Automated Analysis of Data-Dependent Programs with Dynamic Memory

Parosh Aziz Abdulla[1], Muhsin Atto[2], Jonathan Cederberg[1], and Ran Ji[3]

[1] Uppsala University, Sweden
[2] University of Duhok, Kurdistan-Iraq
[3] Chalmers University of Technology, Gothenburg, Sweden

Abstract. We present a new approach for automatic verification of data-dependent programs manipulating dynamic heaps. A heap is encoded by a graph where the nodes represent the cells, and the edges reflect the pointer structure between the cells of the heap. Each cell contains a set of variables which range over the natural numbers. Our method relies on standard backward reachability analysis, where the main idea is to use a simple set of predicates, called *signatures*, in order to represent bad sets of heaps. Examples of bad heaps are those which contain either garbage, lists which are not well-formed, or lists which are not sorted. We present the results for the case of programs with a single next-selector, and where variables may be compared for (in)equality. This allows us to verify for instance that a program, like bubble sort or insertion sort, returns a list which is well-formed and sorted, or that the merging of two sorted lists is a new sorted list. We report on the result of running a prototype based on the method on a number of programs.

1 Introduction

We consider the automatic verification of data-dependent programs that manipulate dynamic linked lists. The contents of the linked lists, here refered to as a *heap*, is represented by a graph. The nodes of the graph represent the cells of the heap, while the edges reflect the pointer structure between the cells (see Figure 1 for a typical example).

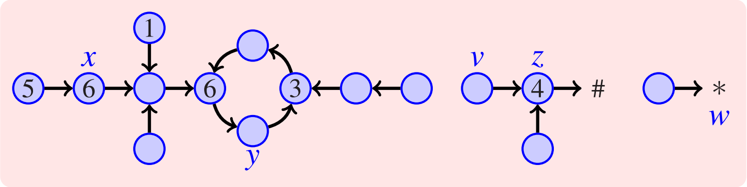


Fig. 1. A typical graph representing the heap

The program has a dynamic behaviour in the sense that cells may be created and deleted; and that pointers may be re-directed during the execution of the program. The program is also data-dependent since the cells contain variables, ranging over the natural numbers, that can be compared for (in)equality and whose values may be updated by the program. The values of the local variables are provided as attributes to the corresponding cells. Finally, we have a set of (pointer) variables which point to different cells inside the heap.

Z. Liu and A.P. Ravn (Eds.): ATVA 2009, LNCS 5799, pp. 197–212, 2009.

In this paper, we consider the case of programs with a single next-selector, i.e., where each cell has at most one successor. For this class of programs, we give a method for automatic verification of safety properties. Such properties can be either *structural properties* such as absence of garbage, sharing, and dangling pointers; or *data properties* such as sortedness and value uniqueness. We provide a simple symbolic representation, which we call *signatures*, for characterizing (infinite) sets of heaps. Signatures can also be represented by graphs. One difference, compared to the case of heaps, is that some parts may be *missing* from the graph of a signature. For instance, the absence of a pointer means that the pointer may point to an arbitrary cell inside a heap satisfying the signature. Another difference is that we only store information about the *ordering* on values of the local variables rather than their exact values. A signature can be interpreted as a *forbidden pattern* which should not occur inside the heap. The forbidden pattern is essentially a set of minimal conditions which should be satisfied by any heap in order for the heap to satisfy the signature. A heap satisfying the signature is considered to be *bad* in the sense that it contains a bad pattern which in turn implies that it violates one of the properties mentioned above. Examples of bad patterns in heaps are garbage, lists which are not well-formed, or lists which are not sorted. This means that checking a safety property amounts to checking the reachability of a finite set of signatures. We perform standard backward reachability analysis, using signatures as a symbolic representation, and starting from the set of bad signatures. We show how to perform the two basic operations needed for backward reachability analysis, namely checking entailment and computing predecessors on signatures.

For checking entailment, we define a pre-order $\sqsubseteq$ on signatures, where we view a signature as three separate graphs with identical sets of nodes. The edge relation in one of the three graphs reflects the structure of the heap graph, while the other two reflect the ordering on the values of the variables (equality resp. inequality). Given two signatures g_1 and g_2, we have $g_1 \sqsubseteq g_2$ if g_1 can be obtained from g_2 by a sequence of transformations consisting of either deleting an edge (in one of the three graphs), a variable, an isolated node, or contracting segments (i.e., sequence of nodes) without sharing in the structure graph. In fact, this ordering also induces an ordering on heaps where $h_1 \sqsubseteq h_2$ if, for all signatures g, h_2 satisfies g whenever h_1 satisfies g.

When performing backward reachability analysis, it is essential that the underlying symbolic representation, signatures in our case, is closed under the operation of computing predecessors. More precisely, for a signature g, let us define $Pre(g)$ to be the set of *predecessors* of g, i.e., the set of signatures which characterize those heaps from which we can perform one step of the program and as a result obtain a heap satisfying g. Unfortunately, the set $Pre(g)$ does not exist in general under the operational semantics of the class of programs we consider in this paper. Therefore, we consider an over-approximation of the transition relation where a heap h is allowed first to move to smaller heap (w.r.t. the ordering $\sqsubseteq$) before performing the transition. For the approximated transition relation, we show that the set $Pre(g)$ exists, and that it is finite and computable.

One advantage of using signatures is that it is quite straightforward to specify sets of bad heaps. For instance, forbidden patterns for the properties of list well-formedness and absence of garbage can each be described by 4-6 signatures, with 2-3 nodes in each signature. Also, the forbidden pattern for the property that a list is sorted consists of only one signature with two nodes. Furthermore, signatures offer a very compact symbolic representation of sets of bad heaps. In fact, when verifying our programs, the number of nodes in the signatures which arise in the analysis does not exceed ten. In addition, the rules for computing predecessors are *local* in the sense that they change only a small part of the graph (typically one or two nodes and edges). This makes it possible to check entailment and compute predecessors quite efficiently.

The whole verification process is fully automatic since both the approximation and the reachability analysis are carried out without user intervention. Notice that if we verify a safety property in the approximate transition system then this also implies its correctness in the original system. We have implemented a prototype based on our method, and carried out automatic verification of several programs such as insertion in a sorted lists, bubble sort, insertion sort, merging of sorted lists, list partitioning, reversing sorted lists, etc. Although the procedure is not guaranteed to terminate in general, our prototype terminates on all these examples.

Outline. In the next section, we describe our model of heaps, and introduce the programming language together with the induced transition system. In Section 3, we introduce the notion of signatures and the associated ordering. Section 4 describes how to specify sets of bad heaps using signatures. In Section 5 we give an overview of the backward reachability scheme, and show how to compute the predecessor and entailment relations on signatures. The experimental results are presented in Section 6. In Section 7 we give some conclusions and directions for future research. Finally, in Section 8, we give an overview of related approaches and the relationship to our work.

2 Heaps

In this section, we give some preliminaries on programs which manipulate heaps.

Let $\mathbb{N}$ be the set of natural numbers. For sets A and B, we write $f : A \rightarrow B$ to denote that f is a (possibly partial) function from A to B. We write $f(a) = \perp$ to denote that $f(a)$ is undefined. We use $f[a \leftarrow b]$ to denote the function f' such that $f'(a) = b$ and $f'(x) = f(x)$ if $x \neq a$. In particular, we use $f[a \leftarrow \perp]$ to denote the function f' which agrees on f on all arguments, except that $f'(a)$ is undefined.

Heaps. We consider programs which operate on dynamic data structures, here called *heaps*. A heap consists of a set of *memory cells* (*cells* for short), where each cell has one next-pointer. Examples of such heaps are singly liked lists and circular lists, possibly sharing their parts (see Figure 1). A cell in the heap may contain a datum which is a natural number. A program operating on a heap may use a finite set of *variables* representing *pointers* whose values are cells inside the heap. A pointer may have the special value `null` which represents a cell without successors. Furthermore, a pointer may be *dangling* which means that it does not point to any cell in the heap. Sometimes, we write the “x-cell” to refer to the the cell pointed to by the variable x. We also write “the value of the x-cell” to refer to the value stored inside the cell pointed to by x. A heap can naturally be encoded by a graph, as the one of Figure 1. A vertex in the graph represents a cell in the heap, while the edges reflect the successor (pointer) relation on the cells. A variable is attached to a vertex in the graph if the variable points to the corresponding cell in the heap. Cell values are written inside the nodes (absence of a number means that the value is undefined).

Assume a finite set X of variables. Formally, a *heap* is a tuple $(M, Succ, \lambda, Val)$ where

- M is a finite set of *(memory) cells*. We assume two special cells # and $*$ which represent the constant `null` and the *dangling* pointer value respectively. We define $M^{\bullet} := M \cup \{\#, *\}$.
- $Succ : M \rightarrow M^{\bullet}$. If $Succ(m_1) = m_2$ then the (only) pointer of the cell m_1 points to the cell m_2. The function *Succ* is total which means that each cell in M has a successor (possibly # or $*$). Notice that the special cells # and $*$ have no successors.
- $\lambda : X \rightarrow M^{\bullet}$ defines the cells pointed to by the variables. The function λ is total, i.e., each variable points to one cell (possibly # or $*$).
- $Val : M \rightarrow \mathbb{N}$ is a partial function which gives the values of the cells.

In Figure 1, we have 17 cells of which 15 are in M, The set X is given by $\{x, y, z, v, w\}$. The successor of the z-cell is `null`. Variable w is attached to the cell $*$, which means that w is dangling (w does not point to any cell in the heap). Furthermore, the value of the x-cell is 6, the value of the y-cell is not defined, the value of the successor of the y-cell is 3, etc.

Remark. In fact, we can allow cells to contain multiple values. However, to simplify the presentation, we keep the assumption that a cell contains only one number. This will be sufficient for our purposes; and furthermore, all the definitions and methods we present in the paper can be extended in a straightforward manner to the general case. Also, we can use ordered domains other than the natural numbers such as the integers, rationals, or reals.

Programming Language. We define a simple programming language. To this end, we assume, together with the earlier mentioned set X of variables, the constant `null` where `null` $\notin X$. We define $X^{\#} := X \cup \{\texttt{null}\}$. A *program* P is a pair (Q, T) where Q is a finite set of *control states* and T is a finite set of *transitions*. The control states represent the locations of the program. A transition is a triple (q_1, op, q_2) where $q_1, q_2 \in Q$ are control states and *op* is an *operation*. In the transition, the program changes location from q_1 to q_2, while it checks and manipulates the heap according to the operation *op*. The operation *op* is of one of the following forms

- $x = y$ or $x \neq y$ where $x, y \in X^{\#}$. The program checks whether the x- and y-cells are identical or different.
- $x := y$ or $x.next := y$ where $x \in X$ and $y \in X^{\#}$. In the first operation, the program makes x point to the y-cell, while in the second operation it updates the successor of the x-cell, and makes it equal to the y-cell.
- $x := y.next$ where $x, y \in X$. The variable x will now point to the successor of the y-cell.
- $new(x)$, $delete(x)$, or $read(x)$, where $x \in X$. The first operation creates a new cell and makes x point to it; the second operation removes the x-cell from the heap; while the third operation reads a new value and assigns it to the x-cell.

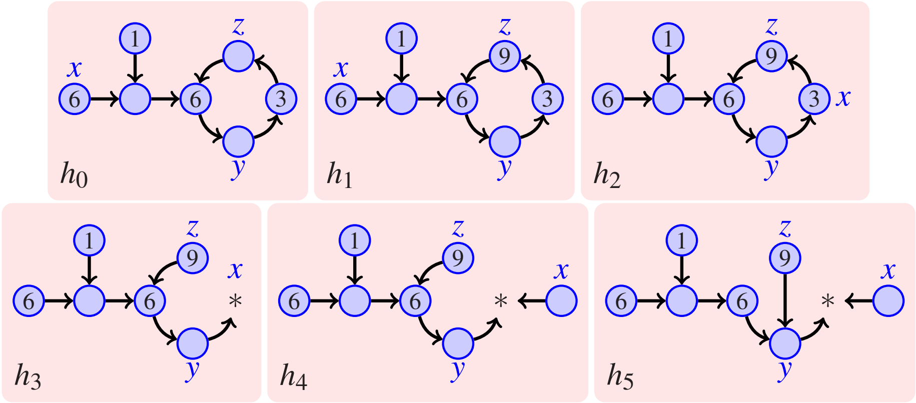


Fig. 2. Starting from the heap h_0, the heaps h_1, h_2, h_3, h_4, and h_5 are generated by performing the following sequence of operations: $z.num :> x.num$, $x := y.next$, $delete(x)$, $new(x)$, and $z.next := y$. To simplify the figures, we omit the special nodes # and $*$ unless one of the variables x, y, z is attached to them. For this reason the cell # is missing in all the heaps, and $*$ is present only in h_3, h_4, h_5.

- $x.num = y.num$, $x.num < y.num$, $x.num := y.num$, $x.num :> y.num$, or $x.num :< y.num$, where $x, y \in X$. The first two operations compare the values of (number stored inside) the x- and y-cells. The third operation copies the value of the y-cell to the x-cell. The fourth (fifth) operation assigns non-deterministically a value to the x-cell which is larger (smaller) than that of the y-cell.

Figure 2 illustrates the effect of a sequence of operations of the forms described above on a number of heaps. Examples of some programs can be found in [2].

Transition System. We define the operational semantics of a program $P = (Q, T)$ by giving the transition system induced by P. In other words, we define the set of configurations and a transition relation on configurations. A *configuration* is a pair (q, h) where $q \in Q$ represents the location of the program, and h is a heap.

We define a transition relation (on configurations) that reflects the manner in which the instructions of the program change a given configuration. First, we define some operations on heaps. Fix a heap $h = (M, Succ, \lambda, Val)$. For $m_1, m_2 \in M$, we use $(h.Succ)[m_1 \leftarrow m_2]$ to denote the heap h' we obtain by updating the successor relation such that the cell m_2 now becomes the successor of m_1 (without changing anything else in h). Formally, $h' = (M, Succ', Val, \lambda)$ where $Succ' = Succ[m_1 \leftarrow m_2]$. Analogously, $(h.\lambda)[x \leftarrow m]$ is the heap we obtain by making x point to the cell m; and $(h.Val)[m \leftarrow i]$ is the heap we obtain by assigning the value i to the cell m. For instance, in Figure 2, let h_i be of the form $(M_i, Succ_i, Val_i, \lambda_i)$ for $i \in \{0,1,2,3,4,5\}$. Then, we have $h_1 = (h_0.Val)[\lambda_0(z) \leftarrow 9]$ since we make the value of the z-cell equal to 9. Also, $h_2 = (h_1.\lambda_1)[x \leftarrow Succ_1(\lambda_1(y))]$ since we make x point to the successor of the y-cell. Furthermore, $h_5 = (h_4.Succ_4)[\lambda_4(z) \leftarrow \lambda_4(y)]$ since we make the y-cell the successor of the z-cell.

Consider a cell $m \in M$. We define $h \ominus m$ to be the heap h' we get by deleting the cell m from h. More precisely, we define $h' := (M', Succ', \lambda', Val')$ where

- $M' = M - \{m\}$.
- $Succ'(m') = Succ(m')$ if $Succ(m') \neq m$, and $Succ'(m') = *$ otherwise. In other words, the successor of cells pointing to m will become dangling in h'.
- $\lambda'(x) = *$ if $\lambda(x) = m$, and $\lambda'(x) = \lambda(x)$ otherwise. In other words, variables pointing to the same cell as x in h will become dangling in h'.
- $Val'(m') = Val(m')$ if $m' \in M'$. That is, the function Val' is the restriction of Val to M': it assigns the same values as Val to all the cells which remain in M' (since $m \notin M'$, it not meaningful to speak about $Val(m)$).

In Figure 2, we have $h_3 = h_2 \ominus \lambda_2(x)$.

Let $t = (q_1, op, q_2)$ be a transition and let $c = (q, h)$ and $c' = (q', h')$ be configurations. We write $c \xrightarrow{t} c'$ to denote that $q = q_1$, $q' = q_2$, and $h \xrightarrow{op} h'$, where $h \xrightarrow{op} h'$ holds if we obtain h' by performing the operation op on h. For brevity, we give the definition of the relation $\xrightarrow{op}$ for three types of operations. The rest of the cases can be found in [2].

- op is of the form $x := y.next$, $\lambda(y) \in M$, $Succ(\lambda(y)) \neq *$, and $h' = (h.\lambda)[x \leftarrow Succ(\lambda(y))]$.
- op is of the from $new(x)$, $M' = M \cup \{m\}$ for some $m \notin M$, $\lambda' = \lambda[x \leftarrow m]$, $Succ' = Succ[m \leftarrow *]$, $Val'(m') = Val(m')$ if $m' \neq m$, and $Val'(m) = \perp$. This operation creates a new cell and makes x point to it. The value of the new cell is not defined, while the successor is the special cell $*$. As an example of this operation, see the transition from h_3 to h_4 in Figure 2.
- op is of the form $x.num :> y.num$, $\lambda(x) \in M$, $\lambda(y) \in M$, $Val(\lambda(y)) \neq \perp$, and $h' = (h.Val)[\lambda(x) \leftarrow i]$, where $i > Val(\lambda(y))$.

We write $c \longrightarrow c'$ to denote that $c \xrightarrow{t} c'$ for some $t \in T$; and use $\xrightarrow{*}$ to denote the reflexive transitive closure of $\longrightarrow$. The relations $\longrightarrow$ and $\xrightarrow{*}$ are extended to sets of configurations in the obvious manner.

Remark. One could also allow deterministic assignment operations of the form $x.num := y.num + k$ or $x.num := y.num - k$ for some constant k. However, according the approximate transition relation which we define in Section 5, these operations will have identical interpretations as the non-deterministic operations given above.

3 Signatures

In this section, we introduce the notion of *signatures*. We will define an ordering on signatures from which we derive an ordering on heaps. We will then show how to use signatures as a symbolic representation of infinite sets of heaps.

Signatures. Roughly speaking, a signature is a graph which is "less concrete" than a heap in the following sense:

- We do not store the actual values of the cells in a signature. Instead, we define an ordering on the cells which reflects their values.
- The functions $Succ$ and λ in a signature are partial (in contrast to a heap in which these functions are total).

Formally, a signature g is a tuple of the form $(M, Succ, \lambda, Ord)$, where M, $Succ$, λ are defined in the same way as in heaps (Section 2), except that $Succ$ and λ are now *partial*. Furthermore, Ord is a partial function from $M \times M$ to the set $\{\prec, \equiv\}$. Intuitively, if $Succ(m) = \perp$ for some cell $m \in M$, then g puts no constraints on the successor of m, i.e., the successor of m can be any arbitrary cell. Analogously, if $\lambda(x) = \perp$, then x may point to any of the cells. The relation Ord constrains the ordering on the cell values. If $Ord(m_1, m_2) = \prec$ then the value of m_1 is strictly smaller than that of m_2; and if $Ord(m_1, m_2) = \equiv$ then their values are equal. This means that we abstract away the actual values of the cells, and only keep track of their ordering (and whether they are equal). For a cell m, we say that the value of m is *free* if $Ord(m, m') = \perp$ and $Ord(m', m) = \perp$ for all other cells m'. Abusing notation, we write $m_1 \prec m_2$ (resp. $m_1 \equiv m_2$) if $Ord(m_1, m_2) = \prec$ (resp. $Ord(m_1, m_2) = \equiv$).

We represent signatures graphically in a manner similar to that of heaps. Figure 3 shows graphical representations of six signatures $g_0, \ldots, g_5$ over the set of variables $\{x, y, z\}$. If a vertex in the graph has no successor, then the successor of the corresponding cell is not defined in g (e.g., the y-cell in g_4). Also, if a variable is missing in the graph, then this means that the cell to which the variable points is left unspecified (e.g., variable z in g_3). The ordering Ord on cells is illustrated by dashed arrows. A dashed single-headed arrow from a cell m_1 to a cell m_2 indicates that $m_1 \prec m_2$. A dashed double-headed arrow between m_1 and m_2 indicates that $m_1 \equiv m_2$. To simplify the figures, we omit self-loops indicating value reflexivity (i.e., $m \equiv m$). In this manner, we can view a signature as three graphs with a common set of vertices, and with three edge relations; where the first edge relation gives the graph structure, and the other two define the ordering on cell values (inequality resp. equality).

In fact, each heap $h = (M, Succ, \lambda, Val)$ induces a unique signature which we denote by $sig(h)$. More precisely, $sig(h) := (M, Succ, \lambda, Ord)$ where, for all cells $m_1, m_2 \in M$, we have $m_1 \prec m_2$ iff $Val(m_1) < Val(m_2)$ and $m_1 \equiv m_2$ iff $Val(m_1) = Val(m_2)$. In other words, in the signature of h, we remove the concrete values in the cells and replace them by the ordering relation on the cell values. For example, in Figure 2 and Figure 3, we have $g_0 = sig(h_0)$.

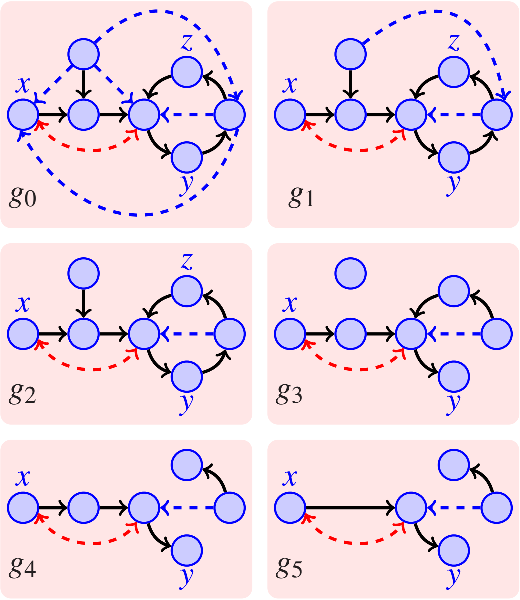


Fig. 3. Examples of signatures

Signature Ordering. We define an *entailment relation*, i.e., ordering $\sqsubseteq$ on signatures. The intuition is that each signature can be interpreted as a predicate which characterizes an infinite set of heaps. The ordering is then the inverse of implication: smaller signatures impose less restrictions and hence characterize larger sets of heaps. We derive a small signature from a larger one, by deleting cells, edges, variables in the graph of the signature, and by weakening the ordering requirements on the cells (the latter corresponds to deleting edges encoding the two relations on data values). To define the ordering, we give first definitions and describe some operations on signatures. Fix a signature $g = (M, Succ, \lambda, Ord)$.

A cell $m \in M$ is said to be *semi-isolated* if there is no $x \in X$ with $\lambda(x) = m$, the value of m is free, $Succ^{-1}(m) = \emptyset$, and either $Succ(m) = \perp$ or $Succ(m) = *$. In other words, m is not pointed to by any variables, its value is not related to that of any other cell, it has no predecessors, and it has no successors (except possibly $*$). We say that m is *isolated* if it is semi-isolated and in addition $Succ(m) = \perp$. A cell $m \in M$ is said to be *simple* if there is no $x \in X$ with $\lambda(x) = m$, the value of m is free, $|Succ^{-1}(m)| = 1$, and $Succ(m) \neq \perp$. In other words, m has exactly one predecessor, one successor and no label. In Figure 3, the topmost cell of g_3 is isolated, and the successor of the x-cell in g_4 is simple. In Figure 1, the cell to the left of the w-cell is semi-isolated in the signature of the heap.

The operations $(g.Succ)[m_1 \leftarrow m_2]$ and $(g.\lambda)[x \leftarrow m]$ are defined in identical fashion to the case of heaps. Furthermore, for cells m_1, m_2 and $\square \in \{\prec, \equiv, \perp\}$, we define $(g.Ord)[(m_1, m_2) \leftarrow \square]$ to be the signature g' we obtain from g by making the ordering relation between m_1 and m_2 equal to $\square$. For a variable x, we define $g \ominus x$ to be the signature g' we get from g by deleting the variable x from the graph, i.e., $g' = (g.\lambda)[x \leftarrow \perp]$. For a cell m, we define the signature $g' = g \ominus m = (M', Succ', \lambda', Ord')$ in a manner similar to the case of heaps. The only difference is that Ord' (rather than Val') is the restriction of Ord to pairs of cells both of which are different from m.

Now, we are ready to define the ordering. For signatures $g = (M, Succ, \lambda, Ord)$ and $g' = (M', Succ', \lambda', Ord')$, we write that $g \lhd g'$ to denote that one of the following properties is satisfied:

- *Variable Deletion*: $g = g' \ominus x$ for some variable x,
- *Cell Deletion*: $g = g' \ominus m$ for some isolated cell $m \in M'$,
- *Edge Deletion*: $g = (g'.Succ)[m \leftarrow \perp]$ for some $m \in M'$,
- *Contraction*: there are cells $m_1, m_2, m_3 \in M'$ and a signature g_1 such that m_2 is simple, $Succ'(m_1) = m_2$, $Succ'(m_2) = m_3$, $g_1 = (g'.Succ)[m_1 \leftarrow m_3]$ and $g = g_1 \ominus m_2$, or
- *Order Deletion*: $g = (g'.Ord)[(m_1, m_2) \leftarrow \perp]$ for some cells $m_1, m_2 \in M'$.

We write $g \sqsubseteq g'$ to denote that there are $g_0 \lhd g_1 \lhd g_2 \lhd \cdots \lhd g_n$ with $n \geq 0$, $g_0 = g$, and $g_n = g'$. That is, we can obtain g from g' by performing a finite sequence of variable deletion, cell deletion, edge deletion, order deletion, and contraction operations. In Figure 3 we obtain: g_1 from g_0 through three order deletions; g_2 from g_1 through one order deletion; g_3 from g_2 through one variable deletion and two edge deletions; g_4 from g_3 through one node deletion and one edge deletion; and g_5 from g_4 through one contraction. It means that $g_5 \lhd g_4 \lhd g_3 \lhd g_2 \lhd g_1 \lhd g_0$ and hence $g_5 \sqsubseteq g_0$.

Heap Ordering

We define an ordering $\sqsubseteq$ on heaps such that $h \sqsubseteq h'$ iff $sig(h) \sqsubseteq sig(h')$. For a heap h and a signature g, we say that h *satisfies* g, denoted $h \models g$, if $g \sqsubseteq sig(h)$. In this manner, each signature characterizes an infinite set of heaps, namely the set $[\![g]\!] := \{h | h \models g\}$. Notice that $[\![g]\!]$ is upward closed w.r.t. the ordering $\sqsubseteq$ on heaps. We also observe that, for signatures g and g', we have that $g \sqsubseteq g'$ iff $[\![g']\!] \subseteq [\![g]\!]$. For a (finite) set G of signatures we define $[\![G]\!] := \bigcup_{g \in G} [\![g]\!]$. Considering the heaps of Figure 2 and the signatures of Figure 3, we have $h_1 \models g_0$, $h_2 \not\models g_0$, $h_0 \sqsubseteq h_1$, $h_0 \not\sqsubseteq h_2$, etc.

Remark. Our definition implies that signatures cannot specify "exact distances" between cells. For instance, we cannot specify the set of heaps in which the x-cell and the y-cell are exactly of distance one from each other. In fact, if such a heap is in the set then, since we allow contraction, heaps where the distance is larger than one will also be in the set. On the other hand, we can characterize sets of heaps where two cells are at distance at least k from each other for some $k \geq 1$.

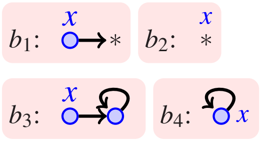
4 Bad Configurations

In this section, we show how to use signatures in order to specify sets of *bad heaps* for programs which produce ordered linear lists. A signature is interpreted as a *forbidden pattern* which should not occur inside the heap. Typically, we would like such a program to produce a heap which is a linear list. Furthermore, the heap should not contain any garbage, and the output list should be ordered. For each of these three properties, we describe the corresponding forbidden patterns as a set of signatures which characterize exactly those heaps which violate the property. Later, we will collect all these signatures into a single set which exactly characterizes the set of bad configurations.

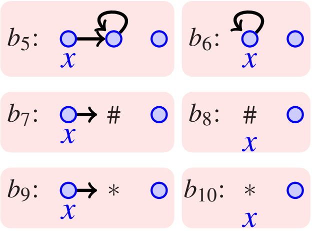
First, we give some definitions. Fix a heap $h = (M, Succ, \lambda, Val)$. A *loop* in h is a set $\{m_0, \ldots, m_n\}$ of cells such that $Succ(m_i) = m_{i+1}$ for all $i : 0 \leq i < n$, and $Succ(m_n) = m_0$. For cells $m, m' \in M$, we say that m' is *visible* from m if there are cells $m_0, m_1, \ldots, m_n$ for some $n \geq 0$ such that $m_0 = m$, $m_n = m'$, and $m_{i+1} = Succ(m_i)$ for all $i : 0 \leq i < n$. In other words, there is a (possibly empty) path in the graph leading from m to m'. We say that m' is *strictly visible* from m if $n > 0$ (i.e. the path is not empty). A set $M' \subseteq M$ is said to be *visible* from m if some $m' \in M'$ is visible from m.

Well-Formedness. We say that h is *well-formed* w.r.t a variable x if # is visible form the x-cell. Equivalently, neither the cell $*$ nor any loop is visible from the x-cell. Intuitively, if a heap satisfies this condition, then the part of the heap visible from the x-cell forms a linear list ending with #. For instance, the heap of Figure 1 is well-formed w.r.t. the variables v and z.

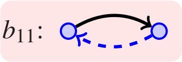
In Figure 2, h_0 is not well-formed w.r.t. the variables x and z (a loop is visible), and h_4 is not well-formed w.r.t. z (the cell $*$ is visible). The set of heaps violating well-formedness w.r.t. x are characterized by the four signatures in the figure to the right. The signatures b_1 and b_2 characterize (together) all heaps in which the cell $*$ is visible from the x-cell. The signatures b_3 and b_4 characterize (together) all heaps in which a loop is visible from the x-cell.



Garbage. We say that h contains *garbage* w.r.t a variable x if there is a cell $m \in M$ in h which is not visible from the x-cell. In Figure 2, the heap h_0 contains one cell which is garbage w.r.t. x, namely the cell with value 1. The figure to the right shows six signatures which together characterize the set of heaps which contain garbage w.r.t. x.



Sortedness. A heap is said to be *sorted* if it satisfies the condition that whenever a cell $m_1 \in M$ is visible from a cell $m_2 \in M$ then $Val(m_1) \leq Val(m_2)$. For instance, in Figure 2, only h_5 is sorted. The figure to the right shows a signature which characterizes all heaps which are not sorted.



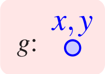
Putting Everything Together. Given a (reference) variable x, a configuration is considered to be *bad* w.r.t. x if it violates one of the conditions of being well-formed w.r.t. x, not containing garbage w.r.t. x, or being sorted. As explained above, the signatures $b_1, \ldots, b_{11}$ characterize the set of heaps which are bad w.r.t. x. We observe that $b_1 \sqsubseteq b_9$, $b_2 \sqsubseteq b_{10}$, $b_3 \sqsubseteq b_5$ and $b_4 \sqsubseteq b_6$, which means that the heaps $b_9.b_{10}, b_5, b_6$ can be discarded from the set above. Therefore, the set of bad configurations w.r.t. x is characterized by the set $\{b_1, b_2, b_3, b_4, b_7, b_8, b_{11}\}$.

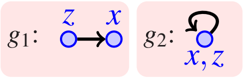
Remark. Other types of bad patterns can be defined in a similar manner. Examples can be found in [2].

5 Reachability Analysis

In this section, we show how to check safety properties through backward reachability analysis. First, we give an abstract transition relation $\longrightarrow_A$ which is an over-approximation of the transition relation $\longrightarrow$. Then, we describe how to compute predecessors of signatures w.r.t. $\longrightarrow_A$; and how to check the entailment relation. Finally, we introduce sets of *initial* heaps (from which the program starts running), and describe how to check safety properties using backward reachability analysis.

Over-Approximation. The basic step in backward reachability analysis is to compute the set of predecessors of sets of heaps characterized by signatures. More precisely, for a signature g and an operation *op*, we would like to compute a finite set G of signatures such that $[\![G]\!] = \left\{ h | \, h \xrightarrow{op} [\![g]\!] \right\}$. Consider the signature g to the right. The set $[\![g]\!]$ contains exactly all heaps where x and y point to the same cell. Consider the operation *op* defined by $y := z.next$. The set H of heaps from which we can perform the operation and obtain a heap in $[\![g]\!]$ are all those where the x-cell is the immediate successor of the z-cell. Since signatures cannot capture the immediate successor relation (see the remark in the end of Section 3), the set H cannot be characterized by a set G of signatures, i.e., there is no G such that $[\![G]\!] = H$. To overcome this problem, we define an approximate transition relation $\longrightarrow_A$ which is an over-approximation of the relation $\longrightarrow$. More precisely, for heaps h and h', we have $h \xrightarrow{op}_A h'$ iff there is a heap h_1 such that $h_1 \sqsubseteq h$ and $h_1 \xrightarrow{op} h'$.





Computing Predecessors. We show that, for an operation *op* and a signature g, we can compute a finite set $Pre(op)(g)$ of signatures such that $[\![Pre(op)(g)]\!] = \left\{h | h \xrightarrow{op}_A [\![g]\!]\right\}$. For instance in the above case the set $Pre(op)(g)$ is given by the $\{g_1, g_2\}$ shown in the figure to the right. Notice that $[\![\{g_1, g_2\}]\!]$ is the set of all heaps in which the x-cell is strictly visible from the z-cell. In fact, if we take any heap satisfying $[\![g_1]\!]$ or $[\![g_2]\!]$, then we can perform deletion and contraction operations (possibly several times) until the x-cell becomes the immediate successor of the z-cell, after which we can perform *op* thus obtaining a heap where x and y point to the same cell.

For each signature g and operation *op*, we show how to compute $Pre(op)(g)$ as a finite set of signatures. Due to lack of space, we show the definition only for the operation *new*. The definitions for the rest of the operations can be found in the [2]. For a cell $m \in M$ and a variable $x \in X$, we define m being *x-isolated* in a manner similar to m being *isolated*, except that we now allow m to be pointed to by x (and only x). More precisely, we say m is *x-isolated* if $\lambda(x) = m$, $\lambda(y) \neq m$ if $y \neq x$, the value of m is free, $Succ^{-1}(m) = \emptyset$, and $Succ(m) = \perp$. We define m being *x-semi-isolated* in a similar manner, i.e., by also allowing $*$ to be the successor of the x-cell. For instance, the leftmost cell of the signature b_1 in Section 4, and the x-cell in the signature $sig(h_5)$ in Figure 2 are x-semi-isolated.

We define $Pre(g)(new(x))$ to be the set of signatures g' such that one of the following conditions is satisfied:

- $\lambda(x)$ is x-semi-isolated, and there is a signature g_1 such that $g_1 = g \ominus \lambda(x)$ and $g' = g_1 \ominus x$.
- $\lambda(x) = \perp$ and $g' = g$ or $g' \in g \ominus m$ for some semi-isolated cell m.

Initial Heaps. A program starts running from a designated set H_{Init} of *initial heaps*. For instance, in a sorting program, H_{Init} is the set of well-formed lists which are (potentially) not sorted. Notice that this set is infinite since there is no bound on the lengths of the input lists. To deal with input lists, we follow the methodology of [7], and augment the program with an *initialization phase*. The program starts from an empty heap (denoted h_ε) and systematically (and non-deterministically) builds an arbitrary initial heap. In the case of sorting, the initial phase builds a well-formed list of an arbitrary length. We can now take the set H_{Init} to be the singleton containing the empty heap h_ε.

Checking Entailment. For signatures g and g', checking whether $g \sqsubseteq g'$ amounts to constructing an injection from the cells of g to those of g'. It turns out that a vast majority (more than 99%) of signatures, compared during the reachability analysis, are not related by entailment. Therefore, we have implemented a number of heuristics to detect negative answers as quickly as possible. An example is that a cell m in g should have (at most) the same labels as its image m' in g'; or that the in- and out-degrees of m are smaller than those of m'. The details of the entailment algorithm are included in [2].

Checking Safety Properties. To check a safety property, we start from the set G_{Bad} of bad signatures, and generate a sequence $G_0, G_1, G_2, \ldots$ of finite sets of signatures, where $G_0 = G_{Bad}$ and $G_{i+1} = \bigcup_{g \in G_i} Pre(g)$. Each time we generate a signature g such that $g' \sqsubseteq g$ for some already generated signature g', we discard g from the analysis.

We terminate the procedure when we reach a point where no new signatures can be added (all the new signatures are subsumed by existing ones). In such a case, we have generated a set G of signatures that characterize all heaps from which we can reach a bad heap through the approximate transition relation $\longrightarrow_A$. The program satisfies the safety property if $g \not\sqsubseteq sig(h_\varepsilon)$ for all $g \in G$.

6 Experimental Results

We have implemented the method described above in a prototype written in Java. We have run the tool on several examples, including all the benchmarks on singly linked lists with data known to us from the TVLA and PALE tools. Table 1 shows the results of our experiments. The column "#Sig." shows the total number of signatures that were computed throughout the analysis, the column "#Final" shows the number of signatures that remain in the visited set upon termination, the column "#Ent" shows the total number of calls to entailment that were made, and the last column shows the percentage of such calls that returned true. We have also considered buggy versions of some programs in which case the prototype reports an error.

All experiments were performed on a 2.2 GHz Intel Core 2 Duo with 4 GB of RAM. For each program, we verify well-formedness, absence of garbage, and sortedness. Also, in the case of the `Partition` program, we verify that the two resulting lists do not have common elements.

Table 1. Experimental results

Prog.	Time	#Sig.	#Final	#Ent	Ratio
`EfficientInsert`	0.1 s	44	40	1570	0.7%
`NonDuplicateInsert`	0.4 s	111	99	8165	0.2%
`Insert`	2.6 s	2343	1601	$2.2 \cdot 10^6$	0.03%
`Insert` (bug)	1.4 s	337	268	86000	0.09%
`Merge`	23.5 s	11910	5830	$3.6 \cdot 10^7$	0.017%
`Reverse`	1.5 s	435	261	70000	0.3%
`ReverseCyclic`	1.6 s	1031	574	375000	0.1%
`Partition`	2 m 49 s	21058	15072	$1.8 \cdot 10^8$	0.003%
`BubbleSort`	35.9 s	11023	10034	$7.5 \cdot 10^7$	0.001%
`BubbleSortCyclic`	36.6 s	11142	10143	$7.7 \cdot 10^7$	0.001%
`BubbleSort` (bug)	1.76 s	198	182	33500	0.07%
`InsertionSort`	11 m 53 s	34843	23324	$4.4 \cdot 10^8$	0.003%

7 Conclusions, Discussion, and Future Work

We have presented a method for automatic verification of safety properties for programs which manipulate heaps containing data. There are potentially two drawbacks of our method, namely the analysis is not guaranteed to *terminate*, and it may generate *false positives* (since we use an over-approximation). A sufficient condition for termination is *well quasi-ordering* of the entailment relation on signatures (see e.g. [3]). The only example known to us for *non-well-quasi-ordering* of this relation is based on a complicated sequence pattern by Nash-Williams (described in [13]) which shows non-well-quasi-ordering of permutations of sequences of natural numbers. Such artificial patterns are unlikely to ever appear in the analysis of typical pointer-manipulating programs. In fact, it is quite hard even to construct artificial programs for which the Nash-Williams pattern arises during backward reachability analysis. This is confirmed by the fact that our implementation terminates on all the given examples. As for false positives, the definition of the heap ordering $\sqsubseteq$ means that the abstract transition relation $\longrightarrow_A$ allows three types of imprecisions, namely it allows: (i) deleting garbage (nodes which are not visible from any variables), (ii) preforming contraction, and (iii) only storing the ordering on cell variables rather than their actual values. Program runs are not changed by (i) since we only delete nodes which are not accessible from the program pointers in the first place. Also, most program behaviors are not sensitive to the exact distances between nodes in a heap and therefore they are not affected by (ii). Finally, data-dependent programs (such as sorting or merge algorithms) check only ordering rather than complicated relations on data inside the heap cells. This explains why we do not get false positives on any of the examples on which we have run our implementation.

The experimental results are quite encouraging, especially considering the fact that our code is still highly unoptimized. For instance, most of the verification time is spent on checking entailment between signatures. We believe that adapting specialized algorithms, e.g. [20], for checking entailment will substantially improve performance of the tool.

Several extensions of our framework can be carried out by refining the considered preorder (and the abstraction it induces). For instance, if needed, our framework can be extended in a straightforward manner to handle arithmetical relations which are more complicated than simple ordering on data values such as *gap-order constraints* [17] or Presburger arithmetic. Given the fact that the analysis terminates on all benchmarks, it is tempting to characterize a class of programs which covers the current examples and for which termination is theoretically guaranteed. Another direction for future work is to consider more general classes of heaps with multiple selectors, and then study programs operating on data structures such as doubly-linked lists and trees both with and without data.

8 Related Work

Several works consider the verification of singly linked lists with data. The paper [14] presents a method for automatic verification of sorting programs that manipulate linked lists. The method is defined within the framework of TVLA which provides an abstract description of the heap structures in 3-valued logic [19]. The user may be required to provide *instrumentation predicates* in order to make the abstraction sufficiently precise. The analysis is performed in a forward manner. In contrast, the search procedure we describe in this paper is backward, and therefore also *property-driven*. Thus, the signatures obtained in the traversal do not need to express the state of the entire heap, but only those parts that contribute to the eventual failure. This makes the two methods conceptually and technically different. Furthermore, the difference in search strategy implies that forward and backward search procedures often offer varying degrees of efficiency in different contexts, which makes them complementary to each other in many cases. This has been observed also for other models such as parameterized systems, timed Petri nets, and lossy channel systems (see e.g. [4,9,1]).

Another approach to verification of linked lists with data is proposed in [6,7] based on *abstract regular model checking (ARMC)* [8]. In ARMC, finite-state automata are used as a symbolic representation of sets of heaps. This means that the ARMC-based approach needs the manipulation of quite complex encodings of the heap graphs into words or trees. In contrast, our symbolic representation uses signatures which provide a simpler and more natural representation of heaps as graphs. Furthermore, ARMC uses a sophisticated machinery for manipulating the heap encodings based on representing program statements as (word/tree) transducers. However, as mentioned above, our operations for computing predecessors are all *local* in the sense that they only update limited parts of the graph thus making it possible to have much more efficient implementations.

The paper [5] uses counter automata as abstract models of heaps which contain data from an ordered domain. The counters are used to keep track of lengths of list segments without sharing. The analysis reduces to manipulation of counter automata, and thus requires techniques and tools for these automata.

Recently, there has been an extensive work to use *separation logic* [18] for performing shape analysis of programs that manipulate pointer data structures (see e.g. [10,21]). The paper [16] describes how to use separation logic in order to provide a semi-automatic procedure for verifying data-dependent programs which manipulate heaps. In contrast, the approach we present here uses a built-in abstraction principle which is different from the ones used above and which makes the analysis fully automatic.

The tool PALE (Pointer Assertion Logic Engine) [15] checks automatically properties of programs manipulating pointers. The user is required to supply assertions expressed in the weak monadic second-order logic of graph types. This means that the verification procedure as a whole is only partially automatic. The tool MONA [11], which uses translations to finite-state automata, is employed to verify the provided assertions.

Recently, there have been several works which aim at algorithmic verification of systems whose configurations are finite graphs (e.g. [12,3]). These works may seem similar since they are all based on backward reachability using finite graphs as symbolic representations. However, they use different orderings on graphs which leads to entirely different methods for computing predecessor and entailment relations. In fact, the main challenge when designing verification algorithms on graphs, is to come up with the "right" notion of ordering: an ordering which allows computing entailment and predecessors, and which is sufficiently precise to avoid too many false positives. For instance, the *graph minor* ordering used in [12] to analyze distributed algorithms, is too weak to employ in shape analysis. The reason is that the contraction operation (in the case of the graph minor relation) is insensitive to the directions of the edges; and furthermore the ordering allows merging vertices which carry different labels (different variables), meaning that we would get false positives in almost all examples since they often rely tests like $x = y$ for termination. In our previous work [3], we combined abstraction with backward reachability analysis for verifying heap manipulating programs. However, the programs in [3] are restricted to be data-independent. The extension to the case of data-dependent programs requires a new ordering on graphs which involves an intricate treatment of structural and data properties. For instance, at the heap level, the data ordering amounts to keeping track of (in)equalities, while the structural ordering is defined in terms of garbage elimination and edge contractions (see the discussion in Section 7). This gives the two orderings entirely different characteristics when computing predecessors and entailment. Also, there is a non-trivial interaction between the structural and the data orderings. This is illustrated by the fact that even specifications of basic data-dependent properties like sortedness require forbidden patterns that contain edges from both orderings (see Section 4). Consequently, none of the programs we consider in this paper can be analyzed in the framework of [3]. In fact, since the programs here are data-dependent, the method of [3] may fail even to verify properties which are purely structural. For instance, the program `EfficientInsert` (described in [2]) gives a false non-well-formedness warning if data is abstracted away.

References

1. Abdulla, P.A., Annichini, A., Bouajjani, A.: Using forward reachability analysis for verification of lossy channel systems. Formal Methods in System Design (2004)
2. Abdulla, P.A., Atto, M., Cederberg, J., Ji, R.: Automated analysis of data-dependent programs with dynamic memory. Technical Report 2009-018, Dept. of Information Technology, Uppsala University, Sweden (2009), `http://user.it.uu.se/~jonmo/datadependent.pdf`
3. Abdulla, P.A., Bouajjani, A., Cederberg, J., Haziza, F., Rezine, A.: Monotonic abstraction for programs with dynamic memory heaps. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 341–354. Springer, Heidelberg (2008)
4. Abdulla, P.A., Henda, N.B., Delzanno, G., Rezine, A.: Regular model checking without transducers (on efficient verification of parameterized systems). In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 721–736. Springer, Heidelberg (2007)
5. Bouajjani, A., Bozga, M., Habermehl, P., Iosif, R., Moro, P., Vojnar, T.: Programs with lists are counter automata. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 517–531. Springer, Heidelberg (2006)
6. Bouajjani, A., Habermehl, P., Moro, P., Vojnar, T.: Verifying programs with dynamic 1-selector-linked structures in regular model checking. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 13–29. Springer, Heidelberg (2005)
7. Bouajjani, A., Habermehl, P., Rogalewicz, A., Vojnar, T.: Abstract tree regular model checking of complex dynamic data structures. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, pp. 52–70. Springer, Heidelberg (2006)
8. Bouajjani, A., Habermehl, P., Vojnar, T.: Abstract regular model checking. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 372–386. Springer, Heidelberg (2004)
9. Ganty, P., Raskin, J., Begin, L.V.: A complete abstract interpretation framework for coverability properties of wsts. In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI 2006. LNCS, vol. 3855, pp. 49–64. Springer, Heidelberg (2006)
10. Guo, B., Vachharajani, N., August, D.I.: Shape analysis with inductive recursion synthesis. In: Proc. PLDI 2007, vol. 42 (2007)
11. Henriksen, J., Jensen, J., Jørgensen, M., Klarlund, N., Paige, B., Rauhe, T., Sandholm, A.: Mona: Monadic second-order logic in practice. In: Brinksma, E., Steffen, B., Cleaveland, W.R., Larsen, K.G., Margaria, T. (eds.) TACAS 1995. LNCS, vol. 1019. Springer, Heidelberg (1995)

12. Joshi, S., König, B.: Applying the graph minor theorem to the verification of graph transformation systems. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 214–226. Springer, Heidelberg (2008)
13. Laver, R.: Well-quasi-orderings and sets of finite sequences. In: Mathematical Proceedings of the Cambridge Philosophical Society, vol. 79, pp. 1–10 (1976)
14. Lev-Ami, T., Reps, T.W., Sagiv, S., Wilhelm, R.: Putting static analysis to work for verification: A case study. In: Proc. ISSTA 2000 (2000)
15. Møller, A., Schwartzbach, M.I.: The pointer assertion logic engine. In: Proc. PLDI 2001, vol. 26, pp. 221–231 (2001)
16. Nguyen, H.H., David, C., Qin, S., Chin, W.-N.: Automated verification of shape and size properties via separation logic. In: Cook, B., Podelski, A. (eds.) VMCAI 2007. LNCS, vol. 4349, pp. 251–266. Springer, Heidelberg (2007)
17. Revesz, P.: Introduction to Constraint Databases. Springer, Heidelberg (2002)
18. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: Proc. LICS 2002 (2002)
19. Sagiv, S., Reps, T., Wilhelm, R.: Parametric shape analysis via 3-valued logic. ACM Trans. on Programming Languages and Systems 24(3), 217–298 (2002)
20. Valiente, G.: Constrained tree inclusion. J. Discrete Algorithms 3(2-4), 431–447 (2005)
21. Yang, H., Lee, O., Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P.W.: Scalable shape analysis for systems code. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 385–398. Springer, Heidelberg (2008)