Chapter 1

Reasoning about Heap Manipulating Programs using Automata Techniques

Supratik Chakraborty
Dept. of Computer Science and Engineering,
I.I.T. Bombay, Powai,
Mumbai 400076, India
supratik@cse.iitb.ac.in

Automatically reasoning about programs is of significant interest to the program verification, compiler development and testing communities. While property checking for programs is undecidable in general, techniques for reasoning about specific classes of properties have been developed and successfully applied in practice. In this article, we discuss three automata-based techniques for reasoning about heap related properties of programs that dynamically allocate and free memory. Specifically, we discuss a regular model checking based approach, an approach based on a storeless semantics and Hoare-style reasoning, and a counter automaton based approach.

1.1. Introduction

Automata theory has been a key area of study in computer science both for the theoretical significance of its results, as well as for the remarkable success of automata based techniques in diverse application areas. Interesting examples of such applications include pattern matching in text files, tokenizing input strings in lexical analyzers (used in compilers), formally verifying properties of programs, solving Presburger arithmetic constraints, machine learning and pattern recognition. In this article, we focus on one such class of applications, and discuss how automata based techniques can be used to formally reason about computer programs that dynamically allocate and free memory, and update link structures between such memory locations.

Formally reasoning about programs has interested computer scientists since the days of Alan Turing. Among the more difficult problems in this area is analysis of programs that manipulate dynamic linked data structures. This article is a survey of some important automata-based techniques to address this problem. Specifically, we wish to answer the following question: Given a sequential program that manipulates dynamic linked data structures by means of creation and deletion of memory cells and updation of links between them, how do we prove assertions about the the
resulting structures in heap memory (e.g. linked lists, trees, etc.)? This problem, also commonly called shape analysis, has been the subject of extensive research over the last few decades. Simple as it may seem, answering the above question in its complete generality is computationally impossible or undecidable. Nevertheless, its practical significance in optimization and verification of programs has motivated researchers to make significant advances for special classes of programs and properties. These advances have borrowed tools and techniques from different areas of computer science and mathematics. We restrict our discussion to a subset of these techniques that are based on automata theory. Specifically, we discuss the following three automata based techniques for shape analysis: (i) regular model checking, (ii) Hoare-style reasoning using a storeless semantics, and (iii) a counter automaton based abstraction technique. These techniques also provide insights into how more general cases of the problems might be solved in the future.

The remainder of this article is organized as follows. Section 1.2 presents notation, definitions and some key automata-theoretic results that are used in the subsequent sections. Section 1.3 discusses a simple imperative programming language equipped with constructs to dynamically allocate and free memory, and to update selectors (fields) of dynamically allocated memory locations. The example programs considered in this article are written in this language. Section 1.4 provides an overview of the challenges involved in reasoning about heap manipulating programs. Sections 1.5, 1.6 and 1.7 then describe three automata based techniques for reasoning about such programs. Specifically, we discuss finite word based regular model checking in Section 1.5 and show how this can be used for shape analysis. Section 1.6 presents a regular language (automaton) based storeless semantics for our language, and a logic for reasoning about programs using this semantics. We show in this section how this logic can be used in Hoare-style reasoning about programs. A counter automaton based abstraction of programs manipulating singly linked lists is discussed in Section 1.7. Finally, section 1.8 concludes the article.

1.2. Automata notation and preliminaries

Let Σ be a finite alphabet and Σ∗ denote the set of all finite words on Σ. A language over Σ is a (possibly empty) subset of Σ∗. A finite-state transition system over Σ is a 4–tuple $\mathcal{B} = (Q, \Sigma, Q_0, \delta)$ where $Q$ is a finite set of states (also called control locations), $Q_0 \subseteq Q$ is the set of initial states, and $\delta \subseteq Q \times \Sigma \times Q$ is the transition relation. If $|Q_0| = 1$ and $\delta$ is a function from $Q \times \Sigma$ to $Q$, we say that $\mathcal{B}$ is a deterministic finite-state transition system. Otherwise, we say that $\mathcal{B}$ is nondeterministic. A finite-state automaton $\mathcal{A}$ over $\Sigma$ is a finite-state transition system equipped with a set of designated final states. Thus, $\mathcal{A} = (Q, \Sigma, Q_0, \delta, F)$, where $\mathcal{B} = (Q, \Sigma, Q_0, \delta)$ is a finite-state transition system and $F \subseteq Q$ is a set of final states. The notation $|\mathcal{A}|$ is often used to refer to the number of states ($|Q|$) of automaton $\mathcal{A}$. 
The transition relation $\delta$ induces a relation $\hat{\delta} \subseteq Q \times \Sigma^* \times Q$, defined inductively as follows: (i) for every $q \in Q$, $(q, \varepsilon, q) \in \hat{\delta}$, and (ii) for every $q_1, q_2, q_3 \in Q$, $w \in \Sigma^*$ and $a \in \Sigma$, if $(q_1, w, q_2) \in \hat{\delta}$ and $(q_2, a, q_3) \in \delta$, then $(q_1, w.a, q_3) \in \hat{\delta}$, where “$\cdot$” denotes string concatenation. A word $w \in \Sigma^*$ is said to be accepted by the automaton $A$ iff $(q_0, w, q_f) \in \hat{\delta}$ for some $q_0 \in Q_0$ and $q_f \in F$. The set of all words accepted by $A$ is called the language of $A$, and is denoted $L(A)$. A language that is accepted by a finite-state automaton is said to be regular.

Given languages $L_1$ and $L_2$, we define the language concatenation operator as $L_1 \cdot L_2 = \{w \mid \exists x \in L_1 \exists y \in L_2, w = x.y\}$. For a language $L$, the language $L^i$ is defined as follows: $L^0 = \{\varepsilon\}$, where $\varepsilon$ is the empty string, and $L^i = L^{i-1} \cdot L$, for all $i \geq 1$. The Kleene-closure operator on languages is defined as $L^* = \{w \mid \exists i \geq 0, w \in L^i\}$. We define the left quotient of $L_2$ with respect to $L_1$ as $L_1^\leftarrow L_2 = \{w \mid w \in \Sigma^*, \exists v \in L_1 \text{ and } v.w \in L_2\}$.

The following results from automata theory are well-known and their proofs can be found in Hopcroft and Ullman’s book [1].

- **(1)** If $A_1$ and $A_2$ are finite-state automata on an alphabet $\Sigma$, there exist effective constructions of finite-state automata accepting each of $L(A_1) \cup L(A_2)$, $L(A_1) \cap L(A_2)$, $\Sigma^* \setminus L(A_1)$, $L(A_1) \cdot L(A_2)$, $L^*(A_1)$ and $L(A_1)^{-1} L(A_2)$.

- **(2)** For every non-deterministic finite-state automaton $A_1$, there exists a deterministic finite-state automaton $A_2$ such that $L(A_1) = L(A_2)$ and $|A_2| \leq 2^{4|A_1|}$.

- **(3)** For every deterministic finite-state automaton $A$, there exists a minimal deterministic finite-state automaton $A_{\text{min}}$ that is unique up to isomorphism and has $L(A) = L(A_{\text{min}})$. Thus, any deterministic finite-state automaton accepting $L(A)$ must have at least as many states as $|A_{\text{min}}|$.

Let $L$ be a language, and let $R_L$ be a binary relation on $\Sigma^* \times \Sigma^*$ defined as follows: $\forall x, y, z \in \Sigma^*$, $(x, y) \in R_L$ iff $\forall z \in \Sigma^*$, $x.z \in L \Leftrightarrow y.z \in L$. The relation $R_L$ is easily seen to be an equivalence relation. Therefore, $R_L$ partitions $\Sigma^*$ into a set of equivalence classes. A famous theorem due to Myhill and Nerode (see [1] for a nice exposition) states that a language $L$ is regular iff the index of $R_L$ is finite. Furthermore, there is no deterministic finite-state automaton that recognizes $L$ and has fewer states than the index of $R_L$.

A finite state transducer over $\Sigma$ is a 5-tuple $\tau = (Q, \Sigma_e \times \Sigma_e, Q_0, \delta_\tau, F)$, where $\Sigma_e = \Sigma \cup \{\varepsilon\}$ and $\delta_\tau \subseteq Q \times \Sigma_e \times \Sigma_e \times Q$. Similar to the case of finite state automata, we define $\hat{\delta}_\tau \subseteq Q \times \Sigma^* \times \Sigma^* \times Q$ as follows: (i) for every $q \in Q$, $(q, \varepsilon, \varepsilon, q) \in \hat{\delta}_\tau$, and (ii) for every $q_1, q_2, q_3 \in Q$, $u, v \in \Sigma^*$ and $a, b \in \Sigma_e$, if $(q_1, u, v, q_2) \in \hat{\delta}_\tau$ and $(q_2, a, b, q_3) \in \delta_\tau$, then $(q_1, u.a, v.b, q_3) \in \hat{\delta}_\tau$. The transducer $\tau$ defines a regular binary relation $R_\tau = \{(u, v) \mid u, v \in \Sigma^* \text{ and } \exists q \in Q_0, \exists q' \in F, (q, u, v, q') \in \hat{\delta}_\tau\}$. For notational convenience, we will use $\tau$ for $R_\tau$, when there is no confusion. Given a language $L \subseteq \Sigma^*$ and a binary relation $R \subseteq \Sigma^* \times \Sigma^*$, we define $R(L) = \{v \mid \exists u \in L, (u, v) \in R\}$. Given binary relations $R_1$ and $R_2$ on $\Sigma^*$, we use $R_1 \circ R_2$ to denote the composed relation $\{(u, v) \mid u, v \in \Sigma^* \text{ and } \exists x \in \Sigma^*, (u, x) \in R_1 \land (x, v) \in R_2\}$.
Let $id \subseteq \Sigma^* \times \Sigma^*$ denote the identity relation on $\Sigma^*$. For every relation $R \subseteq \Sigma^* \times \Sigma^*$, we define $R^0 = id$, and $R^{i+1} = R \circ R^i$ for all $i \geq 0$.

With this background, we now turn our attention to studying some automata based techniques for reasoning about heap manipulating programs.

1.3. A language for heap manipulating programs

Memory locations accessed by a program can be either statically allocated or dynamically allocated. Storage represented by statically declared program variables are allocated on the stack when the program starts executing. If the program also dynamically allocates memory, the corresponding storage comes from a logical pool of free memory locations, called the heap. In order for a program to allocate, deallocate or access memory locations from the heap, special constructs are required in the underlying programming language. We present below a simple imperative programming language equipped with these constructs. Besides supporting allocation and deallocation of memory from the heap, our language also supports updating and reading from selectors (or fields) of allocated memory locations. This makes it possible to write interesting heap manipulating programs using our language. In order to keep the discussion focused on heaps, we will henceforth be concerned only about link structures between memory locations. Therefore we restrict our language to have a single abstract data type, namely pointer to a memory location. All other data-valued selectors (or fields) of memory locations are assumed to be abstracted away, and we consider only pointer-valued selectors.

Dynamically allocated memory locations are also sometimes referred to as heap objects in the literature. Similarly, selectors of such memory locations are sometimes referred to as fields of objects. In this article, we will consistently use the terms memory locations and selectors to avoid confusion with objects and fields in the sense of object-oriented programs. The syntax of our language is given in Table 1.1.

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>PVar</td>
<td>$u \mid v \mid \ldots$ (pointer-valued variables)</td>
</tr>
<tr>
<td>FName</td>
<td>$f \mid g \mid \ldots$ (pointer-valued selectors)</td>
</tr>
<tr>
<td>PExp</td>
<td>$PVar \mid PVar \rightarrow FName$</td>
</tr>
<tr>
<td>BExp</td>
<td>$PVar = PVar \mid PVar = \text{nil} \mid \text{not} BExp \mid BExp \text{ or } BExp \mid BExp \text{ and } BExp$</td>
</tr>
<tr>
<td>Stmt</td>
<td>$\text{AsgnStmt} \mid \text{CondStmt} \mid \text{LoopStmt} \mid \text{SeqCompStmt} \mid \text{AllocStmt} \mid \text{FreeStmt}$</td>
</tr>
<tr>
<td>AsgnStmt</td>
<td>$PExp := PVar \mid PVar := \text{PExp} \mid \text{PExp := nil}$</td>
</tr>
<tr>
<td>AllocStmt</td>
<td>$PVar := \text{new}$</td>
</tr>
<tr>
<td>FreeStmt</td>
<td>$\text{free}(PVar)$</td>
</tr>
<tr>
<td>CondStmt</td>
<td>$\text{if } (\text{BoolExp}) \text{ then } \text{Stmt} \text{ else } \text{Stmt}$</td>
</tr>
<tr>
<td>LoopStmt</td>
<td>$\text{while } (\text{BoolExp}) \text{ do } \text{Stmt}$</td>
</tr>
<tr>
<td>SeqCompStmt</td>
<td>$\text{Stmt} ; \text{Stmt}$</td>
</tr>
</tbody>
</table>

Table 1.1. Here, $PExp$ represents a pointer expression obtained by concatenating at most one selector to a pointer-valued variable. $BExp$ represents Boolean expres-
sions on pointer variables, and are constructed using two basic predicates: the “=” predicate for checking equality of pointer variables, and the “= nil” predicate for checking if a pointer variable has the nil value. AllocStmt generates statements for allocating a fresh memory location in the heap. A pointer to the freshly allocated location is returned and assigned to a pointer variable. FreeStmt generates statements for de-allocating a memory location pointed to by a pointer variable. The remaining constructs are standard and we skip describing their meanings.

We restrict the use of long sequences of selectors in our language. This does not sacrifice generality since reference to a memory location through a sequence of \( n \) selectors can be effected by introducing \( n - 1 \) fresh temporary variables, and using a sequence of assignment statements, where each statement uses at most one selector. Our syntax for assignment statements also disallows statements of the form \( u->f := w->g \). The effect of every such assignment can be achieved by introducing a fresh temporary variable \( z \) and using a sequence of two assignments: \( z := v->g; u->f := z \) instead. For simplicity of analysis, we will further assume that assignment statements of the form \( u := u \) are not allowed. This does not restrict the expressiveness of the language, since \( u := u \) may be skipped without affecting the program semantics. Assignment statements of the form \( u := u->f \) frequently arise in programs that iterate over dynamically created linked lists. We allow such assignments in our language for convenience of programming. However, we will see later that for purposes of analysis, it is simpler to replace every occurrence of \( u := u->f \) by \( z := u->f; u := z \) where \( z \) is a fresh temporary variable.

**Example 1.1.** The following program written in the above language searches a linked list pointed to by \( \text{hd} \) for the element pointed to by \( x \). On finding this element, the program allocates a new memory location and inserts it as a new element in the list immediately after the one pointed to by \( x \). The relative order of all other elements in the list is left unchanged.

| L1 | t1 := hd; | L6 | t2->n := t3; |
| L2 | while (not (t1 = nil)) do | L7 | x->n := t2; |
| L3 | if (t1 = x) then | L8 | t1 := t1->n; |
| L4 | t2 := new; | L9 | else t1 := t1->n |
| L5 | t3 := x->n; | L10 | // end if-then-else, end while-do |

### 1.4. Challenges in reasoning about heap manipulating programs

Given a heap manipulating program such as the one in Example 1.1, there are several interesting questions that one might ask. For example, can the program dereference a null pointer, leading to memory access error? Or, if \( \text{hd} \) points to an (a)cyclic linked list prior to execution of the program, does it still point to an
(a) cyclic linked list after the program terminates? Alternatively, can executing the program lead to memory locations allocated in the heap, but without any means of accessing them by following selectors starting from program variables? The generation of such “orphaned” memory locations, also called garbage, is commonly referred to as memory leak. Yet other important problems concern finding pairs of pointer expressions that refer to the same memory location at a given program point during some or all executions of the program. This is also traditionally called may- or must-alias analysis, respectively.

Unfortunately, reasoning about heap manipulating programs is difficult. A key result due to Landi [2] and Ramalingam [3] shows that even a basic problem like may-alias analysis admits undecidability for languages with if statements, while loops, dynamic memory allocation and recursive data structures (like linked lists and trees). Therefore any reasoning technique that can be used to identify may-aliases in programs written in our language must admit undecidability. This effectively rules out the existence of exact algorithms for most shape analysis problems. Research in shape analysis has therefore focused on approximate techniques that work well in practice for useful classes of programs and properties.

A common issue that all shape analysis techniques must address is that of representing the heap in a succinct yet sufficiently accurate way for answering questions of interest. Since our language permits only pointer-valued selectors, the heap may be viewed as a set of memory locations with a link structure arising from values of selectors. A natural representation of this view of the heap is a labeled directed graph. Given a program \( P \), let \( \Sigma_p \) and \( \Sigma_f \) denote the set of variables and set of selectors respectively in \( P \). We define the heap graph as a labeled directed graph \( G_H = (V, E, v_{\text{nil}}, \lambda, \mu) \), where \( V \) denotes the set of memory locations allocated by the program and always includes a special vertex \( v_{\text{nil}} \) to denote the nil value of pointers, \( E \subseteq (V \setminus \{v_{\text{nil}}\}) \times V \) denotes the link structure between memory locations, \( \lambda : E \to 2^{\Sigma_f} \setminus \{\emptyset\} \) gives the labels of edges, and \( \mu : \Sigma_p \rightharpoonup V \) defines the (possibly partial) mapping from pointer variables to memory locations in the heap. Specifically, there exists an edge \((u, v)\) with label \( \lambda((u, v)) \) in graph \( G_H \) iff for every \( f \in \lambda((u, v)) \), selector \( f \) of memory location \( u \) points to memory location \( v \), or to \( \text{nil} \) (if \( v = v_{\text{nil}} \)). Similarly, for every variable \( x \in \Sigma_p \), \( \mu(x) = v \) iff \( x \) points to memory location \( v \), or to \( \text{nil} \) (if \( v = v_{\text{nil}} \)).

Since a program may allocate an unbounded number of memory locations, the heap graph may become unbounded in general. This makes it difficult to use an explicit representation of the graph, and alternative finite representations must be used. Unfortunately, representing unbounded heap graphs finitely comes at the cost of losing some information about the heap. The choice of representation formalism is therefore important: the information represented must be sufficient for reasoning about properties we wish to study, and yet unnecessary details must be abstracted away. In order to model the effect of program statements, it is furthermore necessary to define the operational semantics of individual statements in terms of the
chosen representation formalism. Ideally, the representation formalism should be such that the operational semantics is definable in terms of efficiently implementable operations on the representation. A reasoning engine must then use this operational semantics to answer questions pertaining to the state of the heap resulting from execution of an entire program. Since the choice of formalism used for representing the heap affects the complexity of analysis, a careful balance must be struck between the expressiveness of the formalism and the decidability or complexity of reasoning with it. In the following three sections, we look at three important automata based techniques for addressing the above issues.

1.5. Shape analysis using regular model checking

Model checking refers to a class of techniques used to determine if a finite or infinite state model of a system satisfies a property specified in a suitable logic [4]. The state transition model is usually derived by defining a notion of system state, and by capturing the small-step operational semantics of the system by means of a state transition relation. In symbolic model checking [4], sets of states are represented symbolically, rather than explicitly. Regular model checking, henceforth called \( \textit{RMC} \), is a special case of symbolic model checking, in which words or trees over a suitable alphabet are used to represent states. Symbolic model checking using word based representation of states was first introduced by Kesten et al [5] and Fribourg [6]. Subsequently, significant advances has been made in this area, with contributions by several researchers (see [7] for an excellent survey). While RMC today refers to a spectrum of techniques that use finite or infinite words, trees or graphs for representing states, the current discussion focuses primarily on finite word based representation of states, and is based on the works of Jonsson, Nilsson, Abdulla, Bouajjani, Moro, Toulli, Habermehl, Vojnar and others [7–14].

Since states are represented as finite words, a set of states is a language of finite words. Moreover, if the set is regular, it can be represented by a finite-state automaton. The small-step operational semantics of the system is a binary relation that relates a word representing the state before executing an action/statement to one representing the state after executing the action/statement. The transition relation can therefore be viewed as a word transducer. For several classes of systems, including programs manipulating singly linked lists, the transition relation can be modeled as a finite state transducer. Given a regular set \( I \) of words representing the initial states, and a finite state transducer \( \tau \), automata theoretic constructions can be used to obtain finite state representations of the sets (languages) \( R^i \tau(I) \), \( i \geq 1 \), where \( R_\tau \) is the binary relation defined by \( \tau \). For notational convenience, we will use \( \tau^i(I) \) to represent the same set, when there is no confusion. The limit language \( \tau^*(I) \) represents the set of all states reachable from some state in \( I \) in finitely many steps. Given a regular set \( \textit{Bad} \) of undesired states, finding if some state in \( \textit{Bad} \) can be reached from \( I \) reduces to checking if the languages \( \textit{Bad} \) and \( \tau^*(I) \) have a
nonempty intersection. Unfortunately, computing $\tau^+(I)$ is difficult in general, and $\tau^+(I)$ may not even be regular even when both $I$ and $\tau$ are regular. A common approach to circumvent this problem is to use an upper approximation of $\tau^+(I)$, that is both regular and efficiently computable. We briefly survey techniques for computing such upper approximations later in this section.

1.5.1. Representing program states as words

To keep things simple, let us consider the class of programs that manipulate dynamically linked data structures, but where each memory location has a single pointer-valued selector. As an example, the program in Example 1.1 belongs to this class. We will treat creation of garbage as an error, and will flag the possibility of garbage creation during our analysis. Hence, for the remainder of the discussion, we will assume that no garbage is created. Under this assumption, the heap graph at any snapshot of execution of a program in our class consists of singly linked lists, with possible sharing of elements and circularly linked structures. Figure 1.1 shows three examples of such graphs.

Adapting the terminology of Manevich et al [15], we say that a node $v$ in the heap graph is heap-shared if either (i) there are two or more distinct nodes with edges to $v$, or (ii) $v$ is pointed to by a program variable and there is a node with an edge to $v$. Furthermore, a node $v$ is called an interruption if it is either heap-shared or pointed to by a program variable. As an example, the heap graph depicted in Figure 1.1a has two heap-shared nodes ($B$ and $D$) and five interruptions ($A$, $B$, $D$, $E$ and $G$). It can be shown that for a program (in our class) with $n$ variables, the number of heap-shared nodes and interruptions in the heap graph is bounded above by $n$ and $2n$, respectively [15]. The heap graph can therefore be represented as a set of at most $2n$ uninterrupted list segments, where each uninterrupted list segment has the following properties: (i) the first node is an interruption, (ii) either the last node is a heap-shared node, or the selector of the last node is uninitialized or points to nil, and (iii) no other node is an interruption. As an example, the heap graph in Figure 1.1a has five uninterrupted list segments: $A \rightarrow B$, $B \rightarrow C \rightarrow D$, $D \rightarrow F \rightarrow D$, $E \rightarrow D$ and $G \rightarrow \text{nil}$.
The above observation motivates representing the heap graph as a set of uninterrupted list segments. To represent a list segment, we first assign a unique name to every heap-shared node, and rank the set of all names (heap-shared node names and program variable names). This allows us to represent a set of names uniquely as a rank-ordered sequence of names. An uninterrupted list segment with \( m \) nodes, each having a single selector named \( f \), can then be represented by listing the set of names (program variable names and/or heap-shared node name) corresponding to the first node, followed by \( m \) copies of \( .f \) (selector name), followed by \( M, \top, \) or \( \bot \), depending on whether the last node is heap-shared with name \( M \), or the selector of the last element is uninitialized or points to \texttt{nil}, respectively.

A heap graph can then be represented as a word obtained by concatenating the representation of each uninterrupted list segment, separated by a special symbol, say \( | \). For example, the heap graph in Figure 1.1a can be represented by the word \( z.fB|xB.f|fD|D.f.fD|w.fD|y.f\bot \), where we have assumed that names in lower case (e.g., \( x \)) are ranked before those in upper case (e.g., \( B \)). Note that the order in which list segments are enumerated is arbitrary. Hence a heap graph may have multiple word representations. Since the number of heap-shared nodes is bounded above by the number of program variables, it is useful to have a statically determined pool of ranked names for heap-shared nodes of a given program. Whenever a new heap-shared node is created (by the action of a program statement), we can assign a name to it from the set of unused names in this pool. Similarly, when a heap-shared node ceases to be heap-shared (by the action of a program statement), we can add its name back to the pool of unused names. While this allows us to work with a bounded number of names for heap-shared nodes, it also points to the need for reclaiming names for reuse.

In order to represent a state of a heap manipulating program, we need to keep track of some additional information beyond representing the heap graph. Given a program with \( n \) variables, let \( \Sigma_M = \{ M_0, M_1, M_2, \ldots, M_n \} \) be a set of \( n+1 \) rank-ordered names for heap-shared nodes of the program, and let \( \Sigma_L \) be the set of program locations. We follow the approach of Bouajjani et al [12], and represent the program state as a word \( w = |w_1|w_2|w_3|w_4|w_5 \), where \( w_5 \) is a word representation of the heap graph, as described above, and \( | \) does not appear in any of \( w_1, w_2, w_3, \) or \( w_4 \). Sub-word \( w_1 \) contains the current program location (\( \in \Sigma_L \)) and a flag indicating the current mode of computation. This flag takes values from the set \( \Sigma_C = \{ C_N, C_0, C_1, C_2, \ldots, C_n \} \), where \( n \) is the number of program variables. A value of \( C_N \) for the flag denotes normal mode of computation. A value of \( C_i \) (\( 0 \leq i \leq n \)) for the flag denotes a special mode of computation used to reclaim \( M_i \) as an unused name for heap-shared nodes. Sub-word \( w_2 \) contains a (possibly empty) rank-ordered sequence of unused names for heap-shared nodes. Sub-words \( w_3 \) and \( w_4 \) contain (possibly empty) rank-ordered sequences of variable names that are uninitialized and set to \texttt{nil}, respectively. Using this convention, every program state can be represented as a finite word over the alphabet \( \Sigma = \Sigma_C \cup \Sigma_L \cup \Sigma_M \cup \Sigma_p \cup \{ \top, \bot, |, .f \} \),
where \( \Sigma_p \) is the set of all program variables, and all selectors have the name \( f \). We also restrict every heap-shared node in a program state to have exactly one name from the set \( \Sigma_M \).

As an example, suppose Figure 1.1b represents the heap graph when the program in Example 1.1 is at location \( L_9 \) during the second iteration of the while loop, and suppose variables \( t_2 \) and \( t_3 \) are uninitialized. Since there are 5 program variables, \( \Sigma_M = \{ M_0, M_1, M_2, M_3, M_4, M_5 \} \). The state of the program at this point of execution can be represented by the word \( \alpha = |C_N L_9| M_0 M_3 M_4 M_5 | t_2 t_3 | || h d.n n M_1 | t_1 M_1.n M_2 | x M_2.n.n.\bot | \). Note that since there are no variables with the \texttt{nil} value, we have an empty list between a pair of consecutive separators (| |) after \( t_2 t_3 \). Similarly, if Figure 1.1c represents the heap graph when the program is at location \( L_{10} \) in the second iteration of the while loop, the corresponding program state is represented by \( \alpha' = |C_N L_{10}| M_0 M_1 M_3 M_4 M_5 | t_2 t_3 | || h d.n.n M_2 | x t_1 M_2.n.n.\bot | \).

### 1.5.2. Representing operational semantics as word transducers

Given the above word based representation of program states, we now represent the operational semantics of program statements as non-deterministic finite state word transducers. For a given program, we construct a separate transducer for the statement at each program location. We also construct transducers for reclaiming names of heap-shared nodes without changing the actual heap graph, program location or values of variables. These component transducers are then non-deterministically combined to give an overall transducer for the entire program.

Given a word \( w = |w_1|w_2|w_3|w_4|w_5| \) representing the current program state, the sub-word \( |w_1|w_2|w_3|w_4| \) is bounded in length, and \( w_5 \) has a bounded number of uninterrupted list segments. Furthermore, each list segment in \( w_5 \) has a bounded set of names as its first element, and a single element as its last element. The unbound edness of \( w \) therefore really originates from unbounded sequences of \( f \)'s in the list segments in \( w_5 \). Hence, we will assume that each transducer reads \( |w_1|w_2|w_3|w_4| \) and remembers the information in this bounded prefix in its finite memory before reading and processing \( w_5 \). Similarly, we will assume that when reading a list segment in \( w_5 \), each transducer reads the (bounded) set of names representing the first element of the segment, and remembers this in its finite memory before reading the unbounded sequence of \( f \)'s. Every component transducer is also assumed to have two special control states, denoted \( q_{\text{mem}} \) and \( q_{\text{err}} \), with self looping transitions on all symbols of the alphabet. Of these, \( q_{\text{mem}} \) is an accepting state, while \( q_{\text{err}} \) is a non-accepting state. A transducer transitions to control state \( q_{\text{mem}} \) on reading an input word if it detects creation of garbage, or dereferencing of an uninitialized or \texttt{nil}-valued pointer. Such a transition is also accompanied by insertion of a special sequence of symbols, say \( \top \top \top \), in the next state. Note that the sequence \( \top \top \top \) never appears in a word based representation of a valid program state. Furthermore, whenever a transducer sees this special sequence in its input, it transitions to
and retains the sequence in the next state. This ensures that the sequence, one generated, survives repeated applications of the transducer, and manifests itself in the final set of reached states. A transducer transitions to control state $q_{err}$ if it reads an unexpected input. In addition, it transitions to $q_{err}$ if it made an assumption (or guess) about an input word, but subsequently, on reading more of the input word, the assumption was found to be incorrect.

The ability to make a non-deterministic guess and verify it subsequently is particularly useful in situations like the following. Suppose the current statement is $t_3 := x \rightarrow n$ and let $w$ and $w'$ be word representations of the current and next program states, respectively. The transducer must determine if $x \rightarrow n$ is uninitialized or nil in $w_5$ before inserting $t_3$ in $w'_3$ or $w'_4$ respectively. However, since the list segments in $w_5$ are unbounded, and since the transducer must generate $w'_3$ and $w'_4$ before generating $w'_5$, it needs to remember unbounded information as it reads $w_5$. This cannot be achieved using a finite state transducer. Therefore, the transducer must non-deterministically guess whether $x \rightarrow n$ is uninitialized or nil in $w_5$, generate $w'_3$ and $w'_4$ accordingly, remember this guess in its finite memory, and proceed to reading $w_5$ and generating $w'_5$. As $w_5$ is read, if the transducer detects that its guess was incorrect, it must abort the transduction. This is achieved by transitioning to $q_{err}$. For notational convenience, we will refer to a state and its word based representation interchangeably in the following discussion.

We first describe transducers for statements at different program locations. The transducer for the statement at location $L_i$ expects the input word to start with $|CN_{Li}|$. On seeing any other input, it transitions to $q_{err}$. Otherwise, the transducer makes a non-deterministic choice to either enter mode $C_i$ ($0 \leq i \leq n$) for reclaiming heap-shared node name $M_i$, or remain in mode $CN$. In the former case, the transducer simply changes $CN$ to $C_i$ and keeps the rest of the input word unchanged. In the latter case, the transducer retains $CN$ as the first letter of the state. It then determines the next program location that would result after executing the statement at location $L_i$. For several statements (e.g., all statements except those at $L_2$ and $L_3$ in Example 1.1), the next program location can be statically determined, and the transducer replaces $L_i$ with the corresponding next location in the next state. For other statements (e.g. those at $L_2$ and $L_3$ in Example 1.1), the next program location has one of two possible values, depending on the truth value of a Boolean expression in the current state. The truth value of the Boolean expression can, of course, be determined from the input word representing the current state, but only after a sufficiently large part of the input word has been read. The transducer therefore non-deterministically replaces $L_i$ by one of the two possible next program locations, and remembers the corresponding guessed truth value for the Boolean expression in its finite memory. Subsequently, if this guess is found to be incorrect, the transducer transitions to $q_{err}$.

Next, the transducer determines the set of unused names for heap-shared nodes in the next state. For all statements other than assignments of the form $x \rightarrow n :=$
t2 and t1 := t1->n (locations L7 and L8 in Example 1.1), the heap graph after executing a statement has no more heap-shared nodes than the heap graph before executing the statement. For all such statements, the transducer keeps the set of unused names for heap-shared nodes unchanged in the next state. A statement of the form x->n := t2 or t1 := t1->n gives rise to a heap-shared node unless the right hand side of the assignment evaluates to \texttt{nil} or is uninitialized. The transducer therefore guesses whether the right hand side is \texttt{nil} or uninitialized, and remembers this guess in its finite memory. If it guesses the right hand side to be \texttt{nil} or uninitialized, the transducer keeps the set of unused names for heap-shared nodes unchanged in the next state. Otherwise, there are two possibilities: (i) If there is a name in $\Sigma_M$ that is outside the set of unused names in the current state, the transducer can guess that the heap-shared node generated by the current statement was already heap-shared earlier. In this case, the transducer non-deterministically guesses a name, say $M_l$, from outside the set of unused names in the current state, and remembers $M_l$ as the name for the heap-shared node. It also keeps the set of unused names for heap-shared nodes unchanged in the next state. (ii) If the set of unused names in the current state is non-empty, the transducer can guess that the current statement generates a new heap-shared node, which must therefore be given an unused name. Hence, the transducer removes the first name, say $M_l$, from the set of unused names in the current state to generate the set for the next state. It also remembers $M_l$ as the name for the new heap-shared node. In all cases, as more of the input word is read, if any of the guesses is found to be incorrect, the transducer transitions to $q_{err}$.

The set of uninitialized and \texttt{nil}-valued variables can also be updated in a similar manner whenever a statement assigns to a variable or frees the memory location pointed to by a variable. In case the transducer needs to make a non-deterministic guess, it remembers the guess in its finite memory. By the time the entire input word has been read, the transducer can determine whether any of its remembered guesses was incorrect. If so, the transducer transitions to $q_{err}$.

As far as the heap graph is concerned, only assignment, memory allocation and deallocation statements can modify it. For assignments of the form t1 := hd (location L1 in Example 1.1), the program state is changed by removing t1 from its current set (either uninitialized variables, or \texttt{nil}-valued variables or head of an uninterrupted list segment) and inserting it in the same set as hd. For assignments of the form x->n := t1 and t1 := t1->n, suppose the transducer has already guessed that the right hand side of the assignment is neither \texttt{nil} nor uninitialized. Let $M_l$ be the guessed name (either already present or an unused name) for the heap-shared node that results from executing the assignment statement. Assuming that all guesses of the transducer are correct, in the case of x->n := t1, the program state is changed by putting t1 in the same set as $M_l$, and by making the uninterrupted list segment starting from x have only one element with its selector pointing to $M_l$. Similarly, in the case of t1 := t1->n, the program state is changed
Reasoning about Heap Manipulating Programs using Automata Techniques

by removing $t_1$ from its current set, and by putting both $t_1$ and $M_i$ at the head of an uninterrupted list segment that starts from the second element of the list originally pointed to by $t_1$. For statements of the form $t_2 := \text{new}$ (location L4 in Example 1.1), $t_2$ is removed from its current set, and a separate uninterrupted list segment, $t_2.f \top$, is appended at the end of the sub-word representing the heap graph. For statements of the form $\text{free}(x)$, the transducer first guesses all variable names and heap-shared node names in the same set as $x$, and remembers this in its memory. All such variable names are removed from the head of any uninterrupted list segment, and added to the list of uninitialized variables. All such heap-shared node names are also removed from the head of any uninterrupted list segment, and all uninterrupted list segments that end with any such heap-shared node name are made to end with $\top$. Of course, if the transducer subsequently detects that its guess was incorrect, it transitions to $q_{err}$. In all cases, if the word representing the current state indicates that an assignment or deallocation statement dereferences a nil-valued or uninitialized pointer, the transducer transitions to the $q_{mem}$ state. In addition, whenever the word representation of the heap graph is changed, if we are left with a list segment without any program variable name or heap-shared node name as the first element of the segment, we can infer that garbage has been created. In such cases too, the transducer transitions to $q_{mem}$.

A transducer for reclaiming the heap-shared node name $M_i$ ($0 \leq i \leq n$) expects its input word to start with $C_i$. Otherwise, the transducer transitions to $q_{err}$. Such a transducer always leaves the program location, and sets of uninitialized and nil-valued variable names unchanged. If $M_i$ is already in the set of unused names for heap-shared nodes, the transducer simply changes $C_i$ to $C_N$ and leaves its input unchanged. If $M_i$ is not in the set of unused names, the transducer assumes that $M_i$ can be reclaimed. This effectively amounts to making one of the following assumptions: (i) $M_i$ does not appear as the head of any uninterrupted list segment, or (ii) $M_i$ appears as the sole name at the head of an uninterrupted list segment, and there is exactly one uninterrupted list segment that has its last node as $M_i$. The transducer non-deterministically chooses one of these assumptions and remembers it in its finite memory. In the first case, the transducer adds $M_i$ to the set of unused names of heap-shared nodes in the next state, changes the flag $C_i$ to $C_N$, and proceeds to replace all occurrences of $M_i$ at the end of uninterrupted list segments with $\top$. However, if it encounters an uninterrupted list segment that has $M_i$ at its head, the transducer transitions to $q_{err}$. In the second case, let $L_1$ be the uninterrupted list segment starting with the sole name $M_i$, and $L_2$ be the uninterrupted list segment ending with $M_i$. The transducer moves one element from the start of $L_1$ to the end of $L_2$, shortening the list $L_1$ pointed to by $M_1$, and lengthening the list $L_2$ ending with $M_1$. It also non-deterministically guesses whether the list pointed to by $M_1$ has shrunk to length zero, and if so, it adds $M_i$ to the list of unused names of heap-shared nodes, and replaces $C_i$ by $C_N$ in the next state. Of course, if the transducer detects that any of its assumptions/guesses
is incorrect, it transitions to \( q_{err} \). Additionally, if the list \( L_1 \) ends in \( M_i \), we have a garbage cycle and the transducer transitions to \( q_{mem} \). Note that in the second case considered above, one application of the transducer may not succeed in reclaiming the name \( M_i \). However, repeated applications of the above transducer indeed reclaims \( M_i \) if assumption (ii) mentioned above holds.

1.5.3. Computing transitive closures of regular transducers

Given a heap manipulating program \( P \), let \( \tau \) represent the finite state transducer obtained by non-deterministically combining all the finite state transducers discussed above, i.e. one for each statement in \( P \), and one for reclaiming each symbol in \( \Sigma_M \).

By abuse of notation, we will use \( \tau \) to represent the binary relation \( R \) represented by \( \tau \) as well, when there is no confusion. Let \( A_I \) be a finite state automaton representing a regular set of initial states \( I \) of \( P \). The language \( \tau^i(I) \) \((i \geq 0)\) represents the set of states reachable from some state in \( I \) in \( i \) steps, and \( \tau^*(I) = \bigcup_{i=0}^{\infty} \tau^i(I) \) represents the set of all states reachable from \( I \). We discuss below approaches to compute \( \tau^*(I) \) or overapproximations of it.

It is straightforward to derive an automaton representing the set \( \tau(I) \) through a product construction. Let \( A_I = (Q_I, \Sigma, \Delta_I, Q_{0,I}, F_I) \) and \( \tau = (Q_{\tau}, \Sigma_{\tau} \times \Sigma_{\tau}, \Delta_{\tau}, Q_{0,\tau}, F_{\tau}) \). To construct an automaton recognizing \( \tau(I) \), we first construct a product automaton \( A_p = (Q_p, \Sigma \times \Sigma_{\tau}, \Delta_p, Q_{0,p}, F_p) \) as follows.

- \( Q_p = Q_I \times Q_{\tau} \)
- For every \( q_1, q'_1 \in Q_I \), \( q_2, q'_2 \in Q_{\tau} \) and \( \sigma_1, \sigma_2 \in \Sigma_{\tau} \), \((q_1, q_2), (\sigma_1, \sigma_2), (q'_1, q'_2)\) \( \in \Delta_p \) iff \((q_1, \sigma_1, q'_1) \in \Delta_I, (q_2, (\sigma_1, \sigma_2), q'_2) \in \Delta_{\tau} \).
- \( Q_{0,p} = Q_{0,I} \times Q_{0,\tau} \)
- \( F_p = F_I \times F_{\tau} \)

A non-deterministic finite state automaton recognizing \( \tau(I) \) is obtained by ignoring the first component of pairs of symbols labeling edges of \( A_p \).

To obtain an automaton recognizing \( \tau^2(I) \), we could compute \( \tau(\tau(I)) \) as above, where an automaton recognizing \( \tau(I) \) is as obtained above. Alternatively, we could precompute an automaton for the transition relation \( \tau^2 = \tau \circ \tau \), and then determine \( \tau^2(I) \). A non-deterministic finite state automaton for \( \tau^2 \) can be obtained from the automaton for \( \tau \) through a simple product construction. We construct \( \tau^2 = (Q_{\tau^2}, \Sigma \times \Sigma_{\tau}, \Delta_{\tau^2}, Q_{0,\tau^2}, F_{\tau^2}) \), where

- \( Q_{\tau^2} = Q_{\tau} \times Q_{\tau} \)
- For every \( q_1, q_2, q'_1, q'_2 \in Q_{\tau}, \sigma_1, \sigma_2 \in \Sigma_{\tau} \), \((q_1, q_2), (\sigma_1, \sigma_2), (q'_1, q'_2) \) \( \in \Delta_{\tau^2} \) iff \( \exists \sigma_3 \in \Sigma_{\tau} \), \((q_1, (\sigma_1, \sigma_3), q'_1) \in \Delta_I \) and \((q_2, (\sigma_3, \sigma_2), q'_2) \in \Delta_{\tau} \).
- \( Q_{0,\tau^2} = Q_{0,\tau} \times Q_{0,\tau} \)
- \( F_{\tau^2} = F_{\tau} \times F_{\tau} \)

The above technique can be easily generalized to obtain a non-deterministic automaton representing \( \tau^i \) for any given \( i > 0 \). Once a representation of \( \tau^i \) is
obtained, we can obtain an automaton for $\tau^i(I)$. However, this does not immediately tell us how to compute a finite state representation of $\tau^* = \bigcup_{i=0}^{\infty} T^i$ or of $\tau^*(I)$. If $\tau$ is a regular transduction relation, the above constructions show that $\tau^i$ and $\tau^*(I)$ is also regular for every $i \geq 0$. However, $\tau^*$ and $\tau^*(I)$ may indeed be non-regular, since regular languages are not closed under infinite union. Even if $\tau^*$ or $\tau^*(I)$ was regular, a finite representation of it may not be effectively computable from representations of $\tau$ and $I$. A central problem in RMC concerns computing a regular upper approximation of $\tau^*$ or $\tau^*(I)$, for a given regular transduction $\tau$ and regular initial set $I$.

Given finite state automata representing $I$, $\tau$ and a set $Bad$ of error states, the safety checking problem is to determine if $\tau^*(I) \cap Bad = \emptyset$. This can be efficiently answered if a regular representation of $\tau^*(I)$ can be obtained. Depending on $\tau$ and $I$, computing a representation of $\tau^*(I)$ may be significantly simpler than computing a representation of $\tau^*$ directly. Several earlier works, e.g. those due to Dams et al [16], Jonsson et al [8], Touili [13], Bouajjani et al [11], Boigelot et al [17], have tried to exploit this and compute a representation of $\tau^*(I)$ directly. A variety of other techniques have also been developed to compute finite state representations of $\tau^*$ or $\tau^*(I)$. We outline a few of them below.

**Quotienting techniques:** In this class of techniques, automata representations of $\tau^i$ are computed for increasing values of $i$ by using the product construction outlined above. A suitable equivalence relation $\simeq$ is then defined on the states of these automata based on the history of their creation during the product construction, and the quotient automata constructed for each $i$. By defining the equivalence relation appropriately, it is possible to establish equivalence between states of the quotient automata for increasing values of $i$. Thus, states of different quotient automata can be merged into states of one automaton that overapproximates $\tau^i$ for all $i$. For arbitrary equivalence relations, the language accepted by the resulting automaton is a superset of the language by $\tau^+$. However, it is possible to classify equivalence relations [9, 10] such that certain classes of relations preserve transitive closure under quotienting, i.e. the language accepted by $(\tau/ \simeq)^+$ coincides with that accepted by $\tau^+$. The use of such equivalence relations, whenever possible, provides a promising way of computing $\tau^*(I)$ accurately for special classes of systems.

**Abstraction-refinement based techniques:** Techniques in this approach can be classified as being either representation-oriented or configuration-oriented. In representation-oriented abstractions, an equivalence relation $\simeq$ of finite index is defined on the set of states of an automaton representation. However, unlike in quotienting techniques, there is no a priori requirement of preservation of transitive closure under quotienting. Therefore, we start with $\tau$ (or $\tau(I)$) and compute $\tau^2$ (or $\tau^2(I)$) as discussed above. The states of the automaton representation of $\tau^2$ (or $\tau^2(I)$) are then quotiented with $\simeq$. The language accepted by the resulting automaton is, in general, a superset of that accepted by $\tau^2$ (or $\tau^2(I)$). The quotiented automaton is then composed with $\tau$ to compute an overapproximation of $\tau^3$ (or
The states of the resulting automaton are further quotiented with \( \simeq \), and the process is repeated until a fixed point is reached. Since \( \simeq \) is an equivalence relation of finite index, convergence of the sequence of automata is guaranteed after a finite number of steps.

In configuration-oriented abstractions, the words (configurations) in the languages \( \tau(I) \), \( \tau^2(I) \), etc. are abstracted by quotienting them with respect to an equivalence relation \( \simeq \) of finite index defined on their syntactic structure. Configuration-oriented abstractions are useful for word based state representations in which syntactically different parts of a word represent information of varying importance. For example, in our word based representation of program states, i.e. in \( w = |w_1|w_2|w_3|w_4|w_5| \), sub-words \( w_1, w_2, w_3 \) and \( w_4 \) contain important information pertaining to current program location, \texttt{nil}-valued and uninitialized variables, number of heap-shared nodes, etc. Furthermore, these sub-words are bounded and hence represent finite information. Therefore, it may not be desirable to abstract these sub-words in \( w \). On the other hand, long sequences of \( .f \)'s in the representation of uninterrupted list segments in \( w_5 \) are good candidates for abstraction. Bouajjani et al have proposed and successfully used other interesting configuration-oriented abstractions, like \( 0-k \) counter abstractions and closure abstractions, as well for reasoning about heap manipulating programs [18].

Once we have a regular overapproximation of \( \tau^* \) or \( \tau^*(I) \), we can use it to conservatively check if \( \tau^*(I) \cap \text{Bad} = \emptyset \). However, since we are working with an overapproximation of \( \tau^*(I) \), safety checking may give false alarms. It is therefore necessary to construct a concrete counterexample from an abstract sequence of states violating safety conditions, and to check for its spuriousness. If the counterexample is not spurious, we have answered the safety checking question negatively. Otherwise, the counterexample can be used to refine the equivalence relation \( \simeq \) (in both representation-oriented and configuration-oriented abstractions) such that the same counterexample is not generated by the analysis starting from the refined relation. The reader is referred to [11, 18] for details of refinement techniques for both representation-oriented and configuration-oriented abstractions in RMC.

**Extrapolation or widening techniques:** In this approach, we compute automata representations of \( \tau^i(I) \) for successive values of \( i \), and detect a regular pattern in the sequence. The pattern is then extrapolated or widened to guess the limit \( \rho \) of \( \bigcup_{i=0}^{\infty} \tau^i(I) \). A check for convergence of the limit determines whether \( I \cup \tau(\rho) \subseteq \rho \). If the check passes, we have computed an overapproximation of \( \tau^*(I) \). Otherwise, the limit must be increased and a new regular pattern detected. The reader is referred to the works of Touili [13], Bouajjani et al [11] and Boigelot et al [17] for details of various extrapolation techniques. As a particularly appealing example of this technique, Touili [13] showed that if \( I \) can be represented as the concatenation of \( n \) regular expressions \( \rho_1, \rho_2, \ldots, \rho_n \), and if \( \tau(\rho_1 \cdots \rho_n) = \bigcup_{i=1}^{n-1} (\rho_1 \cdots \rho_i \Lambda_i \rho_{i+1} \cdots \rho_n) \), where \( \Lambda_i \) are regular expressions, then \( \tau^*(I) \) is given by \( \rho_1 \Lambda_1^* \rho_2 \Lambda_2^* \cdots \Lambda_{n-1}^* \rho_n \).
Regular language inferencing techniques: This approach uses learning techniques developed for inferring regular languages from positive and negative sample sets to approximate \( \tau^*(I) \). The work of Habermehl et al [14] considers length-preserving transducers, and uses increasingly large complete training sets to infer an automaton representation of \( \tau^*(I) \). The increasing training sets are obtained by gradually increasing the maximum size \( i \) of words in the initial set of states, and computing the set of all states of size up to \( i \) reachable from these initial states. Habermehl et al use a variant of the Trakhtenbrot-Barzdin algorithm for inferring regular languages for this purpose. Once an approximate automaton for \( \tau^*(I) \) has been inferred in this manner, a convergence check similar to the one used for extrapolation techniques, is applied to determine if an overapproximation of \( \tau^*(I) \) has been reached. It has been shown [14] that if \( \tau^*(I) \) is indeed regular, safety checking can always be correctly answered using this technique. Even otherwise, good regular upper approximations of \( \tau^*(I) \) can be obtained.

Let \( \text{UpperApprox}(\tau^*(I)) \) denote a regular upper approximation of \( \tau^*(I) \) obtained using one of the above techniques. We can use \( \text{UpperApprox}(\tau^*(I)) \) to answer interesting questions about the program being analyzed. For example, suppose we wish to determine if execution of the program from an initial state represented by \( I \) can create garbage, or cause an uninitialized/nil-valued pointer to be dereferenced. This can be done by search for the special sequence \( \top \top \top \) in the set of words approximating \( \tau^*(I) \). Thus, if \( \text{BadMem} = \Sigma^* \cdot \{ \top \top \top \} \cdot \Sigma^* \), then \( \text{UpperApprox}(\tau^*(I)) \cap \text{BadMem} = \emptyset \) guarantees that no garbage is created, and no uninitialized or nil-valued pointer is dereferenced. However, if \( \text{UpperApprox}(\tau^*(I)) \cap \text{BadMem} \neq \emptyset \), we must construct an (abstract) counterexample leading from a state in \( I \) to a state in \( \text{BadMem} \) and check for its spuriousness. If the counterexample is not spurious, we have a concrete way to generate garbage or to dereference an uninitialized or nil-valued pointer. Otherwise, we must refine or tighten \( \text{UpperApprox}(\tau^*(I)) \) and repeat the analysis.

For the program in Example 1.1, by carefully constructing the transducer \( \tau \) as discussed earlier, and by applying a simple configuration-oriented abstraction technique, we can show that if \( I = \{ [C_N L_1 | M_0 M_1 M_2 M_3 M_4 M_5 \mid t_1 t_2 t_3] \mid \text{hd.n}^+ \perp \} \cup \{ [C_N L_1 | M_1 M_2 M_3 M_4 M_5 \mid t_1 t_2 t_3] \mid \text{hd.n}^+.M_0 \mid xM_0.n^+ \perp \} \), then \( \text{BadMem} \cap \text{UpperApprox}(\tau^*(I)) = \emptyset \). Thus, regardless of whether the list pointed to by \( \text{hd} \) contains an element pointed to by \( x \), there are no memory access errors or creation of garbage.

The above discussion assumed that every memory location had a single pointer-valued selector. This was crucial to representing the heap graph as a bounded set of uninterrupted list segments. Recent work by Bouajjani et al [19] has removed this restriction. Specifically, programs manipulating complex data structures with several pointer-valued selectors and finite-domain non-pointer valued selectors have been analyzed in their work. Regular languages of words no longer suffice to represent the heap graph in such cases. The program state is therefore encoded as a
tree backbone annotated with routing expressions [19] to represent arbitrary link structures. The operational semantics of program statements are modeled as tree transducers. Techniques from abstract regular tree model checking are then used to check memory consistency properties and shape invariants. The reader is referred to [19] for a detailed exposition on this topic.

A primary drawback of RMC based approaches for reasoning about heaps is the rather indirect way of representing heap graphs and program states as words or extended trees. This in turn contributes to the complexity of the transducers as well. Recently, Abdulla et al [20] have proposed an alternative technique for symbolic backward reachability analysis of heaps using upward closed sets of heap graphs with respect to a well-quasi ordering on graphs, and using an abstract program semantics that is monotone with respect to this ordering. Their method allows heap graphs to be directly represented as graphs, and the operational semantics is represented directly as relations on graphs. The work presented in [20] considers programs manipulating singly linked lists (like the class of programs we considered), although the general idea can be extended to programs manipulating more complex data structures as well. A detailed exposition on this technique is beyond the scope of this article. The interested reader is referred to [20] for details.

1.6. An automata-based semantics and Hoare-style reasoning

In this section, we discuss an automata-based heap semantics for our programming language, and present a logic for Hoare-style deductive reasoning using this semantics. We show how this technique can be used to check heap related properties of programs, using the program in Example 1.1 as an example. Unlike RMC, the approach outlined in this section has a deductive (theorem-proving) flavour.

There are two predominant paradigms for defining heap semantics of programming languages. In store-based semantics used by Yorsh et al [21], Podelski et al [22], Reps et al [23], Bouajjani et al [24], Reynolds [25], Calcagno et al [26], Distefano et al [27] and others, the heap is identified as a collection of symbolic memory locations. A program store is defined as a mapping from the set of pointer variables and selectors of memory locations to other memory locations. Various formalisms are then used for representing and reasoning about this mapping in a finite way. These include, among others, representation of program stores as logical structures for specialized logics [21, 25], or over formulae that use specially defined heap predicates [22, 23, 27], graph-based representations [24], etc. In the alternative storeless semantics, originally proposed by Jonkers [28] and subsequently used by Deutsch [29], Bozga [30, 31], Hoare and Jifeng [32] and others, every memory location is identified with the set of paths that leads to the corresponding node in the heap graph. A path in the heap graph is represented by the sequence of edge labels appearing along the path. Thus the heap is identified as a collection of sets of sequences of edge labels, and not as a collection of symbolic memory locations.
Different formalisms have been proposed in the literature for representing sets of edge label sequences in a finite way. Regular languages (or finite state automata), and formulae in suitably defined logics have been commonly used for this purpose. Since the focus of this article is on automata-based techniques, we discuss below an automata-based storeless heap semantics of our programming language. As an aside, we mention that reasoning techniques for heap manipulating programs cannot always be partitioned based on whether they use storeless or store-based semantics. For example, the work of Rinetzky et al [33] uses a novel mix of store-based and storeless semantics in the framework of TVLA [23] to reason about the effect of procedure calls on data structures in the heap.

### 1.6.1. A storeless semantics

Given a program, let $\Sigma_p$ and $\Sigma_f$ be sets of pointer variables and selectors, as defined earlier. Let $G_H = (V, E, v_{\text{nil}}, \lambda, \mu)$ be the heap graph at a snapshot of execution of the program. We define an access path from a variable $x \in \Sigma_p$ to a node $v$ (possibly $v_{\text{nil}}$) in $V$ as a string $x.\sigma$, where $\sigma$ is a sequence of selector names appearing as edge labels along a path from $\mu(x)$ to $v$, if such a path exists in $G_H$. If no such path exists in $G_H$, the access path from $x$ to $v$ is undefined.

Let $\Sigma$ denote $\Sigma_p \cup \Sigma_f$, and $\wp(S)$ denote the powerset of a set $S$. Adapting the definition of Bozga et al [30], we define a storeless structure $\Upsilon$ as a pair $(S_{\text{nil}}, \Gamma)$, where $S_{\text{nil}} \subseteq \Sigma_p \cdot \Sigma_f^*$ and $\Gamma \subseteq \wp(\Sigma_p \cdot \Sigma_f^*)$. Furthermore, $\Gamma$ is either the empty set or a finite set of languages $\{S_1, S_2, \ldots, S_n\}$ with the following properties for all $i, j \in \{1, \ldots, n\}$.

- **$C_1$**: $S_i \neq \emptyset$.
- **$C_2$**: $i \neq j \Rightarrow S_i \cap S_j = \emptyset$. In addition, $S_i \cap S_{\text{nil}} = \emptyset$.
- **$C_3$**: $\forall \sigma \in S_i \ (\forall \tau, \theta \in \Sigma^+ \ (\sigma = \tau \cdot \theta \Rightarrow \exists k \ ((1 \leq k \leq n) \land (\tau \in S_k) \land (S_k \cdot \{\theta\} \subseteq S_i)))$. A similar property holds for all $\sigma \in S_{\text{nil}}$ as well.

Note that unlike languages in $\Gamma$, there is no non-emptiness requirement on $S_{\text{nil}}$. A storeless structure $\Upsilon = (S_{\text{nil}}, \Gamma)$ with $\Gamma = \{S_1, \ldots, S_n\}$ represents $n$ distinct memory locations in the heap and also the $\text{nil}$ value. Recall that in a heap graph, the $\text{nil}$ value is represented by a special node $v_{\text{nil}}$ with no outgoing edges. Language $S_i \in \Gamma$ may be viewed as the set of access paths in $G_H$ to the $i^{th}$ node (distinct from $v_{\text{nil}}$). Similarly, $S_{\text{nil}}$ may be viewed as the set of access paths to $v_{\text{nil}}$. Condition $C1$ requires all nodes other than $v_{\text{nil}}$ represented in $\Upsilon$ to have at least one access path. Consequently, garbage cannot be represented using this formalism, and we will ignore garbage in the current discussion. Condition $C2$ models the requirement that every access path must lead to at most one node. Condition $C3$ states that every prefix $\tau$ of an access path $\sigma$ must itself be an access path to a node represented in $\Gamma$. This is also called the prefix closure property. Condition $C3$ further encodes the requirement that if multiple access paths reach a node represented by $S_k$, extending each of these access paths with the same suffix $\theta$ must lead us to the same node.
Consider a storeless structure $\Upsilon = (S_{\text{nil}}, \Gamma)$, in which $\Gamma = \{S_1, \ldots, S_n\}$ represents a set of $n$ nodes $\{v_1, \ldots, v_n\}$ in the heap graph $G_H$. The structure $\Upsilon$ can be represented by an $n + 3$ state deterministic finite-state transition system $B_\Upsilon$. A natural (but not necessarily the only) way to obtain $B_\Upsilon$ is by considering the subgraph of $G_H$ consisting of nodes $\{v_{\text{nil}}, v_1, \ldots, v_n\}$. Specifically, let $G_H = (V, E, v_{\text{nil}}, \lambda, \mu)$. We define a transition system $B_\Upsilon = (Q, \Sigma, q_{\text{init}}, \delta)$ with $Q = \{q_{\text{init}}, q_{\text{nil}}, q_{\text{err}}, q_1, \ldots, q_n\}$, where $q_{\text{init}}, q_{\text{nil}}$ and $q_{\text{err}}$ are distinguished control states. For notational convenience, let $v_{\text{nil}}, S_{\text{nil}}$ and $q_{\text{nil}}$ be denoted as $v_0$, $q_0$ and $S_0$ respectively. The transition relation of $B_\Upsilon$ can now be defined as follows: for every $i, j$ in 0 through $n$, we let $(q_i, f, q_j) \in \delta$ iff there exist $v_i$ and $v_j$ in $G_H$ (represented by $S_i$ and $S_j$ respectively in $\Upsilon$), such that $(v_i, v_j) \in E$ and $f \in \lambda((v_i, v_j))$. Furthermore, for every $i$ in 0 through $n$ and $x \in \Sigma_p$, we let $(q_{\text{init}}, x, q_i) \in \delta$ iff there exists $v_i$ (represented by $S_i$ in $\Gamma$) such that $\mu(x) = v_i$. Finally, for all states $q$ (including $q_{\text{err}}$) and for all $f \in \Sigma_p \cup \Sigma_f$, we let $(q, f, q_{\text{err}}) \in \delta$ iff the above construction does not create any outgoing edge from $q$ labeled $f$. Right regularity and prefix closure of $\Upsilon$ ensures that for every $i$ in 0 through $n$, the automaton obtained by letting $q_i$ be the sole accepting state in $B_\Upsilon$ accepts language $S_i$. The automaton $A_{\text{err}}$ obtained with $q_{\text{err}}$ as the sole accepting state accepts all sequences that are not valid access paths to nodes represented by $\Upsilon$.

We now present operational semantics of statements in our language with respect to the above storeless representation of the heap. For notational brevity, we will use the following convention:

- The representations of the heap before and after executing a statement are $\Upsilon = (S_{\text{nil}}, \Gamma)$ and $\Upsilon' = (S'_{\text{nil}}, \Gamma')$, respectively.
- If $\theta$ denotes a PExp, then $\Theta$ denotes the singleton regular language consisting of the access path corresponding to $\theta$. For example, if $\theta$ is $u$ or $u->n$, then $\Theta$ is $\{u\}$ or $\{u.n\}$, respectively. We will also use $u$ to denote $\{u\}$ and $u.n$ to denote $\{u.n\}$.
- For $L, M \subseteq \Sigma^+$, $L \cap M$ denotes $L \setminus (M \cdot \Sigma^*)$, i.e. the set of words in $L$ that do not have any prefix in $M$.
- For $L, M, X \subseteq \Sigma^+$ and $n \in \Sigma_f$, $\chi^{L,n,M}(X)$ denotes the function $\lambda X. X \cup (L.n \cdot (M^{-1}L.n) \cdot (M^{-1}X))$. If $L, M$ and $X$ represent sets of access paths to nodes $v_L, v_M$ and $v_X$ respectively in the heap graph, then $\chi^{L,n,M}(X)$ is the augmented set of access paths to $v_X$ after making the $n$-selector of $v_L$ point to $v_M$ [30].

With this notation, the storeless operational semantics of primitive statements, i.e., assignments, memory allocation and memory deallocation, in our language is given in Table 1.3. For purposes of simplicity, we have assumed that statements of the form $u := u$ or $u := u->n$ are not used in programs written in our language. While this may sound restrictive, every program with such statements can be translated
to a semantically equivalent program without such statements, as described in Section 1.3.

The last column in Table 1.3 lists necessary and sufficient conditions for the storeless operational semantics to be defined. If these conditions are violated, we say that the operational semantics is undefined. There are no conditions for \( u := \theta \) since we have assumed that \( \theta \) is neither \( u \) nor \( u->n \). Whenever an assignment is made to \( u \) (or \( u->n \)), the membership (in \( S_{\text{nil}} \) or \( \Gamma_i \in \Gamma \)) of all access paths that have \( u \) (or \( u.n \)) as prefix is invalidated. Therefore, these paths must be removed from all languages in \( Y \) before augmenting the languages with new paths formed as a consequence of the assignment. Similarly, when memory is de-allocated, all paths with a prefix that was an access path to the de-allocated node must be removed from all languages. A formal proof of correctness of the operational semantics involves establishing the following facts for every primitive statement \( \text{Stmt} \):

1. If \( Y = (S_{\text{nil}}, \Gamma) \) is a valid storeless structure, then so is \( Y' = (S'_{\text{nil}}, \Gamma') \).

2. Let \( v_i \neq v_{\text{nil}} \) be a node represented by \( Y \).
   - If \( v_i \) is neither de-allocated nor rendered garbage by executing \( \text{Stmt} \), there exists a language in \( \Gamma' \) that contains all and only access paths to \( v_i \) after executing \( \text{Stmt} \).
   - If executing \( \text{Stmt} \) de-allocates \( v_i \) and if \( \pi \) was an access path to \( v_i \) prior to executing \( \text{Stmt} \), there is no access path with prefix \( \pi \) in any language in \( Y' \).

3. \( S'_{\text{nil}} \) contains all and only access paths to \( v_{\text{nil}} \) after executing \( \text{Stmt} \).

4. If executing \( \text{Stmt} \) allocates a node \( v'_i \), there exists a language in \( \Gamma' \) that contains all and only access paths to \( v'_i \) after executing \( \text{Stmt} \).

We leave the details of the proof as an exercise for the reader.

### Table 1.3. Storeless operational semantics of primitive statements

<table>
<thead>
<tr>
<th>Statement</th>
<th>( S_{\text{nil}}' )</th>
<th>( \Gamma' )</th>
<th>Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \theta := \text{nil} )</td>
<td>( (S_{\text{nil}} \ominus \Theta) \cup {\Theta} )</td>
<td>( {S_i \ominus \Theta \mid S_i \in \Gamma } \setminus {\emptyset} )</td>
<td>if ( \theta ) is ( u-&gt;n ), ( \exists X \in \Gamma, u \in X )</td>
</tr>
<tr>
<td>( u := \theta )</td>
<td>( S_{\text{nil}}'' \cup u \cdot (\Theta^{-1} S_{\text{nil}}'') ), where ( S_{\text{nil}}'' = S_{\text{nil}} \ominus u )</td>
<td>( {S_i'' \cup u \cdot (\Theta^{-1} S_i'') \mid S_i'' \in \Gamma''} )</td>
<td>( {S_i \ominus u \mid S_i \in \Gamma } \setminus {\emptyset} )</td>
</tr>
<tr>
<td>( u-&gt;n := v )</td>
<td>( \chi^{u,n,Y}(S_{\text{nil}} \ominus (u \cdot n)) )</td>
<td>( {\chi^{u,n,Y}(S_i'') \mid S_i'' \in \Gamma''} ), where ( \Gamma'' = {S_i \ominus (u \cdot n) \mid S_i \in \Gamma } \setminus {\emptyset} )</td>
<td>( \exists Y \in {S_{\text{nil}}} \cup \Gamma, v \in Y )</td>
</tr>
<tr>
<td>( u := \text{new} )</td>
<td>( S_{\text{nil}} \ominus u )</td>
<td>( {S_i \ominus u \mid S_i \in \Gamma } \cup {u} \setminus {\emptyset} )</td>
<td>( \exists Y \in {S_{\text{nil}}} \cup \Gamma, u \in X )</td>
</tr>
<tr>
<td>\text{free}(u)</td>
<td>( S_{\text{nil}} \ominus X )</td>
<td>( {S_i \ominus X \mid S_i \in \Gamma } \setminus {\emptyset} )</td>
<td>( \exists X \in \Gamma, u \in X )</td>
</tr>
</tbody>
</table>

It is clear from the expressions for \( S'_{\text{nil}} \) and \( \Gamma' \) that given finite state automata representations of \( S_{\text{nil}} \) and \( \Gamma_i \in \Gamma \) for \( i \in \{1, \ldots, n\} \), finite-state automata representa-
tions of $S_{\text{nil}}$ and also of every language in $\Gamma'$ can be obtained by automata theoretic constructions. Specifically, given a deterministic finite-state transition system $B_1$ representing $\Upsilon$, it is straightforward to construct a deterministic finite-state transition system $B_1'$ representing $\Upsilon'$.

1.6.2. A logic for Hoare-style reasoning

In order to reason about programs using the storeless semantics described above, we choose to use Hoare-style reasoning. The literature contains a rich collection of logics for Hoare-style reasoning using both storeless and store-based representations of the heap. Notable among them are separation logic and its variants [25, 27, 34–36], logic of bunched implications [37], logic of reachable patterns (LRP) [21], several transitive closure logics [38], pointer assertion logic (PAL) [39] based on graph types [40], weak alias logic (wAL) [31], $L_r$ [41] and other assertion logics based on MSO [42]. While separation logic and its variants have arguably received the most attention in recent times, this development has primarily revolved around store-based semantics. Since our focus is on automata-based storeless semantics, we present below a simplified version of Bozga et al’s weak alias logic or wAL [31], called Simplified Alias Logic or SAL. Both wAL and SAL use storeless representations of the heap as structures for evaluating formulae. SAL however has fewer syntactic constructs than wAL. Implication checking in both logics is undecidable [31]. Nevertheless, decidable fragments with restricted expressiveness can be identified, and practically useful sound (but incomplete) inference systems can be defined. Other logics proposed for storeless representations of the heap are PAL [39], $L_r$ [41] and an assertion logic due to Jensen et al [42]. Unlike SAL, implication checking in these logics is decidable. While this is significant and can be very useful for analyzing certain classes of programs, these logics are less expressive than SAL. Our choice of SAL for this discussion is motivated by the need to express complex properties of the heap in a logic that is closed under the weakest precondition operator. Implication checking is addressed separately either by restricting the logic or by using sound (but incomplete) inference systems.

The logic SAL: The syntax and semantics of SAL (adapted from Bozga et al’s wAL [31]) are given in Tables 1.4a and b. Constants in this logic, shown in bold-face letters, are either $c_{\text{nil}}$ (denoting the language $S_{\text{nil}}$) or singleton languages consisting of access paths corresponding to pointer variables in the program. Variables are denoted by $X_i$ where $i$ is a natural number. Each such variable takes values from the set of regular languages in a storeless structure. Note that this differs from Bozga et al’s wAL, where free variables can be assigned arbitrary languages in $\Sigma^+$ that are neither restricted to be regular, nor to coincide with one of the languages in the storeless structure over which the formula is evaluated. Terms are formed by applying regular expression operators to variables, constants and sub-terms. Terms denote (possibly empty) subsets of $\Sigma_p^* \Sigma_f^*$. Formulae are constructed by applying
first-order operators to sub-formulae, where an atomic formula checks language equivalence of two terms \((T_1 = T_2)\). A storeless structure \(\mathcal{Y} = (S_{\text{nil}}, \Gamma)\) satisfies formula \(\varphi\) in this logic iff each free variable \(X_i\) in \(\varphi\) can be assigned a language \(S_i\) from \((S_{\text{nil}}) \cup \Gamma\) such that if every occurrence of \(X_i\) is replaced with \(S_i\), \(\varphi\) evaluates to \text{True}. Given a mapping \(\nu\) of free variables of \(\varphi\) to languages in \(\{S_{\text{nil}}\} \cup \Gamma\), we use \([T]_\nu\) to denote the regular language represented by term \(T\) after replacing each free variable \(X_i\) with \(\nu(X_i)\).

Table 1.4. Syntax and semantics of Simplified Alias Logic

(a) Syntax of \text{SAL}

<table>
<thead>
<tr>
<th>Notation</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Variables &amp; constants)</td>
<td>(V ::= X_i, i \in \mathbb{N} \mid c_{\text{nil}} \mid u, \text{ for all } u \in \Sigma_\nu)</td>
</tr>
<tr>
<td>(Selector name sequences)</td>
<td>(F ::= \emptyset \mid F \cdot F \mid (F + F) \mid F^*, \text{ for all } f \in \Sigma_f)</td>
</tr>
<tr>
<td>(Terms)</td>
<td>(T ::= V \mid T \cdot F \mid T \cdot (T^{-1} \cdot T) \mid T \cup T \mid T \cap T \mid T \ominus T)</td>
</tr>
<tr>
<td>(Formulae)</td>
<td>(\varphi ::= \neg \varphi \mid \exists X_i \varphi \mid \varphi \land \varphi \mid \varphi = \emptyset)</td>
</tr>
</tbody>
</table>

(b) Semantics of \text{SAL}

\[
\begin{align*}
[X_i]_\nu &= \nu(X_i), i \in \mathbb{N} & [F^*]_\nu &= ([F]_\nu)^* \\
[c_{\text{nil}}]_\nu &= S_{\text{nil}} & [T.F]_\nu &= [T]_\nu [F]_\nu \\
[\sigma]_\nu &= \{\sigma\}, \sigma \in \Sigma & [T_1 \cdot (T_2^{-1} \cdot T_3)]_\nu &= [T_1]_\nu \cdot ([T_2]_\nu)^{-1} \cdot [T_3]_\nu \\
[F.f]_\nu &= F \cdot f, f \in \Sigma_f & [T_1 \cup T_2]_\nu &= [T_1]_\nu \cup [T_2]_\nu \\
([F_1 + F_2])_\nu &= [F_1]_\nu \cup [F_2]_\nu & [T_1 \ominus T_2]_\nu &= [T_1]_\nu \ominus [T_2]_\nu \\
\end{align*}
\]

\[
\begin{align*}
\nu \models T_1 = T_2 & \text{ iff } [T_1]_\nu = [T_2]_\nu \\
\nu \models \varphi_1 \land \varphi_2 & \text{ iff } \nu \models \varphi_1 \text{ and } \nu \models \varphi_2 \\
\nu \models \neg \varphi & \text{ iff } \nu \not\models \varphi \\
\nu \models \exists X_i \varphi & \text{ iff } \exists S \in \{S_{\text{nil}}\} \cup \Gamma, \nu \models \varphi[S/X_i] \\
\end{align*}
\]

\(\mathcal{Y} \models \varphi\) iff there exists \(\nu : V(\varphi) \rightarrow \{S_{\text{nil}}\} \cup \Gamma\) such that \(\nu \models \varphi\)

It can be seen that \text{SAL} is expressive enough to describe complex properties of heaps, including some properties of recursive data structures. We list below a few examples to demonstrate the expressiveness of the logic. In each example, we first describe a heap property in English, and then present a shorthand along with a detailed formula in \text{SAL} that evaluates to \text{true} for a storeless structure iff the heap represented by the structure has the given property.

1. Term \(T\) represents a non-empty language: \(\text{nempty}(T) \equiv \neg (T = (T \ominus T))\). We will use \text{empty}(T) to denote \(\neg \text{nempty}(T)\).
2. \(X\) has an access path without any prefix in \(u \cdot g^*\): \(\text{nprefix}(X, u \cdot g^*) \equiv \text{nempty}(X \cap (\Sigma^p \cdot \Sigma^p) \cap u \cdot g^*)\).
3. \(X\) can be reached from \(Y\) using a sequence of \(f\) selectors: \(\text{rch}(X, Y, f) \equiv \text{nempty}(X \cap Y, f^*)\). Note that \(\text{rch}(X, X, f)\) is true for all \(X\) by definition.
(4) \(X\) and \(Y\) lie on a cycle in the heap graph using only \(f\) selectors: \(cyc(X, Y, f) \equiv rch(X, Y, f) \land rch(Y, X, f)\).

(5) \(X\) lies on a lasso or panhandle using only \(f\) selectors: \(lasso(X, f) \equiv \exists Y. (rch(X, Y, f) \land \exists Z. \neg(Z = Y) \land cyc(Y, Z))\).

(6) \(X\) is the root of a tree formed using \(f\) and \(g\) selectors: \(tree(X, f, g) \equiv \forall Y, Z. rch(X, Y, f + g) \land rch(X, Z, f + g) \land \neg rch(Y, Z, f + g) \land \neg rch(Z, Y, f + g) \Rightarrow \neg\exists W. (rch(Y, W, f + g) \land rch(Z, W, f + g))\).

Since terms in \(\text{SAL}\) are constructed by applying regular expression operators, \(\text{SAL}\) formulae cannot be used to express properties of the heap that involve unbounded counting. For example, the property that \(X\) is the root of a balanced binary tree formed using \(f\) and \(g\) selectors cannot be expressed in \(\text{SAL}\).

**Example 1.2.** As an example of how \(\text{SAL}\) can be used to specify properties of programs, let \(P\) be the program in Example 1.1, and let \(\varphi \equiv \text{rch}(c_{\text{nil}}, \text{hd}, f)\). The Hoare triple \(\{\varphi\} P \{\varphi\}\) expresses the property that if \(\text{hd}\) points to a \text{nil}-terminated acyclic list prior to execution of \(P\), and if \(P\) terminates, then \(\text{hd}\) must point to a \text{nil}-terminated acyclic list after termination of \(P\).

**Weakest precondition calculus:** In order to prove the validity of Hoare triples like the one above, we need to capture the operational semantics of statements using Hoare triples with \(\text{SAL}\) as the base logic. This involves computing weakest precondition [43] of formulae in \(\text{SAL}\) with respect to primitive statements in our language. Let \(wp(\text{Stmt}, \varphi)\) denote the weakest precondition of \(\varphi\) with respect to primitive statement \(\text{Stmt}\). It follows from the definition of \(wp\) that \(wp(\text{Stmt}, \varphi_1 \land \varphi_2) = wp(\text{Stmt}, \varphi_1) \land wp(\text{Stmt}, \varphi_2)\). Since the transition relation defined by primitive statements in our language is total (i.e. every state leads to at least one next state after executing a statement) and deterministic (i.e. every state leads to at most one next state after executing a statement), it can be further shown that \(wp(\text{Stmt}, \lnot \varphi) = \lnot wp(\text{Stmt}, \varphi)\) and \(wp(\text{Stmt}, \exists X. \varphi) = \exists X. wp(\text{Stmt}, \varphi)\). Consequently, the weakest precondition of an arbitrary formula \(\varphi\) can be constructed by induction on the structure of \(\varphi\), and we only need to define weakest preconditions of atomic formulae \((T_1 = T_2)\).

Let \(V\) denote the set of free variables of a formula \(\varphi\) in \(\text{SAL}\). For every \(\Omega \subseteq V\), let \(\langle \varphi \rangle_\Omega\) denote the conjunction \(\varphi \land \bigwedge_{X \in \Omega \setminus \Omega_1} (X = c_{\text{nil}}) \land \bigwedge_{Y \in \Omega} (\neg(Y = c_{\text{nil}}) \land \text{empty}(Y))\). Since every model of \(\varphi\) either sets a variable to \(S_{\text{nil}}\) or to an \(S_i \neq S_{\text{nil}}\), it follows that \(\varphi \Leftrightarrow \bigvee_{\Omega \subseteq V} \langle \varphi \rangle_\Omega\). Similarly, for every \(\Omega_2 \subseteq \Omega_1 \subseteq V\) and for every pointer expression \(\theta\) representing a valid access path, let \(\langle \varphi \rangle_{\Omega_1, \Omega_2, \theta}\) denote the conjunction \(\varphi \land \bigwedge_{X \in \Omega_2 \setminus \Omega_1} (X = c_{\text{nil}}) \land \bigwedge_{Y \in \Omega_1 \setminus \Omega_2} (\neg(Y = c_{\text{nil}}) \land (Y = \theta)) \land \bigwedge_{Z \in \Omega_2} (\neg(Z = c_{\text{nil}}) \land \neg(Z = \theta) \land \text{empty}(Z))\). By similar reasoning as above, it is easy to show that \(\text{empty}(\theta \cap c_{\text{nil}}) \Rightarrow (\varphi \Leftrightarrow \bigvee_{\Omega_2 \subseteq \Omega_1 \subseteq V} \langle \varphi \rangle_{\Omega_1, \Omega_2, \theta})\). Since we have already seen that the \(wp\) operator distributes over disjunction for our language and logic, it suffices to compute the weakest precondition of formulae of the form \(\langle \varphi \rangle_{\Omega_1, \Omega_2, \theta}\) if \(\Theta \cap S_{\text{nil}} = \emptyset\).
For notational convenience, let $\alpha$ denote a primitive statement, $X$ denote a variable or $c_{\text{nil}}$, and $\tilde{X}$ denote the term corresponding to $X$ after execution of $\alpha$, as given in Table 1.3, assuming that the object represented by $X$ before executing $\alpha$ is also represented in the storeless structure after executing $\alpha$. When $\alpha$ is clear from the context, we will omit $\alpha$ from the notation and simply use $\tilde{X}$. For example, if $\alpha$ denotes the statement $\theta := \text{nil}$, $\tilde{c_{\text{nil}}} = (c_{\text{nil}} \ominus \Theta) \cup \{\Theta\}$ and $\tilde{X} = X \ominus \Theta$ for variable $X$. Given an atomic formula $\varphi$ and a subset $\Omega$ of its free variables, we use $\varphi[\Omega \mapsto c_{\text{nil}}]$ and $\varphi[\Omega \mapsto \Theta]$ to denote the formula obtained from $\varphi$ by substituting every variable $X$ in $\Omega$ with $c_{\text{nil}}$ or $\Theta$ respectively. Similarly, we use $\varphi[\Omega \mapsto \tilde{X}]$ to denote the formula obtained by substituting every variable $X$ in $\Omega$ with $\tilde{X}$ and by substituting every occurrence of $c_{\text{nil}}$ with $\tilde{c_{\text{nil}}}$. Using this notation, Table 1.5a gives $wp(\alpha, \varphi)$ for various primitive statements $\alpha$ and atomic formula $\varphi \equiv (T_1 = T_2)$. Table 1.5b gives inference rules for looping, sequential composition and conditional branching constructs in our programming language, completing the set of Hoare inference rules for our simple language, with $\text{SAL}$ as the base logic.

**Decidability issues:** In order to prove a property of a program using Hoare-style reasoning, we must formulate the property as a Hoare triple using $\text{SAL}$ as the base logic, and then derive the triple by repeated applications of inference rules in Table 1.5b. Since Hoare logic is relatively complete, if the property holds for the program, then there exists a way to derive the triple by repeated applications of the inference rules, provided we have a way to check implications in $\text{SAL}$. Implication checking is needed in the rules for weakening pre-conditions and strengthening post-conditions, and also in the rule for looping constructs. Given formulas $\varphi$ and $\psi$ in $\text{SAL}$, $\varphi \Rightarrow \psi$ iff $\varphi \land \neg \psi$ is unsatisfiable. Therefore, it suffices to have a satisfiability checker for $\text{SAL}$. Unfortunately, satisfiability checking in $\text{SAL}$ is undecidable. The proof of undecidability follows a similar line of reasoning as used by Bozga et al [31] to show the undecidability of $\text{wAL}$. Various alternative strategies can, however, be adopted to check satisfiability for subclasses of formulae in practice. A simple strategy that is often used in practice is to work with a set of sound (but not complete) inference rules in an otherwise undecidable logic. Thus, if a formula can be shown to be (un)satisfiable using these rules, the formula is indeed (un)satisfiable. However, there is no guarantee that the satisfiability question for all formulae in the logic can be answered using the set of chosen rules. By carefully choosing the set of rules, it is often possible to use an undecidable logic like $\text{SAL}$ quite effectively for proving properties of several programs. Example 1.3 below shows an example of such rule schema for $\text{SAL}$. A second strategy is to use a decidable fragment of the logic. An example of this is the logic $\text{pAL}$ (propositional Alias Logic) [31], a strict subclass of Bozga et al’s $\text{wAL}$, but for which implication checking is in NP. A decidable fragment similar to $\text{pAL}$ can also be defined for $\text{SAL}$, although the ability to express properties of heaps is reduced (e.g., properties like $rch(X, Y, f)$ are not expressible in $\text{pAL}$). Finally, we can define a notion of bounded semantics, in which we only consider storeless structures with
Table 1.5. Weakest preconditions and Hoare inference rules for SAL

(a) Computing weakest preconditions for atomic formulae

<table>
<thead>
<tr>
<th>Statement (α)</th>
<th>wp(α, ϕ)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\theta := \text{nil}$</td>
<td>$\psi_1 \land V_{B \subseteq V}(\varphi)[V \setminus \Omega \rightarrow c_{\text{nil}}][\Omega \rightarrow \tilde{\Omega}]$, where $\psi_1 \equiv \psi_u$ if $\theta$ is $u \rightarrow n$, $\psi_1 \equiv \text{True}$ otherwise.</td>
</tr>
<tr>
<td>$u := \theta$</td>
<td>$V_{B \subseteq V}(\varphi)[V \setminus \Omega \rightarrow c_{\text{nil}}][\Omega \rightarrow \tilde{\Omega}]$</td>
</tr>
<tr>
<td>$u \rightarrow n := v$</td>
<td>$\psi_u \land \exists X. \text{nempty}(X \cap v) \land V_{B \subseteq V}(\varphi)[V \setminus \Omega \rightarrow c_{\text{nil}}][\Omega \rightarrow \tilde{\Omega}]$</td>
</tr>
<tr>
<td>$\text{free}(u)$</td>
<td>$\exists X. \text{nempty}(X \cap u) \land \text{empty}(X \cap c_{\text{nil}}) \land V_{B \subseteq V}(\varphi)[V \setminus \Omega \rightarrow c_{\text{nil}}][\Omega \rightarrow \tilde{\Omega}]$</td>
</tr>
</tbody>
</table>

(b) Hoare inference rules

Notation: $[B]$: SAL formula corresponding to Boolean expression $B$ in programming language

| $[u = v]$ | $\equiv \exists X. \text{nempty}(X \cap u) \land \text{nempty}(X \cap v)$ |
| $[\text{IsNil}(u)]$ | $\equiv \text{nempty}(c_{\text{nil}} \cap u)$ |
| $[B_1 \lor B_2]$ | $\equiv [B_1] \lor [B_2]$ |
| $[\text{not } B]$ | $\equiv \neg [B]$ |

Inference rules:

| $\{\varphi_1\} \text{ Stmt } \{\varphi_2\}$ | $\text{Sequential composition}$ |
| $\{\varphi_1\} \text{ Stmt  } \{\varphi_2\}$ | $\text{Strengthening precondition}$ |
| $\{\varphi_1\} \text{ Stmt  } \{\varphi_2\}$ | $\text{Weakening postcondition}$ |
| $\{\varphi_1 \land [B]\} \text{ Stmt  } \{\varphi_L\}$ | $\text{Conditional branch}$ |
| $\{\varphi_1\} \text{ if (B) then Stmt1 else Stmt2 } \{\varphi_2\}$ | $\text{Looping construct}$ |

at most $k$ languages, for a fixed (possibly large) $k$, to check for satisfiability. Since every storeless structure with $k$ languages can be represented by a deterministic finite-state transition system with $k + 2$ states, and since there are finitely many distinct transition systems with $k + 2$ states, it follows that satisfiability checking
in SAL with bounded semantics is decidable. Note, however, that if a SAL formula \( \varphi \) is found to be unsatisfiable using \( k \)-bounded semantics, it does not mean that the formula is unsatisfiable. Therefore, if a property is proved by applying Hoare inference rules and by using \( k \)-bounded semantics for SAL, then the program satisfies the property as long as the heap contains \( k \) or fewer distinct non-garbage objects. If, however, the heap grows to contain more than \( k \) distinct non-garbage objects, a property proved using \( k \)-bounded semantics is not guaranteed to hold.

**Example 1.3.** The Hoare triple in Example 1.2 can be proved using the rules in Table 1.5b, along with the inference rule schema for SAL in Table 1.6. The loop invariant used at location L2 of the program in Example 1.1 is \( \varphi_{L2} \equiv (t1 = c_{\text{nil}}) \lor (rch(t1, hd, n) \land rch(c_{\text{nil}}, t1, n)) \).

<table>
<thead>
<tr>
<th>Table 1.6. Sound inference rule schema for SAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{nempty}(X \cap Y \cdot F_1) \land \text{nempty}(Z \cap X \cdot F_2) )</td>
</tr>
<tr>
<td>( \text{nempty}(Z \cap Y \cdot F_1 \cdot F_2) )</td>
</tr>
<tr>
<td>( \text{nempty}(X) \lor \text{nempty}(X \cap X \cdot F^*) )</td>
</tr>
</tbody>
</table>

### 1.7. A counter automaton based technique

We have seen above two automata based techniques for reasoning about heap manipulating programs, namely regular model checking and Hoare-style reasoning using SAL that uses automata-based storeless representations of the heap as logical structures. In this section, we describe a third technique based on counter automata.

As in Section 1.5, we restrict our discussion to a class of heap manipulating programs in which each memory location has a single pointer-valued selector. We have seen in Section 1.5 that if we ignore garbage, the heap graph of such a program with \( n \) variables consists of at most \( 2^n \) uninterrupted list segments. This motivates abstracting such a heap graph by mapping each (unbounded) sequence of selector names in an uninterrupted list segment to an abstract sequence of some fixed size. Unfortunately, such an abstraction does not permit remembering the exact count of objects that actually existed in the list segment in the original heap graph. An interesting solution to this problem is to associate a counter with every such sequence in the heap graph, and use it to store the count of objects in the sequence. Bouajjani et al [24] have used this idea to define a *counter automaton* abstraction of the heap graph. More recently, Abdulla et al have proposed a technique using graph minors that achieves a similar abstraction [20].

Let \( X = \{x_1, \ldots, x_n\} \) be a set of counter variables and \( \Phi \) be the set of Presburger logic formulae with free variables from \( \{x_i, x'_i \mid x_i \in X\} \). A *counter automaton* with the set \( X \) of counter variables is a tuple \( A_c = (Q, X, \Delta) \), where \( Q \) is a finite
set of control states, and $\Delta \subseteq Q \times \Phi \times Q$ represents the transition relation. A configuration of the counter automaton is a tuple $(q, \beta)$, where $\beta : X \rightarrow \mathbb{N}$ assigns a natural number to each counter variable. The automaton is said to have a transition from $(q, \beta)$ to $(q', \beta')$ iff $(q, \varphi, q') \in \Delta$ for some Presburger formula $\varphi$ and the following conditions hold: (i) $\beta'(x_i) = \beta(x_i)$ for every $x_i'$ that is not free in $\varphi$, and (ii) if we substitute $\beta(x_i)$ for all free variables $x_i$ and $\beta'(x_i)$ for all free variables $x_i'$ in $\varphi$, then $\varphi$ evaluates to True. A run of $\mathcal{A}_c$ is a sequence of configurations $(q_0, \beta_0), (q_1, \beta_1), \ldots$ such that $\mathcal{A}_c$ has a transition from $(q_i, \beta_i)$ to $(q_{i+1}, \beta_{i+1})$ for every $i \geq 0$.

In order to construct a counter automaton abstraction of the heap graph, we first build a structural abstraction of the heap graph. This is done by defining an abstract structure and then establishing a mapping from nodes in the heap graph to nodes in the abstract structure, such that certain technical conditions are met [24]. These conditions ensure that two distinct nodes in the heap graph are mapped to the same node in the abstract structure only if they are not interruptions and belong to the same uninterrupted list segment. Intuitively, two nodes are mapped to the same node in the abstract if they are “internal” to the same uninterrupted list segment. For the class of programs under consideration, this abstraction is similar to the canonical abstraction of Reps et al [23], in which two nodes “internal” to the same list segment are abstracted into the same summary node. We also associate a counter variable with each node in the abstract structure to keep track of the actual count of objects in the heap graph that have been mapped to it. Furthermore, the abstract structure is constructed in a way such that for every sequence of two abstract nodes in the structure, one of the nodes is necessarily pointed to by a program variable, or has an indegree exceeding 1. Given this condition, it can be shown [24] that the number of abstract structures over a finite number of variables is always finite.

A counter automaton abstraction of the heap graph is obtained by letting the control states of the automaton be pairs of program location and structural abstraction of the heap graph. Counters associated with nodes in the structural abstraction become counters associated with the control state. Transitions of the counter automaton are guarded by Presburger logic formulae that capture the operational semantics of different primitive program statements. Bouajjani et al have shown [24] how such Presburger formula can be calculated for assignment, memory allocation and memory deallocation statements. A transition of the counter automaton therefore corresponds to the execution of a program statement. In general, this can lead to both a change in the counter values as well as change in the shape represented by the abstract structure. The change in counter values, however, allows us to track the lengths of different uninterrupted list segments precisely. Note that a counter automaton abstraction effectively maps a set of memory locations in the heap to a node in the abstract structure. The identity of a memory location is therefore the name of the node in the abstract structure to which it is mapped, and not the set
of paths to this node. In this sense, a counter automaton abstraction uses more of a store-based semantics than a storeless one.

A counter automaton abstraction of a data-insensitive program manipulating the heap can be shown to be bisimilar to the original program. Hence this abstraction preserves all temporal properties of such programs. It has been shown by Bouajjani et al that the counter automaton abstraction has additional properties that can be used to answer questions about the original program. Although the location reachability problem is undecidable in general for counter automata, these additional properties can be used to prove properties of special classes of programs. The reader is referred to [24] for a detailed exposition on this topic.

1.8. Conclusion

Analysis and formal verification of computer programs is a challenging task, especially for programs that manipulate unbounded structures in the heap. Automata theory provides a rich set of tools and techniques for reasoning about unbounded objects like words, trees, graphs etc. It is therefore not surprising that automata based techniques have attracted the attention of researchers in program analysis and verification. In this article, we surveyed three interesting techniques based on automata and logic for reasoning about programs manipulating the heap. This reading is intended to provide an introductory perspective about the use of automata theoretic techniques for analyzing heap manipulating programs. The serious reader is strongly encouraged to refer to the bibliography for further readings.

Acknowledgments

The author thanks the anonymous reviewer and Bhargav Gulavani for their critical comments.

References


Index

abstract regular tree model checking, 18
abstract structure, 28
abstraction
  configuration-oriented, 16
  representation-oriented, 15
access path, 19
bisimilar, 29
bounded semantics, 25
canonical abstraction, 28
concatenation, 3
configuration, 28
counter automaton, 27
counter automaton abstraction, 28
counterexample, 16, 17
decidable fragment, 25
dynamic linked data structures, 1
finite state automaton, 2
finite state transducer, 3, 10
finite state transition system, 2
garbage, 6, 8, 21
heap, 4
heap graph, 6
heap manipulating programs, 4
heap nodes
  heap-shared, 8, 9
  interruption, 8
Hoare inference rules, 25
Hoare-style reasoning, 18, 22
Implication checking, 22
inference rules
  schema, 27
  sound but not complete, 25
  Kleene-closure, 3
  left quotient, 3
may-alias analysis, 6
memory leak, 6
memory locations, 4
model checking, 7
must-alias analysis, 6
Myhill-Nerode theorem, 3
non-deterministic
  finite state automaton, 3
  guess, 11
  transition system, 2
operational semantics, 6
  small-step, 7
  storeless, 20
prefix closure, 19
Presburger logic, 27, 28
primitive statements, 20
product construction, 14
recursive data structures, 23
refinement, 16
regular model checking, 7
representation formalism, 7
right regularity, 20
safety checking, 15
satisfiability checking, 25
selectors, 4
semantics
  automata based, 18
  storeless, 18
  semantics:store-based, 18
  separation logic, 22
Index

shape analysis, 2, 6
Simplified Alias Logic (SAL), 22
stack, 4
statements
  assignment, 4
  conditional branch, 4
  loop, 4
  memory allocation, 4
  memory deallocation, 4
  sequential composition, 4
storeless structure, 19
structural abstraction, 28
temporal properties, 29
transition relation
  deterministic, 24
  total, 24
transitive closure
  regular transducer, 14
tree transducers, 18
unbounded information, 11
unbounded structures, 29
undecidability, 6, 25
uninterrupted list segment, 8, 9, 27
upper approximation
  extrapolation, 16
  quotienting, 15
  regular, 15, 16
  regular language inferencing, 17
  widening, 16
upper approximation:
  abstraction-refinement, 15
weakest precondition, 24
weakest precondition calculus, 24
well-quasi ordering, 18