Unité Mixte de Recherche 5104 CNRS - INPG - UJF
Centre Equation
2, avenue de VIGNATE
F-38610 GIERES
tel : +33 456 52 03 40
fax : +33 456 52 03 50
http://www-verimag.imag.fr

Programs with Lists are Counter Automata[1]

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

Report n^o TR-2006-3

May 24, 2006

Programs with Lists are Counter Automata[2]

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

May 24, 2006

Abstract

We address the verification problem of programs manipulating one-selector linked data structures. We propose a new automated approach for checking safety and termination for these programs. Our approach is based on using counter automata as accurate abstract models: control states correspond to abstract heap graphs where list segments without sharing are collapsed, and counters are used to keep track of the number of elements in these segments. This allows to apply automatic analysis techniques and tools for counter automata in order to verify list programs. We show the effectiveness of our approach, in particular by verifying automatically termination of some sorting programs.

Keywords:

Reviewers:

Notes:

How to cite this report:

@techreport { ,
title = { Programs with Lists are Counter Automata[3]},
authors = { Ahmed Bouajjani, Marius Bozga, Peter Habermehl, Radu Iosif, Pierre Moro, Tomáš Vojnar},
institution = { Verimag Technical Report },
number = {TR-2006-3},
year = { },
note = { }
}

1 Introduction

The design of automatic verification methods for programs manipulating dynamic linked data structures is a challenging problem. Indeed, the analysis of 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 (pointer manipulations). The heap of such programs may have in fact an arbitrary size and shape (a graph structure). There are several approaches for tackling this problem addressing different subclasses of programs and using different kinds of formalisms for representing and reasoning about infinite sets of heap structures, e.g., [18, 16, 20, 7].

We consider in this paper the class of programs manipulating linked data structures with a single data-field selector. It corresponds to programs manipulating linked lists with the possibility of sharing and circularities. We propose a new approach for the 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 details 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: it is either a tree (where edges are directed towards the root) or a set of trees having all their roots connected to a simple cycle. 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 not accessing nor modifying the data stored in lists as, e.g., a list reversal program), 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 does it allow to define accurate abstractions, it allows us also to handle quantitative properties depending on the sizes of some parts of the heap. Thus, we can handle programs with integer variables whose value is somehow 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 interesting since it means that our abstraction preserves all properties of the class of data-insensitive programs. 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. It represents a modification in the shape together with a modification on the counters (attached to vertices abstracting sequences between sharing 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 new 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 go further by considering the issue of data-sensitivity. We consider the class of programs manipulating objects ranging over a potentially infinite data domain supplied with an ordering relation, and we assume that the only allowed operation on these data values is the comparison w.r.t. this ordering relation. 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 choice of ordering predicates.

Finally, we show the application of our approach on three examples of programs (list reversal, insertion sort, and bubble sort). We have derived systematically their counter automata models, and then we used (1) our ARMC tool [8] (and some compile-time techniques) for checking safety properties, and (2) the Terminator tool based on [11] for termination.

Related Work. Programs manipulating singly-linked lists have gained a lot of attention within the past two years, as shown by the fairly large number of recent publications on the subject [4, 6, 17, 3, 7]. Interestingly, the idea of abstracting away all the list segments with no incoming edges is common to many of these works, even though they are independent and use different approaches and frameworks (e.g., static analysis [17], predicate abstraction [3] symbolic reachability analysis [4] and proof search [6]). The fact that the number of sharing points in abstract heap structures is bounded by the number of variables in the program is also behind the techniques proposed in [17, 7].

In [9], the authors use an abstract shape model with counters, but their concerns are mostly related to the decidability of a specification logic. The approach that is the closest to ours is [4]. However, it is rather pointed towards showing particular properties such as absence of segmentation faults and memory leak errors, than checking general safety properties, and the work does not address the problem of verifying termination. Moreover, the work reported in [4] offers less automation of the verification than ours. Recently, the same authors have started independently a work [14] on automatic construction of models based on counter automata similar to our approach. The use of ordering predicates in order to handle sorting programs is similar to the one considered in [13, 20] based on the shape analysis approach. Termination is tackled by works such as [21, 3]. In all of these works, ranking functions must be given manually, whereas our approach is fully automated.

2 Programs with Lists

In this section we define a model for programs manipulating dynamic list data structures. We consider that lists are implemented using reference (pointer) data types with one selector (next) field, as it is the case in most object-oriented imperative programming languages (e.g., Java, C, C++). For the time being we consider programs without recursion or concurrency constructs, therefore all variables are assumed to be global. In addition to the list data structures, the programs can have integer variables. Examples of such programs include: list reversals, list insertion procedures, sorting procedures, programs counting the elements in a list, etc.

2.1 Syntactic Definitions

The abstract syntax of the programs considered in this paper is given in Figure 2.1. Here *Lab* is a finite set of program labels (control locations), *PVar* a finite set of pointer variables, and *IVar* a finite set of integer variables (counters).

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

Figure 1: Abstract Syntax of Programs with Lists

We consider imperative programs working with a set of pointer variables *PVar* and a set of integer counter variables *IVar*. 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`. 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 conditionals (`if-then-else`). The guards of the control constructs are pointer equality `u = w`, data comparisons `u.data <= v.data`, zero tests for counters `i = 0` and boolean combinations of the above. A program is said to be *data insensitive* if it does not use guards of the form `u.data <= v.data`. A program is said to be *flat* if the body of any of its `while` loops does not contain (`while`) statements nor conditionals (`if-then-else`).

An example is the list reversal program in Figure 2. To simplify the definition of the operational semantics below, we consider 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`. In addition, the programs are allowed to increment, decrement and reset the counter variables that range over integers. Conditional statements involve two kinds of tests: structural tests `u = v` and `u = null` testing for equality and definedness of pointer variables, comparisons of the data stored in the lists `u.data` $\leq$ `v.data`, and zero tests `i = 0`.

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

Figure 2: List Reversing

2.2 Concrete Operational Semantics

In order to define the concrete semantics of programs with lists, we have to formalize the notion of *heap*. In principle, a heap is a graph in which each node has at most one successor. In addition, some nodes are designated by special labels (variables from *PVar*). If all the edges are reversed, one can imagine a heap as a set of disjoint trees, in which, for each tree there might be an 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 we denote by $f\downarrow_A$ the projection of f on A i.e. $f \cap A \times A$.

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

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

The *state* of a program with lists is a triple $\langle l, \iota, H \rangle$ where $l \in Lab$ is the current program label, $\iota : IVar \rightarrow \mathbb{Z}$ is the current valuation of counter variables, and $H \in \mathcal{H}(PVar)$ is the current heap configuration. Each assignment modifies the state as follows: $\langle l, \iota, H \rangle \xrightarrow{l:s;l'} \langle l', \iota', H' \rangle$, where l'

is the label of the next statement, ι' is the new valuation of counters, computed as usual, and H' is a heap configuration such that $H \xrightarrow{s} H'$, in conformance with the rules in Figure 3. As a result of removing a node from the heap, other nodes might become unreachable from the pointer variables. This set of nodes whose lifetime *depends exclusively* on $n \in N$ is denoted as $dep_H(n)$. H_{err} is a special sink heap configuration, attained as the result of a null pointer dereference. A pointer equality test $u = v$ evaluates to true in a heap $H = \langle N, S, V \rangle$ if and only if $V(u) = V(v)$. Also, $u =$ null is true if and only if $V(u) = \bot$.

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

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

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

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

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

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

$$\frac{}{H_{err} \xrightarrow{s} H_{err}}\ C_{12}$$

Figure 3: Concrete Semantics of Heap Updates

3 Counter Automata

A counter automaton with n counters is a tuple $A = \langle Q, X, \to \rangle$, where Q is a finite set of control states, $X = \{x_1, \ldots, x_n\}$ are the counter variables and $\to \in Q \times \Phi \times Q$ are the transitions, where Φ is the set of Presburger formulae [19] with free variables from $\{x_i, x_i' \mid 1 \le i \le n\}$. 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 $\mathcal{C}$. The transition relation $\xrightarrow{\mathcal{C}} \subseteq \mathcal{C} \times \mathcal{C}$ is defined by $(q,\nu) \xrightarrow{\mathcal{C}} (q',\nu')$ iff there exists a transition $q \xrightarrow{\varphi} q'$ such that if σ is an assignment of the free variables of φ ($FV(\varphi)$) 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 $i \geq 0$.

The following definition introduces a novel class of counter automata that is useful for our purposes:

Definition 2 *Let $A = \langle Q, X, \rightarrow \rangle$ 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. We can prove:

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

Proof: W.l.o.g. we can restrict our attention to self-loops of the form $q \xrightarrow{\phi} q$, where ϕ is a formula of the form considered in Definition 2. Let $x_i^{(m)}$ denote the value of the counter x_i at the m-th visit of control state q. We have, 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, for some $1 \leq i \leq n$, it is the case that $\sum_{j=1}^{n} a_{ji} = 0$, i.e. all coefficients of x_i are zero, then x_i is not used in computing the next values of $\mathbf{x}$, and we can eliminate x_i from the transition relation by replacing it with the corresponding $f_i(\mathbf{x})$ expression in the transition guard $\varphi(\mathbf{x})$ (see Definition 2). This results in a machine with less counters, whose behavior is the projection of the original one on the new set of counters. Obviously, all temporal properties 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 $\mathbf{x}$ in two disjoint subsets: a set $\mathbf{y}$, for which the first case applies, and which are bounded by a constant throughout the execution of the automaton, and a set $\mathbf{z}$, for which the second case applies, and which must occur exactly once in the computation of $\mathbf{x}'$, i.e. for each $z \in \mathbf{z}$ there exists exactly one $x \in \mathbf{x}$ such that $x' = z + g$, where g is a linear combination not involving z.

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

According to the resulting partition $(\mathbf{y}, \mathbf{z})$, the values taken by the counters at each iteration have the following properties:

- $\mathbf{y}$ range over an effectively computable finite set of values $\Gamma = \{\gamma_1, \ldots, \gamma_N\}$,
- $\mathbf{y}'$ are linear combinations of $\mathbf{y}$,
- $\mathbf{z}' = \mathbf{z} + \delta$, where δ are linear combinations of $\mathbf{y}$.

Since the values taken by $\mathbf{y}$ 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 $\psi = \bigwedge_{1 \leq i \leq n} x_i' = f_i(\mathbf{x})$ is the same as in Definition 2, and a partition of the counters into $(\mathbf{y}, \mathbf{z})$, that satisfies the requirements above, we can build a counter machine $A_{sim} = \langle \mathbf{z}, \blacksquare^{|\mathbf{y}|}, \delta_{\mathbf{sim}} \rangle$, where δ_{sim} is obtained as follows:

$$\gamma \xrightarrow[\mathbf{A_{sim}}]{\varphi[\gamma/\mathbf{y}] \,\wedge\, \mathbf{z}'=\mathbf{z}+\delta} \gamma' \text{ iff } \models \psi[\gamma/\mathbf{y}, \gamma'/\mathbf{y}', \mathbf{z}+\delta/\mathbf{z}']$$

Notice that the new transition relation δ_{sim} is deterministic, since γ' and δ are linear combinations of γ. 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 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 $N^{|y|}$, the number of control states. In other words, A_{sim} is of the form: $\gamma_\mathbf{0} \xrightarrow{\varphi_0 \wedge \mathbf{z}'=\mathbf{z}+\delta_\mathbf{0}} \gamma_\mathbf{1} \xrightarrow{\varphi_1 \wedge \mathbf{z}'=\mathbf{z}+\delta_\mathbf{1}} \gamma_\mathbf{2} \ldots \to \gamma_\mathbf{M-1} \xrightarrow{\varphi_{\mathbf{M-1}} \wedge \mathbf{z}'=\mathbf{z}+\delta_\mathbf{M-1}} \gamma_\mathbf{0}$, where $\varphi_i = \varphi[\gamma_\mathbf{i}/\mathbf{y}]$. The relation between the input $\mathbf{z}$, and output $\mathbf{z}'$ values of the counters of A_{sim}, can be now defined by the following Presburger formula:

$$\exists l \geq 0 \ \bigvee_{j=0}^{M-1} \mathbf{z}' = \mathbf{z} + \mathbf{l} \textstyle\sum_{\mathbf{i=0}}^{\mathbf{M-1}} \delta_\mathbf{i} + \sum_{\mathbf{i=0}}^{\mathbf{j}} \delta_\mathbf{i} \ \wedge \tag{1}$$

$$\forall 0 \leq m \leq l \ \exists q \bigwedge_{j=0}^{M-1} m = qM + j \to \varphi_j(\mathbf{z} + \mathbf{q} \textstyle\sum_{\mathbf{i=0}}^{\mathbf{M-1}} \delta_\mathbf{i} + \sum_{\mathbf{i=0}}^{\mathbf{j}} \delta_\mathbf{i}) \tag{2}$$

Intuitively, the formula from the first row gives the difference between $\mathbf{z}'$ and $\mathbf{z}$, whereas the second one ensures that the guards are satisfied all along the way. Notice that $\varphi_j(\mathbf{z} + \mathbf{q} \sum_{\mathbf{i=0}}^{\mathbf{M-1}} \delta_\mathbf{i} + \sum_{\mathbf{i=0}}^{\mathbf{j}} \delta_\mathbf{i})$ 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 incoming edges into a special (abstract) node. This idea constitutes also the basis of our abstraction. Let $\mathcal{N}$ be a set of *abstract nodes* and $\mathcal{X}$ 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 the set of abstract nodes, and*
- $\overline{S} : \overline{N} \to \overline{N}_\perp$, $\overline{V} : PVar \to \overline{N}_\perp$*, are the successor and variable mappings,*

An abstract structure is moreover said to be in normal form *if, for each* $n \in \overline{N}$*, there exists* $u \in PVar$ *such that* $u \xrightarrow[\overline{H}]{*} n$*, and* n *is a cut point in* $\overline{H}$*.*

Intuitively, each abstract node corresponds to a set of concrete nodes, and the counter associated with it in $\mathcal{X}$ keeps track of the number of nodes in this set. For abstract structures in normal form, we do not allow sequences of successive abstract node that are neither pointed by a variable, nor have the indegree 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. $\overline{\mathcal{H}}(PVar)$ denotes the set of all abstract structures with variables from $PVar$. A result similar to the following has been also proved in [4, 17]:

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, $\|\overline{N}\| \leq 2n$ (cf. [17]). As a consequence, the number of such heaps is bounded asymptotically by $(2n)^{2n}$, and the bound is tight.*

Proof: For a set of nodes $M \subset \overline{N}$, let $succ(M) = \{n \mid m \rightarrow n\}$ denote the set of immediate successors of M, and, $fr_i(M) = succ^i(M) \setminus succ^{i-1}(M)$, where $succ^i(M)$ denotes the i-th application of the $succ$ function to M, for $i > 1$. By convention, we take $fr_0(M) = M$. Since each node is reachable from V, we have $\overline{N} \subseteq \bigcup_{i \geq 0} succ^i(V) = \bigcup_{i \geq 0} fr_i(V)$, therefore $\|\overline{N}\| \leq \Sigma_{i \geq 0} \|fr_i(V)\|$. Let $fr_k^{>1}(M)$, $fr_k^{=1}(M)$ be the sets of nodes from $fr_k(M)$ with two or more predecessors, and with one predecessor respectively. Obviously, $\|fr_k(M)\| = \|fr_k^{>1}(M)\| + \|fr_k^{=1}(M)\|$. A node with only one predecessor and one successor clearly violates the normal form condition of Definition 3, hence each node from $fr_k^{=1}(V)$ has no successors for all $k \geq 0$, so we have $\|fr_k^{>1}(V)\| \leq \frac{\|fr_{k-1}^{>1}(V)\| - \|fr_k^{=1}(V)\|}{2}$ for $k > 1$, and $\|fr_1^{>1}(V)\| \leq \frac{\|fr_0(V)\| - \|fr_1^{=1}(V)\|}{2}$. Summing up, we obtain:

$$
\begin{gathered}
\Sigma_{i>1} \|fr_i^{>1}(V)\| \leq \frac{\|fr_0(V)\|}{2} + \Sigma_{i>1} \frac{\|fr_i^{>1}(V)\|}{2} - \Sigma_{i>1} \frac{\|fr_i^{=1}(V)\|}{2} \\
\Sigma_{i>1} \frac{\|fr_i^{>1}(V)\|}{2} + \Sigma_{i>1} \frac{\|fr_i^{=1}(V)\|}{2} \leq \frac{\|fr_0(V)\|}{2} \\
\Sigma_{i \geq 0} \|fr_i(V)\| \leq 2 \|fr_0(V)\| \\
\|\overline{N}\| \leq 2 \|V\|
\end{gathered}
$$

The number of abstract structures in normal form is bounded from below by the number of partitions of the set *PVar*, i.e. for each possible partition of *PVar*, one can construct a different family of abstract structures. This number is known as the Bell number B_n and is bounded asymptotically by n^n. It is easy to see that this gives also an asymptotic upper bound. □

Let us define now a 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 $\rhd_H \subseteq N \times N$ be a relation on the set of nodes, defined as: $n_1 \rhd_H n_2 : n_1 \underset{H}{\longrightarrow} n_2 \wedge \neg cut(n_2)$. We denote by $\sim_H$ the reflexive, symmetric and transitive closure of $\rhd_H$. 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. 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.

For assume that for some $n \in N$, $S_{/\sim}$ maps $[n]$ into two different equivalence classes, call them $[m]$ and $[p]$. This would imply the existence of two nodes $n_1, n_2 \in [n]$ such that $n_1 \xrightarrow{*} m_0$

and $n_2 \xrightarrow{*} p_0$, for some $m_0 \in [m]$ and some $p_0 \in [p]$. Since either $n_1 \xrightarrow{*} n_2$, or $n_2 \xrightarrow{*} n_1$, there must exist a node in $[n]$ with two distinct direct successors, which contradicts the well-formedness of S. The argument for $V_{/\sim}$ is straightforward.

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 $\beta : N_{/\sim} \cup \{\bot\} \to \overline{N} \cup \{\bot\}$ such that $\beta(\bot) = \bot$, and for all $u \in PVar$:*

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

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 in, e.g., [15].

We define the structural abstraction function $\alpha_s : \mathcal{H}(PVar) \to \overline{\mathcal{H}}(PVar)$, $\alpha_s(H) = \overline{H}$, 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. $\gamma_s(\overline{H}) = \{H \mid \alpha_s(H) = \overline{H}\}$.

Note that according to Definition 4, $\alpha_s(H)$ is an abstract structure in normal form. For reasons that will become clear later, we need to extend the notion of concretisation to abstract structures not in normal form. Let $\overline{H} = \langle \overline{N}, \overline{S}, \overline{V} \rangle$ be an abstract structure not necessarily in normal form, and $\nu : \overline{N} \to \mathbb{N}$ a mapping of nodes to 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 $\mathfrak{D}$. In particular, mapping one node into zero makes the node disappear in the concretization, and all its predecessors automatically point to its successor. Then, $\gamma_s(\overline{H}) = \bigcup\{\nu(\overline{H}) \mid \nu : \overline{N} \to \mathbb{N}\}$. Notice that if $\overline{H}$ is in normal form, the two definitions coincide.

4.1 Data Insensitive Programs

This section is devoted to the description of counter automata that abstract the behaviour of the programs with lists. We formalize the correctness of our construction by proving bisimulation between the semantics of a list program and the semantics of a counter automaton. This entails the strong preservation of temporal logic properties. In particular, safety and termination are strongly preserved by the counter automaton, meaning that one can accept and/or refute them based on the behaviour of the latter.

Consider a 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 $Lab \times (\overline{\mathcal{H}}(PVar) \cup \{H_{err}\})$. Let $\mathcal{N} = \bigcup\{\overline{N} \mid \langle \overline{N}, \overline{S}, \overline{V} \rangle \in \overline{\mathcal{H}}(PVar)\}$ be the set of nodes used in the structural abstraction. The counters are $X = \{x_n \mid n \in \mathcal{N}\} \cup IVar$, one for each node, and including the counter variables from the original program. The transitions are given by the triples $q \xrightarrow{\varphi} q'$ with $q = \langle l, \overline{H} \rangle$, $q' = \langle l', \overline{H'} \rangle$ such that there is a statement $l : s; l'$ in the program and the relation $\overline{H} \xrightarrow[s]{\varphi} \overline{H'}$ is described by the structural rules in Figure 4. The 8 cases for the statement $u := null$ are illustrated in Figure 5.

In order to simplify the treatment of the different cases, we have introduced two low-level operations, that perform merging and splitting of abstract nodes (Figure 4). Intuitively, we need to perform merging of two abstract nodes n and m ($\mu(\overline{H}, n, m)$) in order to re-normalize the abstract structure, after a destructive update.

Lemma 2 *If $\overline{H} = \langle \overline{N}, \overline{S}, \overline{V} \rangle$ is an abstract structure, and $n, m \in \overline{N}$ such that $\overline{S}(n) = m$ and m is not a cut point in $\overline{H}$, then $\gamma_s(\overline{H}) = \gamma_s(\mu(\overline{H}, n, m))$.*

Proof: "$\subseteq$" Let $H \in \gamma_s(\overline{H})$. Then there exists a mapping $\nu : \overline{N} \to \mathbb{N}$ such that $H = \nu(\overline{H})$. Let ν' be like ν, except for $\nu'(n) = \nu(n) + \nu(m)$. One can easily verify that $H = \nu'(\mu(\overline{H}, n, m))$, hence $H \in \gamma_s(\mu(\overline{H}, n, m))$. "$\supseteq$" Let $H \in \gamma_s(\mu(\overline{H}, n, m))$. Then there exists $\nu : \overline{N} \setminus \{m\} \to \mathbb{N}$, such that $H = \nu(\mu(\overline{H}, n, m))$. Let $\nu' : \overline{N} \to \mathbb{N}$ be like ν, except that $\nu'(n)$ and $\nu'(m)$ are such that $\nu'(n) + \nu'(m) = \nu(n)$. Note that this is possible since taking zero as $\nu'(m)$ is allowed. Then one easily verifies that $H = \nu'(H)$, i.e. $H \in \gamma_s(\overline{H})$. □

In the case of `u := w.next`, we need to split ($\sigma(\overline{H}, n, m)$) the abstract node n, into two nodes n and m, based on whether the value of its corresponding counter is greater than one or one ($x_n = 1$, $x_n > 1$).

Lemma 3 *If $\overline{H} = \langle \overline{N}, \overline{S}, \overline{V} \rangle$ is an abstract structure, and $n \in \overline{N}$, $m \notin \overline{N}$, then $\gamma_s(\overline{H}) = \gamma_s(\sigma(\overline{H}, n, m))$.*

Proof: Along the same lines as the proof of Lemma 2. □

The semantics of conditional tests (`u = v` and `u = null`) is similar to the concrete case. For more details concerning the translation, the reader is referred to the list reversal example in Figure 7.

Now we can state the main theorem of this section. Given a data insensitive program P, let $\langle \mathcal{S}, \xrightarrow{c} \rangle$ be its concrete semantics with set of states $\mathcal{S} = Lab \times (IVar \to \mathbb{Z}) \times \mathcal{H}(PVar)$ and $\xrightarrow{c}$ 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.

Theorem 2 *$\langle \mathcal{S}, \xrightarrow{c} \rangle$ and $\langle \overline{\mathcal{S}}, \xrightarrow{s} \rangle$ are bisimilar.*

Proof: We show this theorem by defining a relation between the two transition systems which is proved to be a bisimulation.

Let $H = \langle N, S, V, D \rangle$, $H_{/\sim} = \langle N_{/\sim}, S_{/\sim}, V_{/\sim} \rangle$ and $\overline{H} = \langle \overline{N}, \overline{S}, \overline{V} \rangle$.

Let $\rhd_s \subseteq \mathcal{S} \times \overline{\mathcal{S}}$ be the relation defined by:

$$(l, H, \nu) \rhd_s (l_1, \overline{H}, \overline{\nu})$$

if either $l = l_1$ and $\overline{H}$ is a structural abstraction of H due to a function β and $\overline{\nu} \downarrow_{IVar} = \nu$ and $\forall n \in \overline{N} \,.\, \overline{\nu}(x_n) = \nu_{\beta}(n)$ or $l = l_1 \ \wedge\ H = \overline{H} = H_{err}$.

We show in the following that $\rhd_s$ is a bisimulation between $\langle \mathcal{S}, \xrightarrow{P} \rangle$ and $\langle \overline{\mathcal{S}}, \xrightarrow{\overline{P}} \rangle$. This is done by considering all possible different statements in the program. Suppose that $(l, H, \nu) \rhd_s (l, \overline{H}, \overline{\nu})$ and let $\beta : N_{/\sim} \cup \{\bot\} \rightarrow \overline{N} \cup \{\bot\}$ be the function from definition 4.

We need to show that for each statement s, (1) $(l, H, \nu) \xrightarrow{s} (l', H', \nu')$ implies $(l, \overline{H}, \overline{\nu}) \xrightarrow{s} (l', \overline{H}, \overline{\nu}')$ and $(l', H', \nu') \rhd_s (l', \overline{H}, \overline{\nu}')$ and (2) $(l, \overline{H}, \overline{\nu}) \xrightarrow{s} (l', \overline{H}, \overline{\nu}')$ implies $(l, H, \nu) \xrightarrow{s} (l', H', \nu')$ and $(l', H', \nu') \rhd_s (l', \overline{H}, \overline{\nu}')$. 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 involving pointer variables, this follows directly from the below claim:

Claim 1 (1) *Given* $\overline{H} = \langle \overline{N}, \overline{S}, \overline{V} \rangle$ *such that* $\overline{H}$ *is a structural abstraction of* H*, 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 $\overline{V}(u) = \overline{V}(w) = \bot$. 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 with the fact that $V(w)$ is a cut point. □

For the other cases we need another lemma.

Claim 2 (2) *Given* $H = \langle N, S, V \rangle$ *such that* $\overline{H}$ *is a structural abstraction of* H *due to* β*, for all* $n, m \in N$ *such that* $cut(m)$*,* $n \xrightarrow[H]{*} m$ *iff* $\beta([n]) \xrightarrow[\overline{H}]{*} \beta([m])$*.*

Proof: "⇒" We show that for all $n, m \in N$, $n \xrightarrow{*} m$ implies $\beta([n]) \xrightarrow{*} \beta([m])$, by induction on the length of the path from n to m. If $n = m$ we trivially have $\beta([n]) = \beta([m])$. Else, if $n \xrightarrow{*} n' \rightarrow m$, by the induction hypothesis we have $\beta([n]) \xrightarrow{*} \beta([n'])$. Then either $[n'] = [m]$, case in which $\beta([n]) \xrightarrow{*} \beta([m])$, or $[n'] \neq [m]$, case in which $S_{/\sim}([n']) = [m]$, therefore $\beta([n']) \rightarrow \beta([m])$. "⇐" If $\beta([n]) \xrightarrow{*} \beta([m])$ and $cut(m)$, we necessarily have $\beta([n]) \xrightarrow{+} \beta([m])$. We prove that $n \xrightarrow{*} m$ by induction on the length of the path from $\beta([n])$ to $\beta([m])$. If $\beta([n]) \rightarrow \beta([m])$, then there exist $n_0 \in [n]$ such that $n_0 \rightarrow m$. If $n \xrightarrow{*} n_0$, we are done. Else, we have $n_0 \rightarrow m \xrightarrow{*} n$, leading to a contradiction between $cut(m)$ and the fact that n is reachable from n_0 with no cut points in between. For the induction step, if $\beta([n]) \xrightarrow{*} \beta([n']) \longrightarrow \beta([m])$, and $cut(n')$ we have $n \xrightarrow{*} n'$. By a similar argument, $n' \xrightarrow{*} m$, and, by the induction hypothesis, $n \xrightarrow{*} n'$. □

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

Case $s =$ [u := null]. There are different cases.

- $V(u) = \bot$. We have $V(u) = \bot$ iff $\overline{V}(u) = \bot$, by Claim (1). Therefore, rule C_1 applies to H iff rule A_1 applies to $\overline{H}$, and it is clear that $(l', H, \nu) \rhd_s (l', \overline{H}, \overline{\nu})$.

- $V(u) \neq \bot$ 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 be easily checked that $(l', \langle N, S, V[u \to \bot]\rangle, \nu) \rhd_s (l, \langle \overline{N}, \overline{S}, \overline{V}[u \to \bot], \rangle, \overline{\nu})$.

- $V(u) \neq \bot$ and $\forall w \in Pvar \,.\, V(w) \neq V(u)$ and $\exists w \in Pvar \setminus \{u\} \,.\, w \xrightarrow[H]{*} V(u)$
  Since $V(u)$ is a cut point we have for any w that $V(w) \xrightarrow[H]{*} V(u)$ iff $\overline{V}(w) \xrightarrow[\overline{H}]{*} \overline{V}(u)$, by Claim (2). In this case, $\overline{V}(u) \neq \overline{V}(w)$, by Claim (1), and rule C_2 applies to H iff rule A_2' or rule A_2'' applies to $\overline{H}$.
  Either n 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 $\overline{H}$. It is clear that the structure of $H_{/\sim}$ (except u) does not change after the instruction. Therefore, $(l, H, \nu) \rhd_s (l, \overline{H}, \overline{\nu})$ implies $\langle N, S, V[u \to \bot]\rangle \rhd_s \langle \overline{N}, \overline{S}, \overline{V}[u \to \bot], \overline{\nu}\rangle$.
  - In the latter case, rule C_2 applies to H iff rule A_2'' applies to $\overline{H}$ Let $H' = \langle N, S, V[u \to \bot]\rangle$ be the heap obtained after rule C_2 and $\overline{H'} = \langle \overline{N'}, \overline{S'}, \overline{V'}\rangle$ be the heap obtained after rule A_2''. 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 $H_{/\sim}$, as $[k]$ and $[l]$ become equivalent in H'. Let $[k']$ be this equivalence class 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, $\nu_{\beta'}(m) = \nu_\beta(m) + \nu_\beta(n)$. Therefore, we have $\overline{\nu'}(x_m) = \nu_{\beta'}(m)$ and for all $n \in \overline{N'}$ different from m we have $\overline{\nu'}(x_n) = \overline{\nu}(x_n) = \nu_\beta(n) = \nu_{\beta'}(n)$. Therefore, $(l, H, \nu) \rhd_s (l, \overline{H}, \overline{\nu})$ implies $(l', H' \rhd_s (l', \overline{H'}, \overline{\nu'})$.

- 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.

Case $s = $ [u.next := null]. This case is treated in a similar way to case $u := null$.
$\square$

4.1.1 List Reversal Example

Figure 7 shows the counter automaton for the list reversal program from Figure 2, started with a non-circular list pointed to by i, as input. The counter variable corresponding to each abstract node is depicted inside the node itself. The counter automaton for the same program, working on a circular input, is shown in Figure 8. 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. $\rhd$ relation. If $R \subseteq N \times N$ is any relation on the set of nodes define, for any $[n], [m] \in N_{/\sim}$:*

- $\mathrm{o}^R([n])$ *iff* $\forall n_1, n_2 \in [n] \, . \, n_1 \neq n_2 \, \wedge \, n_1 \rhd n_2 \Rightarrow n_1 \, R \, n_2$
- $[n] \preceq^R_{ff} [m]$ *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* $\forall n_1 \in [n] \, \forall n_2 \in [m] \, . \, n_1 \, R \, n_2$

For a concrete heap $H = \langle N, S, V, D \rangle$, the relation $c \subseteq N \times N$ is defined as $n_1 \, c \, n_2 : D(n_1) \preceq D(n_2)$. Then, $\mathrm{o}^c([n])$ is true for a list segment $[n]$ iff all its elements are ordered w.r.t. $\preceq$. Similarly, $[n] \preceq^c_\diamond [m]$ 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 $[m]$.

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, for each $n, m \in \overline{N}$ we have $\mathrm{o}(n) \Leftarrow \mathrm{o}'(n)$ and $n \preceq_\diamond m \Leftarrow n \preceq'_\diamond m$, for all $\diamond \in \{ff, fa, af, aa\}$. Intuitively, the absence of a predicate indicates incertitude w.r.t. the concrete ordering 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 $\langle \widetilde{\mathcal{H}}(PVar), \sqsubseteq \rangle$.

Definition 7 *Let* $H = \langle N, S, V, D \rangle$ *be a concrete heap with data from the ordered domain* $\langle \mathfrak{D}, \preceq \rangle$ *and* $H_{/\sim} = \langle N_{/\sim}, S_{/\sim}, V_\sim \rangle$ *its quotient. An abstract heap* $\widetilde{H} = \langle \overline{H}, \mathrm{o}, \preceq_{ff}, \preceq_{fa}, \preceq_{af}, \preceq_{aa} \rangle$ *is said to be an* abstraction *of* H *if and only if* $\alpha_s(H) = \overline{H}$ *and for all* $[n], [m] \in N_{/\sim}$, $\diamond \in \{ff, fa, af, aa\}$*:* $\mathrm{o}(\beta([n])) \Rightarrow \mathrm{o}^c([n])$ *and* $\beta([n]) \preceq_\diamond \beta([m]) \Rightarrow [n] \preceq^c_\diamond [m]$ *where* β *is the bijection from Definition 4.*

We define $\alpha : \mathcal{H}(PVar) \rightarrow \widetilde{\mathcal{H}}(PVar)$ as $\alpha(H) = \sqcap \{\widetilde{H} \mid \widetilde{H} \text{ is an abstraction of } H\}$. Note that all abstract heaps that are abstractions of H share the same structure, hence $\sqcap$ is defined for this set. The *concretization* function is $\gamma : \widetilde{\mathcal{H}}(PVar) \rightarrow \mathcal{P}(\widetilde{\mathcal{H}}(PVar))$, defined as $\gamma(\widetilde{H}) = \{H \mid \alpha(H) \sqsubseteq \widetilde{H}\}$. 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.

4.3 Counter Automata Semantics with Ordering Predicates

Taking ordering predicates $\mathrm{o}, \preceq_{ff,fa,af,aa}$ into account refines our notion of counter automaton, previously introduced. The counter automaton defined in this section keeps track of the ordering information, allowing one to verify properties related to the ordering of lists, as it is the case for sorting programs, e.g., insertsort, bubblesort, etc.

A counter automaton with ordering predicates is $A_a = \langle Q_a, X, \xrightarrow{a} \rangle$. The set of control states is defined now as $Q_a = Lab \times (\widetilde{\mathcal{H}}(PVar) \cup \{H_{err}\})$, and the set of configurations is $\mathcal{S}_a = Q_a \times (X \rightarrow \mathbb{N})$, with the usual notation. In addition to updating the abstract structure, the transition relation $\xrightarrow{a}$ has to also update the ordering predicates. Our goal is to define the "best transformer" in the sense of [12]. More precisely, our loss of information is only due to the choice of ordering predicates, the definition of $\xrightarrow{a}$ does not introduce further imprecision. Theorem 4 below formalizes this statement.

In order to achieve completeness of the abstract operational semantics, 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, on a given abstract heap, without changing the corresponding set of concrete heaps. For the remainder of this section, we fix an abstract heap $\widetilde{H} = \langle \overline{H}, \mathrm{o}, \preceq_{ff}, \preceq_{fa}, \preceq_{af}, \preceq_{aa} \rangle$, with its 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.

Let us begin by the presentation of the second stage. Given an abstract heap $\widetilde{H}$, we define the *saturation* of $\widetilde{H}$ to be the most precise abstract heap whose concretization is the concretization of $\widetilde{H}$. More precisely, $\widetilde{H_0}$ is the saturation of $\widetilde{H}$ if and only if $\widetilde{H_0} = \sqcap\{\widetilde{H'} \mid \gamma(\widetilde{H}) = \gamma(\widetilde{H'})\}$. An abstract heap $\widetilde{H}$ is said to be *saturated* if and only if $\widetilde{H} = \sqcap\{\widetilde{H'} \mid \gamma(\widetilde{H}) = \gamma(\widetilde{H'})\}$. Unfortunately, this definition does not allow one to effectively check that $\widetilde{H'}$ is the saturation of $\widetilde{H}$ for arbitrary abstract heaps. The problem is that the set $\gamma(\widetilde{H})$ is infinite. To overcome this problem, we introduce "syntactical" saturation rules in Fig. 9. The closure of an abstract heap $\widetilde{H}$ w.r.t. these rules is denoted as $sat(\widetilde{H})$.

The saturation rules need to be applied with the following premises. Let $(\widetilde{H}, \nu)$ be a configuration of the counter automaton, and n an abstract node of $\widetilde{H}$.

- if $\nu(x_n) = 1$, 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 ordered, and in all possible ordering relations with themselves.
- if $\nu(x_n) = 2$ 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 will test, at each step, for each node $n \in \overline{N}$, that $x_n = 1, 2$ and update the ordering predicates accordingly. Formal arguments for these updates are provided separately in Section 4.4. For the moment, let us carry on with the soundness and completeness proof for our abstraction.

Definition 8 *Let $\widetilde{H} = \langle \overline{H}, \mathrm{o}, \preceq_{ff}, \preceq_{fa}, \preceq_{af}, \preceq_{aa} \rangle$, $\overline{H} = \langle \overline{N}, \overline{S}, \overline{V} \rangle$, $H = \langle N, S, V, D \rangle \in \gamma(\widetilde{H})$, and $\beta : N_{/\sim} \to \overline{N}$ be the bijection from Definition 4. Then $\blacktriangleleft$ is defined to be the* smallest *partial order on N satisfying the following, for all $n, m \in \overline{N}$, $\diamond \in \{ff, fa, af, aa\}$:*

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

Proposition 1 *Let $\widetilde{H} = \langle \overline{H}, \mathrm{o}, \preceq_{ff}, \preceq_{fa}, \preceq_{af}, \preceq_{aa} \rangle$, be an abstract heap, $\overline{H} = \langle \overline{N}, \overline{S}, \overline{V} \rangle$, be its abstract structure in normal form and $H = \langle N, S, V, D \rangle \in \gamma(\widetilde{H})$ a possible concretization of $\widetilde{H}$. Let $\beta : N_{/\sim} \to \overline{N}$ be the bijection from Definition 4, and $sat(\widetilde{H})$ be $\langle \overline{H}, \mathrm{o}^{sat}, \preceq_{ff}^{sat}, \preceq_{fa}^{sat}, \preceq_{af}^{sat}, \preceq_{aa}^{sat} \rangle$. Then for all $n, m \in N$ we have $n \blacktriangleleft m$ only if, either 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)* $n \rhd^* m$ *and* $\mathrm{o}^{sat}(\beta([n]))$,
   - *(b)* $n \not\rhd^* m$ *and* $\beta([n]) \preceq_{aa}^{sat} \beta([m])$.

Proof: The proof is by induction on the length of the argument that gives $n \blacktriangleleft m$. By "argument" we mean here the sequence of derivation steps used to deduce $n \blacktriangleleft m$, according to Definition 8. For the base case we have:

- if $n = hd([n])$, $m = hd([m])$, then either $[n] = [m]$, in which case we have $\beta([n]) \preceq_{ff} \beta([n])$ by the reflexivity rule 9, or $[n] \neq [m]$, in which case $n \blacktriangleleft m$ because $\beta([n]) \preceq_{\diamond} \beta([m])$, for $\diamond \in \{ff, fa, af, aa\}$. In the latter case 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 $\mathrm{o}(\beta([n]))$, 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 $\beta([n]) \preceq_{fa}^{sat} \beta([m])$.
- 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 $\beta([n]) \preceq_{af}^{sat} \beta([m])$.
- if $n \in tl([n])$, $m \in tl([m])$ and
  - $n \rhd^* m$, then $n \blacktriangleleft m$ either because $\mathrm{o}(\beta([n]))$, in which case $\mathrm{o}^{sat}(\beta([n]))$ directly, or because $\beta([n]) \preceq_{aa} \beta([m])$, which implies $\mathrm{o}^{sat}(\beta([n]))$, by rule 10.
  - $n \not\rhd^* m$, then $n \blacktriangleleft m$ because $\beta([n]) \preceq_{aa} \beta([n])$ which directly gives $\beta([n]) \preceq_{aa}^{sat} \beta([m])$.

The induction step:

- if $n = hd([n])$, $m = hd([m])$, then $n \blacktriangleleft m$ because, for some $p \in N$, $p \blacktriangleleft m$, 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 $\beta([n]) \preceq_{ff}^{sat} \beta([p])$. By the induction hypothesis we have $\beta([p]) \preceq_{ff}^{sat} \beta([m])$, and by the transitivity rule 5 we obtain $\beta([n]) \preceq_{ff}^{sat} \beta([m])$.
  - $p \in tl([p])$ 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 $\beta([p]) \preceq_{af}^{sat} \beta([m])$, and by the weakening rule 3, $\beta([p]) \preceq_{ff}^{sat} \beta([m])$. By the transitivity rule 5 we obtain $\beta([n]) \preceq_{ff}^{sat} \beta([m])$.
- if $n = hd([n])$, $m \in tl([m])$, then $n \blacktriangleleft m$ because, for some $p \in N$, $p \blacktriangleleft m$, 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 $\beta([n]) \preceq_{ff}^{sat} \beta([p])$. By the induction hypothesis we have $\beta([p]) \preceq_{fa}^{sat} \beta([m])$, and by the transitivity rule 7 we obtain $\beta([n]) \preceq_{fa}^{sat} \beta([m])$.

  - $p \in tl([p])$ 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, for some $p \in N$, $p \blacktriangleleft m$, and either:
  - $p = hd([p])$ and $\beta([n]) \preceq_{\diamond} \beta([p])$, for some $\diamond \in \{af, aa\}$. By the weakening rules 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 $n \in tl([n])$, $m \in tl([m])$ and:
  - $n \triangleright^* m$, this case is covered by the base case.
  - $n \not\triangleright^* m$, then $n \blacktriangleleft m$ because, for some $p \in N$, $p \blacktriangleleft m$, 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 $\beta([n]) \preceq^{sat}_{af} \beta([p])$, and 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 shows the soundness and completeness of the saturation rules.

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

Proof: We show $sat(\widetilde{H}) \sqsupseteq \sqcap\{\widetilde{H'} \mid \gamma(\widetilde{H'}) = \gamma(H)\}$ by showing that $\gamma(sat(\widetilde{H})) = \gamma(\widetilde{H})$. This is proved by induction on the number of applications of the rules Figure 9. 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 $sat(\widetilde{H}) \sqsubseteq \sqcap\{\widetilde{H'} \mid \gamma(\widetilde{H'}) = \gamma(\widetilde{H})\}$, we let $\widetilde{H'} \in \widetilde{\mathcal{H}}(PVar)$ be any abstract heap such that $\gamma(\widetilde{H'}) = \gamma(\widetilde{H})$, and prove that $sat(\widetilde{H}) \sqsubseteq \widetilde{H'}$. The condition $\gamma(\widetilde{H'}) = \gamma(\widetilde{H})$ is equivalent to the following: $\forall H \,.\, \alpha(H) \sqsubseteq \widetilde{H} \iff \alpha(H) \sqsubseteq \widetilde{H'}$. This can only be true iff $\widetilde{H}$ and $\widetilde{H'}$ share the same abstract structure $\overline{H} = \langle \overline{N}, \overline{S}, \overline{V} \rangle$. In particular, $sat(\widetilde{H})$ has the same abstract structure, since the closure of $\widetilde{H}$ under the rules in Figure 9 does not affect the structure. 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 3 *For any $n,m \in \overline{N}$ 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 $\beta_0 : N_{0/\sim} \to \overline{N}$ 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 chose $D_1(n_0) \succ D_1(m_0)$. This is always possible, by the fact that $\langle \mathfrak{D}, \preceq \rangle$ is infinite. Therefore $H_1 \in \gamma(sat(\widetilde{H})) \setminus \gamma(\widetilde{H'})$, 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 $n_0 \rhd m_0$. 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 exists $n_0, m_0 \in tl(\beta_0^{-1}(n))$, such that $n_0 \not\blacktriangleleft m_0$. 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 chose $D_1(n_0) \succ D_1(m_0)$. This is always possible, by the fact that $\langle \mathfrak{D}, \preceq \rangle$ is infinite. Therefore $H_1 \in \gamma(sat(\widetilde{H})) \setminus \gamma(\widetilde{H'})$, in contradiction with the fact that $\gamma(sat(\widetilde{H})) = \gamma(\widetilde{H}) = \gamma(\widetilde{H'})$. □

This concludes the proof. □

We define now how the change of abstract predicates is being performed. Most of the rules that affect only the abstract structure of the state are very similar with the data insensitive case. To be more precise, all rules from Figure 4, with the exception of the ones that use the merging (μ) or the splitting (σ) functions, will 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 := null` is $\overline{V}' = \overline{V}[u \to \bot]$ and $n \preceq'_{fa} m$. The remaining rules 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 saturated abstract heaps, and another saturation is applied to the result in order to maintain the desired precision.

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 the *merging operation* $\mu(\overline{H}, n, m)$ 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 $p \in \overline{N} \setminus \{m\}$,

$q, r \in \overline{N} \setminus \{n\}$ and $\diamond \in \{ff, fa, af, aa\}$:

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

Lemma 4 *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, where* $\overline{H} = \langle \overline{N}, \overline{S}, \overline{V} \rangle \in \overline{\mathcal{H}}(PVar)$, *and* $n, m \in \overline{N}$ *such that* $\overline{S}(n) = m$ *and* m *is not a cut point in* $\overline{H}$. *Then,* $\alpha(\gamma(\widetilde{H})) = \alpha(\gamma(\mu_o(\widetilde{H}, n, m)))$.

Proof: We first show that $\gamma(\widetilde{H}) \subseteq \gamma(\mu_o(\widetilde{H}, n, m))$.

Let $\mu_o(\widetilde{H}, n, m) = \langle \mu(\overline{H}, n, m), \mathrm{o}', \preceq'_{ff,fa,af,aa} \rangle$ and $H \in \gamma(\widetilde{H})$, $H = \langle N, S, V, D \rangle$. Then, $H \in \gamma_s(\overline{H})$, and by Lemma 2, we have $H \in \gamma_s(\mu(\overline{H}, n, m))$. Let β and β' be the mappings of $\overline{N}$, and $\overline{N} \setminus \{m\}$ into list segments of H, i.e. contiguous sequences of nodes from N, related by S. In particular we have $\beta(p) = \beta'(p)$, for all $p \in \overline{N} \setminus \{n, m\}$ and $\beta^{-1}(n) \circ \beta^{-1}(m) = \beta'^{-1}(n)$. In order to show that $H \in \gamma(\mu_o(\widetilde{H}, n, m))$ it is sufficient to show 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 $\diamond \in \{ff, fa, af, aa\}$.

1. There are two cases:

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

- $p \neq n$: $\mathrm{o}'(p) \Rightarrow \mathrm{o}(p)$. Since $H \in \gamma(\widetilde{H})$, by Definition 7 this implies $\mathrm{o}^c(\beta'^{-1}(n))$.

2. We show the proof for $\preceq_{fa}$, the rest of the cases being similar. There are four cases:

- $p = n, q = n$: $n \preceq'_{fa} n \Rightarrow n \preceq_{fa} n \wedge n \preceq_{fa} m$. Since $H \in \gamma(\widetilde{H})$, 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 $H \in \gamma(\widetilde{H})$, 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 $H \in \gamma(\widetilde{H})$, 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 $H \in \gamma(\widetilde{H})$, by Definition 7 this implies $\beta^{-1}(p) \preceq^{c}_{fa} \beta^{-1}(q)$, i.e. $\beta'^{-1}(p) \preceq^{c}_{fa} \beta'^{-1}(q)$.

Next, from Lemma 2 and from the fact that μ_o is based on μ, we know that $\alpha_s(\gamma(\widetilde{H})) = \alpha_s(\gamma(\mu_o(\widetilde{H},n,m)))$. Thus, $\alpha(\gamma(\widetilde{H}))$ and $\alpha(\gamma(\mu_o(\widetilde{H},n,m)))$ are based on the same abstract structure with a set of nodes $\overline{N}_\alpha$, and they can differ only in what predicates $o(u)$ and $u \preceq_\diamond v$ for $\diamond \in \{ff,af,fa,aa\}$ and $u,v \in \overline{N}_\alpha$ hold in them.

From the above shown fact saying that $\gamma(\widetilde{H}) \subseteq \gamma(\mu_o(\widetilde{H},n,m))$, we get that there cannot be any $u,v \in \overline{N}_\alpha$ such that $o(u)$ or $u \preceq_\diamond v$ for $\diamond \in \{ff,af,fa,aa\}$ holds in $\alpha(\gamma(\mu_o(\widetilde{H},n,m)))$ but not in $\alpha(\gamma(\widetilde{H}))$. This would mean that in some conretization of the latter there is a heap for some of whose nodes the given predicates do not hold whereas such a heap is not a concretization of the former, which contradicts the mentioned inclusion.

Thus, it remains to be shown that for any nodes $u,v \in \overline{N}_\alpha$, we cannot have $o(u)$ or $u \preceq_\diamond v$ for $\diamond \in \{ff,af,fa,aa\}$ in $\alpha(\gamma(\widetilde{H}))$ but not in $\alpha(\gamma(\mu_o(\widetilde{H},n,m)))$:

- Let us start with $o(u)$. If the concretization of u does not involve the concretizations of n,m, the property trivially holds as the ordering predicates for nodes other than n,m are simply preserved by μ_o.

  Suppose now that u involves the concretizations of n,m and that $o(u)$ holds in $\alpha(\gamma(\widetilde{H}))$ but not in $\alpha(\gamma(\mu_o(\widetilde{H},n,m)))$. As the ordering predicates are simply copied for all other nodes other than n, the only reason for the given situation can be that $o'(n)$ is not introduced by μ_o despite semantically it is possible to introduce $o'(n)$.

  However, this is a contradiction as it is easy to see that if either $o(n)$, $o(m)$, or $n \preceq_{aa} m$ does not hold, we can come up with such concretizations of n and m that the joint node will not be ordered. Moreover, we can rely on $o(n)$, $o(m)$, and $n \preceq_{aa} m$ to hold always when possible due to working with a saturated heap $\widetilde{H}$ and due to Theorem 3.

- A very similar reasoning as above can be applied to the binary ordering predicates too.

□

Let us note that Lemmma 4 *cannot be strenghtened* to saying that $\gamma(\widetilde{H}) = \gamma(\mu_o(\widetilde{H},n,m))$. Imagine a situation of merging two ordered nodes where the second node contains bigger values than the first one. Then, the resulting node cannot be claimed ordered. However, we know that it should be possible to split its concretizations to two ordered sequences which is a fact that we cannot record using the predicates that we currently support in our abstraction.

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 previous successor of n becomes the successor of m. In addition, the effect of the split operation on the ordering predicates is modeled by the rules given in the following. Formally, $\sigma_o(\widetilde{H},n,m) = \langle \sigma(\overline{H},n,m), o', \preceq'_{ff}, \preceq'_{fa}, \preceq'_{af}, \preceq'_{aa} \rangle$, where $o', \preceq'_{ff,fa,af,aa}$ 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 $\diamond \in \{ff,fa,af,aa\}$:

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

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

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. The following Lemma formalizes the correctness σ_o:

Lemma 5 *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, where* $\overline{H} = \langle \overline{N}, \overline{S}, \overline{V} \rangle \in \overline{H}(PVar)$, $n \in \overline{N}$ *and* $m \notin \overline{N'}$. *Then,* $\alpha(\gamma(\widetilde{H})) = \alpha(\gamma(\sigma_o(\widetilde{H}, n, m)))$.

Proof: Along the same lines as the proof of Lemma 4. □

A conditional test involving data `u.data` $\leq$ `w.data` evaluates true in the abstract heap $\widetilde{H}$ if and only if $\overline{V}(u) \preceq_{ff} \overline{V}(w)$ holds on $sat(\widetilde{H})$. Otherwise, such tests introduce non-determinism in the generated counter automaton. Therefore, the semantics of the counter automaton is a simulation of the semantics of the original program, but not a bisimulation anymore.

Theorem 4 *Let* $\langle l, \iota, H \rangle \in \mathcal{S}$ *be a concrete program state. Then, there exists* $\langle l', \iota', H' \rangle \in \mathcal{S}$ *such that* $\langle l, \iota, H \rangle \xrightarrow{c} \langle l', \iota', H' \rangle$ *if only if there exists an abstract state* $\langle l, \widetilde{H'}, \nu' \rangle \in \mathcal{S}_a$ *such that* $\langle l, \alpha(H), \nu \rangle \xrightarrow{a} \langle l', \widetilde{H'}, \nu' \rangle$ *and* $H' \in \gamma(\widetilde{H'})$.

Proof: Along the same lines as the proof of Theorem 2, using Lemmas 4 and 5, instead of 2 and 3, respectively. □

The following is a consequence of Theorems 1, 2 and 4.

Corollary 1 *For every 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 single loop iteration in a flat list program is always bounded by a constant, therefore its counter automaton is restrictive. The linear and positive conditions can be established by inspection of the form of the transitions in the abstract semantics[4]. If this automaton is moreover flat, we can apply Theorem 1. The result does not give us a purely syntactic criterion for decidability of verification of list manipulating programs but still allows us to decide whether the program falls into a significant decidable fragment or not.

---

[4] Notice that the only negative coefficients in the transition relations are the base coefficients.

4.4 Saturation w.r.t. Size Information

In this section we show 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 function as follows.

Let us recall the definition of structural concretisation $\gamma_s(\overline{H}) = \bigcup\{\nu(\overline{H}) \mid \nu : \overline{N} \to \mathbb{N}\}$. Let φ be an arithmetic formula with free variables $FV(\varphi) = \mathcal{X}$, where $\mathcal{X}$ denotes the set of counters. A mapping $\nu : \mathcal{N} \to \mathbb{N}$ 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 $\gamma_s^\varphi = \bigcup\{\nu(\overline{H}) \mid \nu \models \varphi\}$. Given an abstract heap $\widetilde{H} = \langle \overline{H}, \mathrm{o}, \preceq_{ff}, \preceq_{fa}, \preceq_{af}, \preceq_{aa} \rangle$, the *concretisation w.r.t.* φ is defined as $\gamma^\varphi = \bigcup\{H \mid H \in \nu(\overline{H}),\ \alpha(H) \sqsubseteq \widetilde{H} \text{ and } \nu \models \varphi\}$. Now an abstract heap $\widetilde{H}$ is said to be *saturated w.r.t.* φ if and only if $\widetilde{H} = \sqcap\{\widetilde{H'} \mid \gamma^\varphi(\widetilde{H}) = \gamma^\varphi(\widetilde{H'})\}$. Notice that saturation w.r.t φ is a generalization of the previous notion of saturation, since saturation coincides with saturation w.r.t $\top$.

Lemma 6 *If φ, ψ are two formulae with $FV(\varphi) = FV(\psi) = \mathcal{X}$, such that $\models \varphi \to \psi$, then $\widetilde{H}$ is saturated w.r.t φ only if it is saturated w.r.t. ψ.*

Proof: Let $\widetilde{H} = \langle \overline{H}, \mathrm{o}, \preceq_{ff}, \preceq_{fa}, \preceq_{af}, \preceq_{aa} \rangle$. By definition, $\widetilde{H}$ is saturated w.r.t φ iff

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

Now let $\widetilde{H'}$ be an arbitrary abstract heap such that $\gamma^\psi(\widetilde{H}) = \gamma^\psi(\widetilde{H'})$. 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 $\alpha(H) \sqsubseteq \widetilde{H}$ 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 $\alpha(H) \sqsubseteq \widetilde{H'}$, therefore $H \in \gamma^\varphi(\widetilde{H'})$. The other direction is symmetrical. $\square$

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 $[x_n > 2]$. Mainly, it proves that size information is treated in a sound and complete fashion. In particular, this theorem shows that the above are the only cases we need to consider in order to saturate with respect to size information.

Theorem 5 *Let $\widetilde{H} = \langle \overline{H}, \mathrm{o}, \preceq_{ff}, \preceq_{fa}, \preceq_{af}, \preceq_{aa} \rangle$ be an abstract heap, $\overline{H} = \langle \overline{N}, \overline{S}, \overline{V} \rangle$ 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: 1. “$\Rightarrow$” If $\widetilde{H}$ is saturated w.r.t. $x_n = 1$ then it is saturated, by Lemma 6 (we have $\models x_n = 1 \rightarrow \top$). 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 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 $\gamma^{x_n=1}(\widetilde{H}) = \gamma^{x_n=1}(\widetilde{H'}) \Rightarrow \gamma(\widetilde{H}) = \gamma(\widetilde{H'})$ and use the fact that $\widetilde{H}$ is saturated to obtain $\widetilde{H} \sqsubseteq \widetilde{H'}$. Let $H \in \gamma(\widetilde{H})$ be an arbitrary concretisation of $\widetilde{H}$, and β be the mapping of list segments 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, $\alpha(H) = \alpha(H')$, hence $\alpha(H') \subseteq \widetilde{H}$, therefore $H' \in \gamma^{x_n=1}(\widetilde{H}) = \gamma^{x_n=1}(\widetilde{H'})$. The latter implies $\alpha(H) = \alpha(H') \subseteq \widetilde{H'}$, i.e. $H \in \gamma(\widetilde{H'})$. The other direction is symmetric.
2. “$\Rightarrow$” This point is similar to the $\Rightarrow$ direction of 1. “$\Leftarrow$” It is sufficient to show that $\gamma^{x_n=2}(\widetilde{H}) = \gamma^{x_n=2}(\widetilde{H'}) \Rightarrow \gamma(\widetilde{H}) = \gamma(\widetilde{H'})$. Let $H \in \gamma(\widetilde{H})$, and β be the corresponding mapping. 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 $\alpha(H) = \alpha(H')$ which leads to $H' \in \gamma^{x_n=2}(\widetilde{H}) = \gamma^{x_n=2}(\widetilde{H'})$, therefore $\alpha(H) \subseteq \widetilde{H'}$, 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 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 consits now of only one element, and namely the one with maximum value in H. One can easily show that $\alpha(H) = \alpha(H')$, since no predicate needs to be updated as result of the transformation. By the same argument as above, we obtain $H \in \gamma(H')$.
    - H' is built now from H by keeping only the minimum and maximum elements of $\beta^{-1}(n)$ possibly in reversed order. In this way one does not introduce $\mathrm{o}(n)$ in $\alpha(H')$ when not necessary (notice that $\mathrm{o}(n)$ might not hold in $\alpha(H)$) and we can still show that $\alpha(H) = \alpha(H')$.

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

5 Experimental Results

In order to obtain experimental evidence about how our techniques behave in practice, we have applied them to several non-trivial procedures manipulating singly-linked lists. In particular, we have considered a procedure for *reversing lists*, whose behaviour we have studied both for an *acyclic* as well as *cyclic* input, and then two procedures for sorting lists, namely *InsertSort* and *BubbleSort*.

For all the examples, we generated (by hand—an implementation of the translation procedure is our future work) the corresponding counter automata. Sizes of the automata—after some trivial simplifications joining sequences of states with no variation in the underlying heap graph—varied as follows: (1) 15 states and 3 counters for reversing acyclic lists (no optimizations were used in this case), (2) 11 states and 3 counters for reversing cyclic lists, (3) 88 states and 6 counters for InsertSort, and (4) 149 states and 7 counters for BubbleSort (we considered the more practical version of the sort with a pointer remembering the already sorted part of the list). For list reversing, no ordering predicates were used.

As for the *safety properties* of the considered programs, we checked that there are no null pointer assignments, no elements are lost, the shape is preserved, and—in the case of the sorting algorithms—that the result is sorted. These properties may be checked by generating a symbolically encoded set of the reachable configurations of the counter automaton corresponding to the program. Using an implementation of the abstract regular model checking technique [8] based on LASH automata libraries [1], the verification took 10 sec for the acyclic list reversion case study and 0.5 sec for cyclic list reversion on a Pentium 4 machine with a 2.6 GHz processor.

Moreover, let us note that all the above properties may often be checked already at the *counter automaton extraction phase*. The checking is mostly straightforward. A slight complication is just checking that no elements of the list are lost via the `u.next := w` operations. However, even here a simple (fully automatable) heuristic may be used. When we generate a counter automaton state containing a new abstract heap and we can grant that some of its nodes have size one (e.g., after a `u := w.next` statement), we remember this fact. Later when we again encounter such a heap and we cannot statically guarantee that the appropriate nodes have size one, we may drop the information. Then, when we see that an `u.next := w` operation is performed on a node for which we remembered that its size is one, we know that we do not loose any list elements. If this is not the case, we have to analyse the dynamic behaviour of the counter automaton and check whether it may actually happen that we loose some elements. In *all* our examples, however, we were able to perform all the checks statically.

In addition to checking safety properties, we have also fully-automatically checked that all the considered programs *terminate*. For checking termination, we analysed the generated counter automata using the tool described in [11]. On the same machine as above, we were able to check termination in 4 sec for reversing acyclic lists, 1.5 sec for reversing cyclic lists, 90 sec for InsertSort, and 150 sec for BubbleSort.

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. These infinite-state models can be handled using various advanced techniques and tools which have been designed recently for their automatic analysis (e.g., [1, 2, 5]), and in particular concerning checking termination and liveness properties (e.g., [11, 10]). 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. Our future work naturally includes extending this approach to more general linked structures such as doubly linked lists, tree-like structures, etc.

References

[1] The LASH toolset. http://www.montefiore.ulg.ac.be/˜ boigelot/research/lash/. 5, 6

[2] A. Bouajjani A. Annichini and M.Sighireanu. TReX: A Tool for Reachability Analysis of Complex Systems. In *Proc. of CAV'01*, volume 2102 of *LNCS*, 2001. 6

[3] I. Balaban, A. Pnueli, and L. Zuck. Shape Analysis by Predicate Abstraction. In *Proc. of VMCAI'05*, volume 3385 of *LNCS*, 2005. 1

[4] S. Bardin, A.Finkel, and D. Nowak. Toward Symbolic Verification of Programs Handling Pointers. In *Proc. of AVIS'04*, 2004. 1, 4

[5] 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. 6

[6] J. Berdine, C. Calcagno, and P. O'Hearn. A Decidable Fragment of Separation Logic. In *Proc. of FSTTCS'04*, volume 3328 of *LNCS*, 2004. 1

[7] 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*, 2005. 1

[8] A. Bouajjani, P. Habermehl, and T. Vojnar. Abstract Regular Model Checking. In *Proc. of CAV'04*, volume 3114 of *LNCS*, 2004. 1, 5

[9] M. Bozga and R. Iosif. Quantitative Verification of Programs with Lists. Technical report, 2005. 1

[10] A. Bradley, Z. Manna, and H. Sipma. Termination Analysis of Integer Linear Loops. In *Proc. of CONCUR'05*, volume 3653 of *LNCS*, 2005. 6

[11] B. Cook, A. Podelski, and A. Rybalchenko. Abstraction Refinement for Termination. In *Proc. of SAS'05*, volume 3672 of *LNCS*, 2005. 1, 5, 6

[12] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. of POPL'97*, 1977. 4.3

[13] N. Dor, M. Rodeh, and S. Sagiv. Checking Cleanness in Linked Lists. In *Proc. of SAS'00*, volume 1824 of *LNCS*, 2000. 1

[14] A. Finkel, 2006. Personal communication. 1

[15] R. Iosif. Symmetry Reductions for Model Checking of Concurrent Dynamic Software. *STTT*, pages 302–319, 2004. 4

[16] S. Ishtiaq and P. O'Hearn. BI as an assertion language for mutable data structures. In *Proc. of POPL'01*, 2001. 1

[17] 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*, 2005. 1, 4, 1

[18] A. Møller and M.I. Schwartzbach. The Pointer Assertion Logic Engine. In *Proc. of PLDI'01*. ACM Press, 2001. 1

[19] M. Presburger. Über die Vollstandigkeit eines Gewissen Systems der Arithmetik. *Comptes Rendus du I Congrés des Pays Slaves*, 1929. 3

[20] S. Sagiv, T.W. Reps, and R. Wilhelm. Parametric Shape Analysis via 3-Valued Logic. *TOPLAS*, 2002. 1

[21] 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*, 2003. 1

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

$$\frac{\begin{array}{c} \overline{V}(u) = n \in \overline{N} \quad \forall w \in Var \setminus \{u\} \ .\ \overline{V}(w) \neq n \\ \exists m, p \in \overline{N} \setminus \{n\} \ .\ p \neq m \ \wedge\ \overline{S}(m) = \overline{S}(p) = n \end{array}}{\overline{H} \xrightarrow[u:=null]{true} \langle \overline{N}, \overline{S}, \overline{V}[u \to \bot] \rangle}\ A_2'$$

$$\frac{\begin{array}{c} \overline{V}(u) = n \in \overline{N} \quad \forall w \in Var \setminus \{u\} \ .\ \overline{V}(w) \neq n \\ \exists m \in \overline{N} \setminus \{n\} \ .\ \overline{S}(m) = n \\ \forall p \in \overline{N} \setminus \{n\} \ .\ \overline{S}(p) \neq n \end{array}}{\overline{H} \xrightarrow[u:=null]{x_m' = x_m + x_n} \mu(\langle \overline{N}, \overline{S}, \overline{V}[u \to \bot] \rangle, m, n)}\ A_2''$$

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

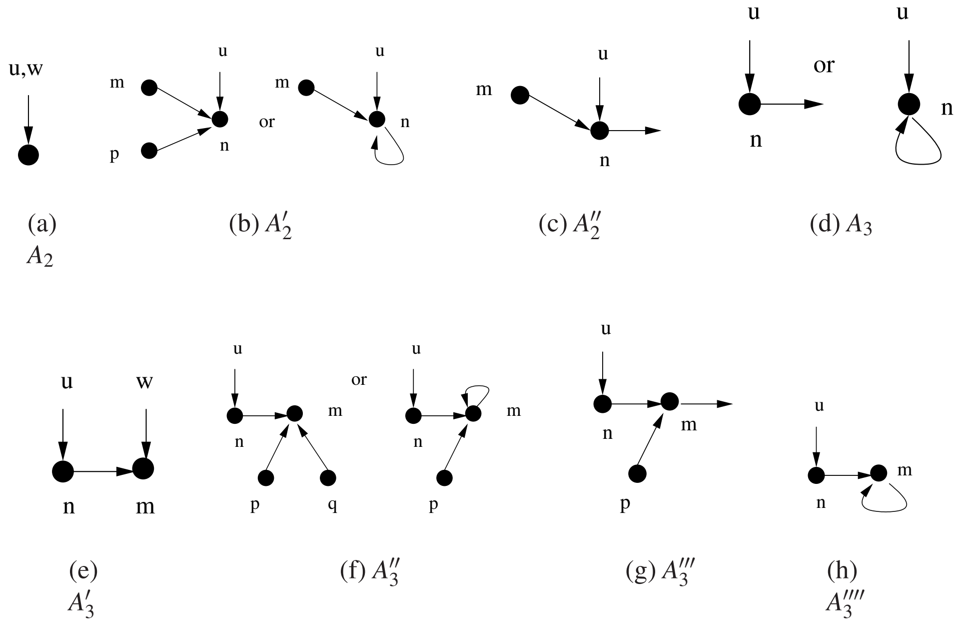
$$\frac{\begin{array}{l} \overline{V}(u) = n \in \overline{N} \quad \forall w \in Var \setminus \{u\} \ .\ w \not\xrightarrow[\overline{H}]{*} n \\ \overline{S}(n) = m \in \overline{N} \setminus \{n\} \\ \exists w \in Var \setminus \{u\} \ .\ V(w) = m \quad \overline{N'} = \overline{N} \setminus \{n\}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

$$\frac{\begin{array}{l} \overline{V}(u) = n \in \overline{N} \quad \forall w \in Var \setminus \{u\} \ .\ w \not\xrightarrow[\overline{H}]{*} n \\ \overline{S}(n) = m \in \overline{N} \setminus \{n\} \\ \forall w \in Var \setminus \{u\} \ .\ \overline{V}(w) \neq m \\ \exists p, q \in \overline{N} \setminus \{n\} \ .\ p \neq q \wedge \overline{S}(p) = m \wedge \overline{S}(q) = m \\ \overline{N'} = \overline{N} \setminus \{n\} \end{array}}{\overline{H} \xrightarrow[u:=null]{true} \langle \overline{N'}, \overline{S}\downarrow_{\overline{N'}}, \overline{V}\downarrow_{\overline{N'}} \rangle}\ A_3''$$

$$\frac{\begin{array}{l} \overline{V}(u) = n \in \overline{N} \quad \forall w \in Var \setminus \{u\} \ .\ w \not\xrightarrow[\overline{H}]{*} n \\ \overline{S}(n) = m \in \overline{N} \setminus \{n\} \\ \forall w \in Var \setminus \{u\} \ .\ \overline{V}(w) \neq m \\ \exists p \in \overline{N} \setminus \{n, m\} \ .\ \overline{S}(p) = m \\ \forall q \in \overline{N} \setminus \{n, p\} \ .\ \overline{S}(q) \neq m \quad \overline{N'} = \overline{N} \setminus \{n\} \end{array}}{\overline{H} \xrightarrow[u:=null]{x_p' = x_p + x_m} \mu(\langle \overline{N'}, \overline{S}\downarrow_{\overline{N'}}, \overline{V}\downarrow_{\overline{N'}} \rangle, p, m)}\ A_3'''$$

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

Figure 4: **Counter Automaton Semantics Part 1** Let $\overline{H} \triangleq \langle \overline{N}, \overline{S}, \overline{V} \rangle$. The merging function is $\mu : \mathcal{H}(Var) \times \mathcal{N} \times \mathcal{N} \to \mathcal{H}(Var)$ given by $\mu(\overline{H}, n, m) = \langle \overline{N'}, \overline{S}\downarrow_{\overline{N'}} [n \to \overline{S}(m)], \overline{V} \rangle$ where $\overline{N'} = \overline{N} \setminus \{m\}$. The splitting function is $\sigma : \mathcal{H}(Var) \times \mathcal{N} \times \mathcal{N} \to \mathcal{H}(Var)$ given by $\sigma(\overline{H}, n, m) = \langle \overline{N} \cup \{m\}, \overline{S'}, \overline{V} \rangle$ where $\overline{S'} = \big(\overline{S} \setminus \{(n, p) \mid n \xrightarrow[\overline{H}]{} p\}\big) \cup \{(m, p) \mid n \xrightarrow[\overline{H}]{} p\} \cup \{(n, m)\}$.



(a) A_2 (b) A'_2 (c) A''_2 (d) A_3

(e) A'_3 (f) A''_3 (g) A'''_3 (h) A''''_3

Figure 5: The different cases for $u := null$ illustrated

$$\frac{n \in \mathcal{N} \setminus \overline{N}}{\overline{H} \xrightarrow[u:=new]{x'_n=1} \langle \overline{N} \cup \{n\}, \overline{S}[n \rightarrow \bot], \overline{V}[u \rightarrow n] \rangle} \; A_5$$

$$\frac{\overline{V}(w) = n \in \overline{N}}{\overline{H} \xrightarrow[u:=w.next]{x_n=1} \langle \overline{N}, \overline{S}, \overline{V}[u \rightarrow \overline{S}(n)] \rangle} \; A_7 \qquad \frac{\overline{V}(w) = n \in \overline{N} \quad m \in \mathcal{N} \setminus \overline{N'}}{\overline{H} \xrightarrow[u:=w.next]{x_n>1 \wedge x'_m=x_n-1} \sigma(\langle \overline{N}, \overline{S}, \overline{V}[u \rightarrow m] \rangle, n, m)} \; A'_7$$

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

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

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

Figure 6: **Counter Automaton Semantics Part 2**

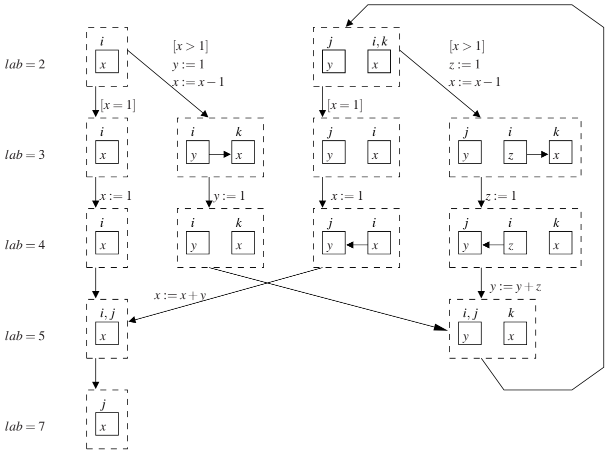


Figure 7: Non-circular List Reversal

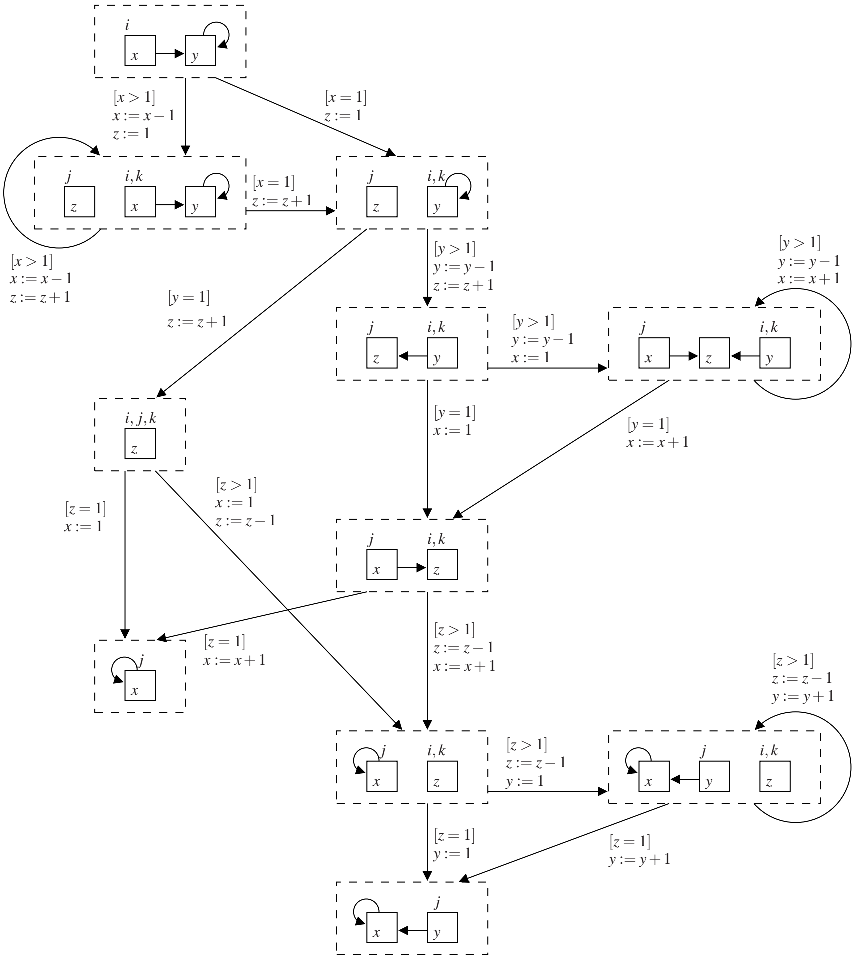


Figure 8: Circular List Reversal

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. $\mathrm{o}(n) \;\Rightarrow\; n \preceq_{fa} n$

Figure 9: Saturation rules