Noname manuscript No.
(will be inserted by the editor)

Programs with Lists are Counter Automata

Ahmed Bouajjani · Marius Bozga · Peter Habermehl · Radu Iosif · Pierre Moro · Tomáš Vojnar

the date of receipt and acceptance should be inserted later

Abstract We address the problem of verifying programs manipulating one-selector linked data structures. We propose and study in detail an application of counter automata as an accurate abstract model for this problem. We let control states of the counter automata correspond to abstract heap graphs where list segments without sharing are collapsed, and use counters to keep track of the number of elements in these segments. As a significant theoretical result, we show that the obtained counter automata are bisimilar to the original programs. Moreover, from a practical point of view, our translation allows one to apply efficient automatic analysis techniques and tools developed for counter automata (integer programs) in order to verify both safety as well as termination of list-manipulating programs. As another theoretical contribution, we prove that if the control of the generated counter automata does not contain nested loops (i.e., these automata are flat), both safety and termination are decidable for the original programs. Subsequently, we generalise our counter-automata-based model to keep track of ordering properties over lists storing ordered data. Finally, we show effectiveness of our approach by verifying automatically safety as well as termination of several sorting programs.

Keywords formal verification · programs with singly-linked lists · safety and termination · counter automata · bisimulation · lists with ordered data

This work is a full and revised version of the extended abstract [10] published in Proceedings of CAV'06. The work was supported in part by the Czech Science Foundation (project P103/10/0306), the Czech Ministry of Education (projects COST OC10009 and MSM 0021630528), the internal BUT FIT grant FIT-10-1 and the French ANR projects Averiles and Veridyc.

A. Bouajjani, P. Habermehl, P. Moro
LIAFA, University Paris Diderot—Paris 7, Case 7014, 2, pl. Jussieu, F-75251 Paris Cedex 13, France
E-mail: {Ahmed.Bouajjani,Peter.Habermehl,Pierre.Moro}@liafa.jussieu.fr

M. Bozga, R. Iosif
VERIMAG, 2 av. de Vignate, F-38610 Gières, France
E-mail: {bozga,iosif}@imag.fr

T. Vojnar
FIT, Brno University of Technology, Božetěchova 2, CZ-61266, Brno, Czech Republic
Tel.: +420541141202
Fax: +420541141270
E-mail: vojnar@fit.vutbr.cz

1 Introduction

The design of automatic verification methods for programs manipulating dynamic linked data structures is a challenging problem. Indeed, analysing the behaviour of such programs requires reasoning about complex transformations of data structures involving both creation and deletion of objects as well as modifications of the links between them (due to pointer manipulations). The heap of such programs may have in fact an arbitrary size and shape—it can be a general graph structure of an in advance unrestricted size. For tackling this problem, various approaches (surveyed below) addressing different subclasses of programs and using different kinds of formalisms for representing and reasoning about infinite sets of heap structures have been proposed.

We consider in this paper the class of programs manipulating linked data structures with a single data-field selector. This class corresponds to programs manipulating linked lists with the possibility of sharing and circularities. We propose and study in detail an approach for automatic verification of such programs which is mainly based on using counter automata as accurate abstract (infinite-state) models. These models can be used for checking both safety properties and termination of the considered programs using techniques such as (abstract) symbolic reachability analysis (for safety and invariance checking) and automatic generation of decreasing ranking functions (for termination checking).

Let us present in more detail the proposed approach. We start from the observation that if we do not consider garbage (parts of the heap not reachable from the pointer variables of the program), the heap graph is always a finite collection of graphs of a special form close to a tree—namely, a tree where edges are directed towards the root and there may be a simple cycle at the root. The number of such graphs is infinite, but it can be proved that for each of them, the number of vertices where sharing occurs is bounded by the number of pointer variables of the program.

Then, for data-insensitive programs (i.e., programs that do not assign nor test the data within the list cells, such as the list reversal program from Figure 2) a natural abstraction consists in mapping each sequence of elements between two sharing points into an abstract sequence of some (fixed) bounded size. However, for each given value of the bound, this abstraction is obviously not precise in general. In order to define a precise abstraction, we need in fact to reason about the size of each sequence between two sharing points. This leads to the idea of using counters in order to keep this information in the abstract model (and therefore to use counter automata as abstract models).

In fact, considering counter automata-based models has several advantages. Not only it allows us to define accurate abstractions, but it allows us to handle quantitative properties depending on the sizes of some parts of the heap as well. Thus, we can handle programs with integer variables whose value is related to the contents of the lists (e.g., to their length). Moreover, it provides a powerful way for checking termination which typically requires reasoning about decreasing values (e.g., the size of the part of the list to be treated).

A first contribution of the paper is to define an abstraction mapping from data-insensitive programs to counter automata for which we prove that the (concrete) program and its abstraction are *bisimilar*. This result is significant since every property that can be proved (or disproved) on the counter automaton (e.g., reachability or termination) automatically holds (or does not hold) on the concrete program. In other words, our abstraction does not loose information for the class of data-insensitive programs.

The idea of the abstraction is the following. The control states of the built automaton correspond to abstract shapes (heap graphs where sequences between shared points are reduced to single vertices), and each transition corresponds to the execution of a program statement.

Each transition represents a modification in the shape together with a modification on the counters which are attached to nodes of the abstract shapes and which count the length of the concrete sequences of list nodes abstracted by the abstract nodes.

The control structure of the built counter automata can be arbitrary in general. However, it turns out that these automata have an important property: we prove that if we consider the evolution of the sum of all counters, the effect of executing any control loop is to increment this sum by a constant which depends on the program. We use this fact to establish a significant decidability result for list programs: for every given (data-insensitive) list program, if the control structure of the generated counter automaton has no nested loops, the verification problems of safety properties and termination are both decidable.

Subsequently, we consider programs that manipulate objects ranging over a potentially infinite data domain supplied with an ordering relation, assuming that the only allowed operations on these data values are comparisons w.r.t. the ordering relation and assignments of data between two list cells. This class of programs includes, for instance, sorting programs. We extend our previous abstraction principle to the heap graphs of these programs by taking into account (in addition to the size) some information about the order of the elements in the abstracted sequences between sharing points, and we provide a construction which associates with each program a counter-automaton-based abstract model. We show that this abstraction is sound w.r.t. the chosen ordering predicates. We briefly discuss ways of dealing with a richer class of data manipulating statements too.

We have implemented our verification method for data-insensitive programs in a prototype tool called L2CA (Lists To Counter Automata) [28], and performed a number of experiments on several textbook list sorting programs (`BubbleSort`, `InsertSort`, `MergeSort`, and `SelectionSort`), in which, for simplicity, the data comparisons were replaced by nondeterministic choices. On these examples, we have successfully checked absence of null pointer dereferences, shape invariance, as well as termination. For two of the sorting algorithms (`BubbleSort`, `InsertSort`), we have then also built counter automata tracking the chosen ordering predicates, which allowed us to prove sortedness of the output of these algorithms.

1.1 Related Work

The area of automated verification of programs manipulating dynamic linked data structures has recently attracted quite a lot of interest. Various approaches to verification of such programs differing in their principles, degree of automation, generality, and scalability have been proposed. The approaches are based, e.g., on monadic second order logic [30], 3-valued predicate logic with transitive closure [33], separation logic [31,21,8,35], other logics [14, 3,29], automata [11,7,19,12], or other symbolic representations such as [36,4,25,17,1].

Interestingly, among the works concentrating on verification of programs manipulating singly-linked lists, the idea of abstracting away the interior of list segments between incoming edges is quite common to many works, even though they are independent and use different approaches and frameworks. Such an abstraction, of course, involves a loss of precision. An idea of partially avoiding this loss by joining a counter with each abstract list segment appears, e.g., in [36] where the possible values of such counters are tracked in an abstract way using a widening operator. This suffices for verification of certain safety properties, but an irrecoverable loss of precision is still involved unlike in our work. In [13], the authors use an abstract shape model with counters, but their concerns are mostly related to the decidability of a specification logic. The approaches that are the closest to ours are [5]

and its later extension [7] where an alternative translation to a bisimilar counter system is also considered, but without the extension to ordered data and without the decidability result for the case when a flat counter automaton is derived.

More recently, in [15], we have also investigated the complexity of several verification problems (e.g., for safety and termination properties) for programs with singly-linked lists. However, this line of work uses a different translation to counter automata in order to guarantee that the obtained models are flat. This translation scheme is less general than the one considered here as it does not consider destructive update statements.

Another similar work that involves tracking the lengths of list segments and considers termination is [20]. However, this approach does not generate a counter automaton simulating the original program. Instead, it first obtains invariants of the program (using separation logic) and then computes the so-called variance relations that say how the invariants change within each loop when the loop is fired once more. When the computed variance relations are well-founded, termination of the program is guaranteed. Unlike our (bisimulation preserving) approach, the analysis of [20] fails if the initial abstraction is not precise enough. The approach of [20] was later generalised in [9] to a general framework that one can use to extend existing invariance analyses to variance analyses that can in turn be used for checking termination. Checking termination of programs with lists is tackled also by [34, 3]. In these works, ranking functions must be given manually whereas our approach is fully automated.

Termination of tree manipulation programs is then considered in [22] using a translation to counter automata which is refinable via a counterexample-guided abstraction refinement (CEGAR) loop—up to the fact that the set of progress measures over the trees being manipulated by the considered programs is fixed in advance. The lossy abstraction involved in this case is unavoidable as one cannot derive a bisimilar counter model like in our case for tree-manipulating programs. This is due to the fact that there are infinitely many reachable abstract shape graphs for such programs. Automated checking of termination of a program manipulating trees is also considered in [26] where the Deutsch-Schorr-Waite tree traversal algorithm was proved to terminate using a manually created progress monitor, encoded in first-order logic.

2 Programs with Lists

In this section, we define a model for imperative programs manipulating dynamic list data structures. We assume that lists are implemented using data types with one selector, i.e., next pointer (or reference) field, as it is the case in most common imperative programming languages. We consider programs without function calls and concurrency constructs, therefore all variables are assumed to be global. We assume the lists to store elements from some ordered domain. In addition to the list data structures, the programs we consider can have (unbounded) integer variables. Simple examples of such programs include list reversion, list insertion, sorting procedures, programs counting the elements in a list, etc.

The programming model presented in this section has been simplified compared to actual programming languages, such as C or C++, in the following respects: (i) the values of pointers are either allocated memory addresses or null (i.e., no dangling pointers), (ii) no pointer arithmetic is allowed, (iii) allocation statements never fail, and (iv) unreachable memory cells are automatically garbage collected.

In the first part of the paper, we assume the data stored in the lists to be completely abstracted away and we concentrate purely on pointer manipulations. In the second part, we allow a limited form of manipulation of the data contents of the lists, namely assignments of

data between memory cells and comparisons of the data contents of the cells. We concentrate mainly on checking ordering properties on programs that reorder the lists which they are given rather than change their contents. Dealing with other forms of data manipulations, such as increment or decrement, is briefly discussed at the end of the second part of the paper too—however, since dealing with such data manipulations is not the goal of this paper, we do not consider them in the supported program syntax given below.

2.1 Syntactic Definitions

The abstract syntax of the programs considered in this paper is given in Figure 2.1. We use *Lab* to denote a finite set of program labels (control locations), *PVar* to denote a finite set of pointer variables, and *IVar* to represent a finite set of integer variables (counters). The pointer variables refer to list cells. Pointers can be used in assignments such as $u := null$, $u := w$, and $u := w.next$; selector updates $u.next := w$ and $u.next := null$; and new cell creation $u := new$. Assignment of data between memory cells is possible via the $u.data := v.data$ statements. Counters can be incremented $i := i + 1$, decremented $i := i - 1$, and reset $i := 0$. The control structure is composed of iteration (`while`) statements and conditional (`if-then-else`) statements. The guards of the control statements are pointer equality $u = w$ and $u = null$, data comparisons $u.data \leq v.data$, zero tests for counters $i = 0$, and boolean combinations of the above. An example of a simple program from the considered class is the list reversal program in Figure 2.

$$
\begin{array}{rcl}
l & \in & Lab \\
u,v,w & \in & PVar \\
i,j,k & \in & IVar \\
Program & := & \{l : Stmnt;\}^* \\
Stmnt & := & WhileStmnt \mid IfStmnt \mid Asgn \\
WhileStmnt & := & \texttt{while}\ Guard\ \texttt{do}\ \{Stmnt;\}^*\ \texttt{od} \\
IfStmnt & := & \texttt{if}\ Guard\ \texttt{then}\ \{Stmnt;\}^*\ \left[\texttt{else}\ \{Stmnt;\}^*\right]\ \texttt{fi} \\
Asgn & := & u := null \mid u := new \mid u := w \mid u := w.next \mid u.next := null \\
 & & \mid u.next := w \mid u.data := v.data \mid i := 0 \mid i := i+1 \mid i := i-1 \\
Guard & := & u = v \mid u = null \mid u.data \leq v.data \mid i = 0 \mid \neg Guard \mid Guard \wedge Guard \mid Guard \vee Guard
\end{array}
$$

Fig. 1 Abstract syntax of programs with lists

To simplify the definition of the operational semantics given below, we assume that all programs are precompiled as follows. Each pointer assignment of the form $u := new$, $u := w$, or $u := w.next$ is immediately preceded by an assignment of the form $u := null$. A pointer assignment of the form $u := u.next$ is turned into $v := u$; $u := null$; $u := v.next$, possibly introducing a fresh variable v. Each pointer assignment of the form $u.next := w$ is immediately preceded by $u.next := null$.

A program is said to be *data-insensitive* if it does not use any data assignment statements ($u.data := v.data$) nor data comparisons ($u.data \leq v.data$).

```
1: j := null;
2: while i ≠ null do
3:     k := i.next;
4:     i.next := j;
5:     j := i;
6:     i := k;
7: od
```

Fig. 2 A list reversal program

2.2 Concrete Operational Semantics

In order to define the concrete semantics of programs with lists, we have to formalise the notion of *heaps*. For our purposes, we view a heap as a graph in which nodes represent allocated memory cells, and *each node has at most one successor*. Moreover, we consider a partial mapping between pointer variables from *PVar* and graph nodes. Intuitively, if all the edges are reversed, one can imagine a heap as a set of disjoint trees in which, for each tree, there may be at most one extra edge from an arbitrary node back to the root.

In the rest of the paper, for a set A, we denote by $A_\perp$ the set $A \cup \{\perp\}$. The element $\perp$ is used to denote that a (partial) function is undefined at a given point, e.g., $f(x) = \perp$. Also, for a function $f : A \to B$, we denote by $f{\downarrow_C}$ the projection of f on C, i.e., $f{\downarrow_C} = f \cap (C \times B)$. Finally, for a function $f : A \to B$, an element $a \in A$, and an element $b \in B$, we use the notation $f[a \to b]$ to denote a function which arises from f by setting the value of a to b, i.e., $f[a \to b] = (f \setminus \{(a, f(a))\}) \cup \{(a,b)\}$.

Definition 1 Let $\langle \mathfrak{D}, \preceq \rangle$ be an infinite, totally ordered set and *PVar* a non-empty finite set of pointer variables. A *heap* is a tuple $H = \langle N, S, V, D \rangle$ where N is a finite set of nodes, $S : N \to N_\perp$ is a *successor function*, $V : PVar \to N_\perp$ is a function associating to each variable a node, and $D : N \to \mathfrak{D}$ is a function associating to each node a data element.

The set of all heaps using variables from *PVar* is denoted by $\mathcal{H}(PVar)$. We denote the fact that $S(n_1) = n_2$ in H by $n_1 \underset{H}{\to} n_2$. We use $u \underset{H}{\to} n$ to denote the formula $\exists m \,.\, V(u) = m \wedge m \underset{H}{\to} n$. Here, H might be omitted when it is clear from the context. We denote by $\overset{*}{\underset{H}{\to}}$ the reflexive and transitive closure of $\underset{H}{\to}$. A node n is said to be a *cut-point* in H, denoted as $cut_H(n)$, if it has two predecessors or it is pointed to by a variable. Formally, let

$$cut_H(n) \;:\; \exists n_1, n_2 \in N \,.\, n_1 \neq n_2 \wedge S(n_1) = S(n_2) = n \;\vee\; \exists u \in PVar \,.\, V(u) = n.$$

When a node n is removed from a heap, or when its successor link is changed, some successors of n may become unreachable from the pointer variables. Due to the implicit garbage collection, such nodes are removed from the heap. The set of nodes whose lifetime depends exclusively on their reachability through the node n is denoted as $dep_H(n)$ in the following. Formally, for a heap $H = \langle N, S, V, D \rangle$ and $n \in N$:

$$dep_H(n) = \left\{ m \in N \mid \forall u \in PVar. \neg\left(u \xrightarrow[\langle N, S[n \to \perp], V, D \rangle]{*} m \right) \right\}$$

The *state* of a program with lists is a triple $\langle l, H, \iota \rangle$ where $l \in Lab$ is the current program label, $H \in \mathcal{H}(PVar)$ is the current heap configuration, and $\iota : IVar \to \mathbb{Z}$ is the current valuation of counter variables. A pointer assignment statement s between locations l and

l' (denoted $l : s;l'$) changes the state from $\langle l,H,\imath\rangle$ to $\langle l',H',\imath\rangle$, where H' is a new heap configuration such that $H \xrightarrow{s} H'$ in conformance with the rules presented in Figure 3.

Rules C_1, C_2, and C_3 describe the effect of assigning null to a pointer variable u. If u is not allocated, then there is no change in the heap (C_1). Otherwise, the location n pointed to by u and its dependent nodes are removed if and only if it is not reachable from another variable than u (C_2, C_3). The assignment between two pointer variables u and w assumes that u is previously null (C_4). Analogously, the allocation statement described by (C_5) assumes that the allocated variable was previously null. Notice that the data element of the newly allocated cell is chosen randomly from the data domain $\mathfrak{D}$. Further, setting the *next* field of an allocated memory cell to null may generate garbage cells, which must be collected (C_9). Notice that assigning the *next* field to another pointer variable is possible only provided that it was previously set to null (C_{10}). When a null pointer dereference is encountered, the heap is changed to a special sink heap configuration H_{err} (C_6, C_8, C_{11}), $H_{err} \notin \mathcal{H}(PVar)$.

A pointer equality test $u = v$ evaluates to true in a heap $H = \langle N,S,V,D\rangle$ iff $V(u) = V(v)$. Also, $u = null$ is true iff $V(u) = \perp$. A data comparison $u.data \leq v.data$ leads to H_{err} iff $V(u) = \perp$ or $V(v) = \perp$. Otherwise, it evaluates to true iff $D(V(u)) \leq D(V(v))$. The semantics of tests and updates on counters is as usual, and so is the semantics of the conditional statements and loops.

$$\frac{V(u) = \perp}{H \xrightarrow{u := null} H}\ C_1 \qquad \frac{V(u) \neq \perp \quad \exists w \in PVar \setminus \{u\}\ .\ w \xrightarrow[H]{*} V(u)}{H \xrightarrow{u := null} \langle N,S,V[u \to \perp],D\rangle}\ C_2$$

$$\frac{V(u) = n \in N \quad \forall w \in PVar \setminus \{u\}\ .\ \neg\big(w \xrightarrow[H]{*} n\big) \quad N' = N \setminus (\{n\} \cup dep_H(n))}{H \xrightarrow{u := null} \langle N',S{\downarrow_{N'}},V[u \to \perp],D{\downarrow_{N'}}\rangle}\ C_3$$

$$\frac{V(u) = \perp}{H \xrightarrow{u := w} \langle N,S,V[u \to V(w)],D\rangle}\ C_4 \qquad \frac{V(u) = \perp \quad n \notin N \text{ is a fresh node} \quad d \in \mathfrak{D}}{H \xrightarrow{u := new} \langle N \cup \{n\},S[n \to \perp],V[u \to n],D[n \to d]\rangle}\ C_5$$

$$\frac{V(w) = \perp}{H \xrightarrow{u := w.next} H_{err}}\ C_6 \qquad \frac{V(u) = \perp \quad V(w) = n \in N}{H \xrightarrow{u := w.next} \langle N,S,V[u \to S(n)],D\rangle}\ C_7$$

$$\frac{V(u) = \perp}{H \xrightarrow{u.next := null} H_{err}}\ C_8 \qquad \frac{V(u) = n \in N \quad N' = N \setminus dep_H(n)}{H \xrightarrow{u.next := null} \langle N',S[n \to \perp]{\downarrow_{N'}},V,D{\downarrow_{N'}}\rangle}\ C_9$$

$$\frac{V(u) = n \in N \quad S(n) = \perp}{H \xrightarrow{u.next := w} \langle N,S[n \to V(w)],V,D\rangle}\ C_{10} \qquad \frac{V(u) = \perp \text{ or } V(v) = \perp}{H \xrightarrow{u.data := v.data} H_{err}}\ C_{11}$$

$$\frac{V(u) = n \in N \quad V(v) = m \in N}{H \xrightarrow{u.data := v.data} \langle N,S,V,D[n \to D(m)]\rangle}\ C_{12} \qquad \frac{}{H_{err} \xrightarrow{s} H_{err}}\ C_{13}$$

Fig. 3 Concrete semantics of heap updates

3 Counter Automata

A counter automaton with n counters is a tuple $A = \langle Q, X, q_0, \rightarrow \rangle$ where Q is a finite set of control states, $q_0 \in Q$ is an initial state, $X = \{x_1, \ldots, x_n\}$ are counter variables, and $\rightarrow \in Q \times \Phi \times Q$ are transitions where Φ is the set of Presburger formulae with free variables from $\{x_i, x_i' \mid 1 \leq i \leq n\}$. A configuration of a counter automaton with n counters is a tuple $\langle q, \nu \rangle$ where ν is a mapping from X to $\mathbb{N}$. The set of all configurations is denoted by $\mathcal{C}$. Let $FV(\varphi)$ denote the set of free variables of a formula φ. The transition relation $\xrightarrow{\mathcal{C}} \subseteq \mathcal{C} \times \mathcal{C}$ is defined by letting $(q, \nu) \xrightarrow{\mathcal{C}} (q', \nu')$ iff there exists a transition $q \xrightarrow{\varphi} q'$ such that if σ is an assignment of $FV(\varphi)$ where $\sigma(x) = \nu(x)$ and $\sigma(x') = \nu'(x)$, we have that $\varphi(FV(\varphi)\sigma)$ holds and $\nu(x) = \nu'(x)$ for all variables x with $x' \notin FV(\varphi)$. A *run* of A is a sequence of configurations $(q_0, \nu_0), (q_1, \nu_1), (q_2, \nu_2) \ldots$ such that $(q_i, \nu_i) \xrightarrow{\mathcal{C}} (q_{i+1}, \nu_{i+1})$ for each $i \geq 0$.

Definition 2 Let $A = \langle Q, X, q_0, \rightarrow \rangle$ be a counter automaton where $X = \{x_1, \ldots, x_n\}$ are counter variables that range over positive integers. A is said to be *linear* if all its transitions are of the form $\varphi(X) \wedge \bigwedge_{1 \leq i \leq n} x_i' = f_i(X)$ where φ is a formula of Presburger arithmetic and $f_i = \sum_{j=1}^{n} a_{ij} x_j + b_i$, $1 \leq i \leq n$, are linear functions with integer coefficients. Moreover, A is said to be *positive* if $a_{ij} \geq 0$ for all $1 \leq i, j \leq n$. A is also said to be *restrictive* if there exists a constant $\alpha \in \mathbb{N}$ such that for each control state $q \in Q$, on each run π that visits q, the sum of values taken by the counters, $\sum_{i=1}^{n} x_i$, increases by at most α between any two consecutive times when the control state is q.

The *control graph* of a counter automaton A is the graph having as vertices the set Q of control states, and, for any two states q and q', there is an edge between q and q' in the control graph if and only if there exists a transition $q \xrightarrow{\varphi} q'$ in A. A counter automaton is said to be *flat* if its control graph has no nested loops. A *control path scheme* is a regular expression over the edges of a control graph describing a set of paths between two control states. For a flat counter automaton $A = \langle Q, X, q_0, \rightarrow \rangle$ and any two control states $q, q' \in Q$, the set of paths between q and q' is a finite union of path schemes of the form $L_1^* S_1 L_2^* S_2 \ldots S_{n-1} L_n^*$ where L_i are possibly empty elementary loops (i.e., paths starting and ending in the same control state, with no repeated states in between), and S_i are sequential paths between loops. We use $q \xrightarrow{E} q'$ to denote that E is a control path scheme describing a set of paths between the states q and q'.

Theorem 1 *The problems of reachability and termination for flat linear positive restrictive counter automata are decidable.*

Proof Let $A = \langle Q, X, q_0, \rightarrow \rangle$ be a flat counter automaton where $X = \{x_1, \ldots, x_n\}$. We reduce the problems of reachability and termination for flat counter automata to the problem of computing the relation between input and output values of the counters, for any execution path. This amounts essentially to computing the transitive closure of a transition relation labelling a loop $q \rightarrow \ldots \rightarrow q$ in the automaton, where the input values of the counters are restricted by an initial condition $I_q(X)$. This initial condition summarises all possible executions up to the beginning of the loop. Formally, we have the following for each control state $q \in Q$:

$$I_q(X) = \exists X_0 \exists X_0' \ldots \exists X_{n-1} \,.\, \bigvee_{q_0 \xrightarrow{L_1^* S_1 L_2^* S_2 \ldots S_{n-1} L_n^*} q} \lambda_1^*(X_0, X_0') \wedge \sigma_1(X_0', X_1) \wedge \ldots \wedge \sigma_{n-1}(X_{n-2}', X_{n-1}) \wedge \lambda_n^*(X_{n-1}, X)$$

where λ_i is the transition relation of L_i, $1 \leq i \leq n$, and σ_i is the transition relation of S_i, $1 \leq i < n$, respectively. Here, R^* denotes the transitive closure of a relation R.

Hence we can restrict our attention w.l.o.g. to one self-loop of the form $q \xrightarrow{\phi} q$, parameterised by an initial condition on counters $I_q(X)$. The transition relation ϕ is of the form $\varphi(X) \wedge X' = f(X)$ where $f(X) = X \cdot A + B$ is an affine transformation, according to Definition 2. Here, $A \in \mathbb{Z}^{|X| \times |X|}$ is a square matrix and $B \in \mathbb{Z}^{|X|}$ is a row vector. We assume w.l.o.g. that all columns of A are linearly independent, for else the value of some counters are linear combinations of the other counters and could be encoded as a part of φ. As a consequence, A is invertible, i.e., there exists $A^{-1} \in \mathbb{Q}^{|X| \times |X|}$ such that $AA^{-1} = A^{-1}A = \mathbf{1}$, where $\mathbf{1}$ is the identity matrix.

Let $x_i^{(m)}$ denote the value of the counter x_i at the m-th visit of control state q. We have the following for all $m \geq 0$:

$$\begin{aligned}
\sum_{i=1}^{n} x_i^{(m+1)} - \sum_{i=1}^{n} x_i^{(m)} &\leq \alpha \\
\sum_{i=1}^{n} (f_i(x_1^{(m)}, \ldots, x_n^{(m)}) - x_i^{(m)}) &\leq \alpha \\
\sum_{i=1}^{n} (\sum_{j=1}^{n} a_{ij} x_j^{(m)} + b_i - x_i^{(m)}) &\leq \alpha \\
\sum_{i=1}^{n} (\sum_{j=1}^{n} a_{ji} - 1) x_i^{(m)} &\leq \alpha - \sum_{i=1}^{n} b_i
\end{aligned}$$

Since all coefficients are positive, we have $\sum_{j=1}^{n} a_{ji} \geq 0$ for all $1 \leq i \leq n$. If it is the case that $\sum_{j=1}^{n} a_{ji} = 0$ for some $1 \leq i \leq n$, i.e., all coefficients of x_i are zero, then x_i is not used in computing the next values of X, and we can eliminate x_i from the transition relation as described in the following.

Let $\underline{X} = X \setminus \{x_i\}$, and $\underline{f}(\underline{X})$ be the projection of f onto $\underline{X}$. Then the loop executes as follows:

$$\begin{aligned}
I_q(X^{(0)}) \wedge \varphi(X^{(0)}) \ \wedge \ X^{(1)} &= f(\underline{X}^{(0)}) \\
\varphi(X^{(1)}) \ \wedge \ X^{(2)} &= f(\underline{X}^{(1)}) \\
\varphi(X^{(2)}) \ \wedge \ X^{(3)} &= f(\underline{X}^{(2)}) \\
&\ldots
\end{aligned}$$

which is equivalent to:

$$\begin{aligned}
\exists x_i^{(0)} \,.\, [I_q(X^{(0)}) \wedge \varphi(X^{(0)})] \wedge \underline{X}^{(1)} &= \underline{f}(\underline{X}^{(0)}) \\
\exists x_i^{(1)} \exists X_p \,.\, [X^{(1)} = f(X_p) \wedge \varphi(f(X_p))] \wedge \underline{X}^{(2)} &= \underline{f}(\underline{X}^{(1)}) \\
\exists x_i^{(2)} \exists X_p \,.\, [X^{(2)} = f(X_p) \wedge \varphi(f(X_p))] \wedge \underline{X}^{(3)} &= \underline{f}(\underline{X}^{(2)}) \\
&\ldots
\end{aligned}$$

The equivalence between $\varphi(X^{(i)}) \wedge X^{(i+1)} = f(\underline{X}^{(i)})$ and $\exists x_i^{(i)} \exists X_p \,.\, [X^{(i)} = f(X_p) \wedge \varphi(f(X_p))] \wedge \underline{X}^{(i+1)} = f(\underline{X}^{(i)})$, for $i > 0$, can be seen by the fact that $X^{(i)} = X^{(i-1)}A + B$ (by the previous transition). Hence there exist values X_p such that $X^{(i)} = f(X_p)$. Now it remains to be shown

that X_p and $X^{(i-1)}$ are the same values. But since A is invertible, we have $X_p = X^{(i-1)} = (X^{(i)} - B)A^{-1}$.

Hence it is enough to unfold the loop once and consider the first transition separately. Obviously, all incoming and outgoing transitions will have to be duplicated, but this does not change the flatness of the control structure. The result is a machine with the following control structure and with counters $\underline{X}$:

$$q_0 \xrightarrow{\exists x_i \,.\, [I_q(X) \wedge \varphi(X)] \wedge \underline{X}' = \underline{f}(\underline{X})} q_1$$
$$q_1 \xrightarrow{\exists x_i \exists X_p \,.\, [X = f(X_p) \wedge \varphi(f(X_p))] \wedge \underline{X}' = \underline{f}(\underline{X})} q_1$$

Obviously, all properties (reachability, termination) of the original machine are preserved by the transformation.

We can therefore restrict w.l.o.g. to the case where $\sum_{j=1}^{n} a_{ji} > 0$ for all $1 \leq i \leq n$. Then either:

- $\sum_{j=1}^{n} a_{ji} > 1$, in which case $0 \leq x_i^{(m)} \leq \frac{\alpha - \sum_{i=1}^{n} b_i}{\sum_{j=1}^{n} a_{ji} - 1}$, or
- $\sum_{j=1}^{n} a_{ji} = 1$, i.e., $a_{ki} = 1$ for some k and $a_{ji} = 0$ for all $j \neq k$.

This (static) case split partitions the set of counters X into two disjoint subsets: a set of counters Y for which the first case applies and which are bounded by a constant throughout the execution of the automaton, and a set of counters Z for which the second case applies and which must occur exactly once in the computation of X'; i.e., for each $z \in Z$ there exists exactly one $x \in X$ such that $x' = z + g$ where g is a linear combination not involving z.

We now distinguish three cases. (1) If $x \in Y$, the value of z is also bounded by a constant. Otherwise, (2) if $x \in Z$ and g contains another occurrence of a variable t from Z, this means that there exists another variable s from Z whose next value depends only on values from Y, or else there would be a variable from Z occurring in two places. In this case, the value of s is also bounded by a constant. The last case (3) is $z' = t + g(Y)$ for $z, t \in Z$. In the first two cases, the partition can be modified by moving bounded variables from Z to Y until a fixpoint is reached.

According to the resulting partition (Y, Z), the values taken by the counters at each iteration have the following properties:

- Y range over an effectively computable finite set of values $V = \{v_1, \ldots, v_N\}$,
- Y' are linear combinations of Y,
- $Z' = Z + \delta$ where δ are linear combinations of Y.

Since the values taken by Y are bounded, they can be encoded in the control of a new counter machine. Given a self loop $q \xrightarrow{\varphi \wedge \psi} q$ where $\psi = \bigwedge_{1 \leq i \leq n} x_i' = f_i(X)$ is the same as in Definition 2, and a partition of the counters into (Y, Z) that satisfies the requirements above, we can build a counter machine $A_{sim} = \langle V^{\|Y\|} \cup \{q_0\}, Z, q_0, \delta_{sim} \rangle$ where δ_{sim} is defined as follows:

- $q_0 \xrightarrow{\top} \mathbf{v}$ if $\mathbf{v} \in V^{\|Y\|}$ is an initial tuple of values for Y,
- for all $\mathbf{v}, \mathbf{v}' \in \mathbb{Z}^{\|Y\|}$ and $\mathbf{w}, \mathbf{w}' \in \mathbb{Z}^{\|Z\|}$, we have:

$$\mathbf{v} \xrightarrow[A_{sim}]{\varphi[\mathbf{v}/Y] \wedge \psi[\mathbf{v}/Y, \mathbf{v}'/Y', Z + \delta/Z'] \wedge Z' = Z + \delta} \mathbf{v}'$$

Notice that the new transition relation δ_{sim} is deterministic since $\mathbf{v}'$ and δ are linear combinations of $\mathbf{v}$. This means that the control structure of A_{sim} is in fact a loop, corresponding to a finite number of unfoldings of $q \xrightarrow{\varphi \wedge \psi} q$. The new machine simulates the original loop in the sense that any execution of the former corresponds to an execution of the latter and vice-versa. Moreover, the set of configurations of the original loop is in a one-to-one relation with the set of configurations of A_{sim}.

Let M be the length of the loop constituting the control of A_{sim}. This is an effectively computable constant, bounded by $N^{|\gamma|}$, the number of control states. In other words, A_{sim} is of the form: $\mathbf{v}_0 \xrightarrow{\varphi_0 \wedge Z' = Z + \delta_0} \mathbf{v}_1 \xrightarrow{\varphi_1 \wedge Z' = Z + \delta_1} \mathbf{v}_2 \ldots \rightarrow \mathbf{v}_{M-1} \xrightarrow{\varphi_{M-1} \wedge Z' = Z + \delta_{M-1}} \mathbf{v}_0$ where $\varphi_i = \varphi[\mathbf{v}_i/Y] \wedge \psi[\mathbf{v}_i/Y, \mathbf{v}'_i/Y', Z + \delta_i/Z']$. The relation between the input Z and output Z' values of the counters of A_{sim} can now be defined by the Presburger formula

$$\exists l \geq 0 \,.\, \bigvee_{j=0}^{M-1} Z' = Z + l \sum_{i=0}^{M-1} \delta_i + \sum_{i=0}^{j} \delta_i \wedge \tag{1}$$

$$\forall 0 \leq m < lM + j \,.\, \varphi_{m \mod M}(Z + \tfrac{m}{M} \sum_{i=0}^{M-1} \delta_i + \sum_{i=0}^{m \mod M} \delta_i) \tag{2}$$

where $\frac{m}{M}$ denotes integer division. Intuitively, the formula from the first row gives the difference between Z' and Z whereas the second one ensures that the guards are satisfied all along the way. Notice that $\varphi_j(Z + \frac{m}{M} \sum_{i=0}^{M-1} \delta_i + \sum_{i=0}^{m \mod M} \delta_i)$, $0 \leq i < M$, are indeed formulae of Presburger arithmetic provided that φ is.

Given a flat linear positive restrictive automaton, one can compute the above formula for each individual loop. The reachability and termination problems for these automata can be reduced to satisfiability of Presburger formulae. □

4 Abstract Semantics of Programs with Lists

A common way of representing heaps compactly consists in mapping an entire list segment with no edges incoming into the middle of it into a special (abstract) node. This idea constitutes the basis of our abstraction too. Let $\mathcal{N}$ be a set of *abstract nodes* and X be a set of *counter variables*, one for each node. We shall first define the abstract structure of heaps.

Definition 3 An *abstract structure* is a tuple $\overline{H} = \langle \overline{N}, \overline{S}, \overline{V} \rangle$ where:

- $\overline{N} \subseteq \mathcal{N}$ is a set of abstract nodes,
- $\overline{S} : \overline{N} \rightarrow \overline{N}_\perp$ is a successor mapping, and
- $\overline{V} : PVar \rightarrow \overline{N}_\perp$ is a variable mapping.

A *cut-point* in $\overline{H}$ is a node $n \in \overline{N}$ such that there exist $n_1, n_2 \in \overline{N}$ with $n_1 \neq n_2$ and $\overline{S}(n_1) = \overline{S}(n_2) = n$ or there exists $u \in PVar$ with $\overline{V}(u) = n$. An abstract structure is said to be in *normal form* iff each node $n \in \overline{N}$ is a *cut-point* in $\overline{H}$ that is reachable from some $u \in PVar$, i.e., $u \xrightarrow[\overline{H}]{*} n$.

Intuitively, each abstract node corresponds to a set of concrete nodes, and the counter corresponding to each node gives the number of nodes in this set. For abstract structures in normal form, we do not allow sequences of successive abstract nodes that are neither pointed by a variable, nor have their in-degree greater than one. This condition is needed in order to ensure that any such abstract structure defined over a finite set of variables is finite. We use $\overline{\mathcal{H}}(PVar)$ to denote the set of all abstract structures with variables from $PVar$ and $\overline{\mathcal{H}}_{nf}(PVar)$ to denote the subset of $\overline{\mathcal{H}}(PVar)$ containing abstract structures in normal form only.

Lemma 1 *Let $PVar = \{u_1, \ldots, u_n\}$ be a set of variables, and $\overline{H} = \langle \overline{N}, \overline{S}, \overline{V} \rangle$ be an abstract structure in normal form such that $dom(\overline{V}) \subseteq PVar$. Then $\|\overline{N}\| \leq 2n$. Moreover, $\|\overline{\mathcal{H}}_{nf}(PVar)\|$ is of the order of $2^{\Theta(nlog(n))}$.*

Proof We show that $\|\overline{N}\| \leq 2n$ by induction on the cardinality n of $PVar$. For $n = 1$, it is obvious that the abstract structures in normal form can have at most two nodes: either there is no node at all, or the unique variable points to a node which points to null or to itself, or the variable points to a first node which points to a second node which points to itself. Now, let us suppose that $\|\overline{N}\| \leq 2n$ holds for abstract structures in normal form with n variables. If we have one more variable u_{n+1}, it either points to a part of the heap completely separated from the other n variables, or not. In the former case, there can be at most two more nodes using the same reasoning as in the base case. In the latter case, there can be at most one node m (pointed to by u_{n+1}) which is not reachable from the other variables. Furthermore, the successor of m must be a cut-point m'. If m' is pointed to by a variable, we can remove m and u_{n+1} and apply the induction hypothesis on the remaining heap which is in normal form since m' is still a cut-point. The same can also be done if m' is pointed to by at least two other nodes distinct from m. If m' is pointed to by exactly one other node m'', it is no more a cut-point after removing u_{n+1} and m. However, if we remove m' by letting m'' point to the successor of m', then the obtained heap is in normal form and we can apply the induction hypothesis on it yielding the desired bound.

For the second part, we compute an upper and lower bound on the number of heaps with $2n$ nodes. For the upper bound, we note that an abstract structure, not necessarily in normal form, with $2n$ nodes can be uniquely encoded by the following mappings:

- $s : \{1, \ldots, 2n\} \to \{1, \ldots, 2n, \bot\}$ is the partial successor function, i.e., $s(n) = \bot$ iff the node n has no successor,
- $v : \{1, \ldots, n\} \to \{1, \ldots, 2n, \bot\}$ maps variables into nodes, i.e., $v(i) = \bot$ iff the i-th variable is undefined.

Taking this into account, the number of abstract structures, not necessarily in normal form, with $2n$ nodes is at most $(2n+1)^{2n} \cdot (2n+1)^n = (2n+1)^{3n}$. At the same time, this formula gives an upper bound on the number of abstract structures in normal form with $2n$ nodes or less as well. This is because for each abstract structure $\overline{H}$ in normal form with $m < 2n$ nodes there is at least one abstract structure not in normal form which has $2n$ nodes and which is equal with $\overline{H}$, up to having $2n - m$ garbage nodes (i.e., nodes unreachable from the pointer variables).

For the lower bound, we consider only trees with n nodes, all of which are pointed to by a variable. The number of unlabelled trees with n nodes is the Catalan number $C_n = \frac{(2n-2)!}{n!(n-1)!}$, and the number of possible variable assignments is $n!$. Hence the number of labelled trees is $n \cdot (n+1) \cdot \ldots \cdot (2n-2) \geq n^{n-1}$. So n^{n-1} is a lower bound on the number of abstract structures in normal form with $2n$ nodes. Hence the number of abstract structures in normal form pointed to by n pointer variables is of the order of $2^{\Theta(nlog(n))}$. □

Let us now define our first abstraction function, denoted by α_s, that maps concrete heaps into abstract structures. Given a concrete heap $H = \langle N, S, V, D \rangle$, let $R_H \subseteq N \times N$ be a relation on the set of nodes defined such that $n_1 R_H n_2$ holds for any $n_1, n_2 \in N$ iff $n_1 \underset{H}{\to} n_2 \wedge \neg cut_H(n_2)$. We denote by $\sim_H$ the reflexive, symmetric, and transitive closure of R_H. The H subscript shall be further omitted for simplicity. For a node $n \in N$, we denote by $[n]$ the equivalence class of n with respect to $\sim$, also referred to as a *list segment*. The *quotient heap* $H_{/\sim} = \langle N_{/\sim}, S_{/\sim}, V_{/\sim} \rangle$ is defined as follows:

- $N_{/\sim} = \{[n] \mid n \in N\}$,
- for all $n, m \in N$, $S_{/\sim}([n]) = [m]$ iff $\exists n_0 \in [n]\ \exists m_0 \in [m]\ .\ S(n_0) = m_0 \wedge cut_H(m_0)$,
- for all $u \in PVar$, $n \in N$, $V_{/\sim}(u) = [n]$ iff $V(u) \in [n]$, and
- $S_{/\sim}$ and $V_{/\sim}$ are undefined, otherwise.

Note that $S_{/\sim}$ and $V_{/\sim}$ are well defined partial functions. Indeed, assume that for some $n \in N$, $S_{/\sim}$ maps $[n]$ into two different equivalence classes, call them $[m]$ and $[p]$. This would imply that there are two nodes $n_1, n_2 \in [n]$ such that $n_1 \underset{H}{\rightarrow} m_0$ and $n_2 \underset{H}{\rightarrow} p_0$ where $m_0 \in [m]$ is the cut-point at the beginning of the list segment $[m]$, and $p_0 \in [p]$ is the cut-point at the beginning of the list segment $[p]$. Since $[m]$ and $[p]$ are different segments, $m_0 \neq p_0$. Now, note that since nodes in a list segment are linked together, each of them has just one successor, and only the first node of a list segment is a cut-point, only the last node of a list segment may point to a cut-point. However, then it must be the case that $n_1 = n_2$, and hence also $m_0 = p_0$, which is a contradiction. The argument for $V_{/\sim}$ is straightforward.

In Figure 4 (a), an example of a concrete heap with its cut-points is given. For example, the nodes n_1 and n_2 form an equivalence class.

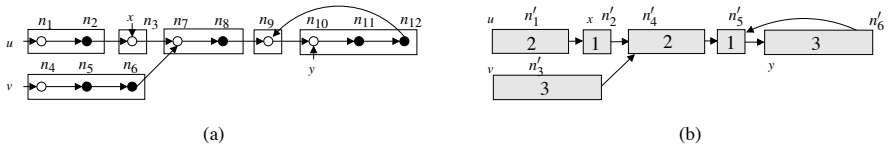

Fig. 4 Concrete and abstract structures. The cut-points are n_1, n_3, n_4, n_7, n_9 and n_{10}.

For an equivalence class $[n] \in N_{/\sim}$, we denote by $hd([n])$, $tl([n])$ the head and tail of the list segment, respectively, and by $[n] \circ [m]$ the concatenation of two list segments.

Definition 4 Let $H = \langle N, S, V, D \rangle$ be a concrete heap and $H_{/\sim} = \langle N_{/\sim}, S_{/\sim}, V_{/\sim} \rangle$ its quotient. An abstract structure $\overline{H} = \langle \overline{N}, \overline{S}, \overline{V} \rangle$ is said to be a *structural abstraction* of H if and only if there exists a bijective function $\beta : N_{/\sim} \cup \{\bot\} \rightarrow \overline{N} \cup \{\bot\}$ such that $\beta(\bot) = \bot$, and for all $u \in PVar$:

- $\overline{S}(\beta([n])) = \beta(S_{/\sim}([n]))$ and
- $\overline{V}(u) = \beta(V_{/\sim}(u))$.

Note that an abstract structure that is a structural abstraction of some concrete heap is necessarily in normal form. Two abstract structures that differ only in the naming of nodes and counter variables are semantically equivalent in the sense that they are abstractions of the same set of concrete heaps. In practice, this increases the number of abstract structures generated by a symbolic state exploration tool. This problem can be overcome by choosing a canonical representation of abstract structures as described, e.g., in [23].

We define the structural abstraction function $\alpha_s : \mathcal{H}(PVar) \rightarrow \overline{\mathcal{H}}_{nf}(PVar)$, $\alpha_s(H) = \overline{H}$ iff $\overline{H}$ is the canonical representative of a structural abstraction of H. Dually, the *concretisation* of an abstract structure $\overline{H}$ is the set of concrete heaps whose structural abstraction is $\overline{H}$, i.e., $\gamma_s(\overline{H}) = \{H \mid \alpha_s(H) = \overline{H}\}$.

Furthermore, let $\overline{H} = \langle \overline{N}, \overline{S}, \overline{V} \rangle$ be an abstract structure in normal form and $\nu : \overline{N} \rightarrow \mathbb{N}^+$ a mapping of nodes to positive natural numbers. By $\nu(\overline{H})$, we denote the set of concrete heaps obtained by replacing each node $n \in \overline{N}$ by a list segment of length $\nu(n)$ and data arbitrarily chosen from $\mathfrak{D}$. Clearly, $\gamma_s(\overline{H}) = \bigcup_{\nu:\overline{N}\rightarrow\mathbb{N}^+} \nu(\overline{H})$.

For example, the concrete heap in Figure 4 (a) is one possible concretisation of the abstract structure in Figure 4 (b), for the valuation: $\nu(n'_1) = 2$, $\nu(n'_2) = 1$, $\nu(n'_3) = 3$, $\nu(n'_4) = 2$, $\nu(n'_5) = 1$, $\nu(n'_6) = 3$.

4.1 Data-Insensitive Programs

This section is devoted to a description of counter automata that abstract the behaviour of data-insensitive programs with lists. We formalise correctness of our construction by proving bisimulation between the semantics of a data-insensitive list program and the semantics of a counter automaton. This entails a strong preservation of temporal logic properties. In particular, safety and termination are strongly preserved by the counter automata, meaning that one can prove as well as disprove safety and termination properties of the programs based on the behaviour of their representation by the counter automata.

Consider a data-insensitive list program with k pointer variables and l counter variables, i.e., $\|PVar\| = k$ and $\|IVar\| = l$. We construct a counter automaton $A = \langle Q, X, \xrightarrow{s} \rangle$ with $2k + l$ counters as follows. The control states Q of the counter automaton are elements of the set $Lab \times (\overline{\mathcal{H}}_{nf}(PVar) \cup \{\overline{H}_{err}\})$ where $\overline{H}_{err} \notin \overline{\mathcal{H}}(PVar)$ is a special sink error state. Let $\mathcal{N} = \bigcup\{\overline{N} \mid \langle \overline{N}, \overline{S}, \overline{V} \rangle \in \overline{\mathcal{H}}_{nf}(PVar)\}$ be the set of nodes used in the structural abstraction. The counters are $X = \{x_n \mid n \in \mathcal{N}\} \cup IVar$, i.e., one counter for each node used in the structural abstraction plus the counter variables from the original program. The transitions are given by triples $q \xrightarrow{\varphi} q'$ such that (1) $q = \langle l, \overline{H} \rangle$, (2) $q' = \langle l', \overline{H'} \rangle$, (3) there is a statement $l : s; l'$ in the program, and (4) it is the case that $\overline{H} \xrightarrow[s]{\varphi} \overline{H'}$ where the relation $\xrightarrow[s]{\varphi}$ is described by the structural rules given in Figures 5 and 7. More precisely, Figures 5 and 7 contain rules A_i which correspond to the rules C_i from Figure 3. Sometimes several rules in the abstract semantics are needed for one rule in the concrete semantics. The rules for data manipulation (C_{11}, C_{12}) are not needed in the abstract semantics for data-insensitive programs. In order to simplify the treatment of the different cases in the rules of Figures 5 and 7, they are based on two low-level operations for merging and splitting abstract nodes, which are explained in the following paragraphs.

Intuitively, when the effect of some pointer manipulating statement is simulated over an abstract structure in normal form, it may happen that we obtain an abstract structure $\overline{H} = \langle \overline{N}, \overline{S}, \overline{V} \rangle$ in which there are two abstract nodes n and m such that m is a successor of n, and m is not a cut-point. In such a case, we need to merge the two abstract nodes n and m in order to re-normalise the abstract structure. For this purpose, we use the *merging function* $\mu : \overline{\mathcal{H}}(PVar) \times \mathcal{N} \times \mathcal{N} \to \overline{\mathcal{H}}(PVar)$ defined as $\mu(\overline{H}, n, m) = \langle \overline{N'}, \overline{S}\downarrow_{\overline{N'}} [n \to \overline{S}(m)], \overline{V} \rangle$ where $\overline{N'} = \overline{N} \setminus \{m\}$.

Furthermore, when simulating the effect of the statement $u := w.next$, we need to split the abstract node n pointed to by w into two nodes n and m provided that the value of its corresponding counter is greater than one (i.e., $x_n > 1$). For this purpose, we use the *splitting* function $\sigma : \overline{\mathcal{H}}(PVar) \times \mathcal{N} \times \mathcal{N} \to \overline{\mathcal{H}}(PVar)$ defined as $\sigma(\overline{H}, n, m) = \langle \overline{N} \cup \{m\}, \overline{S'}, \overline{V} \rangle$ where $\overline{S'} = \big(\overline{S} \setminus \{(n, \overline{S}(n))\}\big) \cup \{(n, m), (m, \overline{S}(n))\}$.

The semantics of tests ($u = v$ and $u = null$) is similar to the concrete case, and the same holds for the conditional statements, `while` loops, and counter manipulating statements.

For an illustration of the translation, the reader is referred to the list reversal example in Figure 8.

Now we can state the main theorem of this section, establishing *bisimilarity* between a data-insensitive program manipulating lists and the counter automaton modelling it. Given

$$\frac{\overline{V}(u)=\bot}{\overline{H}\xrightarrow[u:=null]{true}\overline{H}}\ A_1 \qquad \frac{\exists w\in PVar\setminus\{u\}\quad \overline{V}(w)=\overline{V}(u)\neq\bot}{\overline{H}\xrightarrow[u:=null]{true}\langle\overline{N},\overline{S},\overline{V}[u\to\bot]\rangle}\ A_2$$

$$\frac{\begin{array}{l}\overline{V}(u)=n\in\overline{N}\quad \forall w\in PVar\setminus\{u\}\ .\ \overline{V}(w)\neq n\\ \exists m,p\in\overline{N}\ .\ p\neq m\ \wedge\ \overline{S}(m)=\overline{S}(p)=n\end{array}}{\overline{H}\xrightarrow[u:=null]{true}\langle\overline{N},\overline{S},\overline{V}[u\to\bot]\rangle}\ A'_2 \qquad \frac{\begin{array}{l}\overline{V}(u)=n\in\overline{N}\quad \forall w\in PVar\setminus\{u\}\ .\ \overline{V}(w)\neq n\\ \exists m\in\overline{N}\setminus\{n\}\ .\ \overline{S}(m)=n\\ \forall p\in\overline{N}\setminus\{m\}\ .\ \overline{S}(p)\neq n\end{array}}{\overline{H}\xrightarrow[u:=null]{x'_m=x_m+x_n}\mu(\langle\overline{N},\overline{S},\overline{V}[u\to\bot]\rangle,m,n)}\ A''_2$$

$$\frac{\begin{array}{l}\overline{V}(u)=n\in\overline{N}\quad \forall w\in PVar\setminus\{u\}\ .\ w\not\xrightarrow[\overline{H}]{*}n\\ \overline{S}(n)\in\{\bot,n\}\qquad\qquad \overline{N'}=\overline{N}\setminus\{n\}\end{array}}{\overline{H}\xrightarrow[u:=null]{true}\langle\overline{N'},\overline{S}\downarrow_{\overline{N'}},\overline{V}[u\to\bot]\rangle}\ A_3 \qquad \frac{\begin{array}{l}\overline{V}(u)=n\in\overline{N}\quad \forall w\in PVar\setminus\{u\}\ .\ w\not\xrightarrow[\overline{H}]{*}n\\ \overline{S}(n)=m\in\overline{N}\setminus\{n\}\\ \exists w\in PVar\setminus\{u\}\ .\ V(w)=m\qquad \overline{N'}=\overline{N}\setminus\{n\}\end{array}}{\overline{H}\xrightarrow[u:=null]{true}\langle\overline{N'},\overline{S}\downarrow_{\overline{N'}},\overline{V}[u\to\bot]\rangle}\ A'_3$$

$$\frac{\begin{array}{l}\overline{V}(u)=n\in\overline{N}\quad \forall w\in PVar\setminus\{u\}\ .\ w\not\xrightarrow[\overline{H}]{*}n\\ \overline{S}(n)=m\in\overline{N}\setminus\{n\}\\ \forall w\in PVar\setminus\{u\}\ .\ \overline{V}(w)\neq m\\ \exists p,q\in\overline{N}\setminus\{n\}\ .\ p\neq q\wedge\overline{S}(p)=m\wedge\overline{S}(q)=m\\ \overline{N'}=\overline{N}\setminus\{n\}\end{array}}{\overline{H}\xrightarrow[u:=null]{true}\langle\overline{N'},\overline{S}\downarrow_{\overline{N'}},\overline{V}[u\to\bot]\rangle}\ A''_3 \qquad \frac{\begin{array}{l}\overline{V}(u)=n\in\overline{N}\quad \forall w\in PVar\setminus\{u\}\ .\ w\not\xrightarrow[\overline{H}]{*}n\\ \overline{S}(n)=m\in\overline{N}\setminus\{n\}\\ \forall w\in PVar\setminus\{u\}\ .\ \overline{V}(w)\neq m\\ \exists p\in\overline{N}\setminus\{n,m\}\ .\ \overline{S}(p)=m\\ \forall q\in\overline{N}\setminus\{n,p\}\ .\ \overline{S}(q)\neq m\qquad \overline{N'}=\overline{N}\setminus\{n\}\end{array}}{\overline{H}\xrightarrow[u:=null]{x'_p=x_p+x_m}\mu(\langle\overline{N'},\overline{S}\downarrow_{\overline{N'}},\overline{V}[u\to\bot]\rangle,p,m)}\ A'''_3$$

$$\frac{\begin{array}{l}\overline{V}(u)=n\in\overline{N}\quad \forall w\in PVar\setminus\{u\}\ .\ w\not\xrightarrow[\overline{H}]{*}n\\ \overline{S}(n)=m\in\overline{N}\setminus\{n\}\\ \forall w\in PVar\setminus\{u\}\ .\ \overline{V}(w)\neq m\\ \forall p\in\overline{N}\setminus\{n,m\}\ .\ \overline{S}(p)\neq m\\ \overline{N'}=\overline{N}\setminus\{n,m\}\end{array}}{\overline{H}\xrightarrow[u:=null]{true}\langle\overline{N'},\overline{S}\downarrow_{\overline{N'}},\overline{V}[u\to\bot]\rangle}\ A'''_3$$

Fig. 5 The data-insensitive counter automata semantics of $u := null$ on an abstract structure $\overline{H}=\langle\overline{N},\overline{S},\overline{V}\rangle$ in normal form—see Fig. 6 for an illustration of the different cases

a data-insensitive, list manipulating program P, let $\langle\mathcal{S},\xrightarrow{c}\rangle$ be its concrete semantics with the set of states $\mathcal{S}=Lab\times\mathcal{H}(PVar)\times(IVar\to\mathbb{Z})$ and $\xrightarrow{c}$ being its transition relation. Let $\overline{\mathcal{S}}=Q\times(X\to\mathbb{Z})$ be the set of all configurations of the corresponding counter automaton and $\xrightarrow{s}$ its transition relation. Let $\rhd_s\subseteq\mathcal{S}\times\overline{\mathcal{S}}$ be the relation defined such that $(l,H,\iota)\rhd_s(\overline{l},\overline{H},\overline{\nu})$ holds for a concrete heap $H=\langle N,S,V,D\rangle$, an abstract structure $\overline{H}=\langle\overline{N},\overline{S},\overline{V}\rangle$ in normal form, program labels l and $\overline{l}$, and valuations $\iota:IVar\to\mathbb{Z}$ and $\overline{\nu}:X\to\mathbb{Z}$ iff one of the two following cases happens:

- $l=\overline{l}$, $\overline{H}$ is a structural abstraction of H based on a bijection $\beta:N_{/\sim}\cup\{\bot\}\to\overline{N}\cup\{\bot\}$ in accordance with Definition 4, $\overline{\nu}\downarrow_{IVar}=\iota$, and $\forall n\in\overline{N}\ .\ \overline{\nu}(x_n)=\|\beta^{-1}(n)\|$; or
- $l=\overline{l}\ \wedge\ H=H_{err}\ \wedge\ \overline{H}=\overline{H}_{err}$.

Theorem 2 *The relation $\rhd_s$ is a bisimulation between $\langle\mathcal{S},\xrightarrow{c}\rangle$ and $\langle\overline{\mathcal{S}},\xrightarrow{s}\rangle$.*

Proof We prove the theorem by considering all possible different statements in the program. The case of H_{err} and $\overline{H}_{err}$ is trivial. Suppose that we are given $(l,H,\iota)\rhd_s(l,\overline{H},\overline{\nu})$ such that

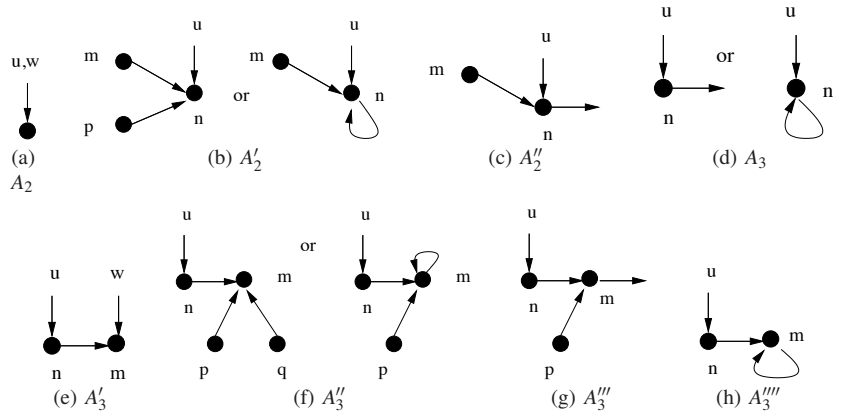

Fig. 6 An illustration of the different cases of the counter automata semantics of $u := null$ from Fig. 5

$H \neq H_{err}$, and let $\beta : N_{/\sim} \cup \{\bot\} \to \overline{N} \cup \{\bot\}$ be the bijection from Definition 4 linking $H_{/\sim}$ and $\overline{H}$.

We need to show that for each statement s, (1) $(l,H,\iota) \xrightarrow{s} (l',H',\iota')$ implies $(l,\overline{H},\overline{\nu}) \xrightarrow{s} (l',\overline{H}',\overline{\nu}')$ and $(l',H',\iota') \rhd_s (l',\overline{H}',\overline{\nu}')$, and (2) $(l,\overline{H},\overline{\nu}) \xrightarrow{s} (l',\overline{H}',\overline{\nu}')$ implies $(l,H,\iota) \xrightarrow{s} (l',H',\iota')$ and $(l',H',\iota') \rhd_s (l',\overline{H}',\overline{\nu}')$.

For statements which are assignments involving integer variables, this is obvious. For statements which are guards involving integer variables, this is also obvious. For guards of the type $u = v$ involving pointer variables, this follows directly from Claim (1) proven below. The case of guards of the type $u = null$ is analogical.

Claim (1) Given an abstract structure $\overline{H} = \langle \overline{N}, \overline{S}, \overline{V} \rangle$ such that $\overline{H}$ is a structural abstraction of a concrete heap $H = \langle N, S, V, D \rangle$, we have for all $u, w \in PVar$, $V(u) = V(w)$ iff $\overline{V}(u) = \overline{V}(w)$.

Proof $V(u) = V(w) = \bot$ iff $V_{/\sim}(u) = V_{/\sim}(w) = \bot$ iff $\overline{V}(u) = \overline{V}(w) = \bot$, by Definition 4. If $V(u) = V(w) \neq \bot$, then $\overline{V}(u) = \overline{V}(w) \neq \bot$ follows. Dually, if $\overline{V}(u) = \overline{V}(w) = \overline{n}$, then $V_{/\sim}(u) = V_{/\sim}(w) = \beta^{-1}(\overline{n})$. Then either $V(u) = V(w)$, or $V(u) \neq V(w)$ and $V(u) \sim_H V(w)$. The latter case leads to a contradiction due the fact that $V(w)$ is a cut-point. □

For the other cases, we need another property.

Claim (2) Given a concrete heap $H = \langle N, S, V, D \rangle$ and its structural abstraction $\overline{H}$ based on a bijection β, for all $n, m \in N$ such that $cut_H(m)$ and $cut_H(n)$, $n \xrightarrow[H]{*} m$ iff $\beta([n]) \xrightarrow[\overline{H}]{*} \beta([m])$.

Proof "⇒" We show that for all $n, m \in N$ such that $cut_H(m)$ and $cut_H(n)$, $n \xrightarrow[H]{*} m$ implies $\beta([n]) \xrightarrow[\overline{H}]{*} \beta([m])$ by induction on the number c of cut-points different from n and m which appear along the shortest path from n to m. For $c = 0$, we have two cases: First, $n = m$, in which case, we trivially have $\beta([n]) = \beta([m])$. Second, $n \neq m$. In this case, there must be some $n' \in [n]$ such that $n \xrightarrow[H]{*} n' \xrightarrow[H]{} m$, and we are done since $[n] = [n']$ and $S_{/\sim}([n']) = [m]$. Assume now that the implication holds for all pairs of cut-points linked by shortest paths of length at most c. Let n and m be cut-points linked by a shortest path of length $c+1$. Then,

$$\frac{\overline{V}(u)=\bot}{\overline{H}\xrightarrow[u:=w]{true}\langle\overline{N},\overline{S},\overline{V}[u\to\overline{V}(w)]\rangle}\ A_4 \qquad \frac{\overline{V}(u)=\bot \quad n\in\mathcal{N}\setminus\overline{N}}{\overline{H}\xrightarrow[u:=new]{x'_n=1}\langle\overline{N}\cup\{n\},\overline{S}[n\to\bot],\overline{V}[u\to n]\rangle}\ A_5$$

$$\frac{\overline{V}(w)=\bot}{\overline{H}\xrightarrow[u:=w.next]{true}\overline{H}_{err}}\ A_6 \qquad \frac{\overline{V}(u)=\bot \quad \overline{V}(w)=n\in\overline{N}}{\overline{H}\xrightarrow[u:=w.next]{x_n=1}\langle\overline{N},\overline{S},\overline{V}[u\to\overline{S}(n)]\rangle}\ A_7$$

$$\frac{\overline{V}(u)=\bot \quad \overline{V}(w)=n\in\overline{N} \quad m\in\mathcal{N}\setminus\overline{N'}}{\overline{H}\xrightarrow[u:=w.next]{x_n>1\wedge x'_m=x_n-1}\sigma(\langle\overline{N},\overline{S},\overline{V}[u\to m]\rangle,n,m)}\ A'_7 \qquad \frac{\overline{V}(u)=\bot}{\overline{H}\xrightarrow[u.next:=null]{true}\overline{H}_{err}}\ A_8$$

$$\frac{\overline{V}(u)=n\in\overline{N} \quad \overline{S}(n)\in\{\bot,n\}}{\overline{H}\xrightarrow[u.next:=null]{x'_n=1}\langle\overline{N},\overline{S}[n\to\bot],\overline{V}\rangle}\ A_9 \qquad \frac{\begin{array}{l}\overline{V}(u)=n\in\overline{N} \quad \overline{S}(n)=m\in\overline{N}\setminus\{n\}\\ \exists v\in PVar\setminus\{u\}\ .\ \overline{V}(v)=m\end{array}}{\overline{H}\xrightarrow[u.next:=null]{x'_n=1}\langle\overline{N},\overline{S}[n\to\bot],\overline{V}\rangle}\ A'_9$$

$$\frac{\begin{array}{l}\overline{V}(u)=n\in\overline{N} \quad \overline{S}(n)=m\in\overline{N}\setminus\{n\}\\ \forall v\in PVar\ .\ \overline{V}(v)\neq m\\ \exists p,q\in\overline{N}\setminus\{n\}\ .\ p\neq q\wedge\overline{S}(p)=\overline{S}(q)=m\end{array}}{\overline{H}\xrightarrow[u.next:=null]{x'_n=1}\langle\overline{N},\overline{S}[n\to\bot],\overline{V}\rangle}\ A''_9 \qquad \frac{\begin{array}{l}\overline{V}(u)=n\in\overline{N} \quad \overline{S}(n)=m\in\overline{N}\setminus\{n\}\\ \forall v\in PVar\ .\ \overline{V}(v)\neq m\\ \exists p\in\overline{N}\setminus\{n,m\}\ .\ \overline{S}(p)=m\\ \forall q\in\overline{N}\setminus\{n,p\}\ .\ \overline{S}(q)\neq m\end{array}}{\overline{H}\xrightarrow[u.next:=null]{x'_n=1\wedge x'_p=x_p+x_m}\mu(\langle\overline{N},\overline{S}[n\to\bot],\overline{V}\rangle,p,m)}\ A'''_9$$

$$\frac{\begin{array}{l}\overline{V}(u)=n\in\overline{N} \quad \overline{S}(n)=m\in\overline{N}\setminus\{n\}\\ \forall v\in PVar\ .\ \overline{V}(v)\neq m\\ \forall p\in\overline{N}\setminus\{n,m\}\ .\ \overline{S}(p)\neq m\\ \overline{N'}=\overline{N}\setminus\{m\}\end{array}}{\overline{H}\xrightarrow[u.next:=null]{x'_n=1}\langle\overline{N'},\overline{S}\downarrow_{\overline{N'}}[n\to\bot],\overline{V}\rangle}\ A''''_9 \qquad \frac{\overline{V}(u)=n\in\overline{N} \quad \overline{S}(n)=\bot}{\overline{H}\xrightarrow[u.next:=w]{true}\langle\overline{N},\overline{S}[n\to\overline{V}(w)],\overline{V}\rangle}\ A_{10}$$

$$\frac{}{\overline{H}_{err}\xrightarrow[s]{true}\overline{H}_{err}}\ A_{13}$$

Fig. 7 The data-insensitive counter automata semantics of program statements other than $u := null$

there must be some cut-point $n' \in N$ such that $n \xrightarrow[H]{*} n' \xrightarrow[H]{} m$ and there is no further cut-point on the shortest path between n' and m. We are done by considering the induction hypothesis over $n \xrightarrow[H]{*} n'$ and the base case over $n' \xrightarrow[H]{} m$.

"$\Leftarrow$" We show that for all $n,m \in N$ such that $cut_H(m)$ and $cut_H(n)$, $\beta([n]) \xrightarrow[\overline{H}]{*} \beta([m])$ implies $n \xrightarrow[H]{*} m$ by induction on the length c of the shortest path between $\beta([n])$ and $\beta([m])$ in $\overline{H}$. If $c = 0$, it must be the case that $n = m$ since both n and m are cut-points, and we are done. If $c = 1$, there must be some $n' \in [n]$ such that $n' \xrightarrow[H]{} m$. Since $n \xrightarrow[H]{*} n'$, we have $n \xrightarrow[H]{*} m$. Assume now that the implication holds for all paths of length up to c. Since any path of length $c+1$ can easily be split to two paths of length at most c, we may apply the induction hypothesis and we are done. □

We now consider all statements involving pointer variables. We suppose that they go from l to l'.

Case $s = [\ u := null\ ]$. There are several different sub-cases:

– $V(u) = \bot$. We have $V(u) = \bot$ iff $\overline{V}(u) = \bot$, by Definition 4. Therefore, rule C_1 applies to H (without changing it) iff the analogical rule A_1 applies to $\overline{H}$ (again, without changing it), and it is clear that $(l', H, \iota) \rhd_s (l', \overline{H}, \overline{\nu})$.

– $V(u) \neq \bot$ and $\exists w \in PVar \setminus \{u\} \,.\, V(w) = V(u)$. We have $V(u) = V(w) \neq \bot$ iff $\overline{V}(u) = \overline{V}(w) \neq \bot$, by Claim (1). In this case, rule C_2 applies to H iff rule A_2 applies to $\overline{H}$, and it can easily be checked that $(l', \langle N, S, V[u \to \bot], D\rangle, \iota) \rhd_s (l, \langle \overline{N}, \overline{S}, \overline{V}[u \to \bot]\rangle, \overline{\nu})$.

– $V(u) \neq \bot$ and $\forall w \in PVar \setminus \{u\} \,.\, V(w) \neq V(u)$ and $\exists w \in PVar \setminus \{u\} \,.\, w \xrightarrow[H]{*} V(u)$. Let $w \in PVar \setminus \{u\}$ be any variable such that $w \xrightarrow[H]{*} V(u)$. Since $V(u)$ and $V(w)$ are cut-points, we have $V(w) \xrightarrow[H]{*} V(u)$ iff $\overline{V}(w) \xrightarrow[\overline{H}]{*} \overline{V}(u)$, by Claim (2). Moreover, $\overline{V}(u) \neq \overline{V}(w)$, by Claim (1), and rule C_2 applies to H iff rule A_2' or rule A_2'' applies to $\overline{H}$. We have to distinguish two cases—either the node $V(u)$ is still a cut-point after $u := null$ or not:
  – In the former case, rule C_2 applies to H iff rule A_2' applies to $\overline{H}$. Since there is no change in H nor $\overline{H}$ (except for the value of u that is set to $\bot$ in both cases) and there is no change in ι and $\overline{\nu}$, we are done.
  – In the latter case, rule C_2 applies to H iff rule A_2'' applies to $\overline{H}$. Let $H' = \langle N, S, V[u \to \bot], D\rangle$ be the heap obtained after rule C_2 and $\overline{H'} = \langle \overline{N'}, \overline{S'}, \overline{V'}\rangle$ be the abstract structure obtained after rule A_2''. Let $\overline{V}(u) = n$ and let $m \in \overline{N}$ be such that $w \xrightarrow[\overline{H}]{*} m \xrightarrow[\overline{H}]{} n$. Then, there exist equivalence classes $[k]$ and $[l]$ such that $\beta^{-1}(m) = [k]$ and $\beta^{-1}(n) = [l]$. $H'_{/\sim}$ contains one less equivalence class than $H_{/\sim}$ because $[k]$ and $[l]$ become equivalent in H' and they together form an equivalence class $[k']$ in $H'_{/\sim}$. We define a function β' which maps $H'_{/\sim}$ into $\overline{H'}$ by $\beta'([p]) = \beta([p])$ for all equivalence classes $[p]$ different from $[k']$ and $\beta'([k']) = m$. Then, it is clear that $\overline{H'} = \langle \overline{N'}, \overline{S'}, \overline{V'}\rangle$ is a structural abstraction of H' due to β'. Furthermore, $\|\beta'^{-1}(m)\| = \|\beta^{-1}(m)\| + \|\beta^{-1}(n)\|$. Therefore, we have $\overline{\nu'}(x_m) = \|\beta'^{-1}(m)\|$, and for all $n \in \overline{N'}$ different from m, we have $\overline{\nu'}(x_n) = \overline{\nu}(x_n) = \|\beta^{-1}(n)\| = \|\beta'^{-1}(n)\|$. Therefore, $(l, H, \nu) \rhd_s (l, \overline{H}, \overline{\nu})$ implies $(l', H', \nu') \rhd_s (l', \overline{H'}, \overline{\nu'})$.
– The other cases are treated in a similar way.

Case $s = [\, u := w.next \,]$. There are two cases: Either the equivalence class of the node pointed to by u in the concrete heap contains one node or more than one node. In the latter case, the structure obtained after the instruction contains one more equivalence class. This is taken care of by splitting an abstract node into two abstract nodes, exploiting the splitting function σ.

Cases $s = [\, u := w \,]$, $s = [\, u.next := null \,]$, $s = [\, u.next := w \,]$, $s = [\, u := new \,]$. These cases are treated in a similar way to the case of $u := null$.

Finally, the cases of the conditional and `while` statements are trivial.

□

The following is a consequence of Theorems 1 and 2.

Corollary 1 *For every data-insensitive program with lists, if its counter automaton is flat, then safety and termination are decidable properties.*

Notice that the number of objects created by a loop iteration in a flat list program is always bounded by a constant, therefore its counter automaton is linear positive and restrictive (but not necessarily flat). If this automaton is moreover flat, we can apply Theorem 1.

The problem of defining syntactic classes of programs that can be translated into flat counter automata has been tackled in detail in [15] using a different translation scheme (which works for a more restricted class of programs). The method of this paper, although more general, makes it hard to predict whether the outcome will be flat or not. The main reason is the translation of the `u := w.next` statement which generates a branch in the counter automaton (rules A_7 and A'_7)—hence even a program without nested loops may translate into a non-flat counter automaton when such statements occur within loops.

An example of a data-insensitive program that yields a flat counter automaton is the list reversal program from Figure 2 (both when applied to non-circular as well as circular lists)[1]. Other examples of data-insensitive programs that yield flat counter automata are, for instance, the programs `ListCounter` and `InsDel` considered in Section 5 (the former program computes the length of a non-circular list and then uses the length to limit a subsequent pass through the list, the latter program creates a list, records its length, and then destroys the list using the recorded length). The other programs considered in Section 5, which are various list sorting programs, lead to non-flat counter automata.

The List Reversal Example. Figure 8 shows the counter automaton bisimulating the list reversal program from Figure 2, started with a non-circular list pointed to by i as its input. The counter variable corresponding to each abstract node is depicted inside the node itself. Variables pointing to null are omitted. For each label of the program the corresponding control states are those reached after execution of the instruction at the label. The counter automaton for the same program working on a circular input is shown in Figure 9. For space reasons, only the control states where branching occurs are depicted.

4.2 Ordered Data Programs

In this section, we complete the definition of abstraction for programs with lists by introducing an abstraction for heaps containing data from an ordered domain $\langle \mathfrak{D}, \preceq \rangle$. More precisely, we need to abstract the order relations that may occur inside a list segment and between two list segments.

Definition 5 Let $H = \langle N, S, V, D \rangle$ be a concrete heap and $H_{/\sim}$ its quotient w.r.t. the R_H relation. If $R \subseteq N \times N$ is any relation on the set of nodes, we define the following for any $[n], [m] \in N_{/\sim}$:

- $\mathrm{o}^R([n])$ iff $\forall n_1, n_2 \in [n] \,.\, (n_1 \neq n_2 \wedge n_1 R_H n_2) \Rightarrow n_1 \; R \; n_2$,
- $[n] \preceq^R_{ff} [m]$ iff $hd([n]) \; R \; hd([m])$,
- $[n] \preceq^R_{fa} [m]$ iff $\forall n_1 \in [m] \,.\, hd([n]) \; R \; n_1$,
- $[n] \preceq^R_{af} [m]$ iff $\forall n_1 \in [n] \,.\, n_1 \; R \; hd([n])$,
- $[n] \preceq^R_{aa} [m]$ iff $\forall n_1 \in [n] \; \forall n_2 \in [m] \,.\, n_1 \; R \; n_2$.

[1] Notice that a statement of the type `u := w.next` is used in a loop in this program, but in the derived counter automaton, only one of the branches that arise from translating this statement stays in a loop—the other branch leads out of the loop.

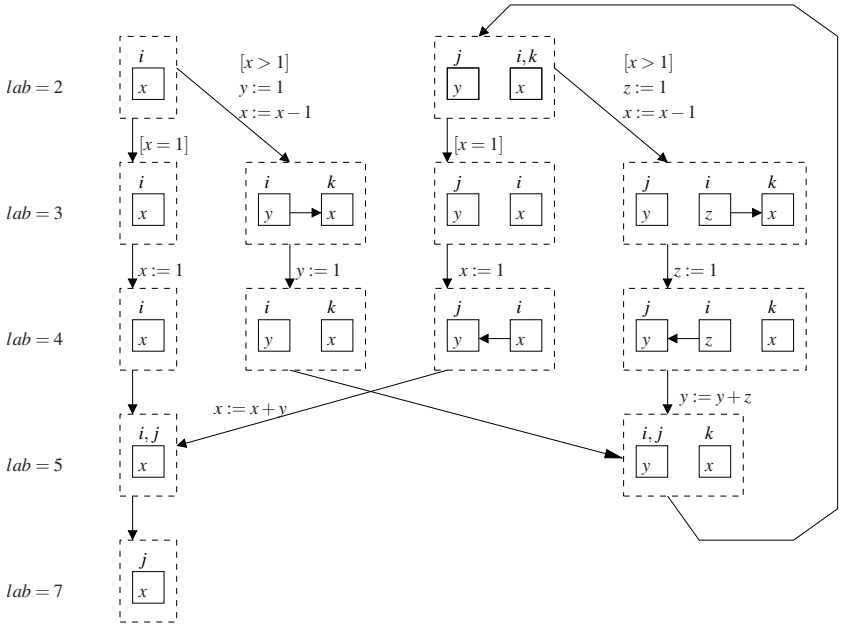

Fig. 8 Non-circular list reversal

For a concrete heap $H = \langle N, S, V, D \rangle$, we define the relation $c \subseteq N \times N$ such that, for any $n_1, n_2 \in N$, $n_1\ c\ n_2$ holds iff $D(n_1) \preceq D(n_2)$. Then, $\mathrm{o}^c([n])$ is true for a list segment $[n]$ iff all its elements are ordered w.r.t. $\preceq$. Similarly, $[n] \preceq^c_\diamond [m]$ holds for $\diamond \in \{ff, fa, af, aa\}$ iff the first (all) element(s) of $[n]$ is (are) less than the first (all) element(s) of $[m]$, respectively.

Definition 6 An *abstract heap* is a tuple $\widetilde{H} = \langle \overline{H}, \mathrm{o}, \preceq_{ff}, \preceq_{fa}, \preceq_{af}, \preceq_{aa} \rangle$ where $\overline{H} = \langle \overline{N}, \overline{S}, \overline{V} \rangle$ is an abstract structure, $\mathrm{o} \subseteq \overline{N}$ is a unary ordering predicate, and $\preceq_{ff,fa,af,aa} \subseteq \overline{N} \times \overline{N}$ are binary ordering predicates.

An abstract heap $\widetilde{H} = \langle \overline{H}, \mathrm{o}, \preceq_{ff}, \preceq_{fa}, \preceq_{af}, \preceq_{aa} \rangle$ sharing the same structure $\overline{H} = \langle \overline{N}, \overline{S}, \overline{V} \rangle$ as another abstract heap $\widetilde{H'} = \langle \overline{H}, \mathrm{o}', \preceq'_{ff}, \preceq'_{fa}, \preceq'_{af}, \preceq'_{aa} \rangle$ is said to be *more precise*, denoted as $\widetilde{H} \sqsubseteq \widetilde{H'}$, if and only if we have $\mathrm{o}(n) \Leftarrow \mathrm{o}'(n)$ and $n \preceq_\diamond m \Leftarrow n \preceq'_\diamond m$ for all $\diamond \in \{ff, fa, af, aa\}$ and for each $n, m \in \overline{N}$. Intuitively, the fact that a predicate does not hold for some abstract node indicates uncertainty w.r.t. the concrete ordering of the configuration. For instance, if $\mathrm{o}(n)$ does not hold, this means that in the concrete setting, n "represents" a list segment that may or may not be ordered.

Given a set S of abstract heaps sharing the same structure, we denote by $\sqcup S$ the least upper bound, and by $\sqcap S$ the greatest lower bound of S with respect to $\sqsubseteq$. Note that $\sqcup$ and $\sqcap$ are undefined for sets of abstract heaps that have different structures. The domain of abstract heaps is denoted by $\langle \widetilde{\mathcal{H}}(PVar), \sqsubseteq \rangle$. We say that an abstract heap is in normal form if the underlying abstract structure is in normal form. We write $\widetilde{\mathcal{H}}_{nf}(PVar)$ for the set of abstract heaps in normal form.

Definition 7 Let $H = \langle N, S, V, D \rangle$ be a concrete heap with data from an ordered domain $\langle \mathfrak{D}, \preceq \rangle$ and let $H_{/\sim} = \langle N_{/\sim}, S_{/\sim}, V_\sim \rangle$ be its quotient. An abstract heap $\widetilde{H} = \langle \overline{H}, \mathrm{o}, \preceq_{ff}, \preceq_{fa}$

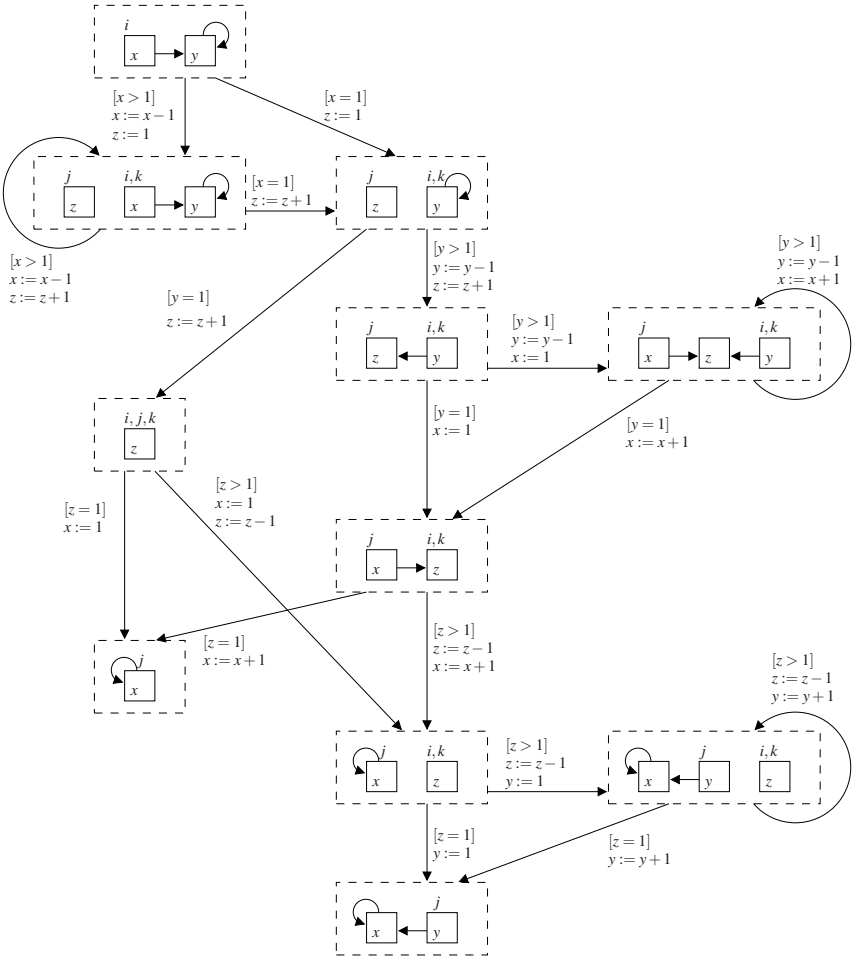

Fig. 9 Circular list reversal

$, \preceq_{af}, \preceq_{aa}\rangle$ is said to be an *abstraction* of H based on $\overline{H}$ iff $\overline{H}$ is a structural abstraction of H and for all $[n],[m] \in N_{/\sim}$ and all $\diamond \in \{ff, fa, af, aa\}$:

- $\mathrm{o}(\beta([n])) \Rightarrow \mathrm{o}^c([n])$ and
- $\beta([n]) \preceq_\diamond \beta([m]) \Rightarrow [n] \preceq^c_\diamond [m]$

where β is the bijection linking $H_{/\sim}$ and $\overline{H}$ in accordance with Definition 4.

For any concrete heap $H \in \mathcal{H}(PVar)$ and one of its structural abstractions $\overline{H}$, we further define the *most precise abstraction* of H w.r.t. $\overline{H}$ as $\alpha_{\overline{H}}(H) = \sqcap\{\widetilde{H} \mid \widetilde{H}$ is an abstraction of H based on $\overline{H}\}$. For any abstract heap $\widetilde{H} \in \widetilde{\mathcal{H}_{nf}}(PVar)$, we define the *concretisation* of $\widetilde{H}$ as $\gamma(\widetilde{H}) = \{H \mid \alpha_{\overline{H}}(H) \sqsubseteq \widetilde{H}$ where $\overline{H}$ is the underlying abstract structure of $\widetilde{H}\}$. Clearly, $\gamma(\widetilde{H}_1) \subseteq \gamma(\widetilde{H}_2)$, if $\widetilde{H}_1 \sqsubseteq \widetilde{H}_2$, but the dual does not necessarily hold. Finally, the canonical most precise abstraction of H is defined as $\alpha_{\alpha_s(H)}(H)$.

Taking ordering predicates o and $\preceq_{ff,fa,af,aa}$ into account refines the notion of counter automata that we have so far introduced to model list manipulating programs. The counter automata defined in this section keep track of the ordering information, allowing one to verify properties related to the ordering of lists, needed, e.g., to establish correctness of various sorting programs, such as InsertSort, BubbleSort, etc.

In order to achieve an enhanced precision of our abstract operational semantics without introducing too many different cases into it, we have designed our abstract state transformer function in two stages. The first stage yields the actual change of the predicates, and the second one is an operation of "saturation" whose goal is to add all the predicates that can be derived from the existing ones for a given abstract heap without changing the corresponding set of concrete heaps. Moreover, we assume to initially start from saturated abstract heaps.

In order to understand the intuition behind the saturation step, let us consider the case depicted in Figure 10. There are three abstract nodes n, m, and p in the figure such that n and m are ordered, and $n \prec_{af} p$ and $p \prec_{fa} m$ hold. Setting the v variable to null in this state causes the nodes n and m to merge into n' which is ordered. In our abstract semantics, we introduce a rule for inferring $\mathrm{o}(n')$ if $n \prec_{aa} m$ is known. In the example, we do not explicitly have $n \prec_{aa} m$, but $n \prec_{aa} m$ is a consequence of the transitivity of the order relation: $n \prec_{af} p \wedge p \prec_{fa} m \Rightarrow n \prec_{aa} m$. In order not to have to introduce the situation of $n \prec_{af} p \wedge p \prec_{fa} m$ as a separate case for when $o(n')$ can be inferred, we saturate the original state by adding $n \prec_{aa} m$ to it before merging the nodes n and m.

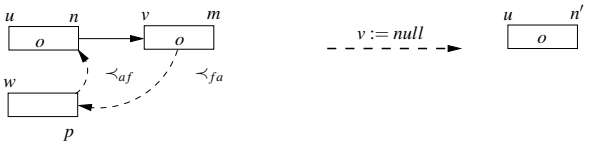

Fig. 10 An example of saturation

For the rest of this section, a counter automaton with ordering predicates is defined as $A_a = \langle Q_a, X, \xrightarrow{a} \rangle$ where the set of control states is now constructed as $Q_a = Lab \times (\widetilde{\mathcal{H}}_{nf}(PVar) \cup \{\overline{H}_{err}\})$ and the set of configurations as $\mathcal{S}_a = Q_a \times (X \to \mathbb{N})$, with the usual notation. In addition to updating the abstract structure, the transition relation $\xrightarrow{a}$ has to also update the ordering predicates.

4.3.1 Saturation

Let $\widetilde{H} = \langle \overline{H}, \mathrm{o}, \preceq_{ff}, \preceq_{fa}, \preceq_{af}, \preceq_{aa} \rangle$ be an abstract heap in normal form based on an abstract structure $\overline{H} = \langle \overline{N}, \overline{S}, \overline{V} \rangle$, and let $\widetilde{H'}$ be just like $\widetilde{H}$ except that all the components of the tuples are primed. We define the *saturation* of $\widetilde{H}$ to be the most precise abstract heap in normal form whose concretisation is the concretisation of $\widetilde{H}$. More precisely, $\widetilde{H_0}$ is the saturation of $\widetilde{H}$ if and only if $\widetilde{H_0} = \sqcap\{\widetilde{H'} \mid \gamma(\widetilde{H}) = \gamma(\widetilde{H'}) \wedge \widetilde{H} =_S \widetilde{H'}\}$ where $\widetilde{H} =_S \widetilde{H'}$ holds for two abstract heaps iff they are based on the same abstract structure. We say that $\widetilde{H}$ is *saturated* if and only if $\widetilde{H} = \widetilde{H_0}$.

Unfortunately, the above definitions do not allow one to effectively check that $\widetilde{H'}$ is the saturation of $\widetilde{H}$ for arbitrary abstract heaps in normal form. The problem is that the set $\gamma(\widetilde{H})$ is infinite. To overcome this problem, we introduce "syntactical" saturation rules, given in

Weakening

1. $n \preceq_{aa} m \Rightarrow n \preceq_{af} m$
2. $n \preceq_{aa} m \Rightarrow n \preceq_{fa} m$
3. $n \preceq_{af} m \Rightarrow n \preceq_{ff} m$
4. $n \preceq_{fa} m \Rightarrow n \preceq_{ff} m$

Transitivity

5. $n \preceq_{ff} m \wedge m \preceq_{ff} p \Rightarrow n \preceq_{ff} p$
6. $n \preceq_{af} m \wedge m \preceq_{ff} p \Rightarrow n \preceq_{af} p$
7. $n \preceq_{ff} m \wedge m \preceq_{fa} p \Rightarrow n \preceq_{fa} p$
8. $n \preceq_{af} m \wedge m \preceq_{fa} p \Rightarrow n \preceq_{aa} p$

Reflexivity

9. $n \preceq_{ff} n$

Order

10. $n \preceq_{aa} n \Rightarrow \mathrm{o}(n)$
11. $\mathrm{o}(n) \Rightarrow n \preceq_{fa} n$

Fig. 11 Saturation rules

Figure 11. The closure of an abstract heap $\widetilde{H}$ in normal form w.r.t. the rules in Figure 11 is denoted as $sat(\widetilde{H})$. The saturation rules need to be applied with the following premises. Let $(\widetilde{H}, \mathrm{v})$ be a configuration of the counter automaton and n an abstract node of $\widetilde{H}$.

- If $\mathrm{v}(x_n) = 1$, then it must be the case that $\mathrm{o}(n)$ and $n \preceq_\diamond n$, $\diamond \in \{ff, fa, af, aa\}$, all hold in $\widetilde{H}$. The reason is that list segments of size one are implicitly ordered, and the ordering relations are reflexive.
- If $\mathrm{v}(x_n) = 2$ and $n \preceq_{fa} n$, then $\mathrm{o}(n)$ must also hold in $\widetilde{H}$. In a list segment of size two, if the first element is less than the second, then the segment must be ordered.

The generated counter automaton tests, at each step, for each node $n \in \overline{N}$, whether $x_n \in \{1,2\}$ and update the ordering predicates accordingly. Formal arguments for these updates are provided separately in Section 4.3.2. For the moment, let us carry on with the soundness proof for our abstraction.

Definition 8 Let $\widetilde{H} = \langle \overline{H}, \mathrm{o}, \preceq_{ff}, \preceq_{fa}, \preceq_{af}, \preceq_{aa} \rangle$ be an abstract heap in normal form based on an abstract structure $\overline{H} = \langle \overline{N}, \overline{S}, \overline{V} \rangle$, let $H = \langle N, S, V, D \rangle \in \gamma(\widetilde{H})$ be a possible concretisation of $\widetilde{H}$, and let $\beta : N_{/\sim} \cup \{\bot\} \to \overline{N} \cup \{\bot\}$ be the bijection from Definition 4. Then $\blacktriangleleft$ is defined to be the *strongest* partial order on N satisfying the following for all $n, m \in \overline{N}$, $\diamond \in \{ff, fa, af, aa\}$:

- $\mathrm{o}(n) \Rightarrow \mathrm{o}^{\blacktriangleleft}(\beta^{-1}(n))$ and
- $n \preceq_\diamond m \Rightarrow \beta^{-1}(n) \preceq_\diamond^{\blacktriangleleft} \beta^{-1}(m)$.

Proposition 1 *Let $\widetilde{H} = \langle \overline{H}, \mathrm{o}, \preceq_{ff}, \preceq_{fa}, \preceq_{af}, \preceq_{aa} \rangle$ be an abstract heap in normal form based on an abstract structure $\overline{H} = \langle \overline{N}, \overline{S}, \overline{V} \rangle$, and let $H = \langle N, S, V, D \rangle \in \gamma(\widetilde{H})$ be a possible concretisation of $\widetilde{H}$. Let $\beta : N_{/\sim} \cup \{\bot\} \to \overline{N} \cup \{\bot\}$ be the bijection from Definition 4, and let $sat(\widetilde{H})$ be $\langle \overline{H}, \mathrm{o}^{sat}, \preceq_{ff}^{sat}, \preceq_{fa}^{sat}, \preceq_{af}^{sat}, \preceq_{aa}^{sat} \rangle$. We have that, for all $n, m \in N$, if $n \blacktriangleleft m$ holds, then one of the following holds:*

1. *$n = hd([n])$, $m = hd([m])$ and $\beta([n]) \preceq_{ff}^{sat} \beta([m])$,*
2. *$n = hd([n])$, $m \in tl([m])$ and $\beta([n]) \preceq_{fa}^{sat} \beta([m])$,*
3. *$n \in tl([n])$, $m = hd([m])$ and $\beta([n]) \preceq_{af}^{sat} \beta([m])$,*
4. *$n \in tl([n])$, $m \in tl([m])$ and either:*
   (a) *$n\, R_H^*\, m$ and $\mathrm{o}^{sat}(\beta([n]))$ or*
   (b) *not $n\, R_H^*\, m$ and $\beta([n]) \preceq_{aa}^{sat} \beta([m])$.*

Proof The proof is by induction on the length of the argument that one can use to show that $n \blacktriangleleft m$ holds according to Definition 8, i.e., according to the number of times one uses the defining implications of Definition 8 to show that $n \blacktriangleleft m$ holds. For the base case, we have:

- If $n = hd([n])$, $m = hd([m])$, we distinguish whether $[n] = [m]$ or $[n] \neq [m]$. In the former case, we have $\beta([n]) \preceq_{ff}^{sat} \beta([n])$ by the reflexivity rule 9. In the latter case, $n \blacktriangleleft m$ holds because $\beta([n]) \preceq_\diamond \beta([m])$ for $\diamond \in \{ff, fa, af, aa\}$, and we use the weakening rules 2, 3, 4 to obtain $\beta([n]) \preceq_{ff}^{sat} \beta([m])$.

- If $n = hd([n])$, $m \in tl([m])$, then $n \blacktriangleleft m$ either because $[n] = [m]$ and $o(\beta([n]))$, or because $\beta([n]) \preceq_\diamond \beta([m])$ for some $\diamond \in \{fa, aa\}$. In the first case, we apply the ordering rule 11, and in the second case, the weakening rule 2 to obtain $\beta([n]) \preceq^{sat}_{fa} \beta([m])$.
- If $n \in tl([n])$, $m = hd([m])$, then $n \blacktriangleleft m$ because $\beta([n]) \preceq_\diamond \beta([m])$ for $\diamond \in \{af, aa\}$, in which case we apply the weakening rule 1 to obtain $\beta([n]) \preceq^{sat}_{af} \beta([m])$.
- If $n \in tl([n])$, $m \in tl([m])$, we distinguish two sub-cases:
  - $nR^*_H m$. In this case, $n \blacktriangleleft m$ either because $o(\beta([n]))$, in which case $o^{sat}(\beta([n]))$ directly holds, or because $\beta([n]) \preceq_{aa} \beta([m])$, which implies $o^{sat}(\beta([n]))$, by rule 10.
  - Not $n\ R^*_H\ m$. In this case, $n \blacktriangleleft m$ because $\beta([n]) \preceq_{aa} \beta([n])$, which directly gives $\beta([n]) \preceq^{sat}_{aa} \beta([m])$.

The induction step:

- If $n = hd([n])$, $m = hd([m])$, then $n \blacktriangleleft m$ because $p \blacktriangleleft m$ for some $p \in N$, and either:
  - $p = hd([p])$ and $\beta([n]) \preceq_\diamond \beta([p])$ for some $\diamond \in \{ff, fa, af, aa\}$. By the weakening rules 1–4, we obtain $\beta([n]) \preceq^{sat}_{ff} \beta([p])$. By the induction hypothesis, we have $\beta([p]) \preceq^{sat}_{ff} \beta([m])$, and by the transitivity rule 5, we obtain $\beta([n]) \preceq^{sat}_{ff} \beta([m])$.
  - $p \in tl([p])$ and $\beta([n]) \preceq_\diamond \beta([p])$ for some $\diamond \in \{fa, aa\}$. By the weakening rules 2, 4 we obtain $\beta([n]) \preceq_{ff} \beta([p])$. By the induction hypothesis, we have $\beta([p]) \preceq^{sat}_{af} \beta([m])$, and by the weakening rule 3, $\beta([p]) \preceq^{sat}_{ff} \beta([m])$. By the transitivity rule 5, we obtain $\beta([n]) \preceq^{sat}_{ff} \beta([m])$.
- If $n = hd([n])$, $m \in tl([m])$, then $n \blacktriangleleft m$ because $p \blacktriangleleft m$ for some $p \in N$, and either:
  - $p = hd([p])$ and $\beta([n]) \preceq_\diamond \beta([p])$ for some $\diamond \in \{ff, fa, af, aa\}$. By the weakening rules 1–4, we obtain $\beta([n]) \preceq^{sat}_{ff} \beta([p])$. By the induction hypothesis, we have $\beta([p]) \preceq^{sat}_{fa} \beta([m])$, and by the transitivity rule 7, we obtain $\beta([n]) \preceq^{sat}_{fa} \beta([m])$.
  - $p \in tl([p])$ and $\beta([n]) \preceq_\diamond \beta([p])$ for some $\diamond \in \{fa, aa\}$. By the weakening rules 2, 4, $\beta([n]) \preceq_{ff} \beta([p])$. By the induction hypothesis, we have $\beta([p]) \preceq^{sat}_{aa} \beta([m])$, therefore by the weakening rule 2, we have $\beta([p]) \preceq^{sat}_{fa} \beta([m])$ and by the transitivity rule 7, we obtain $\beta([n]) \preceq^{sat}_{fa} \beta([m])$.
- If $n \in tl([n])$, $m = hd([m])$, then $n \blacktriangleleft m$ because $p \blacktriangleleft m$ for some $p \in N$, and either:
  - $p = hd([p])$ and $\beta([n]) \preceq_\diamond \beta([p])$ for some $\diamond \in \{af, aa\}$. By the weakening rule 1, we obtain $\beta([n]) \preceq^{sat}_{af} \beta([p])$. By the induction hypothesis, we have $\beta([p]) \preceq^{sat}_{ff} \beta([m])$, and by the transitivity rule 6, we obtain $\beta([n]) \preceq^{sat}_{af} \beta([m])$.
  - $p \in tl([p])$ and $\beta([n]) \preceq_{aa} \beta([p])$, hence $\beta([n]) \preceq_{af} \beta([p])$, by the weakening rule 1. By the induction hypothesis, we have $\beta([p]) \preceq^{sat}_{af} \beta([m])$, hence $\beta([p]) \preceq^{sat}_{ff} \beta([m])$, by the weakening rule 3. By the transitivity rule 6, we obtain $\beta([n]) \preceq^{sat}_{af} \beta([m])$.
- If $n \in tl([n])$, $m \in tl([m])$, we distinguish the following sub-cases:
  - $nR^*_H m$. This case is covered by the base case.
  - Not $n\ R^*_H\ m$. In this case, $n \blacktriangleleft m$ because $p \blacktriangleleft m$ for some $p \in N$, and either:
    - $p = hd([p])$ and $\beta([n]) \preceq_\diamond \beta([p])$ for some $\diamond \in \{af, aa\}$. By the weakening rule 1, we have $\beta([n]) \preceq^{sat}_{af} \beta([p])$, and by the induction hypothesis, we have $\beta([p]) \preceq^{sat}_{fa} \beta([m])$. By the transitivity rule 8, we obtain $\beta([n]) \preceq^{sat}_{aa} \beta([m])$.
    - $p \in tl([p])$ and $\beta([n]) \preceq_{aa} \beta([p])$. By the weakening rule 1, we have $\beta([n]) \preceq^{sat}_{af} \beta([p])$. By the induction hypothesis, we have $\beta([p]) \preceq^{sat}_{aa} \beta([m])$, and by the weakening rule 2, we obtain $\beta([p]) \preceq^{sat}_{fa} \beta([m])$. By the transitivity rule 8, we obtain $\beta([n]) \preceq^{sat}_{aa} \beta([m])$.

□

The next theorem establishes soundness and completeness of the saturation rules.

Theorem 3 *Given an abstract heap $\widetilde{H}$ in normal form, we have $sat(\widetilde{H}) = \sqcap\{\widetilde{H'} \mid \gamma(\widetilde{H'}) = \gamma(\widetilde{H}) \wedge \widetilde{H} =_S \widetilde{H'}\}$.*

Proof Clearly, $sat(\widetilde{H}) =_S \sqcap\{\widetilde{H'} \mid \gamma(\widetilde{H'}) = \gamma(\widetilde{H}) \wedge \widetilde{H} =_S \widetilde{H'}\}$ since the saturation rules in Figure 11 do not change the underlying abstract structure.

We show $sat(\widetilde{H}) \sqsupseteq \sqcap\{\widetilde{H'} \mid \gamma(\widetilde{H'}) = \gamma(\widetilde{H}) \wedge \widetilde{H} =_S \widetilde{H'}\}$ by showing that $\gamma(sat(\widetilde{H})) = \gamma(\widetilde{H})$. This is proved by induction on the number of applications of the rules from Figure 11. In particular, we need to show for each such rule R that $\gamma(\widetilde{H}) = \gamma(\widetilde{H'})$ where $\widetilde{H'}$ is the result of applying R to $\widetilde{H}$. This check is straightforward.

To show that $sat(\widetilde{H}) \sqsubseteq \sqcap\{\widetilde{H'} \mid \gamma(\widetilde{H'}) = \gamma(\widetilde{H}) \wedge \widetilde{H} =_S \widetilde{H'}\}$, we let $\widetilde{H'} \in \widetilde{\mathcal{H}}(PVar)$ be any abstract heap such that $\gamma(\widetilde{H'}) = \gamma(\widetilde{H}) \wedge \widetilde{H} =_S \widetilde{H'}$, and prove that $sat(\widetilde{H}) \sqsubseteq \widetilde{H'}$. Let $\overline{H} = \langle \overline{N}, \overline{S}, \overline{V} \rangle$ be the abstract structure underlying both $sat(\widetilde{H})$ and $\widetilde{H'}$. Let $sat(\widetilde{H}) = \langle \overline{H}, \mathrm{o}, \preceq_{ff}, \preceq_{fa}, \preceq_{af}, \preceq_{aa} \rangle$ and $\widetilde{H'} = \langle \overline{H}, \mathrm{o}', \preceq'_{ff}, \preceq'_{fa}, \preceq'_{af}, \preceq'_{aa} \rangle$. It remains to be shown that:

Claim For any nodes $n, m \in \overline{N}$, we have $\mathrm{o}(n) \Leftarrow \mathrm{o}'(n)$ and $n \preceq_\diamond m \Leftarrow n \preceq'_\diamond m$ for all $\diamond \in \{ff, fa, af, aa\}$.

Proof Let $H_0 = \langle N_0, S_0, V_0, D_0 \rangle \in \gamma(sat(\widetilde{H}))$ be a concrete heap and $\beta_0 : N_{0/\sim} \cup \{\bot\} \to \overline{N} \cup \{\bot\}$ the associated bijection. Let $n, m \in \overline{N}$ be arbitrary nodes. To show that $\preceq_\diamond \supseteq \preceq'_\diamond$ for $\diamond \in \{ff, fa, af, aa\}$, we make a case split based on the relation $\preceq_\diamond$:

- $\preceq_\diamond$ is $\preceq_{ff}$: $n = m$ is a special case since $n \preceq_{ff} m$ by the reflexivity rule 9. Otherwise, consider $n \neq m$. Let $n_0 = hd(\beta_0^{-1}(n)), m_0 = hd(\beta_0^{-1}(m))$. If $n_0 \blacktriangleleft m_0$ is the case, by Proposition 1, we have $n \preceq_{ff} m$. Otherwise, we can build a concrete heap $H_1 = \langle N_0, S_0, V_0, D_1 \rangle$ where for all $n', m' \in N_0$, we let $D_1(n') \preceq D_1(m')$ iff $n' \blacktriangleleft m'$. In particular, we can choose $D_1(n_0) \succ D_1(m_0)$. This is always possible due to the fact that $\langle \mathfrak{D}, \preceq \rangle$ is infinite. Assume $n \preceq'_{ff} m$. Then, $H_1 \in \gamma(sat(\widetilde{H})) \setminus \gamma(\widetilde{H'})$, which is in contradiction with the fact that $\gamma(sat(\widetilde{H})) = \gamma(\widetilde{H}) = \gamma(\widetilde{H'})$.
- The rest of the cases are analogous.

To show that $\mathrm{o} \supseteq \mathrm{o}'$, let $n_0, m_0 \in \beta_0^{-1}(n)$ be arbitrary nodes such that $n_0 R_H m_0$. Note that it is always possible to choose $H_0 \in \gamma(sat(\widetilde{H}))$ such that $n_0, m_0 \in tl([n])$. For this, it is sufficient to choose H_0 in such a way that $\|\beta_0^{-1}(n)\| > 2$. If $n_0 \blacktriangleleft m_0$, it must be that $\mathrm{o}(n)$ holds, by Proposition 1. Otherwise, suppose that there exist $n_0, m_0 \in tl(\beta_0^{-1}(n))$ such that $n_0 \blacktriangleleft m_0$ does not hold. Then it is possible to build a concrete heap $H_1 = \langle N_0, S_0, V_0, D_1 \rangle$ where for all $n', m' \in N_0$, we let $D_1(n') \preceq D_1(m')$ iff $n' \blacktriangleleft m'$. In particular, we can choose $D_1(n_0) \succ D_1(m_0)$. This is always possible due to the fact that $\langle \mathfrak{D}, \preceq \rangle$ is infinite. Assume $o'(n)$, we get $H_1 \in \gamma(sat(\widetilde{H})) \setminus \gamma(\widetilde{H'})$, which is in contradiction with the fact that $\gamma(sat(\widetilde{H})) = \gamma(\widetilde{H}) = \gamma(\widetilde{H'})$. □

This concludes the proof of Theorem 3. □

4.3.2 Saturation and Size Information

We show now how a limited amount of information about the sizes of concrete nodes can be used to enhance the precision of the abstraction. The results below provide formal grounds for implementing runtime tests of the form $x_n = 1$, $x_n = 2$, and $x_n > 2$ for each $n \in \mathcal{N}$ as was briefly mentioned in the previous. To do that we need to extend the notion of concretisation as follows.

Let us recall the definition of structural concretisation $\gamma_s(\overline{H}) = \bigcup_{\nu:\overline{N}\to\mathbb{N}^+} \nu(\overline{H})$. Let φ be an arithmetic formula with free variables $FV(\varphi) \subseteq X$ where X denotes the set of counters. A mapping $\nu : X \to \mathbb{N}^+$ satisfies a given formula φ, denoted $\nu \models \varphi$, iff the formula obtained by substituting each variable x_n with $\nu(n)$ is valid. With this notation, we define the *structural concretisation w.r.t.* φ as $\gamma_s^\varphi(\overline{H}) = \bigcup_{\nu\models\varphi} \nu(\overline{H})$. Given an abstract heap $\widetilde{H} = \langle \overline{H}, \mathrm{o}, \preceq_{ff}, \preceq_{fa}, \preceq_{af}, \preceq_{aa} \rangle$ in normal form, the *concretisation w.r.t.* φ is defined as $\gamma^\varphi(\widetilde{H}) = \{H \mid H \in \nu(\overline{H}),\ \alpha_{\overline{H}}(H) \sqsubseteq \widetilde{H},\ \text{and } \nu \models \varphi\}$. Now an abstract heap $\widetilde{H}$ in normal form is said to be *saturated w.r.t.* φ if and only if $\widetilde{H} = \sqcap\{\widetilde{H'} \mid \gamma^\varphi(\widetilde{H}) = \gamma^\varphi(\widetilde{H'}) \wedge \widetilde{H} =_S \widetilde{H'}\}$. Notice that saturation w.r.t. φ is a generalisation of the previous notion of saturation since saturation coincides with saturation w.r.t. $\top$.

Lemma 2 *If φ and ψ are two formulae such that $FV(\varphi) \subseteq X$, $FV(\psi) \subseteq X$, and $\models \varphi \to \psi$, then an abstract heap $\widetilde{H}$ in normal form using X as the set of counters is saturated w.r.t. φ only if it is saturated w.r.t. ψ.*

Proof Let $\widetilde{H} = \langle \overline{H}, \mathrm{o}, \preceq_{ff}, \preceq_{fa}, \preceq_{af}, \preceq_{aa} \rangle$ be an abstract heap in normal form using X as the set of counters. By definition, $\widetilde{H}$ is saturated w.r.t. φ iff

$$\forall \widetilde{H'} \,.\, \gamma^\varphi(\widetilde{H}) = \gamma^\varphi(\widetilde{H'}) \wedge \widetilde{H} =_S \widetilde{H'} \Rightarrow \widetilde{H} \sqsubseteq \widetilde{H'}$$

Now let $\widetilde{H'}$ be an arbitrary abstract heap in normal form over X such that $\gamma^\psi(\widetilde{H}) = \gamma^\psi(\widetilde{H'}) \wedge \widetilde{H} =_S \widetilde{H'}$. We aim at proving that $\widetilde{H} \sqsubseteq \widetilde{H'}$. It is sufficient to show that $\gamma^\varphi(\widetilde{H}) = \gamma^\varphi(\widetilde{H'})$ and apply the fact that $\widetilde{H}$ is saturated w.r.t. φ in order to obtain $\widetilde{H} \sqsubseteq \widetilde{H'}$. "$\subseteq$" Let $H \in \gamma^\varphi(\widetilde{H})$, i.e., $H \in \nu(\overline{H})$ and $\alpha_{\overline{H}}(H) \sqsubseteq \widetilde{H}$ for some ν such that $\nu \models \varphi$. Since $\models \varphi \to \psi$, we have also $\nu \models \psi$, hence $H \in \gamma^\psi(\widetilde{H}) = \gamma^\psi(\widetilde{H'})$. Hence we have $\alpha_{\overline{H}}(H) \sqsubseteq \widetilde{H'}$, and so $H \in \gamma^\varphi(\widetilde{H'})$. The "$\supseteq$" direction is symmetrical. □

The following theorem relates the notions of saturation and saturation w.r.t. φ for the cases of $\varphi = [x_n = 1]$, $[x_n = 2]$, and $[x_n > 2]$. In particular, it shows that the $[x_n = 1]$, $[x_n = 2]$, and $[x_n > 2]$ cases are the only cases we need to consider in order to saturate with respect to size information.

Theorem 4 *Let $\widetilde{H} = \langle \overline{H}, \mathrm{o}, \preceq_{ff}, \preceq_{fa}, \preceq_{af}, \preceq_{aa} \rangle$ be an abstract heap in normal form, let $\overline{H} = \langle \overline{N}, \overline{S}, \overline{V} \rangle$ be its abstract structure, and $n \in \overline{N}$ an arbitrary abstract node. Then the following holds:*

1. *$\widetilde{H}$ is saturated w.r.t. $x_n = 1$ if and only if it is saturated and $\mathrm{o}(n)$, $n \preceq_\diamond n$ hold for all $\diamond \in \{ff, fa, af, aa\}$.*
2. *$\widetilde{H}$ is saturated w.r.t. $x_n = 2$ if and only if it is saturated and $n \preceq_{fa} n \Rightarrow \mathrm{o}(n)$.*
3. *For any $k > 2$, $\widetilde{H}$ is saturated w.r.t. $x_n = k$ if and only if it is saturated.*

Proof We start with point 1:

"$\Rightarrow$" If $\widetilde{H}$ is saturated w.r.t. $x_n = 1$, then it is saturated, by Lemma 2 (we have $\models x_n = 1 \to \top$). Let $H \in \gamma^{x_n=1}(\widetilde{H})$ be an arbitrary concretisation of $\widetilde{H}$ w.r.t. $x_n = 1$, and β be the mapping of list segments of $H_{/\sim}$ into abstract nodes. Then we have $\|\beta^{-1}(n)\| = 1$, hence $\mathrm{o}^c(\beta^{-1}(n))$ and $\beta^{-1}(n) \preceq^c_\diamond \beta^{-1}(n)$. Suppose now that $\mathrm{o}(n)$ is not the case in $\widetilde{H}$, and let $\widetilde{H'}$ be like $\widetilde{H}$ with $\mathrm{o}(n)$ added. Then we have $\widetilde{H'} \sqsubset \widetilde{H}$ and $\gamma^{x_n=1}(\widetilde{H}) = \gamma^{x_n=1}(\widetilde{H'})$, which contradicts with the fact that $\widetilde{H}$ is saturated w.r.t. $x_n = 1$. The same argument works for $n \preceq_\diamond n$, $\diamond \in \{ff, fa, af, aa\}$.

"$\Leftarrow$" It is sufficient to prove that $\gamma^{x_n=1}(\widetilde{H}) = \gamma^{x_n=1}(\widetilde{H'}) \Rightarrow \gamma(\widetilde{H}) = \gamma(\widetilde{H'})$ holds for any $\widetilde{H'}$ such that $\widetilde{H'} =_S \widetilde{H}$, and then use the fact that $\widetilde{H}$ is saturated to obtain $\widetilde{H} \sqsubseteq \widetilde{H'}$. Let $\widetilde{H'}$

be an arbitrary abstract heap in normal form such that $\widetilde{H'} =_S \widetilde{H}$ and $\gamma^{x_n=1}(\widetilde{H}) = \gamma^{x_n=1}(\widetilde{H'})$. We first show $\gamma(\widetilde{H}) \subseteq \gamma(\widetilde{H'})$. Let $H \in \gamma(\widetilde{H})$ be an arbitrary concretisation of $\widetilde{H}$, and β be the mapping of list segments of $H_{/\sim}$ into abstract nodes. Since $n \preceq_{aa} n$, we have $\beta^{-1}(n) \preceq^c_{aa} \beta^{-1}(n)$, i.e., all nodes from $\beta^{-1}(n)$ have equal data. If $\|\beta^{-1}(n)\| > 1$, let H' be the same as H except for $\beta^{-1}(n)$ which is replaced by a single concrete node with the same data. Obviously, $\alpha_{\overline{H}}(H) = \alpha_{\overline{H}}(H')$, hence $\alpha_{\overline{H}}(H') \sqsubseteq \widetilde{H}$, therefore $H' \in \gamma^{x_n=1}(\widetilde{H}) = \gamma^{x_n=1}(\widetilde{H'})$. The latter implies $\alpha_{\overline{H}}(H) = \alpha_{\overline{H}}(H') \sqsubseteq \widetilde{H'}$, i.e., $H \in \gamma(\widetilde{H'})$. The case of $\gamma(\widetilde{H}) \supseteq \gamma(\widetilde{H'})$ is symmetric.

We continue with point 2:

"$\Rightarrow$" This direction is similar to the "$\Rightarrow$" direction of point 1.

"$\Leftarrow$" Like above, it is sufficient to show that $\gamma^{x_n=2}(\widetilde{H}) = \gamma^{x_n=2}(\widetilde{H'}) \Rightarrow \gamma(\widetilde{H}) = \gamma(\widetilde{H'})$ holds for any $\widetilde{H'}$ such that $\widetilde{H'} =_S \widetilde{H}$. Let $\widetilde{H'}$ be an arbitrary abstract heap in normal form such that $\widetilde{H'} =_S \widetilde{H}$ and $\gamma^{x_n=2}(\widetilde{H}) = \gamma^{x_n=2}(\widetilde{H'})$. We first show $\gamma(\widetilde{H}) \subseteq \gamma(\widetilde{H'})$. Let $H \in \gamma(\widetilde{H})$, and β be the mapping of list segments of $H_{/\sim}$ into abstract nodes. There are three cases:

- $\|\beta^{-1}(n)\| = 1$: Let H' be like H except for $\beta^{-1}(n)$ which is replaced by a list segment consisting of two nodes with the same data as $hd(\beta^{-1}(n))$. Obviously, $\alpha_{\overline{H}}(H) = \alpha_{\overline{H}}(H')$, which leads to $H' \in \gamma^{x_n=2}(\widetilde{H}) = \gamma^{x_n=2}(\widetilde{H'})$, therefore $\alpha_{\overline{H}}(H) \sqsubseteq \widetilde{H'}$, i.e., $H \in \gamma(\widetilde{H'})$.
- $\|\beta^{-1}(n)\| = 2$: $H \in \gamma^{x_n=2}(\widetilde{H}) = \gamma^{x_n=2}(\widetilde{H'}) \subseteq \gamma(\widetilde{H'})$.
- $\|\beta^{-1}(n)\| > 2$: There are two sub-cases:
  - $n \preceq_{fa} n$: We have $\beta^{-1}(n) \preceq^c_{fa} \beta^{-1}(n)$, hence $hd(\beta^{-1}(n))$ is the node with the minimal data value of the entire list segment $\beta^{-1}(n)$. Since $\|tl(\beta^{-1}(n))\| > 1$, let H' be the same as H except for $tl(\beta^{-1}(n))$, which consists now of only one element, and namely the one with the maximum value in $tl(\beta^{-1}(n))$. One can easily show that $\alpha_{\overline{H}}(H) = \alpha_{\overline{H}}(H')$ since no predicate needs to be updated as a result of the transformation. In particular, the new segment obtained from $\beta^{-1}(n)$ is ordered, but due to $n \preceq_{fa} n \Rightarrow o(n)$ and $n \preceq_{fa} n$, the original segment $\beta^{-1}(n)$ is ordered too. By the same argument as above, we obtain $H \in \gamma(H')$.
  - H' is now built from H by keeping only the minimum and maximum elements of $\beta^{-1}(n)$, possibly in the reversed order. In this way, one does not introduce $o(n)$ in $\alpha_{\overline{H}}(H')$ when not necessary (notice that $o(n)$ might not hold in $\alpha_{\overline{H}}(H)$), and we can still show that $\alpha_{\overline{H}}(H) = \alpha_{\overline{H}}(H')$.

The case of $\gamma(\widetilde{H}) \supseteq \gamma(\widetilde{H'})$ is symmetric.

Finally, we handle point 3:

"$\Rightarrow$" This direction is similar to the "$\Rightarrow$" direction of point 1.

"$\Leftarrow$" The argument is similar to the "$\Leftarrow$" direction of point 2. Namely, if $H \in \gamma(\widetilde{H})$, we have three cases based on whether $\|\beta^{-1}(n)\| < k$, $= k$, or $> k$. The construction of H' such that $\alpha_{\overline{H}}(H') = \alpha_{\overline{H}}(H)$ is similar to the one of point 2. □

4.3.3 Abstract Pointer Updates and Tests

We now describe how the ordering predicates are to be treated and updated when simulating the effect of program statements. It turns out that, compared to the data-insensitive case, most of the rules describing the abstract execution of pointer updates and tests needs not be modified more than by adding a copy of the ordering predicates from the source to the destination state. To be more precise, all rules from Figures 5 and 7 except for the ones that use the merging (μ) or the splitting (σ) functions (and the *new* statement which we treat in the next section) have to simply maintain the same predicates between the source and

destination of the transition. For example, if we had $\overline{V}(u) = \overline{V}(w) = n$ and $n \preceq_{fa} m$, then the result of applying the statement $u := \mathit{null}$ is $\overline{V}' = \overline{V}[u \to \bot]$ and $n \preceq'_{fa} m$. Clearly, the same holds for pointer equality tests too.

The rules from Figures 5 and 7 that use merging and splitting are dealt with by introducing *ordered* versions of the merging and splitting functions, called μ_o and σ_o, respectively. As a general rule, the new merging and splitting operations are performed on abstract heaps resulting from an application of a basic pointer update to a saturated abstract heap in normal form. Another saturation is then applied to the result in order to maintain the desired precision.

Let $\widetilde{H} = \langle \overline{H}, \mathrm{o}, \preceq_{ff}, \preceq_{fa}, \preceq_{af}, \preceq_{aa} \rangle$ be an abstract heap based on an abstract structure $\overline{H} = \langle \overline{N}, \overline{S}, \overline{V} \rangle$, and let $n, m \in \overline{N}$ be such that $\overline{S}(n) = m$ and m is not a cut-point in $\overline{H}$. We recall that the result of $\mu(\overline{H}, n, m)$ in this case is the abstract structure in which n takes the place of both n and m. Then, $\mu_o(\widetilde{H}, n, m) = \langle \mu(\overline{H}, n, m), \mathrm{o}', \preceq'_{ff}, \preceq'_{fa}, \preceq'_{af}, \preceq'_{aa} \rangle$ where $\mathrm{o}', \preceq'_{ff,fa,af,aa}$ are the (unique) relations on $\overline{N}$ and $\overline{N} \times \overline{N}$ satisfying the following constraints for all $p \in \overline{N} \setminus \{m\}$, $q, r \in \overline{N} \setminus \{n, m\}$, and $\diamond \in \{ff, fa, af, aa\}$:

$$
\begin{array}{ll}
\mathrm{o}(n) \wedge \mathrm{o}(m) \wedge n \preceq_{aa} m \Leftrightarrow \mathrm{o}'(n) & \mathrm{o}(q) \Leftrightarrow \mathrm{o}'(q) \text{ and } q \preceq_{\diamond} r \Leftrightarrow q \preceq'_{\diamond} r \\
n \preceq_{ff} p \Leftrightarrow n \preceq'_{ff} p & p \preceq_{ff} n \Leftrightarrow p \preceq'_{ff} n \\
p \preceq_{fa} n \wedge p \preceq_{fa} m \Leftrightarrow p \preceq'_{fa} n & n \preceq_{fa} p \Leftrightarrow n \preceq'_{fa} p \\
n \preceq_{af} p \wedge m \preceq_{af} p \Leftrightarrow n \preceq'_{af} p & p \preceq_{af} n \Leftrightarrow p \preceq'_{af} n \\
n \preceq_{aa} p \wedge m \preceq_{aa} p \Leftrightarrow n \preceq'_{aa} p & p \preceq_{aa} n \wedge p \preceq_{aa} m \Leftrightarrow p \preceq'_{aa} n
\end{array}
$$

The splitting operation on abstract structures replaces one node n with two nodes n and m such that m becomes the successor of n, and the original successor of n becomes the successor of m. The effect of the split operation on the ordering predicates is modelled by the rules given in the following. Let $\widetilde{H} = \langle \overline{H}, \mathrm{o}, \preceq_{ff}, \preceq_{fa}, \preceq_{af}, \preceq_{aa} \rangle$ be an abstract heap based on an abstract structure $\overline{H} = \langle \overline{N}, \overline{S}, \overline{V} \rangle$. We define $\sigma_o(\widetilde{H}, n, m) = \langle \sigma(\overline{H}, n, m), \mathrm{o}', \preceq'_{ff}, \preceq'_{fa}, \preceq'_{af}, \preceq'_{aa} \rangle$ where $\mathrm{o}', \preceq'_{ff,fa,af,aa}$ are the (unique) relations on $\overline{N}$ and $\overline{N} \times \overline{N}$ that satisfy the following constraints for all $p \in \overline{N} \setminus \{n\}$, $q, r \in \overline{N} \setminus \{p, n\}$, and all $\diamond \in \{ff, fa, af, aa\}$:

$$\mathrm{o}'(n),\ n \preceq'_{\diamond} n,\ \diamond \in \{ff, fa, af, aa\}$$

$$
\begin{array}{ll}
\mathrm{o}(n) \Leftrightarrow n \preceq'_{aa} m \wedge \mathrm{o}'(m) & n \preceq_{aa} n \Leftrightarrow n \preceq'_{aa} m \wedge m \preceq'_{aa} n \wedge m \preceq'_{aa} m \\
n \preceq_{ff} p \Leftrightarrow n \preceq'_{ff} p & p \preceq_{ff} n \Leftrightarrow p \preceq'_{ff} n \\
n \preceq_{fa} p \Leftrightarrow n \preceq'_{fa} p & p \preceq_{fa} n \Leftrightarrow p \preceq'_{fa} n \wedge p \preceq'_{fa} m \\
n \preceq_{af} p \Leftrightarrow n \preceq'_{af} p \wedge m \preceq'_{af} p & p \preceq_{af} n \Leftrightarrow p \preceq'_{af} n \\
n \preceq_{aa} p \Leftrightarrow n \preceq'_{aa} p \wedge m \preceq'_{aa} p & p \preceq_{aa} n \Leftrightarrow p \preceq'_{aa} n \wedge p \preceq'_{aa} m \\
\mathrm{o}(q) \Leftrightarrow \mathrm{o}'(q) & q \preceq_{\diamond} r \Leftrightarrow q \preceq'_{\diamond} r
\end{array}
$$

The first conditions concerning $\mathrm{o}'(n)$ and $n \preceq'_{\diamond} n$ are due to the fact that the actual size of the list segment represented by n is one, i.e., a split operation separates the head from the tail of a list segment.

4.3.4 Abstract Data Updates and Tests

In this section, we describe how the ordering relations are to be treated within assignments of data between list cells, within the *new* statement, and within comparisons of data stored in list cells. At the end, we also briefly discuss possible treatment of further data manipulating statements.

We start by defining how the ordering relations are modified by a data assignment statement of the form $u.data := v.data$. Let $\widetilde{H} = \langle \overline{H}, \mathrm{o}, \preceq_{ff}, \preceq_{fa}, \preceq_{af}, \preceq_{aa} \rangle \in \widetilde{\mathcal{H}}(PVar)$ be a saturated abstract heap in normal form with $\overline{H} = \langle \overline{N}, \overline{S}, \overline{V} \rangle \in \overline{H}(PVar)$ being its underlying abstract structure. We suppose that both u and v are defined in $\overline{H}$, i.e., $\overline{V}(u) = n$, $\overline{V}(v) = m$, $n, m \in \overline{N}$, and $n \neq m$ (if $\overline{V}(u) = \bot$ or $\overline{V}(v) = \bot$, the result of the statement is $\overline{H}_{err}$; on the other hand, if $n = m \neq \bot$, the assignment $u.data := v.data$ has no effect whatsoever). In the following, let $p \in \overline{N} \setminus \{n, m\}$ be an arbitrary node. Below, we provide rules that establish (1) which relations of the form $n \preceq_\diamond x$, $x \preceq_\diamond n$ for $\diamond \in \{ff, fa, af, aa\}$ and $x \in \{n, m, p\}$ hold and (2) whether $o(n)$ holds after performing the assignment. Since n is the only node changed by the assignment, $p \preceq_\diamond q$ and $o(p)$ are clearly the same before and after the assignment for any $p, q \in \overline{N} \setminus \{n\}$.

$$\frac{m \preceq_{ff} p}{n \preceq_{ff} p} \qquad \frac{p \preceq_{ff} m}{p \preceq_{ff} n} \qquad \frac{}{n \preceq_{ff} n} \qquad \frac{}{n \preceq_{ff} m} \qquad \frac{}{m \preceq_{ff} n}$$

$$\frac{m \preceq_{fa} p}{n \preceq_{fa} p} \qquad \frac{p \preceq_{ff} m \quad p \preceq_{fa} n}{p \preceq_{fa} n} \qquad \frac{m \preceq_{fa} n}{n \preceq_{fa} n} \qquad \frac{m \preceq_{fa} m}{n \preceq_{fa} m} \qquad \frac{m \preceq_{fa} n}{m \preceq_{fa} n}$$

$$\frac{n \preceq_{af} p \quad m \preceq_{ff} p}{n \preceq_{af} p} \qquad \frac{p \preceq_{af} m}{p \preceq_{af} n} \qquad \frac{n \preceq_{af} m}{n \preceq_{af} n} \qquad \frac{n \preceq_{af} m}{n \preceq_{af} m} \qquad \frac{m \preceq_{af} m}{m \preceq_{af} n}$$

$$\frac{n \preceq_{aa} p \quad m \preceq_{fa} p}{n \preceq_{aa} p} \qquad \frac{p \preceq_{aa} n \quad p \preceq_{af} m}{p \preceq_{aa} n} \qquad \frac{n \preceq_{aa} n \quad m \preceq_{fa} n \quad n \preceq_{af} m}{n \preceq_{aa} n}$$

$$\frac{n \preceq_{aa} m \quad m \preceq_{fa} m}{n \preceq_{aa} m} \qquad \frac{m \preceq_{aa} n \quad m \preceq_{af} m}{m \preceq_{aa} n} \qquad \frac{\mathrm{o}(n) \quad m \preceq_{ff} n}{\mathrm{o}(n)}$$

The *new* statement is treated in a simple way: No ordering relation is established between the value of the new node and the already existing ones, hence all ordering relations involving the new node are left purely random.

For evaluating a conditional test involving data (i.e., $u.data \leq v.data$), it is crucial whether $\overline{V}(u) \neq \bot$ and $\overline{V}(v) \neq \bot$—if either of these conditions does not hold, the result is $\overline{H}_{err}$. Assume that $\overline{V}(u) \neq \bot$ and $\overline{V}(v) \neq \bot$ holds. If the test $u.data \leq v.data$ is used in an `if` statement and $\overline{V}(u) \preceq_{ff} \overline{V}(v)$ holds, only the `then` branch is enabled. Otherwise, both the `then` and `else` branches are enabled. In such a case, when following the `then` branch, the relation $\overline{V}(u) \preceq_{ff} \overline{V}(v)$ is established, and when following the `else` branch, the relation $\overline{V}(v) \preceq_{ff} \overline{V}(u)$ is established (as is usual also in other works involving some data abstraction). Conditional tests on data used in `while` loops are handled analogously.

In principle, the rules for handling the $u.data := v.data$ assignment can be extended to arithmetic data updates, such as increment, decrement, reset, or other transformations involving data within memory cells. The idea relies on computing weakest preconditions of the abstract predicates with respect to the updates, and finding combinations of predicates that imply the weakest preconditions.

For example, for statements of the form $u.data := u.data + c$ where c is a positive constant, if $\overline{V}(u) = n \neq \bot$, then for any $m \in \overline{N} \setminus \{n\}$, we can establish $m \preceq_\diamond n$ for $\diamond \in \{ff, fa, af, aa\}$ in the target state if and only if $m \preceq_\diamond n$ holds in the source state. On the other hand, no other ordering relations involving n can be established (apart from $o(n)$ in case $x_n = 1$, of course). Dealing with negative constants is analogous. Statements of the form $u.data := v.data + c$ can then be decomposed into $u.data := v.data$ and $u.data := u.data + c$.

Another way in which the data can be handled is to use integer abstract domains (intervals, octagons, polyhedra) to characterise data within nodes, and between nodes. Since most common abstract domains for integers come with abstract transformers, we could use these abstract transformers to define the updates. However, such extensions are beyond the scope of this work.

4.3.5 Simulation between the Abstract and Concrete Semantics

Due to the non-determinism introduced in the execution of the various programs statements w.r.t. the ordering relations, the semantics of the counter automaton for a data-sensitive program is a *simulation* of the semantics of the original program, but not a bisimulation anymore. Formally, given a data-sensitive list manipulating program P, let $\langle \mathcal{S}, \xrightarrow{c} \rangle$ be its concrete semantics with the set of states $\mathcal{S} = Lab \times \mathcal{H}(PVar) \times (IVar \to \mathbb{Z})$ and $\xrightarrow{c}$ being its transition relation. Let $\mathcal{S}_a = Q_a \times (X \to \mathbb{Z})$ be the set of all configurations of the corresponding counter automaton and $\xrightarrow{a}$ its transition relation. Let $\rhd_a \subseteq \mathcal{S} \times \mathcal{S}_a$ be the relation defined such that $(l, H, \iota) \rhd_a (\overline{l}, \widetilde{H}, \overline{\nu})$ holds for a concrete heap $H = \langle N, S, V, D \rangle$, an abstract heap $\widetilde{H} = \langle \overline{H}, o, \preceq_{ff}, \preceq_{fa}, \preceq_{af}, \preceq_{aa} \rangle$ in normal form, program labels l and $\overline{l}$, and valuations $\iota : IVar \to \mathbb{Z}$ and $\overline{\nu} : X \to \mathbb{Z}$ iff one of the two following cases happens:

- $l = \overline{l}$, $\widetilde{H}$ is an abstraction of H such that $\overline{H}$ is a structural abstraction of H based on a bijection $\beta : N_{/\sim} \cup \{\bot\} \to \overline{N} \cup \{\bot\}$ in accordance with Definition 4, $\overline{\nu}{\downarrow_{IVar}} = \iota$, and $\forall n \in \overline{N} \,.\, \overline{\nu}(x_n) = \|\beta^{-1}(n)\|$; or
- $l = \overline{l} \wedge H = H_{err} \wedge \overline{H} = \overline{H}_{err}$.

Theorem 5 *The relation $\rhd_a$ is a simulation between $\langle \mathcal{S}, \xrightarrow{c} \rangle$ and $\langle \mathcal{S}_a, \xrightarrow{a} \rangle$.*

Proof Like in the case of Theorem 2, the proof can be done by considering all possible different statements in the program. The case of H_{err} and $\overline{H}_{err}$ is trivial. Suppose that we are given $(l, H, \iota) \rhd_a (l, \widetilde{H}, \overline{\nu})$ such that $H \neq H_{err}$. Let $H = \langle N, S, V, D \rangle$ and let $\widetilde{H} = \langle \overline{H}, o, \preceq_{ff}, \preceq_{fa}, \preceq_{af}, \preceq_{aa} \rangle$ where $\overline{H} = \langle \overline{N}, \overline{S}, \overline{V} \rangle$. Let $\beta : N_{/\sim_H} \cup \{\bot\} \to \overline{N} \cup \{\bot\}$ be the bijection from Definition 4 linking $H_{/\sim_H}$ and $\overline{H}$. We need to show that for each statement s, $(l, H, \iota) \xrightarrow{s} (l', H', \iota')$ implies $(l, \widetilde{H}, \overline{\nu}) \xrightarrow{s} (l, \widetilde{H'}, \overline{\nu}')$ and $(l', H', \iota') \rhd_a (l', \widetilde{H'}, \overline{\nu}')$.

For most of the statements and most of the semantic rules describing their abstract semantics, the proof of the above is a simple consequence of Theorem 2 and the fact that the ordering predicates are copied from the source to the destination state. The only exceptions are the abstract semantic rules based on the ordered merging and splitting functions and then the rules describing statements testing or manipulating the data contents of list cells. Proving the above for these different cases is very similar, and so we provide here just one of the cases.

In particular, we consider the case of $s = [\, u := null \,]$ in the situation when $V(u) \neq \bot$, $\forall w \in PVar \setminus \{u\} \,.\, V(w) \neq V(u)$, $\exists w \in PVar \setminus \{u\} \,.\, w \xrightarrow[H]{*} V(u)$, and the node $V(u)$ is no more a cut-point after $u := null$. We know that $H' = \langle N, S, V[u \to \bot], D \rangle$ due to rule C_2, and due to rule A_2'', $\overline{H}' = \langle \overline{N}', \overline{S}', \overline{V}' \rangle = \mu(\langle \overline{N}, \overline{S}, \overline{V}[u \to \bot] \rangle, n, m)$ where $m = \overline{V}(u)$ and $\overline{S}(n) = m$. Let $\beta' : N_{/\sim_{H'}} \cup \{\bot\} \to \overline{N}' \cup \{\bot\}$ be the bijection from Definition 4 linking $H'_{/\sim_{H'}}$ and $\overline{H}'$. We have $\beta(p) = \beta'(p)$ for all $p \in \overline{N} \setminus \{n, m\}$ and $\beta^{-1}(n) \circ \beta^{-1}(m) = \beta'^{-1}(n)$.

Taking into account Theorem 2, it remains to be shown that for all $p, q \in \overline{N} \setminus \{m\}$:

1. $\mathrm{o}'(p) \Rightarrow o^c(\beta'^{-1}(p))$ and
2. $p \preceq'_\diamond q \Rightarrow \beta'^{-1}(p) \preceq^c_\diamond \beta'^{-1}(q)$ for all $\diamond \in \{ff, fa, af, aa\}$.

As for item 1, there are two cases:

- $p = n$: $\mathrm{o}'(n) \Rightarrow \mathrm{o}(n) \wedge \mathrm{o}(m) \wedge n \preceq_{aa} m$. Since $\widetilde{H}$ is an abstraction of H, by Definition 7, this implies $\mathrm{o}^c(\beta^{-1}(n))$, $\mathrm{o}^c(\beta^{-1}(m))$, and $\beta^{-1}(n) \preceq^c_{aa} \beta^{-1}(m)$. Since $\beta^{-1}(n) \circ \beta^{-1}(m) = \beta'^{-1}(n)$, we obtain $\mathrm{o}^c(\beta'^{-1}(n))$.
- $p \neq n$: $\mathrm{o}'(p) \Rightarrow \mathrm{o}(p)$. Since $\widetilde{H}$ is an abstraction of H, by Definition 7, this implies $\mathrm{o}^c(\beta'^{-1}(n))$.

As for item 2, we give the proof for $\preceq_{fa}$, the rest of the cases being similar. There are four sub-cases:

- $p = n, q = n$: $n \preceq'_{fa} n \Rightarrow n \preceq_{fa} n \ \wedge \ n \preceq_{fa} m$. Since $\widetilde{H}$ is an abstraction of H, by Definition 7, this implies $\beta^{-1}(n) \preceq^c_{fa} \beta^{-1}(n)$ and $\beta^{-1}(n) \preceq^c_{fa} \beta^{-1}(m)$, i.e., the first element of $\beta^{-1}(n)$ is less than all other elements of $\beta^{-1}(n)$ and all elements of $\beta^{-1}(m)$. Then it is less than or equal to all elements of $\beta^{-1}(n) \circ \beta^{-1}(m) = \beta'^{-1}(n)$. Since this element is also the first of $\beta'^{-1}(n)$, we have $\beta'^{-1}(n) \preceq^c_{fa} \beta'^{-1}(n)$.
- $p = n, q \neq n$: $n \preceq'_{fa} q \Rightarrow n \preceq_{fa} q$. Since $\widetilde{H}$ is an abstraction of H, by Definition 7, this implies $\beta^{-1}(n) \preceq^c_{fa} \beta^{-1}(q)$. Since $hd(\beta^{-1}(n)) = hd(\beta'^{-1}(n))$, we also have $\beta'^{-1}(n) \preceq^c_{fa} \beta'^{-1}(q)$.
- $p \neq n, q = n$: $p \preceq'_{fa} n \Rightarrow p \preceq_{fa} n \wedge p \preceq_{fa} m$. Since $\widetilde{H}$ is an abstraction of H, by Definition 7, this implies $\beta^{-1}(p) \preceq^c_{fa} \beta^{-1}(n)$ and $\beta^{-1}(p) \preceq^c_{fa} \beta^{-1}(m)$. But then we have $\beta^{-1}(p) \preceq^c_{fa} \beta^{-1}(n) \circ \beta^{-1}(m)$, and hence $\beta'^{-1}(p) \preceq^c_{fa} \beta'^{-1}(n)$.
- $p, q \neq n$: $p \preceq'_{fa} q \Rightarrow p \preceq'_{fa} q$. Since $\widetilde{H}$ is an abstraction of H, by Definition 7, this implies $\beta^{-1}(p) \preceq^c_{fa} \beta^{-1}(q)$, i.e., $\beta'^{-1}(p) \preceq^c_{fa} \beta'^{-1}(q)$.

□

5 Experimental Results

We have implemented our translation of data-insensitive list manipulating programs into (bisimilar) counter automata in a tool called L2CA (Lists To Counter Automata) [28]. In order to obtain experimental evidence about the practical behaviour of our method, we have checked several programs handling lists for both *safety* and *termination* properties. The safety properties considered are absence of null pointer dereferences and shape invariance (mainly non-circularity and absence of sharing between lists). The results are given in Table 1. The experiments were performed on an Intel Pentium 3.00GHz machine with 2GB of RAM.

The reported test cases are mostly sorting programs working on lists (`BubbleSort`, `InsertSort`, `MergeSort`, and `SelectionSort`), in which the data comparisons are replaced by non-deterministic choices. The remaining test cases are the `ListReversal` program (Figure 2), a `ListCounter` procedure that computes the length of non-circular lists in a counter and then traverses the list while decrementing the counter, and a program that creates a list, records its length in an integer counter, and then destroys it using the counter value (`InsDel`).

Our first set of experiments was done using the counter automata models generated by L2CA according to the translation scheme in this paper, without any particular optimisation of the models. Table 1 reports the number of lines of the particular considered programs (Lines); the size of the generated counter automaton in terms of the number of control states

Table 1 Experimental results

	Lines	Locations	Transitions	Counters	Safety (sec)	Termination (sec)
Initial examples						
`BubbleSort`	32	506	577	8	3.62	2.41
`InsertSort`	30	1337	1633	8	14.34	6.80
`MergeSort`	26	109	119	6	0.50	0.38
`SelectionSort`	36	3564	4109	9	81.48	19.58
`ListReversal`	8	96	103	6	0.43	0.33
`ListCounter`	15	33	37	5	-	0.09
`InsDel`	14	30	33	3	-	0.07
Reduced examples						
`BubbleSort`	32	6	86	8	0.36	0.31
`InsertSort`	30	27	104	8	0.51	0.43
`MergeSort`	26	4	13	6	0.07	0.06
`SelectionSort`	36	63	844	9	4.42	3.72
`ListReversal`	8	4	11	6	0.06	0.05
`ListCounter`	15	4	8	5	-	0.04
`InsDel`	14	4	6	3	-	0.03
Faulty examples						
`BubbleSort`	32	734	857	8	6.17	5.95
`InsertSort`	30	1336	1667	8	20.70	20.00
`MergeSort`	26	607	677	6	4.22	4.10
`SelectionSort`	36	8522	10051	9	37.86	37.65
`ListReversal`	8	110	118	6	0.70	0.66
`ListCounter`	15	34	37	5	0.15	0.13
`InsDel`	14	30	32	3	0.31	0.31

(Locations), transitions (Transitions), and counters (Counters), as well as the time needed by the counter automata analyser for checking safety (Safety) and termination (Termination). In particular, we used the ARMC [32] tool for analysing the generated counter automata. In all cases, but `InsDel` and `ListCounter`, ARMC managed to prove the properties of interest, while in these two cases the analysis did not terminate in 30 minutes, and we stopped it.

In the second attempt, we used the FLATA toolset [27] to reduce the size of the models, prior to the analysis with ARMC. The principle of this tool is to eliminate as many control states as possible while preserving the input-output relation of the automaton. When all but the initial and final states are eliminated, FLATA can also establish reachability of a final state, using an SMT solver. In particular, the `InsDel` and `ListCounter` procedures were analysed for safety using FLATA only—in both cases the reduction took roughly one second, and the result was unreachability of the error state.

For most of the test cases, we have then also checked a faulty version, created by abstracting away a null pointer test (`BubbleSort`, `InsertSort`, `MergeSort`, `SelectionSort` and `ListReversal`) or by simply forgetting to initialise an integer counter (`InsDel` and `ListCounter`). For the analysis of the faulty examples, we have also used ARMC. In all the examples, we found a counterexample leading to the error state. At the same time, they all proved to terminate.

L2CA in its current version does not support data-sensitive programs. In order to evaluate the analysis method for sorting programs with data, we have manually constructed the counter automata tracking the ordering predicates for two of the sorting algorithms, namely `BubbleSort` and `InsertSort`. When constructing the automata, we compacted sequences of states with no incoming or outgoing edges. Due to this optimisation, we obtained significantly smaller automata than the ones directly obtained from L2CA: 88 states and 6 counters

for `InsertSort` and 149 states and 7 counters for `BubbleSort`. All the reachable final states of the generated automata represented sorted lists, and hence we were able to prove that all algorithms output sorted lists on all possible inputs.

Finally, let us note that all the above mentioned safety properties may often be checked already at the counter automata extraction phase by checking that no faulty state (i.e., $\overline{H}_{err}$, a final unordered state, or a final state of an undesired shape) is ever generated. Moreover, when the translation is slightly extended, one can often check also another safety property—namely, preservation of list nodes (i.e., the fact that no nodes are accidentally lost)—at the extraction phase. For this, one has to pay a special attention to ensure that no elements of a list are lost via the $u.next := w$ operations. A simple heuristics may be used for this purpose. When we generate a counter automaton state containing a new abstract heap after a $u := w.next$ or $w := new$ statement, we are sure that the node pointed by w has size one. We record this fact by attaching a special flag to this node in the abstract heap. If we later encounter the same abstract heap, we drop the flag if there is no static guarantee that the node has size one. Then, when a $u.next := w$ operation is performed on a flagged node, it is clear that no list nodes are lost. If this is not the case, the dynamic behaviour of the generated counter automaton must be analysed to see whether it may actually happen that some list nodes get really lost (the same is needed when some unsorted final state or $\overline{H}_{err}$ is generated). In fact, we have used the described heuristics when manually constructing the automata for `BubbleSort` and `InsertSort`, and in both cases, we were able to check all the considered safety properties—including preservation of list nodes—statically.

6 Conclusion

We have presented an approach for automatic verification of programs with 1-selector dynamic linked structures. It is based on using counter automata as accurate abstract models for such programs. We have proved several significant properties of the derived models, including their bisimilarity to the original programs in the data-independent case, decidability of both safety and termination of data-insensitive programs for which a flat counter-automata-based model is derived, as well as soundness of the counter-automata-based model extended with selected ordering predicates for the data-sensitive case. Moreover, from a more practical point of view, the obtained counter-automata-based models can be handled using various advanced techniques and tools which have been designed for automatic analysis of counter automata (or, equivalently, programs with integer variables) in the past years— c.f., e.g., [24, 2, 6, 27], and in particular for checking termination and liveness properties, e.g., [18, 16]. Indeed, using counters referring to the sizes of parts of the heap structure (e.g., list segments) of a program is a powerful means for dealing with quantitative reasoning about programs, and in particular about their termination. An extension of this work to trees has been considered in [22] where, however, a bisimilar counter-automata-based model cannot be derived and hence a CEGAR loop was used instead. Further improvements of the techniques for fully automatically proving termination of programs handling doubly-linked lists, trees, and more complex data structures are still of a large interest.

References

1. P.A. Abdulla, A. Bouajjani, J. Cederberg, F. Haziza, and A. Rezine. Monotonic Abstraction for Programs with Dynamic Memory Heaps. In *Proc. of CAV'08*, volume 5123 of *LNCS*. Springer, 2008.

2. A. Annichini, A. Bouajjani, and M.Sighireanu. TReX: A Tool for Reachability Analysis of Complex Systems. In *Proc. of CAV'01*, volume 2102 of *LNCS*, 2001.
3. I. Balaban, A. Pnueli, and L.D. Zuck. Shape Analysis by Predicate Abstraction. In *Proc. of VMCAI'05*, volume 3385 of *LNCS*. Springer, 2005.
4. P. Baldan, A. Corradini, J. Esparza, T. Heindel, B. König, and V. Kozioura. Verifying Red-Black Trees. In *Proc. of COSMICAH'05*, Technical report RR-05-04. Queen Mary, University of London, 2005.
5. S. Bardin, A.Finkel, and D. Nowak. Toward Symbolic Verification of Programs Handling Pointers. In *Proc. of AVIS'04*, 2004.
6. S. Bardin, A. Finkel, J. Leroux, and L. Petrucci. FAST: Fast Acceleration of Symbolic Transition systems. In *Proc. of CAV'03*, volume 2725 of *LNCS*, 2003.
7. S. Bardin, A. Finkel, and E. Lozes. From Pointer Systems to Counter Systems Using Shape Analysis. In *Proc. of AVIS'06*, 2006.
8. J. Berdine, C. Calcagno, B. Cook, D. Distefano, P.W. O'Hearn, T. Wies, and H. Yang. Shape Analysis for Composite Data Structures. In *Proc. of CAV'07*, volume 4590 of *LNCS*. Springer, 2007.
9. J. Berdine, A. Chawdhary, B. Cook, D. Distefano, and P.W. O'Hearn. Variance analyses from invariance analyses. In *Proc. of POPL'07*. ACM Press, 2007.
10. A. Bouajjani, M. Bozga, P. Habermehl, R. Iosif, P. Moro, and T. Vojnar. Programs with Lists are Counter Automata. In *Proc. of CAV'06*, volume 4144 of *LNCS*. Springer, 2006.
11. 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'05*, volume 3440 of *LNCS*. Springer, 2005.
12. A. Bouajjani, P. Habermehl, A. Rogalewicz, and T. Vojnar. Abstract Regular Tree Model Checking of Complex Dynamic Data Structures. In *Proc. of SAS'06*, volume 4134 of *LNCS*. Springer, 2006.
13. M. Bozga and R. Iosif. Quantitative Verification of Programs with Lists. In *Proc. of VISSAS'05*, 2005.
14. M. Bozga, R. Iosif, and Y. Lakhnech. Storeless Semantics and Alias Logic. In *Proc. of PEPM'03*. ACM Press, 2003.
15. Marius Bozga and Radu Iosif. On flat programs with lists. In *VMCAI'07: Proceedings of the 8th international conference on Verification, model checking, and abstract interpretation*, pages 122–136. Springer-Verlag, 2007.
16. A. Bradley, Z. Manna, and H. Sipma. Termination Analysis of Integer Linear Loops. In *Proc. of CONCUR'05*, volume 3653 of *LNCS*, 2005.
17. M. Češka, P. Erlebach, and T. Vojnar. Pattern-Based Verification of Programs with Extended Linear Linked Data Structures. *ENTCS*, 145:113–130, 2006.
18. B. Cook, A. Podelski, and A. Rybalchenko. Abstraction Refinement for Termination. In *Proc. of SAS'05*, volume 3672 of *LNCS*. Springer, 2005.
19. J.V. Deshmukh, E.A. Emerson, and P. Gupta. Automatic Verification of Parameterized Data Structures. In *Proc. of TACAS'06*, volume 3920 of *LNCS*. Springer, 2006.
20. D. Distefano, J. Berdine, B. Cook, and P.W. O'Hearn. Automatic Termination Proofs for Programs with Shape-shifting Heaps. In *Proc. of CAV'06*, volume 4144 of *LNCS*. Springer, 2006.
21. D. Distefano, P.W. O'Hearn, and H. Yang. A Local Shape Analysis Based on Separation Logic. In *Proc. of TACAS'06*, volume 3920 of *LNCS*. Springer, 2006.
22. P. Habermehl, R. Iosif, A. Rogalewicz, and T. Vojnar. Proving Termination of Tree Manipulating Programs. Technical Report TR-2007-1, Verimag, 2007.
23. R. Iosif. Symmetry Reductions for Model Checking of Concurrent Dynamic Software. *STTT*, pages 302–319, 2004.
24. The LASH toolset. http://www.montefiore.ulg.ac.be/ ~ boigelot/research/lash/.
25. O. Lee, H. Yang, and K. Yi. Automatic Verification of Pointer Programs Using Grammar-Based Shape Analysis. In *Proc. of ESOP'05*, volume 3444 of *LNCS*. Springer, 2005.
26. A. Loginov, T.W. Reps, and M. Sagiv. Automated Verification of the Deutsch-Schorr-Waite Tree-Traversal Algorithm. In *Proc. of SAS'06*, volume 4134 of *LNCS*. Springer, 2006.
27. R. Iosif M. Bozga and F. Konecny. Flata.
URL: `http://www-verimag.imag.fr/FLATA.html`.
28. R. Iosif M. Bozga and S. Perarnau. L2CA: Lists to Counter Automata.
URL: `http://www-verimag.imag.fr/L2CA-homepage.html`.
29. R. Manevich, E. Yahav, G. Ramalingam, and M. Sagiv. Predicate Abstraction and Canonical Abstraction for Singly-Linked Lists. In *Proc. of VMCAI'05*, volume 3385 of *LNCS*. Springer, 2005.
30. A. Møller and M.I. Schwartzbach. The Pointer Assertion Logic Engine. In *Proc. of PLDI'01*. ACM Press, 2001.
31. J.C. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *Proc. of LICS'02*. IEEE CS Press, 2002.

32. A. Rybalchenko. ARMC: Abstraction Refinement Model Checker.
URL: http://www7.in.tum.de/ rybal/armc/.
33. S. Sagiv, T.W. Reps, and R. Wilhelm. Parametric Shape Analysis via 3-valued Logic. *TOPLAS*, 24(3), 2002.
34. E. Yahav, T. Reps, M. Sagiv, and R. Wilhelm. Verifying Temporal Heap Properties Specified via Evolution Logic. In *Proc. of ESOP'03*, volume 2618 of *LNCS*. Springer, 2003.
35. H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P.W. O'Hearn. Scalable Shape Analysis for Systems Code. In *Proc. of CAV'08*, volume 5123 of *LNCS*. Springer, 2008.
36. T. Yavuz-Kahveci and T. Bultan. Automated Verification of Concurrent Linked Lists with Counters. In *Proc. of SAS'02*, volume 2477 of *LNCS*. Springer, 2002.