>.<q>},[s,t,p]) = \mathbf{get}(\text{eval}(\mathfrak{h},[s,t,p]), \mathcal{F}[\![\texttt{<ptr>}]\!]([s,t,p]), \texttt{<q>})$

Figure 1: Symbolic evaluation of heap operations

gram context $[s,t,p]$ (see [12; 13]). The function $\delta(s; v_1 = e_1, \ldots, v_k = e_l)$ compresses the state description by listing only those variables whose value changes after evaluating the associated statement whereby state s is the state before evaluating the statement and variables v_i ($1 \leq i \leq l$) are changed by the statement and get new symbolic values e_i.
We distinguish between two different pointer assignments (D1), and (D4). (D1) is a pointer assignment for a field of a dynamic record. An assignment for a program variable (D4) produces a new entry $\mathbf{put}(\texttt{<var>}, \perp, r)$ in the chain. This redundant entry facilitates subsequent analyses. Rule (D2) assigns a non-pointer field of a dynamic record a new symbolic value. (D3) allocates a new dynamic record and assigns it to a pointer field. (D5) allocates a new dynamic record and assigns it to a program variable. The semantic of the free operation is given by (D6). There are two different types of field read accesses: (1) A pointer field is read (D9), (2) a basic type field is accessed (D10).

3.3 Examples

The code fragment in Figure 2(a) illustrates the symbolic evaluation of a sequence of program statements. Three pointer variables `p1`, `p2`, `p3` of type `bna` are modified in the sequence. The type declaration of `bna` is given in Figure 2(b). For all statements ℓ_i in Figure 2(a) there exists a program context $[s_{i-1}, t_{i-1}, p_{i-1}]$ before statement ℓ_i and a program context $[s_i, t_i, p_i]$ after statement ℓ_i. In context $[s_0, t_0, p_0]$ heap $\mathfrak{h}$ is set to symbolic expression β and symbolic counter $\mathfrak{c}$ is assigned symbolic value α which denote arbitrary heap configurations depending on the preceding statements of the code fragment. The values of variables `p1`, `p2`, and `p3` are set to $\perp$, which means that they do not have a valid value so far. In the first statement ℓ_1 three dynamic records are allocated and assigned to the variables. The allocation is symbolically described by incrementing symbolic counter $\mathfrak{c}$ and by adding three **new** and three **put** functions to the heap $\mathfrak{h}$. Note that the **put** functions are necessary due to the variable assignment (see Figure 1). In the next two statements ℓ_2 and ℓ_3 field assignments are symbolically evaluated. For each statement a new **put** function is added to $\mathfrak{h}$. In ℓ_4 a list of qualifiers is used. Note that Figure 2(c) depicts the heap after statement ℓ_5. The last statement ℓ_7 frees the first dynamic record in the tree which results in a memory leak. The memory leak occurs as it is no longer possible to access the dynamic records.
The second example in Figure 3(b) shows a code fragment, which is supposed to build a simply linked list. It is not able to build up the list due to the fact that the reference to the first element is lost. The type declaration of variable `p` is given in Figure 3(a). Before evaluating the first statement the heap variables $\mathfrak{h}$ and $\mathfrak{c}$ are initialized to $\perp$ and 0. The variables `i` and `p` do not have a value so far and variable `n` is set to γ, which denotes an arbitrary symbolic value.
The variables `i`, `p`, $\mathfrak{c}$, and $\mathfrak{h}$ change their values in each iteration of the loop. Therefore, we have four recurrences induced by variable `i`, pointer variable `p`, symbolic counter $\mathfrak{c}$ and heap $\mathfrak{h}$. Note that the result of operator **get** in statement ℓ_5 is given by $\mathfrak{c}+1$ since the right-most function **put** can be directly found in the recurrence relation of $\mathfrak{h}(k)$. After the loop terminates the values of variables `i`, `p`, $\mathfrak{c}$ and $\mathfrak{h}$ are determined by their recurrences whereby the recurrence index z is derived from the loop exit condition. Closed forms can be found for `i`, `p`, $\mathfrak{c}$. The heap $\mathfrak{h}$ in ℓ_7 symbolically describes all elements of the simply linked list. Figure 3(c) depicts the graphical representation of the recursively described linked list for $\gamma \geq 2$.

4. DETECTION OF MEMORY LEAKS

We now show how to detect memory leaks by providing a logical predicate defined over program input. Clearly, symbolic analysis can only track down a subclass of all possible memory leaks, which is a limitation of our approach as well. In order to statically detect memory leaks we introduce a

```
p1,p2,p3:  bna;
...
```

$[s_0 = \{p1 = \perp, p2 = \perp, p3 = \perp, \mathfrak{c} = \alpha, \mathfrak{h}_0 = \beta\}, t_0, p_0]$

ℓ_1 : `p1 := new bin; p2:= new bin; p3:= new bin;`

$[s_1 = \delta(s_0; p1 = \alpha+1, p2 = \alpha+2, p3 = \alpha+3, \mathfrak{c} = \alpha+3,$
$\mathfrak{h}_1 = \mathfrak{h}_0 \oplus \mathbf{new}(\alpha+1, \mathtt{bin}) \oplus \mathbf{put}(\mathrm{p1}, \perp, \alpha+1) \oplus \mathbf{new}(\alpha+2, \mathtt{bin}) \oplus$
$\mathbf{put}(\mathrm{p2}, \perp, \alpha+2) \oplus \mathbf{new}(\alpha+3, \mathtt{bin}) \oplus \mathbf{put}(\mathrm{p3}, \perp, \alpha+3)),$
$t_1 = t_0, p_1 = p_0]$

ℓ_2 : `p1.left := p3;`

$[s_2 = \delta(s_1; \mathfrak{h}_2 = \mathfrak{h}_1 \oplus \mathbf{put}(\alpha+1, \mathtt{left}, \alpha+3)), t_2 = t_1, p_2 = p_1]$

ℓ_3 : `p1.right := p2;`

$[s_3 = \delta(s_2; \mathfrak{h}_3 = \mathfrak{h}_2 \oplus \mathbf{put}(\alpha+1, \mathtt{right}, \alpha+2)), t_3 = t_2, p_3 = p_2]$

ℓ_4 : `p1.right.left := p3;`

$[s_4 = \delta(s_3; \mathfrak{h}_4 = \mathfrak{h}_3 \oplus \mathbf{put}(\mathbf{get}(\mathfrak{h}_3, \alpha+1, \mathtt{right}), \mathtt{left}, \alpha+3)),$
$t_4 = t_3, p_4 = p_3] =$
$[s'_4 = \delta\left(s_4; \mathfrak{h}'_4 = \mathfrak{h}_3 \oplus \mathbf{put}(\alpha+2, \mathtt{left}, \alpha+3)\right), t'_4 = t_4, p'_4 = p_4]$

ℓ_5 : `p3.right := p2;`

$[s_5 = \delta\left(s'_4; \mathfrak{h}_5 = \mathfrak{h}'_4 \oplus \mathbf{put}(\alpha+3, \mathtt{right}, \alpha+2)\right), t_5 = t'_4, p_5 = p'_4]$

ℓ_6 : `p2 := nil; p3:= nil;`

$[s_6 = \delta(s_5; p1 = \perp, p2 = \perp, \mathfrak{h}_6 = \mathfrak{h}_5 \oplus \mathbf{put}(\mathrm{p2}, \perp, \perp) \oplus \mathbf{put}(\mathrm{p3}, \perp, \perp)),$
$t_6 = t_5, p_6 = p_5]$

ℓ_7 : `free(p1);`

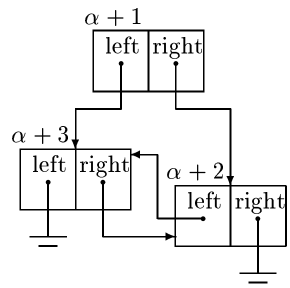
$[s_7 = \delta(s_6; \mathfrak{h}_7 = \mathfrak{h}_6 \oplus \mathbf{free}(\alpha+1)), t_7 = t_6, p_7 = p_6]$

```
...
```

(a)

```
type bin;
type bna is access bin;
type bin is record
   left:  bna;
   right:  bna;
end record;
```

(b)



(c)

Figure 2: Code segment and its graphical representation

pointer graph that describes the connectivity of the dynamic records on the heap. A heap operation destroys the property of being connected if at least one dynamic record can no longer be referenced in the program.

Formally, we introduce a *pointer graph* $\mathbb{G}(c_i) = (\mathfrak{N}, \mathfrak{E}, \mathfrak{r})$ of a program context c_i. The graph consists of a set of *nodes* $\mathfrak{N}$, a set $\mathfrak{E} \subseteq \mathfrak{N} \times \mathfrak{N}$ of *edges*, and a *root node* denoted as $\mathfrak{r} \in \mathfrak{N}$. A pair $(x, y) \in \mathfrak{E}$ is called edge from x to y. A *path* from x to y, $\{x, y\} \subset \mathfrak{N}$ is a sequence $x_0, x_1, \ldots, x_l$ of nodes such that $x_0 = x$, $x_l = y$ and $(x_i, x_{i+1}) \in \mathfrak{E}$ for $0 \leq i < l$; l is the length of the path. A dynamic record is represented by a node in $\mathfrak{N}$. The root node $\mathfrak{r}$ is an artificial node and is not related to any dynamic records on the heap. If there exists at least one field q of dynamic record x that points to dynamic record y then edge (x, y) is in $\mathbb{G}$. Pointer variables are handled as artifical dynamic records without fields. If there exists a pointer variable v in state s_i that points to dynamic record x then $(\mathfrak{r}, v)$ and (v,x) are in $\mathfrak{E}$. If $(x, y) \in \mathfrak{E}$ we say that node x is a *predecessor* of node y. The set of predecessors $\mathrm{pred}(y) = \{x \mid (x, y) \in \mathfrak{E}\}$ are all predecessors of node y.

One can easily prove that a memory leak occurs if and only if the graph is not connected. In other words for every dynamic record x there must exist a pointer path from the root node $\mathfrak{r}$ to x.

For creating pointer graph $\mathbb{G}$ we analyze the heap functions of $\mathfrak{h}$ from the left to the right of chain $\mathfrak{h}$. Initially, the pointer graph contains only root node $\mathfrak{r}$. For each heap function of $\mathfrak{h}$ we update pointer graph $\mathbb{G}$ as follows:

1. Heap function $\mathbf{new}(r, \texttt{<t>})$ adds a new node r to $\mathbb{G}$.

2. Heap function $\mathbf{put}(x, \texttt{<q>}, z)$, $x \neq \perp$ updates pointer graph $\mathbb{G}$. If h is the sub-chain before $\mathbf{put}(x, \texttt{<q>}, z)$, and $y = \mathbf{get}\,(h, x, \texttt{<q>})$ the old reference $(y \neq \perp)$ then edge (x, y) is removed from $\mathbb{G}$ and a new edge (x, z), $z \neq \perp$, is added to $\mathbb{G}$. Figure 4(a) shows how the pointer graph and heap are updated before and after setting field <q> of dynamic record x under the assumption that y and z are not $\perp$ and no memory leak occurs. A pointer path and an edge in $\mathbb{G}$ are shown as a dotted and solid arrow, respectively.

3. Heap function $\mathbf{put}(v, \perp, z)$ removes edge (v, y) (if variable v refers to y and y is unequal to $\perp$) and adds edge (v, z), $z \neq \perp$.

4. Heap function $\mathbf{free}(x)$ removes node x and all edges (x, y) and (y, x) from pointer graph $\mathbb{G}$. Figure 4(b) displays the free operation under the assumption that no memory leak occurs. The dynamic record x can have several pointer fields, which point to dynamic record $y_1, \ldots, y_n$.

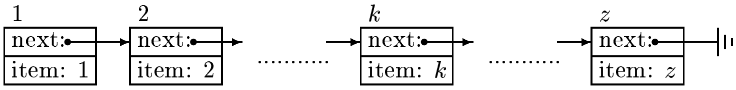
In the following we derive a symbolic function that determines the set of predecessors of dynamic record y. For constructing the set $\mathrm{pred}(y)$ we need an intermediate set S_y which consists of pairs $(x, \texttt{<q>})$ – field <q> of dynamic record x points to y – where x is a reference and <q> a qualifier. The intermediate set is created by applying the operator $\odot$ to an intermediate set. Initially, the intermediate set is empty. The intermediate set is incrementally updated by traversing the heap $h \in \mathbb{H}$ from left to right.

$$S_y \odot \perp = S_y \tag{5}$$

$$S_y \odot \left(\perp \oplus \mathbf{new}(x, \texttt{<t>}) \bigoplus_{l=1}^{m} \mu_l\right) = S_y \odot \perp \bigoplus_{l=1}^{m} \mu_l \tag{6}$$

```
type el;
type ela is access el;
 type el is record
 item:  natural;
 next:  ela;
end record;
```

(a)



(c)

```
p:  ela; i,n:natural;
...
```

$[s_0 = \{n = \gamma, p = \perp, i = \perp, \mathfrak{c} = 0, \mathfrak{h}_0 = \perp\}, t_0 = true, p_0 = true]$

ℓ_1 : `p := new el;`

$[s_1 = \delta(s_0; p = 1, \mathfrak{c} = 1, \mathfrak{h}_1 = \mathfrak{h}_0 \oplus \mathbf{new}(1, \mathtt{el}) \oplus \mathbf{put}(p, \perp, 1)), t_1 = t_0, p_1 = p_0]$

ℓ_2 : `p.item := 1;`

$[s_2 = \delta(s_1; \mathfrak{h}_2 = \mathfrak{h}_1 \oplus \mathbf{put}(1, \mathtt{item}, 1)), t_2 = t_1, p_2 = p_1] =$

ℓ_3 : `for i in 2..n loop`

$[s_3 = \delta(s_2; i = i(k), p = p(k), \mathfrak{c} = \mathfrak{c}(k), \mathfrak{h}_3 = \mathfrak{h}(k)), t_3 = (i(0) = 2 \wedge p(0) = 1 \wedge \mathfrak{c}(0) = 1 \wedge \mathfrak{h}(0) = \mathfrak{h}_2),$
$p_3 = i(k) <= \gamma]$

ℓ_4 : `    p.next := new el;`

$[s_4 = \delta(s_3; \mathfrak{c} = \mathfrak{c}(k) + 1, \mathfrak{h}_4 = \mathfrak{h}(k) \oplus \mathbf{new}(\mathfrak{c}(k) + 1, \mathtt{el}) \oplus \mathbf{put}(p(k), \mathtt{next}, \mathfrak{c}(k) + 1)), t_4 = t_3, p_4 = p_3]$

ℓ_5 : `    p := p.next;`

$[s_5 = \delta(s_4; p = \mathbf{get}(\mathfrak{h}_5, p(k), \mathtt{next})), t_5 = t_4, p_5 = p_4] =$
$[s'_5 = \delta(s_5; p = \mathfrak{c}(k) + 1), t'_5 = t_5, p'_5 = p_5]$

ℓ_6 : `    p.item := i;`

$[s_6 = \delta\left(s'_5; \mathfrak{h}_6 = \mathfrak{h}_5 \oplus \mathbf{put}(\mathfrak{c}(k) + 1, \mathtt{item}, i(k))\right), t_6 = t'_5, p_6 = p'_5]$

ℓ_7 : `end loop;`

$[s_7 = \delta(s_6; i = z + 2, p = z + 1, \mathfrak{c} = z + 1, \mathfrak{h}_7 = \mathfrak{h}(z)), t_7 = (z = \max(0, \gamma - 2) \wedge$
$\mathfrak{h}(0) = \mathfrak{h}_2 \wedge \mathfrak{h}(k + 1) = \mathfrak{h}(k) \oplus \mathbf{new}(k + 2, \mathtt{el}) \oplus \mathbf{put}(k + 1, \mathtt{next}, k + 2) \oplus \mathbf{put}(k + 2, \mathtt{item}, k + 2)), p_7 = true]$

```
...
```

(b)

Figure 3: Symbolic evaluation of a program with recurrences

$$S_y \odot \left(\perp \oplus \mathbf{put}(x, \texttt{<q>}, y) \bigoplus_{l=1}^{m} \mu_l \right) = (S \cup \{(x, \texttt{<q>})\})_y \odot \perp \bigoplus_{l=1}^{m} \mu_l \tag{7}$$

$$S_y \odot \left(\perp \oplus \mathbf{put}(x, \texttt{<q>}, z) \bigoplus_{l=1}^{m} \mu_l \right) = (S \setminus \{(x, \texttt{<q>})\})_y \odot \perp \bigoplus_{l=1}^{m} \mu_l, \quad \text{if } z \neq y \tag{8}$$

$$S_y \odot \left(\perp \oplus \mathbf{free}(z) \bigoplus_{l=1}^{m} \mu_l \right) = (\{(x, \texttt{<q>}) | (x, \texttt{<q>}) \in S \wedge x \neq z\})_y \odot \perp \bigoplus_{l=1}^{m} \mu_l \tag{9}$$

The first Rule (5) is a termination rule. If the heap h is $\perp$ the operator's result is the original intermediate set. Allocating a dynamic record does not change the set of predecessors of y. Therefore, the result of operator $\odot$ in Rule (6) is the original intermediate set. In the third (7) and forth (8) rule a dynamic record is either added or removed from the intermediate set depending on the value of reference y. If value r of pointer field q in function $\mathbf{put}(x, \texttt{<q>}, r)$ is equal to y, x must be a predecessor of y. If value r of pointer field `<q>` in function **put** is not y, the dynamic record $(x, \texttt{<q>})$ must be removed from the intermediate set. Note that for heap functions $\mathbf{put}(\mathrm{v}, \perp, r)$, where v is a program variable and r is a reference, rule (7) and rule (8) are valid. In the last rule (9) all elements $(x, \texttt{<q>})$ are deleted. The free operation removes dynamic record x from the intermediate set. Record x can no longer be a predecessor of y.

The intermediate set is given by $\emptyset_y \odot h$ for a program context and a dynamic record y. Based on the intermediate set we derive the set of predecessors $\mathrm{pred}(y)$ as follows,

$$\mathrm{pred}(y) = \begin{cases} \{x \mid (x, \texttt{<q>}) \in \emptyset_y \odot h, \texttt{<q>} \neq \perp\} \cup \\ \quad \{\mathfrak{r} \mid \text{if } \exists x : (x, \perp) \in \emptyset_y \odot h\} & \text{, if } y \neq \mathfrak{r} \\ \emptyset, & \text{otherwise} \end{cases} \tag{10}$$

The transitive closure of the set of predecessors pred^* is inductively defined as follows,

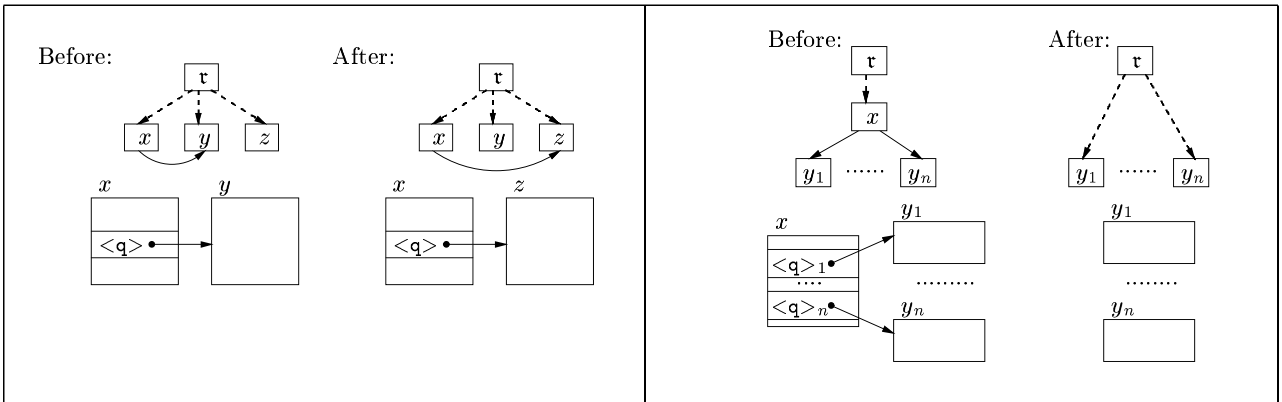
$$\mathrm{pred}^{(0)}(y) = \mathrm{pred}(y) \tag{11}$$

$$\mathrm{pred}^{(k+1)}(y) = \bigcup_{p \in \mathrm{pred}^{(k)}(y)} \mathrm{pred}(p) \tag{12}$$

$$\mathrm{pred}^*(y) = \bigcup_{k \geq 0} \mathrm{pred}^{(k)}(y) \tag{13}$$

Now it is easy to prove that the pointer graph $\mathbb{G}$ is connected if and only if the root $\mathfrak{r}$ is in $\mathrm{pred}^*(y)$ for all $y \in \mathfrak{N}$.

Since (12) and (13) do only involve set union operations, an algorithm calculating $\mathrm{pred}^*(y)$ in order to decide if $\mathbb{G}$ is connected can stop if it finds $\mathfrak{r} \in \mathrm{pred}(p)$ for some p because if $\mathfrak{r} \in \mathrm{pred}^{(\ell)}(y)$ for some ℓ, then also $\mathfrak{r} \in \mathrm{pred}^*(y)$. This



(a) **put** function

(b) **free** function

Figure 4: Pointer graph — update operations

certainly will improve the performance of determining the connectivity of $\mathbb{G}$.

Let us continue the example code of Figure 2(a). In order to detect memory leaks every single statement must be examined. For the sake of demonstration we only inspect statement ℓ_7, which invokes a free operation $\mathfrak{h}_6 \oplus \mathbf{free}(\alpha+1)$ on the heap. There are two pointer fields `left` and `right` of record $\alpha+1$. Let us consider `left`. The value of `left` can be determined by applying $\mathbf{get}(\mathfrak{h}_7, \alpha+1, \texttt{left})$. Thus, in order to detect the memory leak we need to compute the symbolic set of predecessors $\mathrm{pred}^*(\alpha+3)$.

$$
\begin{aligned}
&\emptyset_{\alpha+3} \odot (\perp \oplus \mathbf{new}(\alpha+1, \texttt{bin}) \oplus \mathbf{put}(p_1, \perp, \alpha+1)\\
&\quad \oplus \mathbf{new}(\alpha+2, \texttt{bin}) \oplus \mathbf{put}(p_2, \perp, \alpha+2)\\
&\quad \oplus \mathbf{new}(\alpha+3, \texttt{bin}) \oplus \mathbf{put}(p_3, \perp, \alpha+3)\\
&\quad \oplus \mathbf{put}(\alpha+1, \texttt{left}, \alpha+3) \oplus \mathbf{put}(\alpha+1, \texttt{right}, \alpha+2)\\
&\quad \oplus \mathbf{put}(\alpha+2, \texttt{left}, \alpha+3) \oplus \mathbf{put}(\alpha+3, \texttt{right}, \alpha+2)\\
&\quad \oplus \mathbf{put}(p_2, \perp, \perp) \oplus \mathbf{put}(p_3, \perp, \perp)\\
&\quad \oplus \mathbf{free}(\alpha+1)) = && \text{(Rule 6)}\\
&\emptyset_{\alpha+3} \odot (\perp \oplus \mathbf{put}(p_1, \perp, \alpha+1) \oplus \mathbf{new}(\alpha+2, \texttt{bin})\\
&\quad \oplus \mathbf{put}(p_2, \perp, \alpha+2) \oplus \dots) = && \text{(Rule 8)}\\
&\emptyset_{\alpha+3} \odot (\perp \oplus \mathbf{new}(\alpha+2, \texttt{bin}) \oplus \mathbf{put}(p_2, \perp, \alpha+2)\\
&\quad \oplus \mathbf{new}(\alpha+3, \texttt{bin}) \oplus \dots) = && \text{(Rule 6)}\\
&\emptyset_{\alpha+3} \odot (\perp \oplus \mathbf{put}(p_2, \perp, \alpha+2) \oplus \mathbf{new}(\alpha+3, \texttt{bin})\\
&\quad \oplus \mathbf{put}(p_3, \perp, \alpha+3) \oplus \dots) = && \text{(Rule 8)}\\
&\emptyset_{\alpha+3} \odot (\perp \oplus \mathbf{new}(\alpha+3, \texttt{bin}) \oplus \mathbf{put}(p_3, \perp, \alpha+3)\\
&\quad \oplus \mathbf{put}(\alpha+1, \texttt{left}, \alpha+3) \oplus \dots) = && \text{(Rule 6)}\\
&\emptyset_{\alpha+3} \odot (\perp \oplus \mathbf{put}(p_3, \perp, \alpha+3)\\
&\quad \oplus \mathbf{put}(\alpha+1, \texttt{left}, \alpha+3) \oplus \dots) = && \text{(Rule 7)}\\
&\{(p_3, \perp)\}_{\alpha+3} \odot (\perp \oplus \mathbf{put}(\alpha+1, \texttt{left}, \alpha+3)\\
&\quad \oplus \mathbf{put}(\alpha+1, \texttt{right}, \alpha+2) \oplus \dots) = && \text{(Rule 7)}\\
&\{(p_3, \perp), (\alpha+1, \texttt{left})\}_{\alpha+3} \odot (\perp\\
&\quad \oplus \mathbf{put}(\alpha+1, \texttt{right}, \alpha+2) \oplus \dots) = && \text{(Rule 8)}\\
&\{(p_3, \perp), (\alpha+1, \texttt{left})\}_{\alpha+3} \odot\\
&\quad (\perp \oplus \mathbf{put}(\alpha+2, \texttt{left}, \alpha+3) \oplus \dots) = && \text{(Rule 7)}\\
&\{(p_3, \perp), (\alpha+1, \texttt{left}), (\alpha+2, \texttt{left})\}_{\alpha+3} \odot\\
&\quad (\perp \oplus \mathbf{put}(\alpha+3, \texttt{right}, \alpha+2) \oplus \dots) = && \text{(Rule 8)}\\
&\{(p_3, \perp), (\alpha+1, \texttt{left}), (\alpha+2, \texttt{left})\}_{\alpha+3} \odot (\perp\\
&\quad \oplus \mathbf{put}(p_2, \perp, \perp) \oplus \mathbf{put}(p_3, \perp, \perp) \oplus \dots) = && \text{(Rule 8)}\\
&\{(p_3, \perp), (\alpha+1, \texttt{left}), (\alpha+2, \texttt{left})\}_{\alpha+3} \odot\\
&\quad (\perp \oplus \mathbf{put}(p_3, \perp, \perp) \oplus \mathbf{free}(\alpha+1)) = && \text{(Rule 8)}\\
&\{(\alpha+1, \texttt{left}), (\alpha+2, \texttt{left})\}_{\alpha+3} \odot\\
&\quad (\perp \oplus \mathbf{free}(\alpha+1)) = && \text{(Rule 9)}\\
&\{(\alpha+2, \texttt{left})\}_{\alpha+3} \odot \perp = && \text{(Rule 5)}\\
&\{(\alpha+2, \texttt{left})\}_{\alpha+3}
\end{aligned}
$$

Thus we obtain $\mathrm{pred}(\alpha+3) = \{\alpha+2\}$. In order to compute $\mathrm{pred}^*(\alpha+3)$ we have to determine $\mathrm{pred}(\alpha+2)$, which by a similar computation gives $\mathrm{pred}(\alpha+2) = \{\alpha+3\}$. Hence $\mathrm{pred}^*(\alpha+3) = \{\alpha+2, \alpha+3\}$. Since $\mathfrak{r} \notin \mathrm{pred}^*(\alpha+2)$, we conclude that the pointer graph is disconnected, which means that we have detected a memory leak.

If a recursive heap is given by (2) we obtain a recursively defined intermediate set $S_y(k)$ for $k \geq 0$

$$
\begin{aligned}
S_y(0) &= S_y \odot h(0)\\
S_y(\ell+1) &= S_y(\ell) \odot \perp \bigoplus_{l=1}^{m} \mu_l(\ell),
\end{aligned}
$$

where S_y denotes the intermediate set immediately before the recursive heap description. Based on the recursive intermediate set we define recurrence relations for $\mathrm{pred}(y)$ and $\mathrm{pred}^*(y)$ in an obvious way.

Studying the example in Figure 3 we obtain the following recurrence relation $(k \geq 1)$

$$
\begin{aligned}
S_{k+1}(0) &= \emptyset\\
S_{k+1}(\ell+1) &= S_{k+1}(\ell) \odot (\perp \oplus \mathbf{new}(k+2, \texttt{el})\\
&\qquad \oplus \mathbf{put}(k+1, \texttt{next}, k+2)\\
&\qquad \oplus \mathbf{put}(k+1, \texttt{item}, k+2))\\
&= S_{k+1}(\ell) \odot (\perp \oplus \mathbf{put}(k+1, \texttt{next}, k+2)\\
&\qquad \oplus \mathbf{put}(k+1, \texttt{item}, k+2))\\
&= S_{k+1}(\ell) \odot (\perp \oplus \mathbf{put}(k+1, \texttt{item}, k+2))\\
&= S_{k+1}(\ell).
\end{aligned}
$$

Thus we get $S_{k+1}(z) = \text{pred}(k+1) = \text{pred}^*(k+1) = \emptyset$, which implies that the corresponding pointer graph is disconnected and we have encountered a memory leak. For $k = 0$ no memory leak occurs.

In general, if the recurrence relation for the intermediate set or for the pred-sets cannot be solved, we try to find approximations (cf. [10; 11]). If we do not succeed in approximating the solution, we conservatively assume that the appropriate sets are empty, thus warning the programmer that a memory leak may occur.

We have implemented a prototype of our symbolic framework [12; 13; 3] which includes a system for manipulating symbolic expressions and constraints, techniques for simplifying expressions, automatically generating program contexts, and a recurrence solver. The current implementation of our recurrence solver handles recurrences of the following kind: linear recurrence variables (incremented inside a loop by a symbolic expression defined over constants and invariants), polynomial recurrence variables (incremented by a linear symbolic expression defined over constants, invariants and recurrence variables) and geometric recurrence variables (incremented by a term which contains a recurrence variable multiplied by an invariant). Our algorithm [11] for computing lower and upper bounds of symbolic expressions based on a set of constraints is used to detect whether a recurrence variable monotonically increases or decreases. Even if no closed form can be found for a recurrence variable, monotonicity information may be useful, for instance, to determine whether a pair of references can ever touch the same address. The current implementation of our symbolic evaluation framework models assignments, GOTO, IF, simple I/O and array statements, loops and procedures.

5. RELATED WORK

Much work has been done in the field of statically detecting memory errors including misuses to `nil` pointers, failures to allocate or deallocate memory, uses of undefined or deallocated storage, and dangerous or unexpected aliasing.

There is a variety of conventional heap-based pointer analysis [18; 22; 5; 21; 4; 25; 15] which develops specialized algorithms to solve specific memory (pointer) problems without annotations of the program. The deficiencies of these approaches mainly stem from their imprecise modeling of the control flow.

In [9] a more general approach for memory analysis is described that is based on user-provided annotations about function interfaces, variables and types. Constraints necessary to satisfy these annotations are checked at compile-time. Anomalies are reported for program positions where these constraints are violated. The underlying analysis is not as accurate as our approach due to its imprecise modeling of loops, recurrences, array subscript expressions, and program control flow.

A runtime technique for detecting large classes of pointer and array access errors has been described in [1]. The method is based on pointer conversion, insertion of access checks, pointer operator conversion, and runtime support. Inherently due to the fact of a runtime analysis, this approach can detect more memory errors than static analysis. However, experiments shown in [1] also report on significant execution time overhead due to runtime checking.

A Hoare-like logic is proposed in [14] in order to find dereferences of invalid pointers for a subset of C programs. This analysis is based on an axiomatization of alias and connectivity properties and can deal with circular data structures. Loop invariants, however, have to be supplied by the user, which can be a tedious task.

Another approach describes a static verification of pointer programs using monadic second-order logic [19]. Stores are modeled as strings and decidable specification logic for properties of stores with pointers is employed. This technique depends on user-provided loop invariants and targets a small subset of Pascal programs.

A combination of existing pointer analyses, point-to analysis, and connection analysis is presented in [16]. Pointers referring to stack are modeled based on a "store-based points-to" analysis [8]. Heap pointers are analyzed by a hierarchy of storeless heap analysis, connection analysis [17], and shape analysis [20]. A main drawback of these methods is that control flow information for resolving pointer problems are not considered.

A static analysis of programs that perform destructive updating on heap-allocated storage is described in [23]. This technique is based on shape-analysis. For every program point a conservative, finite characterization of the possible shapes that the program's heap-allocated data structures can have at that point is provided. For certain programs it is possible to infer the underlying data structures, e.g. linked lists and trees. However, in [7] it has been reported that for detecting some memory errors (memory leaks, dereferencing of `nil` pointers, etc.) the relationship between different variables as well as control flow must be modeled, which is not included in the shape analysis of [23]. This drawback has been overcome by a refined shape analysis as described in [7]. On the other hand the approach of [7] limits the number of loop iterations which accounts also for the method specified in [20]. In [15] a method is presented that approximates the shape of dynamic data structures. For programs that make major structural changes to data structures, the shape abstraction is not powerful enough to give accurate results.

6. CONCLUSIONS AND FUTURE WORK

We have presented a unified symbolic evaluation framework for detecting and locating memory leaks (losing references to dynamic records without freeing them). The contributions of our approach are as follows:

- A heap algebra $\mathbb{H}$ symbolically describes the heap and the heap operations of a program. Most other approaches approximate heap operations, which can result in many false alarms. The complexity of the heap structure directly relates to the complexity of our heap modeling.
- Recurrences over the heap algebra $\mathbb{H}$ accurately express the dynamic behavior of heap operations inside of loops.
- Arbitrary pointer structures can be modeled.
- A *pointer graph* $\mathbb{G}$ is introduced based on which we detect memory leaks (if and only if the graph is disconnected).

Our framework is not restricted to a specific programming language. The underlying techniques are equally applicable to most existing imperative programming languages. Moreover, our future work will be in three different directions. Firstly, we want to investigate different types of memory anomalies such as illegal accesses to already freed dynamic records or dereferencing nil pointers. Secondly, we plan to build a similar analysis for object-oriented programming languages, in particular for Java. This includes also a Java garbage collector that can greatly benefit by our techniques. Thirdly, we want to develop analyses that prove specific heap properties ("listness", "treeness", etc.) at certain program points.

7. REFERENCES

[1] T. Austin, S. Breach, and G. Sohi. Efficient detection of all pointer and array access errors. In *Conference on Programming Language Design and Implementation*, Jun. 1994.

[2] J. Blieberger, B. Burgstaller, and B. Scholz. Interprocedural Symbolic Evaluation of Ada Programs with Aliases. In *Ada-Europe'99 International Conference on Reliable Software Technologies*, pages 136–145, Santander, Spain, June 1999.

[3] J. Blieberger, T. Fahringer, and B. Scholz. Symbolic cache analysis for real-time systems. To appear in Real-Time Systems Journal., 1999.

[4] M. Burke, P. Carini, J.-D. Choi, and M. Hind. Flow-insensitive interprocedural alias analysis in the presence of pointers. In K. Pingali, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Proceedings of the 7th International Workshop on Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science, pages 234–250, Ithaca, New York, Aug. 8–10, 1994. Springer-Verlag.

[5] A. Deutsch. Semantic models and abstract interpretation techniques for inductive data structures and pointers. In *Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, June 1995.

[6] E. Dijkstra. *A discipline of programming.* Prentice Hall, New Jersey, 1976.

[7] N. Dor, M. Rodeh, and M. Sagiv. Detecting memory errors via static pointer analysis. In *Workshop on Program Analysis for Software Tools and Engineering PARLE'98.* ACM Press, 1998.

[8] M. Emami, R. Ghiya, and L. J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. *ACM SIGPLAN Notices*, 29(6):242–256, June 1994.

[9] D. Evans. Static detection of dynamic memory errors. In *SIGPLAN Conference on Programming Language Design and Implementation (PLDI '96)*, May 1996.

[10] T. Fahringer. Efficient Symbolic Analysis for Parallelizing Compilers and Performance Estimators. *Journal of Supercomputing, Kluwer Academic Publishers*, 12(3), 1998.

[11] T. Fahringer. Symbolic Analysis Techniques for Program Parallelization. *Journal of Future Generation Computer Systems, Elsevier Science, North-Holland*, 13(1997/98):385–396, March 1998.

[12] T. Fahringer and B. Scholz. Symbolic Evaluation for Parallelizing Compilers. In *Proc. of the ACM International Conference on Supercomputing*, July 1997.

[13] T. Fahringer and B. Scholz. A unified symbolic evaluation framework for parallelizing compilers. Submitted. `http://www.par.univie.ac.at/~tf/papers/symbolic/symbol_eval.ps`, 1998.

[14] P. Fradet, R. Caugne, and D. L. Métayer. Static detection of pointer errors: An axiomatisation and a checking algorithm. In H. R. Nielson, editor, *Programming Languages and Systems—ESOP'96, 6th European Symposium on Programming*, volume 1058 of *LNCS*, Linköping, Sweden, 22–24 Apr. 1996. Springer.

[15] R. Ghiya and L. Hendren. Is it a tree, a DAG, or a cyclic graph? A shape analysis for heap-directed pointers in C. In *Symposium on Principles of Programming Languages.* ACM Press, Jan. 1996.

[16] R. Ghiya and L. Hendren. Putting pointer analysis to work. In *Symposium on Principles of Programming Languages*, Jan. 1998.

[17] R. Ghiya and L. J. Hendren. Connection analysis: A practical interprocedural heap analysis for C. *International Journal of Parallel Programming*, 24(6):547–578, Dec. 1996.

[18] L. Hendren and A. Nicolau. Parallelizing programs with recursive data structures. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):35–47, Jan. 1990.

[19] J. L. Jensen, M. E. Jørgensen, M. I. Schwartzbach, and N. Klarlund. Automatic verification of pointer programs using monadic second-order logic. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI-97)*, volume 32, 5 of *ACM SIGPLAN Notices*, pages 226–234, New York, June15–18 1997. ACM Press.

[20] N. D. Jones and S. S. Muchnick. Flow analysis and optimization of Lisp-like structures. In S. S. Muchnick and N. D. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 102–131. Englewood Cliffs, N.J.: Prentice-Hall, 1981.

[21] W. Landi and B. G. Ryder. Safe approximate algorithm for interprocedural pointer aliasing. *ACM SIGPLAN Notices*, 27(7):235–248, 1992.

[22] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. In *Symposium on Principles of Programming Languages*, St. Petersburg Beach, FL, Jan.Jan. 1996.

[23] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Transactions on Programming Languages and Systems*, 20(1):1–50, Jan. 1998.

[24] R. Tennent. Denotational semantics of programming languages. *Communication of the ACM*, 19(8), Aug. 1976.

[25] R. P. Wilson and M. S. Lam. Efficient context-sensitive pointer analysis for C programs. In *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation (PLDI)*, La Jolla, California, 18–21 June 1995.