Chapter 7

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 software 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 programs that dynamically allocate and free memory from the heap. Specifically, we discuss a regular model checking based approach, an approach based on storeless semantics of programs and Hoare-style reasoning, and a counter automaton based approach.

7.1. 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, converting input strings to tokens in lexical analyzers (used in compilers), formally verifying properties of programs, solving Presburger arithmetic constraints, machine learning and pattern recognition, among others. In this article, we focus on one such class of applications, and discuss how automata techniques can be used to formally reason about computer programs that dynamically allocate and free memory from the heap.

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 an overview of three important automata based techniques to address this problem. It is not meant to be an exhaustive overview of all automata-based techniques for reasoning about programs. Instead, we look at three different and interesting approaches, and explain them in some detail. To lend concreteness to the problem, 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 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 invest significant effort in studying special classes of programs and properties. The resulting advances in program analysis techniques have borrowed tools and techniques from different areas of computer science and mathematics. In this article, we restrict our discussion to a subset of these techniques that are based on automata theory. Specifically, we discuss the following three 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 problem might be solved in future.

The remainder of this article is organized as follows. Section 7.2 presents notation, definitions and some key automata-theoretic results that are used subsequently. Section 7.3 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 7.4 provides an overview of the challenges involved in reasoning about heap manipulating programs. Sections 7.5, 7.6 and 7.7 describe three automata based techniques for reasoning about programs manipulating heaps. Specifically, we discuss finite word based regular model checking in Section 7.5, and show how this can be used for shape analysis. Section 7.6 presents a regular language (automaton) based storeless semantics for our programming 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 7.7. Finally, section 7.8 concludes the article.

7.2. Automata notation and preliminaries

Let Σ be a finite alphabet and let Σ^* denote the set of all finite words on Σ. Note that Σ^* contains ε – the empty word of length 0. 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 δ 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 non-deterministic. A finite-state automaton $\mathcal{A}$ over Σ 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 (i.e., $|Q|$) of automaton $\mathcal{A}$.

The transition relation δ induces a relation $\widehat{\delta} \subseteq Q \times \Sigma^* \times Q$, defined inductively as follows: (i) for every $q \in Q$, $(q, \varepsilon, q) \in \widehat{\delta}$, and (ii) for every $q_1, q_2, q_3 \in Q$, $w \in \Sigma^*$ and $a \in \Sigma$, if $(q_1, w, q_2) \in \widehat{\delta}$ and $(q_2, a, q_3) \in \delta$, then $(q_1, w.a, q_3) \in \widehat{\delta}$, where "." denotes string concatenation. A word $w \in \Sigma^*$ is said to be *accepted* by the automaton $\mathcal{A}$ iff $(q, w, q') \in \widehat{\delta}$ for some $q \in Q_0$ and $q' \in F$. The set of all words accepted by $\mathcal{A}$ is called the language of $\mathcal{A}$, and is denoted $L(\mathcal{A})$. 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: $L^0 = \{\varepsilon\}$, 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 $L_1^{-1} L_2 = \{w \mid w \in \Sigma^* \text{and } \exists v \in L_1, \ v.w \in L_2\}$. If $L_1 = \emptyset$, we define $L_1^{-1} L_2 = \emptyset$ in all cases, including when $L_2 = \emptyset$.

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

(1) If $\mathcal{A}_1$ and $\mathcal{A}_2$ are finite-state automata on an alphabet Σ, there exist effective constructions of finite-state automata accepting each of $L(\mathcal{A}_1) \cup L(\mathcal{A}_2)$, $L(\mathcal{A}_1) \cap L(\mathcal{A}_2)$, $\Sigma^* \setminus L(\mathcal{A}_1)$, $L(\mathcal{A}_1) \cdot L(\mathcal{A}_2)$, $L^*(\mathcal{A}_1)$ and $L(\mathcal{A}_1)^{-1} L(\mathcal{A}_2)$.
(2) For every non-deterministic finite-state automaton $\mathcal{A}_1$, there exists a deterministic finite-state automaton $\mathcal{A}_2$ such that $L(\mathcal{A}_1) = L(\mathcal{A}_2)$ and $|\mathcal{A}_2| \leq 2^{|\mathcal{A}_1|}$.
(3) For every deterministic finite-state automaton $\mathcal{A}$, there exists a minimal deterministic finite-state automaton $\mathcal{A}_{min}$ that is unique up to isomorphism and has $L(\mathcal{A}) = L(\mathcal{A}_{min})$. Thus, any deterministic finite-state automaton accepting $L(\mathcal{A})$ must have at least as many states as $|\mathcal{A}_{min}|$.

Let L be a language, and let R_L be a binary relation on $\Sigma^* \times \Sigma^*$ defined as follows: $\forall x, y \in \Sigma^*, \ (x, y) \in R_L$ 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 Σ^* 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 Σ is a 5$-$tuple $\tau = (Q, \Sigma_\varepsilon \times \Sigma_\varepsilon, Q_0, \delta_\tau, F)$, where $\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$ and $\delta_\tau \subseteq Q \times \Sigma_\varepsilon \times \Sigma_\varepsilon \times Q$. Similar to the case of finite state automata, we define $\widehat{\delta_\tau} \subseteq Q \times \Sigma^* \times \Sigma^* \times Q$ as follows: (i) for every $q \in Q$, $(q, \varepsilon, \varepsilon, q) \in \widehat{\delta_\tau}$, and (ii) for every $q_1, q_2, q_3 \in Q$, $u, v \in \Sigma^*$ and $a, b \in \Sigma_\varepsilon$, if $(q_1, u, v, q_2) \in \widehat{\delta_\tau}$ and $(q_2, a, b, q_3) \in \delta_\tau$, then $(q_1, u.a, v.b, q_3) \in \widehat{\delta_\tau}$. The transducer τ defines a regular binary relation $R_\tau = \{(u, v) \mid u, v \in \Sigma^* \text{ and } \exists q \in Q_0, \exists q' \in F, \ (q, u, v, q') \in \widehat{\delta_\tau}\}$. For notational convenience, we will use τ for R_τ 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 Σ^*, we use $R_1 \circ R_2$ to denote the composed relation $\{(u, v) \mid u, v \in \Sigma^* \text{ and } \exists x \in \Sigma^*,\ ((u, x) \in R_1 \text{ and } (x, v) \in R_2)\}$. Let $id \subseteq \Sigma^* \times \Sigma^*$ denote the identity relation on Σ^*. 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 reasoning about heap manipulating programs using automata based techniques.

7.3. 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 de-allocation 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 with link structures between allocated 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, leaving 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 7.1. Syntax of our programming language

`PVar`	::=	`u` | `v` | ... (pointer-valued variables)
`FName`	::=	`n` | `f` | ... (pointer-valued selectors)
`PExp`	::=	`PVar` | `PVar->FName`
`BExp`	::=	`PVar = PVar` | `Pvar =` **nil** | **not** `BExp` | `BExp` **or** `BExp` | `BExp` **and** `BExp`
`Stmt`	::=	`AsgnStmt` | `CondStmt` | `LoopStmt` | `SeqCompStmt` | `AllocStmt` | `FreeStmt`
`AsgnStmt`	::=	`PExp := PVar` | `PVar := PExp` | `PExp :=` **nil**
`AllocStmt`	::=	`PVar :=` **new**
`FreeStmt`	::=	**free**`(PVar)`
`CondStmt`	::=	**if** `(BoolExp)` **then** `Stmt` **else** `Stmt`
`LoopStmt`	::=	**while** `(BoolExp)` **do** `Stmt`
`SeqCompStmt`	::=	`Stmt ; Stmt`

Table 7.1. Here, `PExp` represents a pointer expression obtained by concatenating at most one selector to a pointer-valued variable. `BExp` represents Boolean expressions on pointer variables, and are constructed using two basic predicates: the "`=`" predicate for checking equality of two pointer variables, and the "`=` **nil**" predicate for checking if a pointer variable has the **nil** value. `AllocStmt` represents a statement 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` represents a statement for de-allocating a previously allocated heap 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 k selectors can be effected by introducing $k - 1$ 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 := v->n`. 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->n; 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->n` 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->n` by `z := u->n; u := z` where `z` is a fresh temporary variable.

Example 7.1. The following program written in the above language searches a linked list pointed to by `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.

Table 7.2. A program manipulating a linked list

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

7.4. Challenges in reasoning about heap manipulating programs

Given a heap manipulating program such as the one in Example 7.1, there are several interesting questions that one might ask. For example, can the program de-reference a null pointer, leading to memory access error? Or, if `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 sound techniques that work well in practice for useful classes of programs and properties, but are conservative in general.

A common problem 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 Σ_p and Σ_f denote the set of variables and set of selectors respectively in P. We define the *heap graph* as a labeled directed graph $G_H = (V, E, v_{\mathrm{nil}}, \lambda, \mu)$, where V denotes the set of memory locations allocated by the program and always includes a special vertex v_{nil} to denote the **nil** value of pointers, $E \subseteq (V \setminus \{v_{\mathrm{nil}}\}) \times V$ denotes the link structure between memory locations, $\lambda : E \rightarrow 2^{\Sigma_f} \setminus \{\emptyset\}$ gives the labels of edges, and $\mu : \Sigma_p \hookrightarrow V$ 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 **nil** (if $v = v_{nil}$). Similarly, for every variable $x \in \Sigma_p$, $\mu(x) = v$ iff x points to memory location v, or to **nil** (if $v = v_{\mathrm{nil}}$).

Since a program may allocate an unbounded number of memory locations, the size of 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 further 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 for representing the heap affects the complexity of analysis, a careful balance must be struck between expressiveness of the formalism and 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.

7.5. Shape analysis using regular model checking

Model checking refers to a class of techniques for determining 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 obtained by defining a notion of system state, and by defining a transition relation between states to represent the small-step operational semantics of the system. In symbolic model checking [4], sets of states are represented symbolically, rather than explicitly. Regular model checking, henceforth called *RMC*, is a special kind 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 have been made in this area (see [7] for an excellent survey). While RMC is today used to refer to a spectrum of techniques that use finite/infinite words, trees or graphs to represent states, we will focus on finite word based representation of states in the present discussion. Specifically, the works of Jonsson, Nilsson, Abdulla, Bouajjani, Moro, Touilli, Habermehl, Vojnar and others [7–14] form the basis of our discussion on RMC.

If individual states are represented as finite words, a set of states can be represented as a language of finite words. Moreover, if the set is regular, it can be represented by a finite-state automaton. In the remainder of this section, we will refer to a state and its word-representation interchangeably. Similarly, we will refer to a set of states and its corresponding language representation interchangeably. The small-step operational semantics of the system is a binary relation that relates pairs of words representing the states before and after executing a statement. The state transition relation can therefore be viewed as a word transducer. For several classes of systems, including programs manipulating singly linked lists, the state

transition relation can indeed be modeled as a finite state transducer. Given a regular set I of words representing the initial states, and a finite state transducer τ, automata theoretic constructions can be used to obtain finite state representations of the sets (languages) $R_\tau^i(I)$, $i \geq 1$, where R_τ is the binary relation defined by τ and $R_\tau^i = \underbrace{R_\tau \circ (R_\tau \circ (\cdots (R_\tau \circ R_\tau) \cdots))}_{i}$. For notational convenience, we will use $\tau^i(I)$ to denote $R_\tau^i(I)$, when there is no confusion. The limit language $\tau^*(I)$, defined as $\bigcup_{i \geq 0} \tau^i(I)$, represents the set of all states reachable from some state in I in finitely many steps. Given a regular set Bad of undesired states, the problem of determining if some state in Bad can be reached from I therefore reduces to checking if the languages Bad and $\tau^*(I)$ have a non-empty intersection. Unfortunately, computing $\tau^*(I)$ is difficult in general, and $\tau^*(I)$ may not be regular even when both I and τ 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.

7.5.1. *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. The program in Example 7.1 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 this 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 chosen class) consists of singly linked lists, with possible sharing of elements and circularly linked structures. Figure 7.1 shows three examples of such heap graphs.

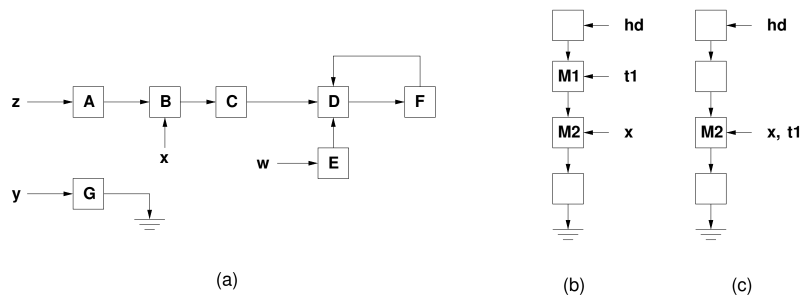

Fig. 7.1. Shared lists

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 7.1a 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 in the uninterrupted list segment is an interruption. As an example, the heap graph in Figure 7.1a has five uninterrupted list segments: $A \to B$, $B \to C \to D$, $D \to F \to D$, $E \to D$ and $G \to$ **nil**.

The above observation motivates us to represent a 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). Note that names are assigned only to heap-shared nodes and not to all nodes in the heap graph. Since the number of heap-shared nodes is bounded by the number of program variables, a finite number of names suffices for our purpose. Ranking names allows us to represent a set of names uniquely as a rank-ordered sequence of names. An uninterrupted list segment with r nodes, each having a single selector named n, can then be represented by listing the set of names (program variable names and/or heap-shared node name) corresponding to the first node in the list segment, followed by r copies of $.n$ (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 **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, if the selector in the heap graph shown in Figure 7.1a is named n, then this graph can be represented by the word $z.nB \,|\, xB.n.nD \,|\, D.n.nD \,|\, w.nD \,|\, y.n\bot$, 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 the 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. We will soon see details of how special modes of computation are used to reclaim unused names for heap-shared nodes.

In order to represent the state of a heap manipulating program, we need to keep track of some additional information beyond representing the heap graph. Given a program with k variables, let $\Sigma_M = \{M_0, M_1, M_2, \ldots M_k\}$ be a set of $k+1$ rank-ordered names for heap-shared nodes of the program, and let Σ_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 $\Sigma_C = \{C_N, C_0, C_1, C_2, \ldots C_k\}$, where k is the number of program variables. A value of C_N for the flag denotes normal mode of computation. A value of C_i $(0 \leq i \leq k)$ 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 **nil**, respectively. Using this convention, every program state can be represented as a finite word over the alphabet $\Sigma = \Sigma_C \cup \Sigma_L \cup \Sigma_M \cup \Sigma_p \cup \{\top, \bot, |, .n\}$, where Σ_p is the set of all program variables, and all selectors have the name n. We also restrict every heap-shared node in a program state to have exactly one name from the set Σ_M.

As an example, suppose Figure 7.1b represents the heap graph when the program in Example 7.1 is at location `L9` during the second iteration of the while loop, and suppose variables `t2` and `t3` 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 $\alpha = |\, C_N\, L9\, |\, M_0\, M_3\, M_4\, M_5\, |\, t2\, t3\, |\, |\, hd.nM_1\, |\, t1\, M_1.nM_2\, |\, x\, M_2.n.n\bot\, |$. The sub-word $t2\, t3$ of α encodes the fact that both `t2` and `t3` are uninitialized. Moreover, since there are no variables with the **nil** value, we have an empty list between a pair of consecutive separators (i.e., $||$) after $t2\, t3$. Similarly, if Figure 7.1c represents the heap graph when the program is at location `L10` in the second iteration of the while loop, the corresponding program state can be represented by the word $\alpha' = |\, C_N\, L10\, |\, M_0\, M_1\, M_3\, M_4\, M_5\, |\, t2\, t3\, |\, |\, hd.n.nM_2\, |\, x\, t1\, M_2.n.n\bot|$. Note that M_1 has been reclaimed as an unused name for heap-shared nodes in α', since the node named M_1 in Figure 7.1b is no longer heap-shared in Figure 7.1c.

7.5.2. *Operational semantics as word transducers*

Having seen how program states can be represented as finite words, we now discuss how operational semantics of program statements can be represented as non-deterministic finite state word transducers. Given a program in our language, we assume without loss of generality that each program location is associated with at most one primitive statement, i.e., `AsgnStmt`, `AllocStmt` or `FreeStmt` as described in Table 7.1. Each compound statement, i.e. `CondStmt`, `LoopStmt` or `SeqCompStmt`

as described in Table 7.1, is assumed to be split across multiple program locations to ensure that no program location is associated with more than one primitive statement. Specifically, for conditional statements, we assume that the first program location is associated with the partial statement "**if** (BoolExp) **then**", while for loop statements, the first program location is assumed to be associated with the partial statement "**while** (BoolExp) **do**". Example 7.1 illustrates how a program with multiple compound statements can be written in this manner. Given a program such that no program location is associated with more than one primitive statement, we construct a separate transducer for the statement (or part of it) 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. Finally, these individual transducers are non-deterministically combined to give an overall word transducer for the entire program. For notational convenience, we will henceforth refer to both the statement and part of a statement at a given program location as the statement at that location.

Given a word $w = |w_1|w_2|w_3|w_4|w_5|$ representing the current program state, it follows from the discussion in Section 7.5.1 that (i) the sub-word $|w_1|w_2|w_3|w_4|$ is bounded in length, and (ii) w_5 encodes a bounded number of uninterrupted list segments. Furthermore, each list segment encoded in w_5 has a bounded set of names for its first element, and either a name or $\top$ or $\bot$ as its last element. Therefore, the only source of unboundedness in w is the length of sequences of $.n$'s in the list segments represented 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 sequence of $.n$'s. Every transducer is also assumed to have two special "sink" states, denoted q_{mem} and q_{err}, with self looping transitions on all symbols of the alphabet. Of these, q_{mem} is an accepting state, while q_{err} is a non-accepting state. A transducer transitions to q_{mem} on reading an input word if it detects creation of garbage, or de-referencing of an uninitialized or **nil**-valued pointer. Such a transition is also accompanied by insertion of a special sequence of symbols, say $\top\top\top$, in the word representation of the next state. Note that the sequence $\top\top\top$ never appears in a word representation of a valid program state. Subsequently, whenever a transducer sees this special sequence in the word representation of the current state, it transitions to q_{mem} and retains the $\top\top\top$ sequence in the word representation of the next state. This ensures that the $\top\top\top$ sequence, one generated, survives repeated applications of the transducer, and manifests itself in the word representation of the final set of reached states. A transducer transitions to 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 our context. To see why this is so, suppose the current program statement is `t3 := x->n` (statement at location `L5` in Example 7.1) and the current program state is $w = |w_1|w_2|w_3|w_4|w_5|$. The transducer must read w and generate the word representation of the next program state, say $w' = |w'_1|w'_2|w'_3|w'_4|w'_5|$. Recall that w'_3 and w'_4 are required to be rank-ordered sequences of program variables names that are uninitialized and set to **nil**, respectively, in the next program state. In order to determine w'_3 and w'_4, the transducer must determine if program variable `t3` is set to an uninitialized value or to **nil** after execution of `t3 := x->n`. This requires knowledge of whether `x->n` is uninitialized or **nil** in the current program state. This information is encoded in sub-word w_5 that represents the uninterrupted list segments in the current program state. Therefore, the transducer must read w_5 before it can generate either w'_3 or w'_4. In addition, the transducer also needs to read w_5 in order to generate w'_5. This is because the uninterrupted list segments in the next program state (encoded by w'_5) are the same as those in the current program state (encoded by w_5), modulo changes effected by the current program statement. Since w'_5 appears to the right of w'_3 and w'_4 in w', it can be generated only after w'_3 and w'_4 have been generated. However, as seen above, generating w'_3 and w'_4 requires reading the whole of w_5 in general. Therefore, the transducer must remember w_5 as it reads w. Unfortunately, the uninterrupted list segments encoded in w_5 are unbounded in general, and a finite state transducer cannot remember unbounded information . One way to circumvent this problem is to have the transducer non-deterministically guess whether `x->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}.

Transducers for program statements: We now describe how transducers for statements at different program locations are constructed. Using the same notation as before, let $w = |w_1|w_2|w_3|w_4|w_5|$ be the input word read by the transducer and $w' = |w'_1|w'_2|w'_3|w'_4|w'_5|$ be the output word generated by it. We will see how each of the five components of w' are generated. Let us begin with w'_1. The transducer for the statement at location `Li` expects its input to begin with $|C_N L_i|$, i.e. w_1 should be $C_N L_i$. On seeing any input with a different prefix, the transducer simply transitions to q_{err}. Otherwise, the transducer non-deterministically chooses to enter mode C_j ($0 \leq j \leq k$) for reclaiming heap-shared node name M_j, or remain in mode C_N. In the former case, the transducer changes C_N to C_j and copies the rest of the input word w unchanged to its output. In the latter case, the transducer retains C_N as the first letter of w'_1. It then determines the next program location that would result after executing the statement at location `Li`. For several statements (e.g., all statements except those at `L2` and `L3` in Example 7.1), the next program location

can be statically determined, and the transducer replaces L_i with the corresponding next program location in w'_1. For other statements (e.g. those at `L2` and `L3` in Example 7.1), 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 word representation of 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 in w'_1, 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}.

Having generated w'_1, the transducer must next determine the set of unused names for heap-shared nodes in the next state, in order to generate w'_2. Recalling the constructs in our language (see Table 7.1) and the fact that each program location is associated with at most one primitive statement, it is easy to see that the number of heap-shared nodes in the heap graph can potentially increase only if the primitive statement associated with the current program location is an assignment of the form `PVar->FName := PVar` or `PVar := PVar->FName`. The execution of any other statement either keeps the number of heap-shared nodes unchanged or reduces it by one. For all these other statements, the transducer keeps the set of unused names for heap-shared nodes unchanged in the next state. Note that this may temporarily give rise to a situation wherein the number of heap-shared nodes has reduced by one, but the set of unused names for heap-shared nodes has not changed. Fortunately, this situation is easily remedied in the construction of the overall transducer, since the transducer for individual program statements is non-deterministically combined with transducers for reclaiming heap-shared node names in the overall transducer. Thus, if heap-shared node name M_l was rendered unused by execution of the statement at location L_i, the overall transducer can non-deterministically choose to transition to mode C_l (for reclaiming unused heap-shared node name M_l) after the transducer corresponding to program location L_i has completed its action. We will discuss further about transducers for reclaiming unused heap-shared node names later in this section.

If the statement at the current program location is of the form `PVar->FName := PVar` or `PVar := PVar->FName`, its execution gives rise to a (not necessarily new) heap-shared node unless the right hand side of the assignment evaluates to **nil** or is uninitialized. The statements at locations `L5`, `L6`, `L7` and `L8` in Example 7.1 are examples of such statements. In such cases, the transducer first guesses whether the right hand side of the assignment is **nil** or uninitialized, and remembers this guess in its finite memory. Accordingly, there are two cases to consider.

- If the right hand side is guessed to be **nil** or uninitialized, the number of heap-shared nodes cannot increase (but can potentially reduce by 1) as a result of executing the current statement. In this case, the transducer keeps the set of unused names for heap-shared nodes unchanged in the next state. The case

where the number of heap-shared nodes actually reduces by 1 but the set of unused names is unchanged is eventually taken care of by non-deterministically combining the current transducer with transducers for reclaiming unused names, as discussed above.

- If the right hand side is guessed to be neither **nil** nor uninitialized, execution of the current statement gives rise to a (not necessarily new) heap-shared node, say nd, in the heap graph. The transducer can now make one of two non-deterministic choices.
  - If there is at least one name in Σ_M that is outside the set of unused names in the current state (i.e., not in w_2), the transducer can guess that nd was already heap-shared earlier and was named M_l, where M_l is non-deterministically chosen from outside the set of unused names. The transducer then remembers M_l as the guessed name for the heap-shared node resulting from execution of the current statement. It also keeps the set of unused names for heap-shared nodes unchanged in the next state.
  - If the set of unused names in the current state, i.e. w_2, is non-empty, the transducer can guess that nd is a newly generated heap-shared node. The transducer then removes the first name, say M_l, from w_2, and remembers this as the name for the heap-shared node resulting from execution of the current statement. The set of unused names in the next state is obtained by removing M_l from the corresponding set in the current state.

Once the set of unused names for heap-shared nodes in the next state is determined, it is straightforward to generate w_2'. In all cases, as more of the input word w is read, if any guess made by the transducer is found to be incorrect, the transducer transitions to q_{err}.

In order to generate w_3' and w_4', the transducer must determine the sets of program variables that are uninitialized and set to **nil**, respectively, in the next program state. These sets are potentially changed when the current statement is of the form `PVar := PExp`, `PVar :=` **nil**, `PVar :=` **new** or **free**`(PVar)`. In all other cases, w_3' and w_4' are the same as w_3 and w_4, respectively. If the current statement is of the form `PVar := PVar`, `PVar :=` **nil** or `PVar :=` **new**, the corresponding updations to the sets of uninitialized and **nil**-valued program variables are straightforward, and w_3' and w_4' can be determined after reading w_3 and w_4. For de-allocation statements of the form **free**`(u)`, the sub-word w_4' (encoding the set of **nil**-valued variables) is the same as w_4. However, to determine w_3', we need to guess the set of variables that point to the same memory location as `u`, and are thereby rendered uninitialized by **free**`(u)`. The generation of w_3' in this case is explained later when we discuss generation of w_5'. Finally, if the current statement is of the form `PVar := PVar->FName`, sub-word w_5 of the input may need to be read and remembered, in general, before w_3' and w_4' can be generated. As discussed earlier, this leads to the problem of storing unbounded information in a finite state transducer. To circum-

vent this problem, the transducer makes a non-deterministic guess about whether the right hand side of the assignment evaluates to an uninitialized value, **nil** or the address of an allocated memory location. It then remembers this guess in its finite memory and generates w'_3 and w'_4 accordingly. By the time the entire input word w has been read, the transducer can determine whether any of its guesses was incorrect. If so, the transducer transitions to q_{err}.

Generating sub-word w'_5 requires determining how the uninterrupted list segments in the current program state (encoded in w_5) are modified by the current program statement. It is not hard to see that only primitive statements, i.e. assignment, memory allocation and de-allocation statements (`AsgnStmt`, `AllocStmt` and `FreeStmt` in Table 7.1), at the current program location can modify the encoding of the heap graph. For all other statements, the encoding of the heap graph in the next program state is the same as that in the current state; in other words, w'_5 is the same as w_5. Let us now look at what happens when each of `AsgnStmt`, `AllocStmt` and `FreeStmt` is executed.

Suppose the current statement is of the form `PVar := PVar`, as exemplified by `t1 := hd` at location `L1` in Example 7.1. In this case, w'_5 is obtained by removing `t1` from the head of any uninterrupted list segment in which it appears in w_5, and by inserting `t1` in the head of any uninterrupted list segment in which `hd` appears in w_5. Next, consider an assignment of the form `PVar := PVar->FName` or `PVar->FName := PVar`, as exemplified by `x->n := t1` and `t1 := t1->n` at locations `L7` and `L8`, respectively, in Example 7.1. Suppose the transducer has guessed that the right hand side of the assignment is neither **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 current statement. Let us also assume that all guesses made by the transducer thus far are correct. In the case of `x->n := t1`, sub-word w'_5 is obtained by inserting M_l in the head of the uninterrupted list segment in which `t1` appears in w_5, and by removing M_l from the head of any other list segment in w_5. In addition, the uninterrupted list segment starting from x is made to have only one element with its selector pointing to M_l in w'_5. Similarly, in the case of `t1 := t1->n`, we remove `t1` from the head of any uninterrupted list segment in which it appears in w_5, and putting both $t1$ and M_l at the head of the uninterrupted list segment in w'_5 that starts from the second element of the list originally pointed to by `t1` in w_5. If the current statement is of the form `PVar :=` **nil**, sub-word w'_5 is obtained by simply removing the variable name corresponding to `PVar` from the head of any uninterrupted list segment in which it appears in w_5. If, however, the current statement is of the form `PVar->FName :=` **nil**, say `u->n :=` **nil**, the uninterrupted list segment starting from `u` is made to have a sequence of only one $.n$ selector pointing to $\perp$ in w'_5. For statements of the form `PVar :=` **new**, as exemplified by `t2 := new` at location `L4` in Example 7.1, $t2$ is removed from the head of any uninterrupted list segment in which it appears in w_5, and a separate uninterrupted list segment, $t2.n\top$, is appended at the end of sub-word w'_5. For statements of the

form `free(u)`, the transducer first guesses all program variable names and heap-shared node name that appear together with `u` at the head of an uninterrupted list segment in w_5, and remembers this set in its finite memory. All program variable names in this set are removed from the head of the uninterrupted list segment in w_5, and added to the list of uninitialized variables, i.e. w'_3. All heap-shared node names in the above set are also removed from the head of the uninterrupted list segment in w_5, and all list segments that end with any such heap-shared node name are made to end with $\top$. As before, 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 de-allocation statement de-references a **nil**-valued or uninitialized pointer, the transducer transitions to the control state q_{mem}. 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 control state q_{mem}.

Transducers for reclaiming heap-shared node names: A transducer for reclaiming the heap-shared node name M_i $(0 \leq i \leq k)$ expects sub-word w_1 of its input 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 in the output word. If M_i is already in the set of unused names for heap-shared nodes, i.e. in w_2, the transducer simply changes C_i to C_N in sub-word w'_1 and leaves the rest of its input unchanged. If M_i is not in w_2, the transducer assumes that M_i is an unused heap-shared node name and 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 in sub-word w_5, or (ii) M_i appears as the sole name at the head of an uninterrupted list segment in w_5, and there is exactly one uninterrupted list segment in w_5 that has the name of its last node as M_i. The transducer non-deterministically chooses one of these cases and remembers its choice 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, i.e. to sub-word w'_2, changes the flag C_i to C_N in w'_1, and proceeds to replace all occurrences of M_i at the end of uninterrupted list segments in w_5 with $\top$. However, if it encounters an uninterrupted list segment in w_5 that has M_i at its head, the guess made by the transducer was incorrect, and hence it transitions to q_{err}. In the second case, let L_1 be the uninterrupted list segment starting with the sole name M_i in w_5, and let L_2 be the uninterrupted list segment ending with M_i in w_5. The transducer moves one element from the start of L_1 to the end of L_2 in w'_5, thereby shortening the list L_1 pointed to by M_i, and lengthening the list L_2 ending with M_i. It also non-deterministically guesses whether the list pointed to by M_i in w'_5 has shrunk to length zero, and if so, it adds M_i to the list of un-

used names of heap-shared nodes in w'_2, and replaces C_i by C_N in w'_1. Note that in this case, 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. Of course, if the transducer detects that any of its assumptions/guesses is incorrect, it transitions to control state q_{err}. Additionally, if the list L_1 ends in M_i, we have a garbage cycle and the transducer transitions to control state q_{mem}.

7.5.3. *Computing transitive closures of regular transducers*

Given a heap manipulating program P, let τ represent the overall finite state transducer obtained by non-deterministically combining all the finite state transducers discussed above, i.e. one for the statement (or part of it) at each program location, and one for reclaiming each heap-shared node name in Σ_M. By abuse of notation, we will use τ to also represent the binary relation, R_τ, on words induced by τ, when there is no confusion. Let A_I be a finite state automaton representing a regular set of initial states, say 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 over-approximations of it.

It is easy to use a product construction to obtain an automaton representing the set $\tau(I)$. Suppose $A_I = (Q_I, \Sigma_\varepsilon, \Delta_I, Q_{0,I}, F_I)$ and $\tau = (Q_\tau, \Sigma_\varepsilon \times \Sigma_\varepsilon, \Delta_\tau, Q_{0,\tau}, F_\tau)$. To construct an automaton recognizing $\tau(I)$, we first construct a product automaton $A_p = (Q_p, \Sigma_\varepsilon \times \Sigma_\varepsilon, \Delta_p, Q_{0,p}, F_p)$ 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 $\sigma_1, \sigma_2 \in \Sigma_\varepsilon$, $((q_1, q_2), (\sigma_1, \sigma_2), (q'_1, q'_2)) \in \Delta_p$ 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 $\tau^2(I) = \tau(\tau(I))$, we can use the same product construction, where an automaton recognizing $\tau(I)$ is first obtained as described above. Alternatively, we can precompute an automaton that induces the binary relation $\tau^2 = \tau \circ \tau$, and then determine $\tau^2(I)$. A non-deterministic finite state automaton that induces τ^2 can be obtained from the automaton that induces τ through a simple product construction. We construct $\tau^2 = (Q_{\tau^2}, \Sigma_\varepsilon \times \Sigma_\varepsilon, \Delta_{\tau^2}, Q_{0,\tau^2}, F_{\tau^2})$, where

- $Q_{\tau^2} = Q_\tau \times Q_\tau$
- For every $q_1, q_2, q'_1, q'_2 \in Q_\tau$, $\sigma_1, \sigma_2 \in \Sigma_\varepsilon$, $((q_1, q_2), (\sigma_1, \sigma_2), (q'_1, q'_2)) \in \Delta_{\tau^2}$ iff $\exists \sigma_3 \in \Sigma_\varepsilon$. $(q_1, (\sigma_1, \sigma_3), q'_1) \in \Delta_\tau$ 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 finite state automaton inducing τ^i for any given $i > 0$. Once a finite state automaton representation of τ^i is obtained, we can obtain a finite state automaton for $\tau^i(I)$, where I is a regular set of words, through the product construction illustrated above. However, this does not immediately tell us how to compute a finite state automaton representation of $\tau^* = \bigcup_{i=0}^{\infty} T^i$ or of $\tau^*(I)$. If τ is a regular transduction relation, the above constructions show that τ^i and $\tau^i(I)$ is also regular for every $i \geq 0$. However, τ^* and $\tau^*(I)$ may indeed be non-regular, since regular languages are not closed under infinite union. Even if τ^* or $\tau^*(I)$ was regular, a finite state automaton representation of it may not be effectively computable from finite state automata representations of τ and I. A central problem in regular model checking (RMC) concerns computing a regular upper approximation of τ^* or $\tau^*(I)$, for a given regular transduction relation τ and a regular initial set I.

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

Quotienting techniques: In this class of techniques, the product construction outlined above is used to compute finite state automata representations of τ^i for increasing values of i. A suitable equivalence relation $\simeq$ on the states of these automata is then defined based on the history of their creation during the product construction, and the quotient automaton 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 over-approximates τ^i for all i. For arbitrary equivalence relations, the language accepted by the resulting automaton is a superset of the language by τ^+. However, it is possible to classify equivalence relations such that certain classes of relations preserve transitive closure under quotienting. In other words, the language accepted by $(\tau/\simeq)^+$ *coincides* with that accepted by τ^+. The reader is referred to the works of Abdulla et al [9, 10] for details of these special relations. 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 $\simeq_r$ 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 τ (or $\tau(I)$) and compute τ^2 (or $\tau^2(I)$ respectively) as discussed above. The states of the automaton representation of τ^2 (or $\tau^2(I)$) are then quotiented with $\simeq_r$. The language accepted by the resulting automaton is, in general, a superset of that accepted by τ^2 (or $\tau^2(I)$ respectively). The quotiented automaton is then composed with τ to compute an over-approximation of τ^3 (or $\tau^3(I)$ respectively). The states of the resulting automaton are further quotiented with $\simeq_r$, and the process is repeated until a fixed point is reached. Since $\simeq_r$ 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 $\simeq_c$ 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 encode important information pertaining to current program location, **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 $.n$'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, for reasoning about heap manipulating programs [18].

Once we have a regular over-approximation of τ^* or $\tau^*(I)$, we can use it to conservatively check if $\tau^*(I) \cap Bad = \emptyset$. However, since we are working with an over-approximation of $\tau^*(I)$, safety checking may give false alarms. It is therefore necessary to construct a counterexample from an abstract sequence of states from a state in I to a state in Bad, and check if the counterexample is spurious. 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 $\simeq_r$ or $\simeq_c$ such that the same counterexample is not generated again 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 finite state automata representations of $\tau^i(I)$ for successive values of i, and detect a reg-

ular pattern in the sequence. The pattern is then *extrapolated* or *widened* to guess the limit ρ of $\bigcup_{i=0}^{m} \tau^i(I)$ as m approaches ∞. Convergence of the limit can be checked by determining if $I \cup \tau(\rho) \subseteq \rho$. If the check passes, we have computed an over-approximation of $\tau^*(I)$. Otherwise, the guessed limit must be modified to include more states (configurations), 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 k regular expressions $\rho_1.\rho_2.\cdots\rho_k$, and if $\tau(\rho_1.\rho_2\cdots\rho_k) = \bigcup_{i=1}^{k-1}(\rho_1\cdots\rho_i.\Lambda_i.\rho_{i+1}\cdots\rho_k)$, where the Λ_i are regular expressions, then $\tau^*(I)$ is given by $\rho_1.\Lambda_1^*.\rho_2.\Lambda_2^*\cdots\Lambda_{k-1}^*.\rho_k$.

Regular language inferencing techniques: This approach uses learning techniques, originally 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 increasingly large training sets are obtained by gradually increasing the maximum size i of words in the initial set of states, and by computing the set of all states (words) of size up to i reachable from these initial states. Habermehl et al use a variant of the Trakhtenbrot-Barzdin algorithm [19] 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 can be used to determine if an over-approximation of $\tau^*(I)$ has indeed been reached. It has been shown [14] that if $\tau^*(I)$ is regular, safety checking can always be correctly answered using this technique. Even otherwise, good regular upper approximations of $\tau^*(I)$ can be computed.

Let $UpperApprox(\tau^*(I))$ denote a regular upper approximation of $\tau^*(I)$ obtained using one of the above techniques. We can use $UpperApprox(\tau^*(I))$ 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 in the regular set I can create garbage, or cause an uninitialized/**nil**-valued pointer to be de-referenced. This can be done by searching for the special sequence $\top\top\top$ in the set of words approximating $\tau^*(I)$. Thus, if $BadMem = \Sigma^*\cdot\{\top\top\top\}\cdot\Sigma^*$, then $UpperApprox(\tau^*(I)) \cap BadMem = \emptyset$ guarantees that no garbage is created, and no uninitialized or **nil**-valued pointer is de-referenced. However, if $UpperApprox(\tau^*(I)) \cap BadMem \neq \emptyset$, we must construct an (abstract) counterexample leading from a state in I to a state in $BadMem$ and check for its spuriousness. If the counterexample is not spurious, we have a concrete way to generate garbage or to de-reference an uninitialized or **nil**-valued pointer starting from a state in I. Otherwise, we must refine or tighten $UpperApprox(\tau^*(I))$ and repeat the analysis.

Consider the program in Example 7.1. By carefully constructing the transducer τ as discussed earlier, and by applying a simple configuration-oriented abstraction technique, we can show that if $I = \{|C_N\, L_1 \mid M_0 M_1 M_2 M_3 M_4 M_5 \mid t1\, t2\, t3 \mid\cdot$

$|\, hd.n^+\bot \,|\,\} \;\cup\; \{|C_N\, L_1 \,|\, M_1M_2M_3M_4M_5 \,|\, t1\, t2\, t3 \,|\,|\, hd.n^+.M_0 \,|\, xM_0.n^+\bot \,|\,\}$, then $BadMem \cap UpperApprox(\tau^*(I)) = \emptyset$. Thus, regardless of whether the list pointed to by `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 [20] 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 [20] 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 [20] for a detailed exposition on this topic.

A primary drawback of RMC based approaches for reasoning about heap manipulating programs 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. Recently, Abdulla et al [21] 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 [21] 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 the present article. The interested reader is referred to [21] for details.

7.6. An automata based semantics and Hoare-style reasoning

We now present a completely different approach for reasoning about heap manipulating programs. Specifically, 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 7.1 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 [22], Podelski et

al [23], Reps et al [24], Bouajjani et al [25], Reynolds [26], Calcagno et al [27], Distefano et al [28] 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 [22, 26] or over formulae that use specially defined heap predicates [23, 24, 28], graph based representations [25], etc. In the alternative *storeless semantics*, originally proposed by Jonkers [29] and subsequently used by Deutsch [30], Bozga [31, 32], Hoare and Jifeng [33] and others, every memory location is identified with the set of paths that lead 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 for our programming language. As an aside, we note 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 [34] uses a novel mix of store based and storeless semantics in the framework of TVLA [24] to reason about the effect of procedure calls on data structures in the heap.

7.6.1. *A storeless semantics*

Given a program, let Σ_p and Σ_f denote sets of pointer variables and selectors respectively, as discussed earlier. Let $G_H = (V, E, v_{\mathrm{nil}}, \lambda, \mu)$ 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_{nil}) in V as a string $x.\sigma$, where σ 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 Σ denote $\Sigma_p \cup \Sigma_f$, and $\wp(S)$ denote the powerset of a set S. Adapting the definition of Bozga et al [31], we define a *storeless structure* Υ as a pair $(S_{\mathrm{nil}}, \Gamma)$, where $S_{\mathrm{nil}} \subseteq \Sigma_p{\cdot}\Sigma_f^*$ and $\Gamma \subseteq \wp(\Sigma_p{\cdot}\Sigma_f^*)$. Furthermore, Γ is either the empty set or a finite set of languages $\{S_1, S_2, \ldots S_n\}$ satisfying the following conditions for all $i, j \in \{1, \ldots n\}$.

- $C1 : S_i \neq \emptyset$.
- $C2 : i \neq j \Rightarrow S_i \cap S_j = \emptyset$. In addition, $S_i \cap S_{\mathrm{nil}} = \emptyset$.
- $C3 : \forall \sigma \in S_i\ (\forall \tau, \theta \in \Sigma^+\ (\sigma = \tau \cdot \theta\ \Rightarrow\ \exists k\ ((1 \leq k \leq n) \wedge (\tau \in S_k) \wedge (S_k{\cdot}\{\theta\} \subseteq S_i))))$. A similar property holds for all $\sigma \in S_{\mathrm{nil}}$ as well.

Unlike languages in Γ, there is no non-emptiness requirement on $S_{\rm nil}$. A storeless structure $\Upsilon = (S_{\rm nil}, \Gamma)$ with $\Gamma = \{S_1, \ldots S_n\}$ represents n distinct memory locations in the heap and also the **nil** value. Recall that in a heap graph, the **nil** value is represented by a special node $v_{\rm nil}$ with no outgoing edges. Language $S_i \in \Gamma$ may be viewed as the set of access paths in G_H to the ith node (distinct from v_{nil}). Similarly, $S_{\rm nil}$ may be viewed as the set of access paths to $v_{\rm nil}$. Condition $C1$ requires all nodes other than $v_{\rm nil}$ represented in Υ 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 $C2$ encodes the requirement that every access path must lead to at most one node. Condition $C3$ states that every prefix τ of an access path σ must itself be an access path to a node represented in Γ. 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 θ must lead us to the same node (represented by S_i or $S_{\rm nil}$ in condition $C3$). This is also called *right regularity.*

Consider a storeless structure $\Upsilon = (S_{\rm nil}, \Gamma)$, 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 Υ can be represented by an $n + 3$ state deterministic finite-state transition system B_Υ. A natural (but not necessarily the only) way to obtain B_Υ is by considering the sub-graph of G_H consisting of nodes $\{v_{\rm nil}, v_1, \ldots v_n\}$. Specifically, we define a transition system $B_\Upsilon = (Q, \Sigma, q_{init}, \delta)$, where $\Sigma = \Sigma_p \cup \Sigma_f$ as defined earlier, and $Q = \{q_{init}, q_{\rm nil}, q_{\rm err}, q_1, \ldots q_n\}$, with $q_{\rm init}, q_{\rm nil}$ and $q_{\rm err}$ as distinguished control states. For notational convenience, we will refer to $v_{\rm nil}$, $S_{\rm nil}$ and $q_{\rm nil}$ as v_0, S_0 and q_0 respectively in the following construction. The transition relation of B_Υ is defined as follows: for every i, j in 0 through n, we let $(q_i, f, q_j) \in \delta$ iff the nodes v_i and v_j in $G_H = (V, E, v_{\rm nil}, \lambda, \mu)$ are 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 $(q_{init}, x, q_i) \in \delta$ iff there exists v_i (represented by S_i in Γ) such that $\mu(x) = v_i$. Finally, for all states q (including $q_{\rm err}$) and for all $f \in \Sigma_p \cup \Sigma_f$, we let $(q, f, q_{\rm err}) \in \delta$ iff the above construction does not create any outgoing edge from q labeled f. Right regularity and prefix closure of Υ ensure that for all i in 0 through n, the automaton obtained by letting q_i be the sole accepting state in B_Υ accepts language S_i. The automaton $\mathcal{A}_{\rm err}$ obtained by letting $q_{\rm err}$ be the sole accepting state accepts all sequences that are *not* valid access paths to nodes represented by Υ.

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

- The representations of the heap before and after executing a statement are $\Upsilon = (S_{\rm nil}, \Gamma)$ and $\Upsilon' = (S'_{\rm nil}, \Gamma')$, respectively.
- If θ denotes a `PExp`, then $\mathbf{\Theta}$ denotes the singleton regular language consisting of the access path corresponding to θ. For example, if θ is `u` or `u->n`, then $\mathbf{\Theta}$ is $\{u\}$ or $\{u.n\}$, respectively. We will also use $\mathbf{u}$ to denote $\{u\}$ and $\mathbf{u} \cdot \mathbf{n}$ to denote $\{u.n\}$.

- For $L, M \subseteq \Sigma^+$, $L \ominus M$ denotes $L \setminus (M \cdot \Sigma^*)$, i.e. the set of words in L that do not have any prefix in M.
- For $L, X, \subseteq \Sigma^+$, $M \subseteq \Sigma^*$ and $\mathtt{n} \in \Sigma_f$, $\chi^{L,\mathtt{n},M}(X)$ denotes the function $\lambda X.\ X \cup (L \cdot \mathbf{n} \cdot ((M^{-1}L) \cdot \mathbf{n})^* \cdot (M^{-1}X))$. 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 $\chi^{L,\mathtt{n},M}(X)$ is the augmented set of access paths to v_X after making the n-selector of v_L point to v_M [31]. Note that if $M = \emptyset$, then $\chi^{L,\mathtt{n},M}(X) = X$ for all X.
- If θ denotes a `PExp` and $\mathcal{S} = \{S_1, S_2, \ldots S_r\}$ denotes a set of mutually disjoint languages, then $FindSet(\mathbf{\Theta}, \mathcal{S})$ is defined to be $S_i \in \mathcal{S}$ if $\mathbf{\Theta} \cap S_i \neq \emptyset$. If $\mathbf{\Theta} \cap S_i = \emptyset$ for all $S_i \in \mathcal{S}$, then $FindSet(\mathbf{\Theta}, \mathcal{S})$ is defined to be $\emptyset$.

With this convention, the storeless operational semantics of primitive statements in our language, i.e., assignment, memory allocation and memory de-allocation statements, is given in Table 7.3. To keep the discussion simple, we have assumed that statements of the form `u := u` or `u := u->n` are not present in programs that we wish to analyze. While this may sound restrictive, every program containing such statements can be translated to a semantically equivalent program without any such statement, as described in Section 7.3.

Table 7.3. Storeless operational semantics of primitive statements

Notation: θ denotes a `PExp`, `u` and `v` are program variables, `n` is a selector name

Statement	S'_{nil}	Γ'	Conditions
θ `:=` **nil**	$(S_{\mathrm{nil}} \ominus \mathbf{\Theta}) \cup \{\mathbf{\Theta}\}$	$\{S_i \ominus \mathbf{\Theta} \mid S_i \in \Gamma\} \setminus \{\emptyset\}$	if θ is `u->n`, $\exists X \in \Gamma,\ u \in X$
`u :=` θ	$S''_{\mathrm{nil}} \cup \mathbf{u} \cdot (\mathbf{\Theta}^{-1} S''_{\mathrm{nil}})$, where $S''_{\mathrm{nil}} = S_{\mathrm{nil}} \ominus \mathbf{u}$	$\{S''_j \cup \mathbf{u} \cdot (\mathbf{\Theta}^{-1} S''_j) \mid S''_j \in \Gamma''\}$, where $\Gamma'' = \{S_i \ominus \mathbf{u} \mid S_i \in \Gamma\} \setminus \{\emptyset\}$	if θ is `v->n`, $\exists X \in \Gamma,\ v \in X$
`u->n := v`	$\chi^{\mathbf{u},\mathtt{n},Y}(S_{\mathrm{nil}} \ominus (\mathbf{u} \cdot \mathbf{n}))$	$\{\chi^{\mathbf{u},\mathtt{n},Y}(S''_j) \mid S''_j \in \Gamma''\}$, where $\Gamma'' = \{S_i \ominus (\mathbf{u} \cdot \mathbf{n}) \mid S_i \in \Gamma\} \setminus \{\emptyset\}$	$Y = FindSet(\mathbf{v}, \{S_{\mathrm{nil}}\} \cup \Gamma)$ $\exists X \in \Gamma,\ u \in X$
`u :=` **new**	$S_{\mathrm{nil}} \ominus \mathbf{u}$	$(\{S_i \ominus \mathbf{u} \mid S_i \in \Gamma\} \cup \{\mathbf{u}\}) \setminus \{\emptyset\}$	
free`(u)`	$S_{\mathrm{nil}} \ominus X$	$\{S_i \ominus X \mid S_i \in \Gamma\} \setminus \{\emptyset\}$	$\exists X \in \Gamma,\ u \in X$

The last column in Table 7.3 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. Whenever an assignment is made to `u` (or `u->n`), the membership in S_{nil} or $S_i \in \Gamma$ of all access paths that have u (or $u.n$ respectively) as prefix is invalidated. Therefore, these paths must be removed from all languages in Υ 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 `Stmt`.

(1) If $\Upsilon = (S_{\text{nil}}, \Gamma)$ is a storeless structure (i.e., satisfies all conditions in the definition of storeless structures), then so is $\Upsilon' = (S^{'}_{\text{nil}}, \Gamma')$.
(2) Let $v_i \neq v_{\text{nil}}$ be a node represented by Υ.
   (a) If v_i is neither de-allocated nor rendered garbage by executing `Stmt`, there exists a language in Γ' that contains all and only access paths to v_i after executing `Stmt`.
   (b) If executing `Stmt` de-allocates v_i and if π was an access path to v_i prior to executing `Stmt`, there is no access path with prefix π in any language in Υ'.
(3) $S^{'}_{\text{nil}}$ contains all and only access paths to v_{nil} after executing `Stmt`.
(4) If executing `Stmt` allocates a node v_i', there exists a language in Γ' that contains all and only access paths to v_i' after executing `Stmt`.

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

It is clear from the expressions for S'_{nil} and Γ' in Table 7.3 that if we are given finite state automata representations of S_{nil} and $S_i \in \Gamma$ for $i \in \{1, \ldots n\}$, then finite-state automata representations of $S^{'}_{\text{nil}}$ and also of every language in Γ' can be obtained by automata theoretic constructions. Specifically, given a deterministic finite-state transition system B_Υ representing Υ, it is possible to construct a deterministic finite-state transition system B'_Υ representing Υ'.

7.6.2. *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 [26, 28, 35–37], logic of bunched implications [38], logic of reachable patterns (LRP) [22], several transitive closure logics [39], pointer assertion logic (PAL) [40] based on graph types [41], weak alias logic (wAL) [32], L_r [42] and other assertion logics based on monadic second order logic [43]. 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 [32]. We call this *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 [32]. 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 [40], L_r [42] and an assertion logic due to Jensen et al [43]. Unlike SAL, im-

plication checking in these logics is decidable. While this represents a significant difference and can be very useful for analyzing certain classes of programs, these logics are less expressive than SAL. Our choice of SAL for the current discussion is motivated by the need to express complex heap properties in a logic that is closed under the weakest pre-condition 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 [32]) are given in Tables 7.4a and b. Constants in this logic, shown in bold-

Table 7.4. Syntax and semantics of Simplified Alias Logic

(a) Syntax of SAL

(Variables & constants)	VC	::=	$X_i, i \in \mathbb{N} \mid \mathbf{c}_{\text{nil}} \mid \mathbf{u}$, for all $\mathtt{u} \in \Sigma_p$
(Selector name sequences)	F	::=	$\mathbf{f} \mid \text{F}.\mathbf{f} \mid (\text{F} + \text{F}) \mid \text{F}^*$, for all $\mathtt{f} \in \Sigma_f$
(Terms)	T	::=	$\text{VC} \mid \text{T} \cdot \text{F} \mid \text{T} \cdot (\text{T}^{-1}\ \text{T}) \mid \text{T} \cup \text{T} \mid \text{T} \cap \text{T} \mid \text{T} \ominus \text{T}$
(Formulae)	φ	::=	$\text{T} = \text{T} \mid \varphi \wedge \varphi \mid \neg\varphi \mid \exists X_i\ \varphi$

(b) Semantics of SAL

Notation:
$V(\varphi)$: Free variables in φ, $\Upsilon = (S_{\text{nil}}, \Gamma)$: A storeless structure, $\nu : V(\varphi) \to \{S_{\text{nil}}\} \cup \Gamma$

			$[\![F^*]\!]_\nu$	=	$([\![F]\!]_\nu)^*$
$[\![X_i]\!]_\nu$	=	$\nu(X_i)$, $i \in \mathbb{N}$	$[\![T.F]\!]_\nu$	=	$[\![T]\!]_\nu \cdot [\![F]\!]_\nu$
$[\![\mathbf{c}_{\text{nil}}]\!]_\nu$	=	S_{nil}	$[\![T_1.(T_2^{-1}T_3)]\!]_\nu$	=	$[\![T_1]\!]_\nu \cdot ([\![T_2]\!]_\nu^{-1} [\![T_3]\!]_\nu)$
$[\![\mathbf{u}]\!]_\nu$	=	$\{u\}$, $\mathtt{u} \in \Sigma_p$	$[\![T_1 \cup T_2]\!]_\nu$	=	$[\![T_1]\!]_\nu \cup [\![T_2]\!]_\nu$
$[\![F.\mathbf{f}]\!]_\nu$	=	$[\![F]\!]_\nu \cdot \{f\}$, $\mathtt{f} \in \Sigma_f$	$[\![T_1 \cap T_2]\!]_\nu$	=	$[\![T_1]\!]_\nu \cap [\![T_2]\!]_\nu$
$[\![(F_1 + F_2)]\!]_\nu$	=	$[\![F_1]\!]_\nu \cup [\![F_2]\!]_\nu$	$[\![T_1 \ominus T_2]\!]_\nu$	=	$[\![T_1]\!]_\nu \setminus ([\![T_2]\!]_\nu \cdot \Sigma^*)$,

$\nu \models T_1 = T_2$	iff	$[\![T_1]\!]_\nu = [\![T_2]\!]_\nu$
$\nu \models \varphi_1 \wedge \varphi_2$	iff	$\nu \models \varphi_1$ and $\nu \models \varphi_2$
$\nu \models \neg\varphi$	iff	$\nu \not\models \varphi$
$\nu \models \exists X_i\ \varphi$	iff	$\exists S \in \{S_{\text{nil}}\} \cup \Gamma,\ \nu \models \varphi[S/X_i]$

We say that $\Upsilon \models \varphi$ iff there exists $\nu : V(\varphi) \to \{S_{\text{nil}}\} \cup \Gamma$ such that $\nu \models \varphi$

face, are either $\mathbf{c}_{\text{nil}}$ (denoting the language S_{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 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 Σ^+ that are neither restricted to be regular, nor required 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 $\Sigma_p \cdot \Sigma_f^*$. Formulae are constructed by applying first-order operators to sub-formulae, where an atomic formula checks language

equivalence of two terms ($T_1 = T_2$). We will use the usual shorthand notations $\forall X_i\ \varphi$ and $\varphi_1 \vee \varphi_2$ for $\neg\,(\exists X_i\ \neg\varphi)$ and $\neg\,(\neg\varphi_1 \wedge \neg\varphi_2)$ respectively, whenever necessary. Given a storeless structure $\Upsilon = (S_{\text{nil}}, \Gamma)$ and an assignment ν that assigns a language from $\{S_{\text{nil}}\} \cup \Gamma$ to each free variable of φ, we use $[\![T]\!]_\nu$ to denote the regular language obtained by replacing every occurrence of every free variable X in term T with $\nu(X)$. We say that Υ is a model of φ and ν is a satisfying assignment for φ in Υ iff $\nu \models \varphi$, as defined in Table 7.4b. If $\nu \models \varphi$, we will also say that the assignment ν renders φ true.

It can be seen that 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 SAL that evaluates to true for a storeless structure iff the heap represented by the structure has the given property.

(1) Term T represents a non-empty set of access paths to a node (possibly v_{nil}) in the heap graph: $\mathit{nempty}(T) \equiv \neg(T = (T \ominus T))$. We will use $\mathit{empty}(T)$ to denote $\neg \mathit{nempty}(T)$.
(2) X has an access path without any prefix in $\mathbf{u}{\cdot}\mathbf{f}^*$: $\mathit{nprefix}(X, \mathbf{u}{\cdot}\mathbf{f}^*) \equiv \mathit{nempty}(X \cap (\Sigma_p{\cdot}\Sigma_f^* \ominus \mathbf{u}{\cdot}\mathbf{f}^*))$.
(3) X can be reached from Y using a sequence of `f` selectors: $\mathit{rch}(X, Y, \mathbf{f}) \equiv \mathit{nempty}(X \cap Y.\mathbf{f}^*)$. Note that $\mathit{rch}(X, X, \mathbf{f})$ is true for all non-empty X by definition.
(4) X can be reached from Y using exactly one `f` selector: $\mathit{edge}(X, Y, \mathbf{f}) \equiv \mathit{nempty}(X \cap Y.\mathbf{f})$.
(5) X and Y lie on a cycle in the heap graph formed using only `f` selectors: $\mathit{cyc}(X, Y, \mathbf{f}) \equiv \neg(X = Y) \wedge \mathit{rch}(X, Y, \mathbf{f}) \wedge \mathit{rch}(Y, X, \mathbf{f})$.
(6) X lies on a lasso or panhandle formed using only `f` selectors: $\mathit{lasso}(X, \mathbf{f}) \equiv \exists Y\ (\mathit{rch}(Y, X, \mathbf{f}) \wedge \neg \mathit{rch}(X, Y, \mathbf{f}) \wedge \exists Z\ (\neg(Z = Y) \wedge \mathit{cyc}(Y, Z, \mathbf{f})))$.
(7) X is the root of a tree formed using `f` selectors: $\mathit{tree}(X, \mathbf{f}) \equiv \forall Y \forall Z \forall V\ ((\mathit{rch}(Y, X, \mathbf{f}) \wedge \mathit{rch}(Z, X, \mathbf{f}) \wedge \mathit{edge}(V, Y, \mathbf{f}) \wedge \mathit{edge}(V, Z, \mathbf{f})) \Rightarrow (Y = Z))$.

Since terms in SAL are formed by applying regular expression operators, SAL formulae cannot be used to express non-regular properties of the heap, such as those involving unbounded counting. For example, we cannot express "X is the root of a balanced binary tree formed using `f` selectors" in SAL.

Example 7.2. As an illustration of how SAL can be used to specify properties of programs, let P be the program in Example 7.1 and let $\varphi \equiv \mathit{rch}(\mathbf{c}_{\text{nil}}, \mathbf{hd}, \mathbf{f})$. The Hoare triple $\{\varphi\}\ P\ \{\varphi\}$ asserts that if program P is started in a state in which `hd` points to a **nil**-terminated acyclic list, and if P terminates, then `hd` always points to a **nil**-terminated acyclic list after termination of P as well.

Weakest pre-condition calculus: In order to prove the validity of Hoare triples like the one in Example 7.2, we must express the operational semantics of statements using Hoare triples with SAL as the base logic. This involves computing *weakest pre-conditions* [44] of formulae in SAL with respect to primitive statements (i.e., assignment, memory allocation and de-allocation statements) in our language. Let $wp(\mathsf{Stmt}, \varphi)$ denote the weakest pre-condition of φ with respect to primitive statement Stmt. It follows from the definition of wp that $wp(\mathsf{Stmt}, \varphi_1 \wedge \varphi_2) = wp(\mathsf{Stmt}, \varphi_1) \wedge wp(\mathsf{Stmt}, \varphi_2)$. 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 $wp(\mathsf{Stmt}, \neg\varphi) = \neg wp(\mathsf{Stmt}, \varphi)$ and $wp(\mathsf{Stmt}, \exists X\, \varphi) = \exists X\, wp(\mathsf{Stmt}, \varphi)$. Consequently, the weakest pre-condition of an arbitrary SAL formula φ with respect to a primitive statement can be computed by induction on the structure of φ, and we only need to define weakest pre-conditions of atomic formulae of the form $(T_1 = T_2)$.

Let $V(\varphi)$ denote the set of free variables of a formula φ. For brevity of notation, we will use V for $V(\varphi)$ when φ is clear from the context. For every $\Omega \subseteq V$, let $\langle\varphi\rangle_\Omega$ denote the conjunction $\varphi \wedge \bigwedge_{X \in V \setminus \Omega}(X = \mathbf{c}_{\mathrm{nil}}) \wedge \bigwedge_{Y \in \Omega}(\neg(Y = \mathbf{c}_{\mathrm{nil}}) \wedge \mathit{nempty}(Y))$. Since every satisfying assignment of φ in every model Υ $(= (S_{\mathrm{nil}}, \Gamma))$ sets a (possibly empty) subset of the free variables of φ to S_{nil} and the remaining free variables to languages in Γ, 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 θ representing a valid access path, let $\langle\varphi\rangle_{\Omega_1,\Omega_2,\Theta}$ denote the conjunction $\varphi \wedge \bigwedge_{X \in V \setminus \Omega_1}(X = \mathbf{c}_{\mathrm{nil}}) \wedge \bigwedge_{Y \in \Omega_1 \setminus \Omega_2}(\neg(Y = \mathbf{c}_{\mathrm{nil}}) \wedge (Y = \mathbf{\Theta})) \wedge \bigwedge_{Z \in \Omega_2}(\neg(Z = \mathbf{c}_{\mathrm{nil}}) \wedge \neg(Z = \mathbf{\Theta}) \wedge \mathit{nempty}(Z))$. By reasoning similar to that above, it can also be shown that $\mathit{empty}(\mathbf{\Theta} \cap \mathbf{c}_{\mathrm{nil}}) \Rightarrow (\varphi \Leftrightarrow \bigvee_{\Omega_2 \subseteq \Omega_1 \subseteq V}\langle\varphi\rangle_{\Omega_1,\Omega_2,\mathbf{\Theta}})$.

Let α be a primitive statement in a program written in our language, and let $\Upsilon = (S_{\mathrm{nil}}, \Gamma)$ be a storeless structure representing the program state before execution of α. Furthermore, let φ be a SAL formula such that $\Upsilon \models \varphi$. By definition, there exists an assignment of free variables, say $\nu : V(\varphi) \to \{S_{\mathrm{nil}}\} \cup \Gamma$, such that $\nu \models \varphi$. For $X \in V(\varphi)$, suppose $\nu(X) = S_i \neq S_{\mathrm{nil}}$. Then X represents the set of access paths to a node, say nd_i, distinct from v_{nil}, in the heap graph prior to execution of α. Suppose nd_i is neither de-allocated nor rendered garbage by executing statement α from the program state Υ. One can then ask: Can the set of access paths to nd_i in the heap graph resulting after execution of α be expressed in terms of X? The operational semantics given in Table 7.3 tells us that this can indeed be done. We will use $\widetilde{X}^\alpha$ to denote the term representing the (potentially new) set of access paths to nd_i after execution of α, where X represented the set of access paths to nd_i before execution of α. Similarly, we will use $\widetilde{\mathbf{c}_{\mathrm{nil}}}^\alpha$ to denote the term representing the set of access paths to v_{nil} after execution of α, where $\mathbf{c}_{\mathrm{nil}}$ represented the set of access paths to v_{nil} prior to execution of α. As an example, if α denotes the statement θ **:= nil**, the first row of Table 7.3 tells us that $\widetilde{\mathbf{c}_{\mathrm{nil}}}^\alpha = (\mathbf{c}_{\mathrm{nil}} \ominus \mathbf{\Theta}) \cup \{\mathbf{\Theta}\}$ and $\widetilde{X}^\alpha = X \ominus \mathbf{\Theta}$ for variable $X \neq \mathbf{c}_{\mathrm{nil}}$. For notational clarity, we will henceforth

use $\widetilde{X}$ instead of $\widetilde{X}^\alpha$ when α is clear from the context. Given a SAL formula φ and a subset Ω of $V(\varphi)$, we will use $\varphi[\Omega \mapsto \mathbf{c}_{\text{nil}}]$ and $\varphi[\Omega \mapsto \mathbf{\Theta}]$ to denote the formulae obtained from φ by substituting every variable X in Ω with $\mathbf{c}_{\text{nil}}$ and $\mathbf{\Theta}$ respectively. Similarly, we will use $\varphi[\Omega \mapsto \widetilde{\Omega}]$ to denote the formula obtained by substituting every variable X in Ω with $\widetilde{X}$ *and* by substituting every occurrence of $\mathbf{c}_{\text{nil}}$ with $\widetilde{\mathbf{c}_{\text{nil}}}$. Extending the notation, if Ω_1 and Ω_2 are subsets of $V(\varphi)$, we will use $\varphi[\Omega_1 \mapsto \mathbf{c}_{\text{nil}}][\Omega_2 \mapsto \widetilde{\Omega_2}]$ to denote $(\varphi[\Omega_1 \mapsto \mathbf{c}_{\text{nil}}])[\Omega_2 \mapsto \widetilde{\Omega_2}]$. The interpretation of $\varphi[\Omega_1 \mapsto \mathbf{c}_{\text{nil}}][\Omega_2 \mapsto \mathbf{\Theta}][\Omega_3 \mapsto \widetilde{\Omega_3}]$ is similar.

The intuition behind the computation of $wp(\alpha, \varphi)$ can now be explained as follows. As before, let $\Upsilon = (S_{\text{nil}}, \Gamma)$ be a storeless structure representing the program state before execution of α. Let $\Upsilon' = (S'_{\text{nil}}, \Gamma')$ be the corresponding storeless structure after execution of α. Suppose $\Upsilon' \models \varphi$ and $\nu' : V(\varphi) \to \{S'_{\text{nil}}\} \cup \Gamma'$ is a satisfying assignment for φ in Υ'. Let $\Omega \subseteq V(\varphi)$ be the subset of free variables that are assigned languages in Γ' (and not S'_{nil}) by ν', i.e. $\nu'(X) \in \Gamma'$ for all $X \in \Omega$ and $\nu'(Y) = S'_{\text{nil}}$ for all $Y \in V \setminus \Omega$. It follows that $\Upsilon' \models \widehat{\varphi}$, where $\widehat{\varphi} \equiv \varphi[V \setminus \Omega \mapsto \mathbf{c}_{\text{nil}}]$. This is because the assignment $\widehat{\nu} : \Omega \to \Gamma'$, given by $\widehat{\nu}(X) = \nu(X)$ for every $X \in \Omega$, causes $\widehat{\varphi}$ to evaluate to the same truth value that φ evaluates to under the assignment ν', i.e. true. If the execution of α does not allocate any new memory location, every node in the heap graph after execution of α was also present in the heap graph before execution of α. From Table 7.3, we also know how the representations of the sets of access paths to these nodes and to v_{nil} in Υ change to their corresponding representations in Υ', as a result of executing α. It therefore follows that $\Upsilon \models \widehat{\varphi}[\Omega \mapsto \widetilde{\Omega}]$. The corresponding satisfying assignment $\nu : \Omega \mapsto \Gamma$ is such that $\nu(X) = S_i \in \Gamma$ iff $\nu'(X) = S'_i \in \Gamma'$, where S_i and S'_i represent sets of access paths to the same node in the heap graph before and after executing α respectively. Now suppose the formula φ is such that *for all* models $\Upsilon' = (S'_{\text{nil}}, \Gamma')$, every satisfying assignment ν' of φ in Υ' sets all free variables in a subset Ω of $V(\varphi)$ to languages in Γ' and all other free variables to S'_{nil}. We will call such formulae *model-constraining with respect to* Ω. It is easy to see that if φ is model-constraining with respect to Ω, then $wp(\alpha, \varphi)$ is essentially given by $\widehat{\varphi}[\Omega \mapsto \widetilde{\Omega}]$. In reality, $wp(\alpha, \varphi) \equiv \zeta \wedge \widehat{\varphi}[\Omega \mapsto \widetilde{\Omega}]$, where ζ is a SAL formula that asserts conditions to ensure that the execution of α doesn't lead to a memory error. In other words, ζ encodes the conditions listed in Table 7.3 for the operational semantics of primitive statements to be defined. Unfortunately, a general SAL formula φ may not be model-constraining with respect to any $\Omega \subseteq V(\varphi)$. In order to circumvent this problem, we express φ in the equivalent form $\bigvee_{\Omega \subseteq V} \langle \varphi \rangle_\Omega$. Note that for every Ω, the formula $\langle \varphi \rangle_\Omega$ is model-constraining with respect to Ω. Since the wp operator distributes over negation and conjunction (and hence, over disjunction) of formulae with respect to primitive statements in our language, it is now easy to see that $wp(\alpha, \varphi) \equiv \zeta \wedge \bigvee_{\Omega \subseteq V} \left(\langle \varphi \rangle_\Omega [V \setminus \Omega \mapsto \mathbf{c}_{\text{nil}}][\Omega \mapsto \widetilde{\Omega}] \right)$.

The above discussion assumed that the statement α does not allocate a new memory location. However, the same intuition can be generalized even when α

Table 7.5. Weakest pre-conditions and Hoare inference rules for SAL

(a) Computing weakest pre-conditions for atomic formulae

Notation: $\varphi \equiv (T_1 = T_2)$, V = set of free variables of φ
$\psi_u \equiv \exists X \ (nempty(X \cap \mathbf{u}) \wedge empty(X \cap \mathbf{c}_{\text{nil}}))$

Statement (α)	$\mathbf{wp}(\alpha, \varphi)$
θ `:=` **nil**	$\zeta \wedge \bigvee_{\Omega \subseteq V} \left(\langle\varphi\rangle_\Omega [V \setminus \Omega \mapsto \mathbf{c}_{\text{nil}}][\Omega \mapsto \widetilde{\Omega}] \right)$, where $\zeta \equiv \psi_u$ if θ is `u->n`, and True otherwise.
`u := ` θ	$\zeta \wedge \bigvee_{\Omega \subseteq V} \left(\langle\varphi\rangle_\Omega [V \setminus \Omega \mapsto \mathbf{c}_{\text{nil}}][\Omega \mapsto \widetilde{\Omega}] \right)$, where $\zeta \equiv \psi_v$ if θ is `v->n`, and True otherwise.
`u->n := v`	$\zeta \wedge \bigvee_{\Omega \subseteq V} \left(\langle\varphi\rangle_\Omega [V \setminus \Omega \mapsto \mathbf{c}_{\text{nil}}][\Omega \mapsto \widetilde{\Omega}] \right)$, where $\zeta \equiv \psi_u$
`u :=` **new**	$\bigvee_{\Omega_2 \subseteq \Omega_1 \subseteq V} \left(\langle\varphi\rangle_{\Omega_1, \Omega_2, \mathbf{u}} [V \setminus \Omega_1 \mapsto \mathbf{c}_{\text{nil}}][\Omega_1 \setminus \Omega_2 \mapsto \mathbf{u}][\Omega_2 \mapsto \widetilde{\Omega_2}] \right)$
free`(u)`	$\zeta \wedge \bigvee_{\Omega \subseteq V} \left(\langle\varphi\rangle_\Omega [V \setminus \Omega \mapsto \mathbf{c}_{\text{nil}}][\Omega \mapsto \widetilde{\Omega}] \right)$, where $\zeta \equiv \psi_u$

(b) Hoare inference rules

Notation: [`B`]: SAL formula corresponding to Boolean expression `B` in programming language
[`u = v`] $\equiv \exists X \ (nempty(X \cap \mathbf{u}) \wedge nempty(X \cap \mathbf{v}))$
[`IsNil(u)`] $\equiv nempty(\mathbf{c}_{\text{nil}} \cap \mathbf{u})$
[`B1 or B2`] $\equiv [B1] \vee [B2]$
[`not B`] $\equiv \neg[B]$

Inference rules:

$$\frac{}{\{wp(\texttt{Stmt},\varphi)\}\ \texttt{Stmt}\ \{\varphi\}}$$ `Stmt` $\in$ `AsgnStmt`, `AllocStmt` or `FreeStmt`

$$\frac{\{\varphi_1\}\ \texttt{Stmt1}\ \{\varphi_2\} \qquad \{\varphi_2\}\ \texttt{Stmt2}\ \{\varphi_3\}}{\{\varphi_1\}\ \texttt{Stmt1;Stmt2}\ \{\varphi_3\}}$$ Sequential composition

$$\frac{\{\varphi_1\}\ \texttt{Stmt}\ \{\varphi_2\} \qquad \varphi_3 \Rightarrow \varphi_1 \qquad \varphi_2 \Rightarrow \varphi_4}{\{\varphi_3\}\ \texttt{Stmt}\ \{\varphi_4\}}$$ Strengthening pre-condition
Weakening post-condition

$$\frac{\{\varphi_1 \wedge [B]\}\ \texttt{Stmt1}\ \{\varphi_2\} \qquad \{\varphi_1 \wedge \neg[B]\}\ \texttt{Stmt2}\ \{\varphi_2\}}{\{\varphi_1\}\ \texttt{if (B) then Stmt1 else Stmt2}\ \{\varphi_2\}}$$ Conditional branch

$$\frac{\varphi_1 \Rightarrow \varphi_L \qquad \{\varphi_L \wedge [B]\}\ \texttt{Stmt}\ \{\varphi_L\} \qquad \varphi_L \wedge \neg[B] \Rightarrow \varphi_2}{\{\varphi_1\}\ \texttt{while (B) do Stmt}\ \{\varphi_2\}}$$ Looping construct

is a memory allocating primitive statement like `u :=` **new**. The only difference in this case is that we need to express a SAL formula φ as $\bigvee_{\Omega_2 \subseteq \Omega_1 \subseteq V} \langle\varphi\rangle_{\Omega_1, \Omega_2, \mathbf{u}}$. Since $empty(\mathbf{u} \cap \mathbf{c}_{\text{nil}})$ necessarily holds after execution of `u :=` **new**, the formula $\bigvee_{\Omega_2 \subseteq \Omega_1 \subseteq V} \langle\varphi\rangle_{\Omega_1, \Omega_2, \mathbf{u}}$ is equivalent to φ in all program states after execution of `u :=` **new**. Table 7.5a lists $wp(\alpha, \varphi)$ for various primitive statements α in our language and for $\varphi \equiv (T_1 = T_2)$. As discussed earlier, this suffices for computing $wp(\alpha, \varphi)$ for all SAL formulae φ. Table 7.5b gives Hoare inference rules for looping, sequential

composition and conditional branching constructs in our programming language. This completes the set of Hoare inference rules for our simple language, with SAL as the base logic.

Decidability issues: In order to prove heap-related properties of programs in our language using Hoare-style reasoning, we must formulate the property as a Hoare triple using SAL as the base logic, and then derive the triple by repeated applications of inference rules in Table 7.5b. Since Hoare logic is relatively complete, if the property holds for the program, there exists a way to derive the triple by repeated applications of inference rules, provided we have an algorithm to check implications in SAL. Implication checking is needed in the rule for weakening pre-conditions and strengthening post-conditions, and also in the rule for looping constructs in Table 7.5b. Given formulas φ and ψ in SAL, $\varphi \Rightarrow \psi$ iff $\varphi \wedge \neg\psi$ is unsatisfiable. In other words, for every storeless structure $\Upsilon = (S_{\text{nil}}, \Gamma)$ and for every assignment $\nu : V(\varphi) \cup V(\psi) \rightarrow \{S_{\text{nil}}\} \cup \Gamma$, we have $\nu \not\models (\varphi \wedge \neg\psi)$. Clearly, it suffices to have a satisfiability checker for SAL in order to check implications between SAL formulae that arise in our Hoare-style proofs. Unfortunately, satisfiability checking in SAL is undecidable. The proof of undecidability follows a similar line of reasoning as used by Bozga et al [32] to show the undecidability of wAL.

Various alternative strategies can, however, be adopted to check satisfiability of subclasses of formulae in practice. A simple strategy that is often used is to work with a set of *sound* (but not *complete*) inference rules in an 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 SAL quite effectively for proving useful properties of several interesting programs. Example 7.3 below shows an example of such rule schema for SAL. A second strategy is to use a decidable fragment of the logic. An example of this is the logic pAL (propositional Alias Logic) [32], a strict subclass of Bozga et al's wAL, but for which implication checking is in NP. A decidable fragment similar to pAL can also be defined for SAL, although the ability to express properties of heaps is reduced (e.g., properties like $rch(X, Y, \mathsf{f})$ are not expressible in pAL). Finally, we can define a notion of *bounded semantics*, in which we only consider storeless structures with at most k languages other than S_{nil}, 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 $k + 3$ states, and since there are finitely many distinct transition systems with $k+3$ states, it follows that satisfiability checking in SAL with bounded semantics is decidable. Note, however, that if a SAL formula φ 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

memory locations. If, however, the heap grows to contain more than k distinct non-garbage memory locations, a property proved using k-bounded semantics is not guaranteed to hold.

Example 7.3. The Hoare triple in Example 7.2 can be proved using the rules in Table 7.5b, along with the inference rule schema for SAL in Table 7.6. The loop invariant used at location L2 of the program in Example 7.1 is $\varphi_{L2} \equiv ((\mathbf{t1} = \mathbf{c}_{\text{nil}}) \wedge rch(c_{\text{nil}}, \mathbf{hd}, \mathsf{n})) \vee (\neg(\mathbf{t1} = \mathbf{c}_{\text{nil}}) \wedge (rch(\mathbf{t1}, \mathbf{hd}, \mathsf{n}) \wedge rch(\mathbf{c}_{\text{nil}}, \mathbf{t1}, \mathsf{n})))$.

Table 7.6. Sound inference rule schema for SAL

Rule	Conditions
$\dfrac{nempty(X \cap Y{\cdot}F_1) \quad nempty(Z \cap X{\cdot}F_2)}{nempty(Z \cap Y{\cdot}F_1{\cdot}F_2)}$	X: variable or constant, Y, Z: terms F_1, F_2: regular expressions on selector names
$\dfrac{nempty(X)}{nempty(X \cap X{\cdot}F^*)}$	X: term, F: regular expression on selector names

7.7. A counter automaton based technique

We have seen above two different automata based techniques for reasoning about heap manipulating programs: regular model checking, and Hoare-style reasoning using a logic called 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 7.5, we restrict our attention to a class of heap manipulating programs in which each memory location has a single pointer-valued selector. We have seen earlier that if we ignore garbage, the heap graph of such a program with n variables consists of at most $2n$ 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 nodes 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 to use the counter to store the count of nodes in the sequence. Bouajjani et al [25] have used this idea to define a *counter automaton* abstraction of the state transition behaviour of heap manipulating programs. More recently, Abdulla et al have proposed a technique using graph minors that achieves a similar abstraction [21].

Let $X = \{x_1, \ldots x_n\}$ be a set of counter variables, and let Φ be the set of Presburger logic formulae with free variables in $\{x_i, x'_i \mid x_i \in X\}$. A *counter automaton* with the set X of counter variables is a tuple $\mathcal{A}_c = (Q, X, \Delta)$, 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, β), where $\beta : X \to \mathbb{N}$

assigns a natural number to each counter variable. The automaton is said to have a transition from (q, β) to (q', β') iff $(q, \varphi, q') \in \Delta$ for some Presburger formula $\varphi \in \Phi$ and the following conditions hold: (i) $\beta'(x_i) = \beta(x_i)$ for every x'_i that is not free in φ, and (ii) φ evaluates to true on substituting $\beta(x_i)$ for all free variables x_i and $\beta'(x_i)$ for all free variables x'_i of φ. 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, β_i) to (q_{i+1}, β_{i+1}) for every $i \geq 0$.

In order to construct a counter automaton abstraction of the state transition behaviour of a heap manipulating program, we first build a *structural abstraction* of the heap graph. This is done by first 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 [25]. 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 (see Section 7.5.1 for a definition of "interruptions") and belong to the same uninterrupted list segment. Intuitively, two nodes in the heap graph are mapped to the same node in the abstract structure 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 [24], 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 nodes in the heap graph that have been mapped to it. Furthermore, the abstract structure is constructed in such a way that for every sequence of two abstract nodes connected by an edge, one of the nodes is necessarily pointed to by a program variable, or has an in-degree exceeding 1. Given this condition, it can be shown [25] that the number of different abstract structures representing uninterrupted list segments in the heap graph of a program with a finite number of variables is always finite.

A counter automaton abstraction of the state transition graph of a heap manipulating program is obtained by letting the control states of the automaton be (program location, structural abstraction of heap graph) pairs. Thus, each control state is an abstraction of the program state. 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 encode the operational semantics of various primitive program statements. Bouajjani et al have shown [25] how such Presburger logic formulae can be calculated for assignment, memory allocation and memory de-allocation statements. A transition of the counter automaton 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 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 access paths to this node. In this sense, a counter automaton abstraction uses a store based semantics.

For data-insensitive programs manipulating the heap, it can be shown that a counter automaton abstraction is bisimilar to the state transition graph of the original program. Hence this abstraction preserves all temporal properties of data-insensitive programs. It has been shown by Bouajjani et al that the counter automaton abstraction also 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 [25] for a detailed exposition on this topic.

7.8. 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 article is intended to provide an introductory perspective on 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 editors of the current volume for their invitation to write this article. The author also thanks the anonymous reviewers and Bhargav Gulavani for their critical comments.

References

[1] J. E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory, Languages and Computation*. (Addison-Wesley, 1979).
[2] W. Landi, Undecidability of static analysis, *ACM Letters on Programming Languages and Systems*. **1**(4), 323–337, (1992).
[3] G. Ramalingam, The undecidability of aliasing, *ACM Trans. on Program. Lang. Syst.* **16**(5), 1467–1471, (1994). ISSN 0164-0925.
[4] E. M. Clarke, O. Grumberg, and D. Peled, *Model Checking*. (The MIT Press, 2000).
[5] Y. Kesten, O. Maler, M. Marcus, A. Pnueli, and E. Shahar, Symbolic model checking with rich assertional languages, *Theoretical Computer Science*. **256**(1–2), 93–112, (2001).

[6] L. Fribourg. Reachability sets of parametrized rings as regular languages. In *Proc. of INFINITY*. Elsevier Science, (1997).
[7] P. A. Abdulla, B. Jonsson, M. Nilsson, and M. Saksena. A survey of regular model checking. In *Proc. of CONCUR*, pp. 35–48. Springer, (2004).
[8] B. Jonsson and M. Nilsson. Transitive closures of regular relations for verifying infinite-state systems. In *Proc. of TACAS*, pp. 220–234. Springer, (2000).
[9] P. Abdulla, B. Jonsson, M. Nilsson, and J. d'Orso. Regular model checking made simple and efficient. In *Proc. of CONCUR*, pp. 116–130. Springer, (2002). ISBN 3-540-44043-7.
[10] P. A. Abdulla, B. Jonsson, and M. Nilsson. Algorithmic improvements in regular model checking. In *Proc. of CAV*, pp. 236–248. Springer, (2003).
[11] A. Bouajjani, P. Habermehl, and V. Tomas. Abstract regular model checking. In *Proc. of CAV*, pp. 372–386. Springer, (2004).
[12] A. Bouajjani, P. Habermehl, P. Moro, and T. Vojnar. Verifying programs with dynamic 1-selector-linked structures in regular model checking. In *Proc. of TACAS*, pp. 13–29. Springer, (2005).
[13] T. Touili, Regular model checking using widening techniques, *Electronic Notes in Theoretical Computer Science*. **50**(4), (2001).
[14] P. Habermehl and T. Vojnar, Regular model checking using inference of regular languages, *Electronic Notes in Theoretical Computer Science*. **138**(3), 21–36, (2005).
[15] R. Manevich, E. Yahav, G. Ramalingam, and M. Sagiv. Predicate abstraction and canonical abstraction for singly-linked lists. In *Proc. of VMCAI*, pp. 181–198. Springer, (2005).
[16] D. Dams, Y. Lakhnech, and M. Steffen. Iterating transducers. In *Proc. of CAV*, pp. 52–53. Springer, (2001).
[17] B. Boigelot, A. Legay, and P. Wolper. Iterating transducers in the large. In *Proc. of CAV*, pp. 223–235. Springer, (2003).
[18] A. Bouajjani, P. Habermehl, and A. Rogalewicz. Abstract regular tree model checking of complex dynamic data structures. In *Proc. of SAS*, pp. 52–70. Springer, (2006).
[19] B. A. Trakhtenbrot and Y. A. Barzdin, *Finite Automata: Behaviour and Synthesis*. (North Holland, 1973).
[20] A. Bouajjani, P. Habermehl, A. Rogalewicz, and T. Vojnar. Abstract tree regular model checking of complex dynamic data structures. In *Proc. of SAS*, pp. 52–70. Springer, (2006).
[21] P. A. Abdulla, A. Bouajjani, J. Cederberg, F. Haziza, and A. Rezine. Monotonic abstraction for programs with dynamic heap graphs. In *Proc. of CAV*, pp. 341–354. Springer, (2008).
[22] G. Yorsh, A. Rabinovich, M. Sagiv, A. Meyer, and A. Bouajjani, A logic of reachable patterns in linked data structures, *Journal of Logic and Algebraic Programming*. **73** (1-2), 111–142, (2007).
[23] A. Podelski and T. Wies. Boolean heaps. In *Proc. of SAS*, pp. 268–283. Springer, (2005).
[24] M. Sagiv, T. Reps, and R. Wilhelm, Parametric shape analysis via 3-valued logic, *ACM Trans. on Prog. Lang. Sys.* **24**, 2002, (1999).
[25] A. Bouajjani, M. Bozga, P. Habermehl, R. Iosif, and P. Moro. Programs with lists are counter automata. In *Proc. of CAV*, pp. 517–531. Springer, (2006).
[26] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proc. of LICS*, pp. 55–74. IEEE Computer Society, (2002).
[27] C. Calcagno. *Semantic and Logical Properties of Stateful Programming*. PhD thesis, University of Genova, (2002).

[28] D. Distefano, P. W. O'Hearn, and H. Yang. A local shape analysis based on separation logic. In *Proc. of TACAS*, pp. 287–302. Springer, (2006).

[29] H. Jonkers. Abstract storage structures. In *Algorithmic Languages*, pp. 321–344. North Holland, (1981).

[30] A. Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *Proc. of PLDI*, pp. 230–241. ACM Press, (1994).

[31] M. Bozga, R. Iosif, and Y. Laknech. Storeless semantics and alias logic. In *Proc. of PEPM*, pp. 55–65. ACM Press, (2003).

[32] M. Bozga, R. Iosif, and Y. Lakhnech. On logics of aliasing. In *Proc. of SAS*, pp. 344–360. Springer, (2004).

[33] C. A. R. Hoare and H. Jifeng. A trace model for pointers and objects. In *Proc. of ECOOP*, pp. 1–17. Springer, (1999).

[34] N. Rinetzky, J. Bauer, T. Reps, S. Sagiv, and R. Wilhelm. A semantics for procedure local heaps and its abstractions. In *POPL*, pp. 296–309. ACM, (2005).

[35] J. Berdine, C. Calcagno, and P. W. O'Hearn. A decidable fragment of separation logic. In *Proc. of FSTTCS*, pp. 97–109. Springer, (2004).

[36] C. Calcagno, H. Yang, and P. W. O'Hearn. Computability and complexity results for a spatial assertion language for data structures. In *Proc. of FSTTCS*, pp. 108–119. Springer, (2001).

[37] C. Calcagno, P. Gardner, and M. Hague. From separation logic to first-order logic. In *Proc. of FoSSaCS*, pp. 395–409. Springer, (2005).

[38] S. S. Ishtiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. In *Proc. of POPL*, pp. 14–26. ACM, (2001).

[39] N. Immerman, A. Rabinovich, T. Reps, M. Sagiv, and G. Yorsh. The boundary between decidability and undecidability for transitive-closure logics. In *Proc. of CSL*, pp. 160–174. Springer, (2004).

[40] A. Möller and M. I. Schwartzbach. The pointer assertion logic engine. In *Proc. PLDI*, pp. 221–231. ACM Press, (2001).

[41] N. Klarlund and M. I. Schwartzbach. Graph types. In *Proc. of POPL*, pp. 196–205. ACM, (1993).

[42] M. Benedikt, T. Reps, and M. Sagiv. A decidable logic for describing linked data structures. In *Proc. of ESOP*, pp. 2–19. Springer, (1999).

[43] J. L. Jensen, M. E. Jörgensen, N. Klarlund, and M. I. Schwartzbach. Automatic verification of pointer programs using monadic second-order logic. In *Proc. of PLDI*, pp. 226–236. ACM Press, (1997).

[44] E. W. Dijkstra. and C. S. Scholten, *Predicate calculus and program semantics.* (Springer-Verlag New York, Inc., 1990). ISBN 0-387-96957-8.