Automatic Verification of Dynamic Data-Dependent Programs

Ran Ji
Abstract

**Automatic Verification of Dynamic Data-Dependent Programs**

*Ran Ji*

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 equality or inequality. 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 will report on the result of running a prototype based on the method on a number of programs.
Acknowledgements

First of all, I would like to give the greatest thanks to my reviewer Prof. Parosh Aziz Abdulla. Thank you for giving me the opportunity to do my master thesis in the APV group, for your excellent lectures Programming Theory and Data Structures, which led me to the magic world of verification, and for your patience and encouragement.

Secondly, my warm thanks goes to Jonathan Cederberg, who is not only my thesis supervisor but also a sincere friend. I will always remember the inspired discussions and enjoyable teamworks. The work will never be like now without you.

Thirdly, it is a pleasure for me to share the room with Muhsin Haji Atto, who has been with me all along the thesis work. I am also grateful to the other members of the APV group: Frédéric Haziza, Noomene Ben Henda, Lisa Kaati, Ahmed Rezine and Mayank Saksena.

I have had many interesting courses in IT department, thanks to all the teachers. In particular, I would like to mention Prof. Bengt Jonsson and Prof. Wang Yi. They offered me more than the knowledge given in class.

My friends in Uppsala also help me a lot in both life and studying, thanks.

Last but not least, I owe my thanks to my parents. Their endless love and instant supporting give me the courage to overcome every difficulty and make me trust myself all the time. I wish to make you proud.
## Contents

1 **Introduction** 1  
  1.1 Introduction ........................................... 1  
  1.2 Related Work ........................................... 3  
  1.3 Outline .................................................. 4  

2 **Heaps** 6  
  2.1 Heaps .................................................... 6  
  2.2 Programming Language ................................. 8  
  2.3 Transition System ....................................... 9  

3 **Signatures** 12  
  3.1 Signatures ................................................ 12  
  3.2 Operations on Signatures ............................... 14  
    3.2.1 Operations on Cells ................................ 14  
    3.2.2 Operations on Variables .......................... 15  
    3.2.3 Operations on Edges ............................... 17  
    3.2.4 Operation on Ordering Relations .................. 18  
  3.3 Ordering ............................................... 18  

4 **Bad Configurations** 20  
  4.1 Sorted Linear List ...................................... 20  
    4.1.1 Well-Formedness ................................. 20  
    4.1.2 Garbage ........................................... 21  
    4.1.3 Sortedness ....................................... 21  
    4.1.4 Putting Everything Together ..................... 21  
  4.2 Sorted Cyclic List .................................... 22  
    4.2.1 Well-Formedness ................................. 22  
    4.2.2 Garbage ........................................... 22  
    4.2.3 Sortedness ....................................... 22
4.2.4 Putting Everything Together .......................... 23
4.3 Sorted Partition ........................................... 23
  4.3.1 Well-Formedness ...................................... 23
  4.3.2 Garbage ................................................ 24
  4.3.3 Sharing ............................................... 24
  4.3.4 Sortedness ............................................ 24
  4.3.5 Putting Everything Together ........................ 24
4.4 More Bad on Inverse Sortedness .......................... 25
  4.4.1 Inversely Sorted Linear List ........................ 25
  4.4.2 Inversely Sorted Cyclic List ......................... 25

5 Reachability Analysis ........................................ 26
  5.1 Over-Approximation ...................................... 26
  5.2 Computing Predecessors ................................ 26
  5.3 Initial Heaps .......................................... 32
  5.4 Checking Safety Properties .............................. 32
    5.4.1 Reachability Algorithm ........................... 32
    5.4.2 Checking Entailment ............................... 33
    5.4.3 Searching Strategies ............................. 35

6 Experimental Results ........................................ 36
  6.1 Example Programs ...................................... 36
    6.1.1 Insert ............................................ 36
    6.1.2 Insert(bug) ........................................ 36
    6.1.3 Merge .............................................. 36
    6.1.4 Reverse ........................................... 38
    6.1.5 ReverseCyclic ..................................... 38
    6.1.6 Partition .......................................... 38
    6.1.7 BubbleSort ........................................ 40
    6.1.8 BubbleSort(bug) .................................... 41
    6.1.9 BubbleSortCyclic .................................. 41
    6.1.10 InsertionSort ..................................... 42
  6.2 Experimental Results ................................... 42

7 Conclusions and Future Work ............................... 44
List of Figures

2.1 A typical graph representing the heap. ................................. 7
2.2 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 $\ast$ unless one of the variables $x, y, z$ is attached to them. For this reason the cell $\#$ is missing in all the heaps, and $\ast$ is present only in $h_3, h_4, h_5$. ................................. 9
3.1 Examples of signatures. ................................................. 13
5.1 The procedure of generating a linear list ............................. 32
6.1 The insert program including initialization phase .................. 37
6.2 The merge program without initialization phase ................... 37
6.3 The reverse program without initialization phase .................. 38
6.4 The reverseCyclic program including initialization phase ......... 39
6.5 The partition program including initialization phase .............. 39
6.6 The bubbleSort program without initialization phase ............. 40
6.7 The bubbleSortCyclic program including initialization phase .... 41
6.8 The insertionSort program without initialization phase .......... 42
Chapter 1

Introduction

1.1 Introduction

We consider automatic verification of data-dependent programs that manipulate dynamic linked lists. The contents of the linked lists, here referred 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 2.1 for a typical example). 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.

In this report, 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 provide 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. We provide a simple symbolic representation, which we call signatures, for characterizing (infinite) sets of heaps. Signatures can also be represented by graphs. The 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. In this manner, 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 underly-
ing symbolic representation, signatures in our case, is closed under the operation of
computing predecessors. More precisely, for a signature \( g \), let us define \( \text{Pre}(g) \) 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 \( \text{Pre}(g) \) does not exist in general under the oper-
tional semantics of the class of programs we consider in this report. 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 \( \text{Pre}(g) \) 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.

1.2 Related Work

Several works consider the verification of singly linked lists with data. The paper [11] 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 [16]. 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 report 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. [3, 8, 1]).

Another approach to verification of linked lists with data is proposed in [5, 6] based on abstract regular model checking (ARMC) [7]. 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 [4] 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 [15] for performing shape analysis of programs that manipulate pointer data structures (see e.g. [9, 18]). The paper [13] 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) [12] 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 [10], which uses translations to finite-state automata, is employed to verify the provided assertions.

In our previous work [2], we used backward reachability analysis for verifying heap manipulating programs. However, the programs are restricted to be data-independent. The extension to the case of data-dependent programs is not trivial and requires an intricate treatment of the ordering on signatures. In particular, the interaction between the structural and the data orderings is central to our method. This is used for instance to specify basic properties like sortedness (whose forbidden pattern contains edges from both orderings). Hence, none of the programs we consider in this report can be analyzed in the framework of [2].

1.3 Outline

In the next chapter, we describe our model of heaps, and introduce the programming language together with the induced transition system. In Chapter 3, we introduce the notion of signatures and the associated ordering. Chapter 4 describes how to specify sets of bad heaps using signatures. In Chapter 5 we give an overview of the backward reachability scheme, and show how to compute the predecessor relation on signatures.
The experimental results are presented in Chapter 6. Finally, in Chapter 7 we give some conclusions and directions for future research.
Chapter 2

Heaps

In this chapter, 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.

A binary relation $R$ on a set $A$ is said to be a partial order if it is irreflexive and transitive. We say that $R$ is an equivalence relation if it is reflexive, symmetric, and transitive. We use $f(a) \overset{\triangle}{=} f(b)$ to denote that $f(a) \neq \perp$, $f(b) \neq \perp$, and $f(a) = f(b)$, i.e., $f(a)$ and $f(b)$ are defined and equal. Analogously, we write $f(a) \preceq f(b)$ to denote that $f(a) \neq \perp$, $f(b) \neq \perp$, and $f(a) < f(b)$.

2.1 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 linked lists and circular lists, possibly sharing their parts (see Figure 2.1). 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 2.1. 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 $M^* := M \cup \{#, *\}$.

- $Succ : M \rightarrow M^*$. 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.

- $\lambda : X \rightarrow M^*$ defines the cells pointed to by the variables. The function $\lambda$ 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 2.1, 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 report can be extended in a straightforward manner to the general
case. Furthermore, we can use ordered domains other than the natural numbers such
as the integers, rationals, or reals.
2.2 Programming Language

We define a simple programming language. To this end, we assume, together with the earlier mentioned set $X$ of variables, the constant $\text{null}$ where $\text{null} \not\in X$. We define $X^\# := X \cup \{\text{null}\}$. 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 $x.\text{next} := y$ 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.

- $x := y.\text{next}$ where $x, y \in X$. The variable $x$ will now point to the successor of the $y$-cell.

- $\text{new}(x)$, $\text{delete}(x)$, or $\text{read}(x)$, 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.

- $x.\text{num} = y.\text{num}$, $x.\text{num} < y.\text{num}$, $x.\text{num} := y.\text{num}$, $x.\text{num} := y.\text{num}$, or $x.\text{num} := y.\text{num}$, 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 nondeterministically a value to the $x$-cell which is larger (smaller) than that of the $y$-cell.

Figure 2.2 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 Chapter 6.
Figure 2.2: 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: \( z.num \colon> x.num \), \( x:=y.next \), \( delete(x) \), \( new(x) \), and \( z.next:=y \). To simplify the figures, we omit the special nodes \# and \( \ast \) unless one of the variables \( x,y,z \) is attached to them. For this reason the cell \# is missing in all the heaps, and \( \ast \) is present only in \( h_3,h_4,h_5 \).

2.3 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 \( h = (M,\text{Succ},\lambda,\text{Val}) \). For \( m_1,m_2 \in M \), we use \( (h.\text{Succ})[m_1 \leftarrow m_2] \) 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, \( h' = (M,\text{Succ}',\text{Val},\lambda) \) where \( \text{Succ}' = \text{Succ}[m_1 \leftarrow m_2] \).

Analogously, \( (h.\lambda)[x \leftarrow m] \) is the heap we obtain by making \( x \) point to the cell \( m \); and \( (h.\text{Val})[m \leftarrow i] \) is the heap we obtain by assigning the value \( i \) to the cell \( m \). For instance, in Figure 2.2, let \( h_i \) be of the form \( (M_i,\text{Succ}_i,\text{Val}_i,\lambda_i) \) for \( i \in \{0,1,2,3,4,5\} \). Then, we have \( h_1 = (h_0.\text{Val})[\lambda_0(z) \leftarrow 9] \) since we make the value of the \( z \)-cell equal to 9. Also, \( h_2 = (h_1.\lambda_1)[x \leftarrow \text{Succ}_1(\lambda_1(y))] \) since we make \( x \) point to the successor of the \( y \)-cell. Furthermore, \( h_5 = (h_4.\text{Succ}_4)[\lambda_4(z) \leftarrow \lambda_4(y)] \) 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 \( h' := (M',\text{Succ}',\lambda',\text{Val}') \) where
In Figure 2.2, 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 heaps $h$ and $h'$, $h \xrightarrow{op} h'$ holds if one of the following conditions is satisfied:

- $op$ is of the form $x = y$, $\lambda(x) \neq \star$, $\lambda(y) \neq \star$, $\lambda(x) = \lambda(y)$, and $h' = h$. In other words, the transition is enabled if the pointers are not dangling, and they point to the same cell.

- $op$ is of the form $x \neq y$, $\lambda(x) \neq \star$, $\lambda(y) \neq \star$, $\lambda(x) \neq \lambda(y)$, and $h' = h$.

- $op$ is of the form $x := y$, $\lambda(y) \neq \star$, and $h' = (h.\lambda)[x \leftarrow \lambda(y)]$.

- $op$ is of the form $x := y.next$, $\lambda(y) \in M$, $\text{Succ}(\lambda(y)) \neq \star$, and $h' = (h.\lambda)[x \leftarrow \text{Succ}(\lambda(y))]$.

- $op$ is of the form $x.next := y$, $\lambda(x) \in M$, $\lambda(y) \neq \star$, and $h' = (h.\text{Succ})[\lambda(x) \leftarrow \lambda(y)]$.

- $op$ is of the form new$(x)$, $M' = M \cup \{m\}$ for some $m \not\in M$, $\lambda' = \lambda[x \leftarrow m]$, $\text{Succ'} = \text{Succ}[m \leftarrow \star]$, $\text{Val'}(m') = \text{Val}(m')$ if $m' \neq m$, and $\text{Val'}(m) = \bot$. 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 $\star$.

- $op$ is of the form delete$(x)$, $\lambda(x) \in M$, and $h' = h \ominus \lambda(x)$. The operation deletes the $x$-cell.

- $op$ is of the form read$(x)$, $\lambda(x) \in M$, and $h' = (h.\text{Val})[\lambda(x) \leftarrow i]$, where $i$ is the value assigned to $x$-cell.

$M' = M - \{m\}$.

$\text{Succ}'(m') = \text{Succ}(m')$ if $\text{Succ}(m') \neq m$, and $\text{Succ}'(m') = \star$ otherwise. In other words, the successor of cells pointing to $m$ will become dangling in $h'$.

$\lambda'(x) = \star$ 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 $h'$.

$\text{Val}'(m') = \text{Val}(m')$ if $m' \in M'$. That is, the function $\text{Val}'$ is the restriction of $\text{Val}$ to $M'$: it assigns the same values as $\text{Val}$ to all the cells which remain in $M'$ (since $m \not\in M'$, it is not meaningful to speak about $\text{Val}(m)$).
• \( \text{op} \) is of the form \( x.\, \text{num} = y.\, \text{num} \), \( \lambda(x) \in M \), \( \lambda(y) \in M \), \( \text{Val}(\lambda(x)) = \text{Val}(\lambda(y)) \), and \( h' = h \). The transition is enabled if the pointers are not dangling and the values of their cells are defined and equal.

• \( \text{op} \) is of the form \( x.\, \text{num} < y.\, \text{num} \), \( \lambda(x) \in M \), \( \lambda(y) \in M \), \( \text{Val}(\lambda(x)) < \text{Val}(\lambda(y)) \), and \( h' = h \).

• \( \text{op} \) is of the form \( x.\, \text{num} := y.\, \text{num} \), \( \lambda(x) \in M \), \( \lambda(y) \in M \), \( \text{Val}(\lambda(y)) \neq \bot \), and \( h' = (h.\, \text{Val}) [\lambda(x) \leftarrow \text{Val}(\lambda(y))]. \)

• \( \text{op} \) is of the form \( x.\, \text{num} :> y.\, \text{num} \), \( \lambda(x) \in M \), \( \lambda(y) \in M \), \( \text{Val}(\lambda(y)) \neq \bot \), and \( h' = (h.\, \text{Val}) [\lambda(x) \leftarrow i] \), where \( i > \text{Val}(\lambda(y)) \). The case for \( x.\, \text{num} :< y.\, \text{num} \) is defined analogously.

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

**Remark** One could also allow deterministic assignment operations of the form \( x.\, \text{num} := y.\, \text{num} + k \) or \( x.\, \text{num} := y.\, \text{num} - k \) for some constant \( k \). However, according the approximate transition relation which we define in Chapter 5, these operations will have identical interpretations as the non-deterministic operations given above.
Chapter 3

Signatures

In this chapter, 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.

3.1 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 $(M, \text{Succ}, \lambda, \text{Ord})$, where $M$, Succ, λ are defined in the same way as in heaps (Chapter 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 $\text{Succ}(m) = \bot$ for some cell $m \in M$, then this means that $g$ puts no constraints on the successor of $m$, i.e., the successor of $m$ can be any arbitrary cell. Analogously, if $\lambda(x) = \bot$, then $x$ may point to any of the cells. The relation Ord constrains the ordering on the cell values. If $\text{Ord}(m_1, m_2) = \prec$ then the value of $m_1$ is strictly smaller than that of $m_2$; and if $\text{Ord}(m_1, m_2) = \equiv$ 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 $\text{Ord}(m, m') = \bot$ and $\text{Ord}(m', m) = \bot$ for all other cells $m'$. Abusing notation, we write $m_1 \prec m_2$ (resp. $m_1 \equiv m_2$) if $\text{Ord}(m_1, m_2) = \prec$ (resp. $\text{Ord}(m_1, m_2) = \equiv$). A
signature $g$ is said to be saturated if (i) $\equiv$ is an equivalence relation; (ii) $\prec$ is a partial order; (iii) $m_1 \equiv m_2$, $m_2 \prec m_3$ implies $m_1 \prec m_3$; and (iv) $m_1 \prec m_2$, $m_2 \equiv m_3$ implies $m_1 \prec m_3$. For a signature $g = (M, Succ, \lambda, Ord)$, we define its saturation, denoted $sat(g)$, to be the signature $(M, Succ, \lambda, Ord')$ where $Ord' \supseteq Ord$ is the smallest set sufficient for making $g$ saturated.

We represent signatures graphically in a manner similar to that of heaps. Figure 3.1 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., $m \equiv m$). 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 $Val(m_1) = Val(m_2)$.
\(\text{Val}(m_2)\). 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 2.2 and Figure 3.1, we have \(g_0 = \text{sig}(h_0)\).

### 3.2 Operations on Signatures

We use \(M^\#\) to denote \(M \cup \{\#\}\). Assume a saturated signature \(g = (M, \text{Succ}, \lambda, \text{Ord})\). 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, \(\text{Succ}^{-1}(m) = \emptyset\), and either \(\text{Succ}(m) = \bot\) or \(\text{Succ}(m) = \ast\). 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 \(\ast\)). We say that \(m\) is isolated if it is semi-isolated and in addition \(\text{Succ}(m) = \bot\). 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, \(|\text{Succ}^{-1}(m)| = 1\), and \(\text{Succ}(m) \neq \bot\). In other words, \(m\) has exactly one predecessor, one successor and no label. In Figure 3.1, the topmost cell of \(g_3\) is isolated, and the successor of the \(x\)-cell in \(g_4\) is simple. In Figure 2.1, the cell to the left of the \(w\)-cell is semi-isolated in the signature of the heap.

The operations \((g.\text{Succ})[m_1 \leftarrow m_2]\) and \((g.\lambda)[x \leftarrow m]\) are defined in identical fashion to the case of heaps. Furthermore, for cells \(m_1, m_2\) and \(\square \in \{\prec, \equiv, \bot\}\), we define \((g.\text{Ord})[(m_1, m_2) \leftarrow \square]\) to be the signature \(g'\) we obtain from \(g\) by making the ordering relation between \(m_1\) and \(m_2\) equal to \(\square\).

#### 3.2.1 Operations on Cells

For \(m \not\in M\), we define \(g \oplus m\) to be the signature \(g' = (M', \text{Succ}', \lambda', \text{Ord}')\) such that \(M' = M \cup \{m\}\), \(\text{Succ}' = \text{Succ}\), \(\lambda' = \lambda\), and \(\text{Ord}' = \text{Ord}\). i.e. we add a new cell to \(g\). Observe that the added cell is then isolated.

We define \(g \oplus \lambda(x)\) to be the signature \(g' = (M', \text{Succ}', \lambda', \text{Ord}')\) such that \(M' = M \cup \{m\}\), \(\text{Succ}' = \text{Succ}\), \(\lambda' = \lambda[x \leftarrow m]\), and \(\text{Ord}' = \text{Ord}\). i.e. we add a new cell to \(g\) which is pointed by \(x\).

For \(m \in M\), we define \(g \ominus m\) to be the signature \(g' = (M', \text{Succ}', \lambda', \text{Ord}')\) such that

- \(M' = M - \{m\}\).
- \(\text{Succ}'(m') = \text{Succ}(m')\) if \(\text{Succ}(m') \neq m\), and \(\text{Succ}'(m') = \ast\) otherwise.
- \(\lambda'(x) = \ast\) if \(\lambda(x) = m\), and \(\lambda'(x) = \lambda(x)\) otherwise.
• \( \text{Ord}'(m_1, m_2) = \text{Ord}(m_1, m_2) \) if \( m_1, m_2 \in M' \).

### 3.2.2 Operations on Variables

We use \( g \oplus x \) to denote the set of signatures we get from \( g \) by letting \( x \) point anywhere inside \( g \), except on \(*\). Formally, we define \( g \oplus x \) to be the smallest set containing each signature \( g' \) such that one of the following conditions is satisfied:

1. There is a cell \( m \in M^# \), and \( g' = (g.\lambda)[x \leftarrow m] \).

2. There is a cell \( m \notin M \), and a signature \( g_1 \) such that \( g_1 = g \oplus m, \ g' = (g_1.\lambda)[x \leftarrow m] \).

3. There are \( m_1 \in M, m_2 \notin M \), and signatures \( g_1, g_2, g_3 \) such that \( \text{Succ}(m_1) \neq \perp \), \( g_1 = g \oplus m_2, g_2 = (g_1.\text{Succ})[m_2 \leftarrow \text{Succ}(m_1)], g_3 = (g_2.\text{Succ})[m_1 \leftarrow m_2] \), and \( g' = (g_3.\lambda)[x \leftarrow m_2] \).

For variables \( x \) and \( y \), \( \lambda(x) \in M^# \), we use \( g \oplus_{=x} y \) to denote \( (g.\lambda)[y \leftarrow \lambda(x)] \), i.e. we make \( y \) point to the same cell as \( x \). Furthermore, we define \( g \oplus_{\neq x} y \) to be the smallest set containing each signature \( g' \) such that \( g' \in (g \oplus y) \), and \( \lambda'(y) \neq \lambda(x) \), i.e. we make \( y \) point anywhere inside \( g \) except on \( x \)-cell and \(*\). As a special case, we use \( g \oplus_{\neq \#} y \) to denote the smallest set containing each signature \( g' \) such that \( g' \in (g \oplus y) \), and \( \lambda'(y) \neq \# \), i.e. we make \( y \) point anywhere inside \( g \) except on \( \# \) and \(*\).

For variables \( x \) and \( y \), \( \lambda(x) \in M, \text{Succ}(\lambda(x)) \in M^# \), we use \( g \oplus_{x\rightarrow} y \) to denote the set of signatures we get from \( g \) by letting \( y \) point to the successor of \( x \)-cell. Formally, we define \( g \oplus_{x\rightarrow} y \) to be the smallest set containing each signature \( g' \) such that one of the following conditions is satisfied:

1. \( g' = (g.\lambda)[y \leftarrow \text{Succ}(\lambda(x))] \).

2. There is a cell \( m \notin M \), and signatures \( g_1, g_2, g_3 \), such that \( g_1 = g \oplus m, g_2 = (g_1.\text{Succ})[m \leftarrow \text{Succ}(\lambda(x))], g_3 = (g_2.\text{Succ})[\lambda(x) \leftarrow m] \), and \( g' = (g_3.\lambda)[y \leftarrow m] \).

For variables \( x \) and \( y \), \( \text{Succ}(\lambda(x)) = * \), we use \( g \oplus_{x\rightarrow *} y \) to denote the signature we get from \( g \) by letting \( y \) point to the new added cell in between \( x \)-cell and \(*\). Formally, we define \( g \oplus_{x\rightarrow *} y \) to be the signature \( g' \) such that there is a cell \( m \notin M \), and signatures \( g_1, g_2, g_3 \), such that \( g_1 = g \oplus m, g_2 = (g_1.\text{Succ})[m \leftarrow *], g_3 = (g_2.\text{Succ})[\lambda(x) \leftarrow m] \), and \( g' = (g_3.\lambda)[y \leftarrow m] \).

For variables \( x \) and \( y \), \( \lambda(x) \in M^# \), we use \( g \oplus_{x\leftarrow} y \) to denote the set of signatures we get from \( g \) by letting \( y \) point to any cell except \(*\), where it has no successor or its
successor is \(x\)-cell. Formally, we define \(g \oplus_{x} y\) to be the smallest set containing each signature \(g'\) such that one of the following conditions is satisfied:

1. There is a cell \(m \in M\) such that \(\text{Succ}(m) = \bot\) or \(\text{Succ}(m) = \lambda(x)\), and \(g' = (g, \lambda)[y \leftarrow m]\).

2. There is a cell \(m \notin M\), and a signature \(g_{1}\) such that \(g_{1} = g \oplus m\), \(g' = (g_{1}, \lambda)[y \leftarrow m]\).

3. There are \(m_{1} \in M\), \(m_{2} \notin M\), and signatures \(g_{1}, g_{2}, g_{3}\), such that \(\text{Succ}(m_{1}) = \lambda(x)\), \(g_{1} = g \oplus m_{2}\), \(g_{2} = (g_{1}.\text{Succ})[m_{2} \leftarrow \lambda(x)]\), \(g_{3} = (g_{2}.\text{Succ})[m_{1} \leftarrow m_{2}]\), and \(g' = (g_{3}, \lambda)[y \leftarrow m_{2}]\).

For variables \(x\) and \(y\), \(\lambda(x) \in M\), we use \(g \oplus_{=x} y\) to denote the set of signatures we get from \(g\) by letting \(y\) point to any cell such that possibly \(\lambda(y) \equiv \lambda(x)\). Formally, we define \(g \oplus_{=x} y\) to be the smallest set containing each signature \(g'\) such that one of the following conditions is satisfied:

1. There is a cell \(m \in M\) such that \(\text{Ord}(m, \lambda(x)) = \equiv\) or \(\text{Ord}(m, \lambda(x)) = \bot\), and \(g' = (g, \lambda)[x \leftarrow m]\).

2. There is a cell \(m \notin M\), and a signature \(g_{1}\) such that \(g_{1} = g \oplus m\), \(g' = (g_{1}, \lambda)[x \leftarrow m]\).

3. There are \(m_{1} \in M\), \(m_{2} \notin M\), and signatures \(g_{1}, g_{2}, g_{3}\) such that \(\text{Succ}(m_{1}) \neq \bot\), \(g_{1} = g \oplus m_{2}\), \(g_{2} = (g_{1}.\text{Succ})[m_{2} \leftarrow \text{Succ}(m_{1})]\), \(g_{3} = (g_{2}.\text{Succ})[m_{1} \leftarrow m_{2}]\), and \(g' = (g_{3}, \lambda)[x \leftarrow m_{2}]\).

For variables \(x\) and \(y\), \(\lambda(x) \in M\), we use \(g \oplus_{<x} y\) to denote the set of signatures we get from \(g\) by letting \(y\) point to any cell such that possibly \(\lambda(y) < \lambda(x)\). Formally, we define \(g \oplus_{<x} y\) to be the smallest set containing each signature \(g'\) such that one of the following conditions is satisfied:

1. There is a cell \(m \in M\) such that \(\text{Ord}(m, \lambda(x)) = <\), or \(\text{Ord}(m, \lambda(x)) = \bot\), and \(g' = (g, \lambda)[x \leftarrow m]\).

2. There is a cell \(m \notin M\), and a signature \(g_{1}\) such that \(g_{1} = g \oplus m\), \(g' = (g_{1}, \lambda)[x \leftarrow m]\).

3. There are \(m_{1} \in M\), \(m_{2} \notin M\), and signatures \(g_{1}, g_{2}, g_{3}\) such that \(\text{Succ}(m_{1}) \neq \bot\), \(g_{1} = g \oplus m_{2}\), \(g_{2} = (g_{1}.\text{Succ})[m_{2} \leftarrow \text{Succ}(m_{1})]\), \(g_{3} = (g_{2}.\text{Succ})[m_{1} \leftarrow m_{2}]\), and \(g' = (g_{3}, \lambda)[x \leftarrow m_{2}]\).
For variables $x$ and $y$, $\lambda(x) \in M$, we use $g \oplus_{x \leftarrow} y$ to denote the set of signatures we get from $g$ by letting $y$ point to any cell such that possibly $\lambda(x) \prec \lambda(y)$. Formally, we define $g \oplus_{x \leftarrow} y$ to be the smallest set containing each signature $g'$ such that one of the following conditions is satisfied:

1. There is a cell $m \in M$ such that $\text{Ord}(\lambda(x), m) = \prec$, or $\text{Ord}(\lambda(x), m) = \bot$, and $g' = (g.\lambda)[x \leftarrow m]$.

2. There is a $m \not\in M$, and a signature $g_1$ such that $g_1 = g \oplus m$, $g' = (g_1.\lambda)[x \leftarrow m]$.

3. There are $m_1 \in M$, $m_2 \not\in M$, and signatures $g_1, g_2, g_3$ such that $\text{Succ}(m_1) \neq \bot$, $g_1 = g \oplus m_2$, $g_2 = (g_1.\text{Succ})[m_2 \leftarrow \text{Succ}(m_1)]$, $g_3 = (g_2.\text{Succ})[m_1 \leftarrow m_2]$, and $g' = (g_3.\lambda)[x \leftarrow m_2]$.

For a variable $x$, we use $g \ominus x$ to denote $g' = (g.\lambda)[x \leftarrow \bot]$, i.e. we remove $x$ from $g$.

### 3.2.3 Operations on Edges

For variables $x$ and $y$, $\lambda(x) \in M$, $\lambda(y) \in M^\#$, we use $g \ominus_3 (x \rightarrow y)$ to denote $(g.\text{Succ})[\lambda(x) \leftarrow \lambda(y)]$, i.e. we remove the edge between $x$-cell and its successor (if any), and add an edge from $x$-cell to $y$-cell.

For a variable $x$, $\lambda(x) \in M$, we use $g \ominus_1 (x \rightarrow)$ to denote the set of signatures we get from $g$ by making an edge from $x$-cell to anywhere inside $g$, except $\ast$. Formally, we define $g \ominus_1 (x \rightarrow)$ to be the smallest set containing each signature $g'$ such that one of the following conditions is satisfied:

1. There is a $m \in M^\#$, and $g' = (g.\text{Succ})[\lambda(x) \leftarrow m]$.

2. There is a $m \not\in M$ such that $g' = ((g \oplus m).\text{Succ})[\lambda(x) \leftarrow m]$.

3. There are $m_1 \in M$, $m_2 \not\in M$, and signatures $g_1, g_2, g_3$, such that $\text{Succ}(m_1) \neq \bot$, $g_1 = g \oplus m_2$, $g_2 = (g_1.\text{Succ})[m_2 \leftarrow \text{Succ}(m_1)]$, $g_3 = (g_2.\text{Succ})[m_1 \leftarrow m_2]$, and $g' = (g_3.\text{Succ})[\lambda_3(x) \leftarrow m_2]$.

We use $M^L$ to denote the set of cells such that for all $m \in M^L$, $\text{Succ}(m) = \ast$. For a variable $x$, $\lambda(x) \in M$, we define $g \ominus_2 (M^L \rightarrow x)$ to be the smallest set containing each signature $g'$ such that $g' = (g.\text{Succ})[m' \leftarrow \lambda(x)]$, where $m' \in M^L$, $M^L \in P(M^L)$, and $P(M^L)$ is the power set of $M^L$, i.e. we get each $g'$ by picking some cells in $M^L$, and make their successors all point to $x$.

For a variable $x$, $\lambda(x) \in M$, we use $g \ominus_4 (x \rightarrow)$ to denote $(g.\text{Succ})[\lambda(x) \leftarrow \bot]$, i.e. we remove the edge from $x$-cell and its successor (if any).
3.2.4 Operation on Ordering Relations

For variables $x$ and $y$, $\lambda(x) \in M$, $\lambda(y) \in M$, we use $g \oplus (x \equiv y)$ to denote $(g.Ord)[(\lambda(x), \lambda(y)) \leftarrow \equiv], (g.Ord)[(m_1, \lambda(y)) \leftarrow \equiv], \text{ for } m_1 \in M, \text{ Ord}(m_1, \lambda(x)) = \equiv,$ and $(g.Ord)[(m_2, \lambda(x)) \leftarrow \equiv], \text{ for } m_2 \in M, \text{ Ord}(m_2, \lambda(y)) = \equiv, \text{ i.e. we make the ordering relation between } x\text{-cell and } y\text{-cell equal to } \equiv, \text{ and make } g' \text{ saturated.}

For variables $x$ and $y$, $\lambda(x) \in M$, $\lambda(y) \in M$, we use $g \oplus (x \prec y)$ to denote $(g.Ord)[(\lambda(x), \lambda(y)) \leftarrow \prec], (g.Ord)[(m_1, \lambda(y)) \leftarrow \prec], \text{ for } m_1 \in M, \text{ Ord}(m_1, \lambda(x)) = \equiv \text{ or } \text{Ord}(m_1, \lambda(x)) = \prec, \text{ and } (g.Ord)[(\lambda(x), m_2) \leftarrow \prec], \text{ for } m_2 \in M, \text{ Ord}(\lambda(y), m_2) = \equiv \text{ or } \text{Ord}(\lambda(y), m_2) = \prec, \text{ i.e. we make the ordering relation between } x\text{-cell and } y\text{-cell equal to } \prec, \text{ and make } g' \text{ saturated.}

For a variable $x$, $\lambda(x) \in M$, we use $g \ominus_{Ord} \lambda(x)$ to denote that for all $m \in M - \{\lambda(x)\}, (g.Ord)[(\lambda(x), m) \leftarrow \bot], (g.Ord)[(m, \lambda(x)) \leftarrow \bot], \text{ i.e. we remove the ordering relation between } x\text{-cell and other cells.}

3.3 Ordering

We define an ordering $\subseteq$ 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).

For signatures $g = (M, \text{Succ}, \lambda, \text{Ord})$ and $g' = (M', \text{Succ}', \lambda', \text{Ord}')$, we write that $g \triangleleft g'$ 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**: $g = (g'.\text{Succ})[m \leftarrow \bot]$ 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$'(m_2) = m_3$, $g_1 = (g'.\text{Succ})[m_1 \leftarrow m_3]$ and $g = g_1 \ominus m_2$, or
- **Order Deletion**: $g = (g'.\text{Ord})[(m_1, m_2) \leftarrow \bot]$ for some cells $m_1, m_2 \in M'$. 
We write \( g \sqsubseteq g' \) to denote that there are \( g_0 < g_1 < g_2 < \cdots < g_n \) 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 3.1 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 \( g_5 < g_4 < g_3 < g_2 < g_1 < g_0 \) and hence \( g_5 \sqsubseteq g_0 \).

We define an ordering \( \sqsubseteq \) on heaps such that \( h \sqsubseteq h' \) iff \( \text{sig}(h) \sqsubseteq \text{sig}(h') \). For a heap \( h \) and a signature \( g \), we say that \( h \) satisfies \( g \), denoted \( h \models g \), if \( g \sqsubseteq \text{sig}(h) \).

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 2.2 and the signatures of Figure 3.1, we have \( h_1 \models g_0 \), \( h_2 \not\models g_0 \), \( h_0 \sqsubseteq h_1 \), \( h_0 \not\sqsubseteq h_2 \), \( g_1 \sqsubseteq g_0 \), 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 \( k \geq 1 \).

It is also worth mentioning that the classical graph minor relation is too weak for our class of programs. The reason is that the contraction operation is insensitive to the directions of the edges; and furthermore the ordering allows merging vertices which carry different labels (different variables). This means that for almost all the programs considered in this report, we would get spurious examples falsifying the safety property.
Chapter 4

Bad Configurations

In this chapter, we show how to use signatures in order to specify sets of bad heaps for programs which produce sorted single linked lists. In particular, the bad configurations of sorted linear list, sorted cyclic list, sorted partition, inversely sorted linear list and inversely sorted cyclic list are introduced.

A signature is interpreted as a forbidden pattern which should not occur inside the heap. Fix a heap \( h = (M, \text{Succ}, \lambda, \text{Val}) \). A loop in \( h \) is a set \( \{m_0, \ldots, m_n\} \) of cells such that \( \text{Succ}(m_i) = m_{i+1} \) for all \( 0 \leq i < n \), and \( \text{Succ}(m_n) = m_0 \). 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 \( m_{i+1} = \text{Succ}(m_i) \) for all \( 0 \leq i < n \). 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 \( m \).

4.1 Sorted Linear List

Considering programs such as insert, insertion sort, bubble sort and the merging procedure of two sorted linear lists, we would like such programs to produce a heap which is a linear list. Furthermore, the heap should not contain any garbage, and the output list should be sorted. 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. Then, we will collect all these signatures into a single set which characterizes the set of bad configurations.

4.1.1 Well-Formedness
We say that \( h \) is well-formed w.r.t a variable \( x \) if \# is visible from 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 2.1 is well-formed w.r.t. the variables \( v \) and \( z \). In Figure 2.2, \( h_0 \) 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.

### 4.1.2 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 2.2, 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. \( x \).

### 4.1.3 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 \( \text{Val}(m_1) \leq \text{Val}(m_2) \).

For instance, in Figure 2.2, only \( h_5 \) is sorted. The figure to the right shows a signature which characterizes all heaps which are not sorted.

### 4.1.4 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 for a sorted linear list 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 signatures \( b_5, b_6, b_9, b_{10} \) can be discarded from the set above. Actually, a configuration is bad if there is a cell with successor \( * \) or a loop pattern in the heap, no matter where \( x \) points to. The signature \( b_{12} \) characterizes the cell with successor \( * \),
while the loop pattern is characterized by signature $b_{13}$. According to the ordering of signatures, we have $b_{12} \sqsubseteq b_1$, $b_{13} \sqsubseteq b_3$ and $b_{13} \sqsubseteq b_4$. Then, the signatures $b_1, b_3, b_4$ are discarded from the set, while the signatures $b_{12}, b_{13}$ are added into the set. Therefore, the set of bad configurations for a sorted linear list w.r.t. $x$ is characterized by the set $\{b_2, b_7, b_8, b_{11}, b_{12}, b_{13}\}$.

4.2 Sorted Cyclic List

Considering programs such as cyclic bubble sort, we would like such programs to produce a heap which is a cyclic list. Furthermore, the heap should not contain any garbage, and the output list should be sorted. Similar to the sorted linear list case, we describe the forbidden patterns of these three properties and collect all the signatures into a single set which exactly characterizes the set of bad configurations.

4.2.1 Well-Formedness

We say that $h$ is cyclically well-formed w.r.t. a variable $x$ if the $x$-cell belongs to a loop. Intuitively, if a heap satisfies this condition, then the part of the heap visible from the $x$-cell forms a cyclic list. The set of heaps violating cyclic well-formedness w.r.t. $x$ are characterized by the five signatures in the figure to the right. The signatures $b_{14}$ and $b_{15}$ characterize (together) all heaps in which the cell $*$ is visible from the $x$-cell, and the signatures $b_{16}$ and $b_{17}$ characterize (together) all heaps in which the cell $#$ is visible from the $x$-cell. The signature $b_{18}$ characterizes all heaps in which a loop is visible from the $x$-cell, but where the $x$-cell itself is not part of the loop.

4.2.2 Garbage

The garbage for a cyclic list is defined exactly the same as the linear list case in Section 4.3.2, which is characterized by the set of signatures $\{b_5, b_6, b_7, b_8, b_9, b_{10}\}$.

4.2.3 Sortedness

A heap is said to be cyclically sorted with respect to the variable $x$ if whenever a cell $m \in M$ belongs to the same loop as the $x$-cell, then either $\text{Val}(m) \leq \text{Val}(\text{Succ}(m))$ or $\text{Succ}(m) = \lambda(x)$, but not both. Intuitively, this means that the $x$-cell has the smallest value of all cells in the cycle, and each consecutive pair of cells in the cycle is pairwise
4.2.4 Putting Everything Together

As explained above, the signatures $b_5, \ldots, b_{10}, b_{14}, \ldots, b_{20}$ characterize the set of heaps which are bad for a sorted cyclic list w.r.t. $x$. Observe that $b_{14} \sqsubseteq b_9$, $b_{15} \sqsubseteq b_{10}$, $b_{16} \sqsubseteq b_8$, $b_{17} \sqsubseteq b_7$ and $b_{18} \sqsubseteq b_5$, which means that the signatures $b_5, b_7, b_8, b_9, b_{10}$ can be discarded from the set above. Actually, a configuration is bad if one of the following conditions is satisfied no matter where $x$ points to: (i) there is a cell with successor $\ast$, which is characterized by signature $b_{21}$; (ii) there is a cell with successor $\#$, which is characterized by signature $b_{22}$; (iii) there is a cell out of the loop, which is characterized by signature $b_{23}$. According to the ordering of signatures, we have $b_{21} \sqsubseteq b_{14}$, $b_{22} \sqsubseteq b_{17}$, $b_{23} \sqsubseteq b_6$ and $b_{23} \sqsubseteq b_{18}$. Then, the signatures $b_6, b_{14}, b_{17}, b_{18}$ are discarded from the set, while the signatures $b_{21}, b_{22}, b_{23}$ are added into the set. Therefore, the set of bad configurations for a sorted cyclic list w.r.t. $x$ is characterized by the set $\{b_{15}, b_{16}, b_{19}, b_{20}, b_{21}, b_{22}, b_{23}\}$.

4.3 Sorted Partition

Considering the partition program, we would like such program to produce a heap which contains two linear lists. Furthermore, the heap should not contain any garbage, there is no sharing part between these two linear lists and all the elements in one list are greater than or equal to the first element of the original list, while the elements in the other list are strictly smaller than that. We describe the forbidden patterns of these four properties and collect all the signatures into a single set which exactly characterizes the set of bad configurations.

4.3.1 Well-Formedness

We say that $h$ is well-formed w.r.t variables $x$ and $y$ if $\#$ is visible from both $x$-cell and $y$-cell. Equivalently, neither the cell $\ast$ nor any loop is visible from either $x$-cell or $y$-cell. Intuitively, if a heap satisfies this condition, then the part of the heap visible from the $x$-cell or $y$-cell forms a linear list ending with $\#$. The set of heaps violating well-formedness w.r.t. $x$ and $y$ are characterized by the set of signatures $\{b_1, \ldots, b_4, b_{24}, \ldots, b_{27}\}$. The signatures $b_1, b_2$ and $b_{24}, b_{25}$ characterize (together) all heaps in which the cell $\ast$ is
visible from $x$-cell and $y$-cell respectively. The signatures $b_3, b_4$ and $b_{26}, b_{27}$ characterize (together) all heaps in which a loop is visible from $x$-cell and $y$-cell respectively.

### 4.3.2 Garbage

We say that $h$ contains garbage w.r.t variables $x$ and $y$ if there is a cell $m \in M$ in $h$ which is not visible from both $x$-cell and $y$-cell. The figure to the right shows four signatures which together characterize the set of heaps which contain garbage w.r.t. $x$ and $y$, excluding those are subsumed by some signatures which characterize the bad configurations of well-formedness for partition as introduced in Section 4.3.1.

### 4.3.3 Sharing

We say that $h$ exhibits sharing w.r.t. two variables $x$ and $y$ if there is a cell $m \in M$ in $h$ which is visible from both the $x$-cell and the $y$-cell. In Figure 2.2, the heaps $h_0$, $h_1$ and $h_2$ exhibit sharing w.r.t. the variables $x$ and $z$. The figure to the right shows four signatures $b_{32}$, $b_{33}$, $b_{34}$ and $b_{35}$, which together characterize the set of heaps which exhibits sharing w.r.t. the variables $x$ and $y$.

### 4.3.4 Sortedness

A heap is said to be a sorted partition with respect to the variable $x$ and $y$ if whenever a cell $m_1 \in M$ is visible from the $x$-cell then $Val(\lambda(x)) \leq Val(m_1)$, and whenever a cell $m_2 \in M$ is visible from the $y$-cell then $Val(m_2) < Val(\lambda(x))$. Intuitively, this means that the $x$-cell has the smallest value of all cells in the linear list pointed by $x$, and greater than the value of any cell in the linear list pointed by $y$. The figure to the right shows five signatures $b_{36}, \ldots, b_{40}$, which characterize all heaps that are not sorted w.r.t. $x$ and $y$.

### 4.3.5 Putting Everything Together

As explained above, the signatures $b_1, \ldots, b_4, b_{27}, \ldots, b_{40}$ characterize the set of heaps which is bad for a partition w.r.t. $x$ and $y$. Actually, a configuration is bad if there is a cell with successor $\ast$ or a loop pattern in the heap, no matter where $x$ or $y$ points to. The signature $b_{41}$ characterizes the
cell with successor \(*\), while the loop pattern is characterized by signature \(b_{42}\). According to the ordering of signatures, we have \(b_{41} \sqsubseteq b_1, b_{41} \sqsubseteq b_{24}, b_{42} \sqsubseteq b_3, b_{42} \sqsubseteq b_{26}, b_{42} \sqsubseteq b_4\) and \(b_{42} \sqsubseteq b_{27}\). Then, the signatures \(b_1, b_3, b_{24}, b_{26}, b_{27}\) are discarded from the set, while the signatures \(b_{41}, b_{42}\) are added into the set. Therefore, the set of bad configurations for a sorted partition w.r.t. \(x\) and \(y\) is characterized by the set \(\{b_2, b_{25}, b_{28}, \ldots, b_{42}\}\).

4.4 More Bad on Inverse Sortedness

Considering programs such as reversion of a sorted linear list and reversion of a sorted cyclic list, we would like the output list (linear and cyclic respectively) to be well-formed, contains no garbage and inversely sorted.

4.4.1 Inversely Sorted Linear List

A heap is said to be \textit{inversely sorted} if it satisfies the condition that whenever a cell \(m_1 \in M\) is visible from a cell \(m_2 \in M\) then \(\text{Val}(m_2) \leq \text{Val}(m_1)\). The figure to the right shows a signature which characterizes all heaps which are not inversely sorted. Therefore, the set of bad configurations for a inversely sorted linear list w.r.t. \(x\) is characterized by the set \(\{b_2, b_{7}, b_{8}, b_{12}, b_{13}, b_{43}\}\).

4.4.2 Inversely Sorted Cyclic List

A heap is said to be \textit{cyclically inversely sorted} with respect to the variable \(x\) if whenever a cell \(m \in M\) belongs to the same loop as the \(x\)-cell, then either \(\text{Val}(\text{Succ}(m)) \leq \text{Val}(m)\) or \(\text{Succ}(m) = \lambda(x)\), but not both. Intuitively, this means that the \(x\)-cell has the largest value of all cells in the cycle, and each consecutive pair of cells in the cycle is pairwise sorted. The figure to the right shows two signatures \(b_{44}\) and \(b_{45}\), which characterize all heaps that are not cyclically sorted w.r.t. \(x\). Therefore, the set of bad configurations for a inversely sorted cyclic list w.r.t. \(x\) is characterized by the set \(\{b_{15}, b_{16}, b_{21}, b_{22}, b_{23}, b_{44}, b_{45}\}\).
Chapter 5

Reachability Analysis

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

5.1 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 $[G] = \{ h | h \xrightarrow{op} [g] \}$. Consider the signature $g$ to 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 Chapter 3), the set $H$ cannot be characterized by any set $G$ of signatures, i.e., there is no $G$ such that $[G] = H$. To overcome this problem, we define an approximate transition relation $\rightarrow_A$ which is an over-approximation of the relation $\rightarrow$. 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 $h_1 \xrightarrow{op} h'$.

5.2 Computing Predecessors
We show that, for an operation \( op \) and a signature \( g \), we can compute a finite set \( \text{Pre}(op)(g) \) of signatures such that 
\[
[\text{Pre}(op)(g)] = \{ h | h \xrightarrow{\text{op}} A [g] \}.
\]
For instance in the above case the set \( \text{Pre}(op)(g) \) 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 contraction (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 \( \text{Pre}(op)(g) \) as a finite set of signatures. 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 \( \lambda(y) \neq \lambda(x) \) for all \( y \neq x \), \( \text{Succ}(m) \neq \lambda(x) \) for all cells \( m \in M \), \( \text{Succ}(m) = \bot \), and \( \text{Ord}(x,y) = \text{Ord}(y,x) = \bot \) for all \( y \neq x \). We define \( m \) being \( x \)-semi-isolated in a 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 Chapter 4, and the \( x \)-cell in the signature \( \text{sig}(h_5) \) in Figure 2.2 are \( x \)-semi-isolated.

\( \text{Pre}(op)(g) \) is defined to be the set of saturated signatures \( g' \) such that one of the following conditions is satisfied:

1. \( op \) is of the form \( x = y \) and one of the following conditions is satisfied:
   
   \begin{itemize}
   
   \item \( \lambda(x) \in M^\#, \lambda(y) \in M^\#, \lambda(x) = \lambda(y), \text{ and } g' = g. \)
   
   \item \( \lambda(x) \in M^\#, \lambda(y) = \bot, \text{ and } g' = g \oplus_x y. \)
   
   \item \( \lambda(x) = \bot, \lambda(y) \in M^\#, \text{ and } g' = g \oplus_y x. \)
   
   \item \( \lambda(x) = \bot, \lambda(y) = \bot, \text{ and there is a signature } g_1 \text{ such that } g_1 \in g \oplus x, \)
   
   \hspace{1cm} \text{ and } g' = g_1 \oplus_{=x} y. \)
   
   \end{itemize}

In order to be able to perform this \( op \), the variables \( x \) and \( y \) should point to the same cell. If one (or both) of them is missing, then we add them to the signature (with the restriction that they point to the same cell).

2. \( op \) is of the form \( x \neq y \) and one of the following conditions is satisfied:

   \begin{itemize}
   
   \item \( \lambda(x) \in M^\#, \lambda(y) \in M^\#, \lambda(x) \neq \lambda(y), \text{ and } g' = g. \)
   
   \item \( \lambda(x) \in M^\#, \lambda(y) = \bot, \text{ and } g' \in g \oplus_{\neq x} y. \)
   
   \item \( \lambda(x) = \bot, \lambda(y) \in M^\#, \text{ and } g' \in g \oplus_{\neq y} x. \)
   
   \end{itemize}
• \( \lambda(x) = \bot, \lambda(y) = \bot \), and there is a signature \( g_1 \) such that \( g_1 \in g \oplus x, g' \in g_1 \oplus_{\neq x} y \).

We proceed as in case 1, but now under the restriction that \( x \) and \( y \) point to different cells (rather than to the same cell).

3. \( \text{op} \) is of the form \( x := y \) and one of the following conditions is satisfied:

   • \( \lambda(x) \in M^\#, \lambda(y) \in M^\#, \lambda(x) = \lambda(y) \), and \( g' = g \ominus x \).
   
   • \( \lambda(x) \in M^\#, \lambda(y) = \bot \), and there is a signature \( g_1 \) such that \( g_1 = g \ominus_{= x} y \), \( g' = g_1 \ominus x \).
   
   • \( \lambda(x) = \bot, \lambda(y) \in M^\# \), and \( g' = g \).
   
   • \( \lambda(x) = \bot, \lambda(y) = \bot \), and \( g' \in g \oplus y \).

   The difference compared to case 1 is that the variable \( x \) may have had any value before performing the assignment. Therefore, we remove \( x \) from the signature.

4. \( \text{op} \) is of the form \( x := y.\text{next} \) and one of the following conditions is satisfied:

   • \( \lambda(x) \in M^\#, \lambda(y) \in M, \text{Succ}(\lambda(y)) = \lambda(x) \), and \( g' = g \ominus x \).
   
   • \( \lambda(x) \in M^\#, \lambda(y) \in M, \text{Succ}(\lambda(y)) = \bot \), and there is a signature \( g_1 \) such that \( g_1 = \bigoplus (y \rightarrow x), g' = g_1 \ominus x \).
   
   • \( \lambda(x) \in M^\#, \lambda(y) = \bot \), and there are signatures \( g_1, g_2 \) such that \( g_1 \in g \oplus x, g_2 = g_1 \bigoplus (y \rightarrow x), g' = g_2 \ominus x \).
   
   • \( \lambda(x) = \bot, \lambda(y) \in M, \text{Succ}(\lambda(y)) \in M^\# \), and \( g' = g \).
   
   • \( \lambda(x) = \bot, \lambda(y) \in M, \text{Succ}(\lambda(y)) = \ast \), and there is a signature \( g_1 \) such that \( g_1 = g \oplus_{y \rightarrow \ast} x, g' = g_1 \ominus x \).
   
   • \( \lambda(x) = \bot, \lambda(y) \in M, \text{Succ}(\lambda(y)) = \bot \), and \( g' \in g \bigoplus (y \rightarrow) \).
   
   • \( \lambda(x) = \bot, \lambda(y) = \bot \), and there are signatures \( g_1, g_2, g_3 \) such that \( g_1 \in g \oplus x, g_2 \in g_1 \oplus x, g_3 = g_2 \bigoplus (y \rightarrow x), g' = g_3 \ominus x \).

   Similarly to case 3 we remove \( x \) from the signature. The successor of \( y \)-cell should be defined and point to \( x \)-cell. In case the successor is missing, we add an edge explicitly from the \( y \)-cell to \( x \)-cell. Furthermore, if \( x \) is missing then the successor of \( y \) may point anywhere inside the signature.

5. \( \text{op} \) is of the form \( x.\text{next} := y \) and one of the following conditions is satisfied:
• \( \lambda(x) \in M, \lambda(y) \in M^#, \text{Succ}(\lambda(x)) = \lambda(y) \), and \( g' = g \Box (x \rightarrow) \).
• \( \lambda(x) \in M, \text{Succ}(\lambda(x)) \in M^#, \lambda(y) = \bot \), and there is a signature \( g_1 \) such that \( g_1 \in g \boxplus_{\rightarrow y} y \), \( g' = g_1 \Box (x \rightarrow) \).
• \( \lambda(x) \in M, \text{Succ}(\lambda(x)) = \bot, \lambda(y) \in M^#, \text{and } g' = g \).
• \( \lambda(x) = \bot \), \( \lambda(y) \in M^# \), and there is signature \( g_1 \) such that \( g_1 \in g \oplus \lambda(x) \), \( g' = g_1 \Box (x \rightarrow) \).

After performing this \( op \), the successor of the \( x \)-cell should be \( y \)-cell. Before performing this \( op \), the successor could have been anywhere inside the signature, and the corresponding edge is therefore removed.

6. \( op \) is of the form new\((x)\) and one of the following conditions is satisfied:

• \( \lambda(x) \) is \( x\)-semi-isolated, and there is signature \( g_1 \) such that \( g_1 = g \ominus \lambda(x) \) and \( g' = g_1 \ominus x \).
• \( \lambda(x) = \bot \), \( \text{and } g' = g \) or \( g' \in g \oplus m \) for some semi-isolated cell \( m \).

After performing this \( op \), a new \( x\)-semi-isolated cell is added. So we remove the cells which are \( x\)-semi-isolated or semi-isolated (when \( x \) is not shown in the signature).

7. \( op \) is of the form delete\((x)\) and one of the following conditions is satisfied:

• \( \lambda(x) = \ast \), and there are a signature \( g_1, g_2 \) such that \( g_1 = g \ominus x \), \( g_2 = g_1 \ominus \lambda(x), g' = g_1 \boxplus (M^L \rightarrow x) \).
• \( \lambda(x) = \bot \), and there is signatures \( g_1 \) such that \( g_1 = g \oplus \lambda(x), g' = g_1 \boxplus (M^L \rightarrow x) \).

After performing this \( op \), the predecessor of \( x\)-cell should have successor \( \ast \), and \( x \) is on \( \ast \). So we add an \( x\)-isolated cell, pick some cells which has successor \( \ast \) and make them point to \( x\)-cell.

8. \( op \) is of the form read\((x)\) and one of the following conditions is satisfied:

• \( \lambda(x) \in M \), and \( g' = g \Box_{\text{Ord}} \lambda(x) \).
• \( \lambda(x) = \perp \), and there is a signature \( g_1 \) such that \( g_1 \in g \oplus_{\neq} x \), \( g' = g_1 \sqcup_{\text{Ord}} \lambda(x) \)

This \( \text{op} \) reads the value of \( x \)-cell, so we remove all the ordering relations of the \( x \)-cell. When \( x \) is not shown, we should put it somewhere first.

9. \( \text{op} \) is of the form \( x.num = y.num \) and one of the following conditions is satisfied:
   
   • \( \lambda(x) \in M, \lambda(y) \in M, \lambda(x) \equiv \lambda(y), \) and \( g' = g \).
   
   • \( \lambda(x) \in M, \lambda(y) \in M, \text{Ord}(x, y) = \perp, \) and \( g' = g \sqcup (x \equiv y) \).
   
   • \( \lambda(x) \in M, \lambda(y) = \perp, \) there is signature \( g_1 \) such that \( g_1 \in g \oplus_{\equiv x} y, \) \( g' = g_1 \sqcup (x \equiv y) \).
   
   • \( \lambda(x) = \perp, \lambda(y) \in M, \) there is signature \( g_1 \) such that \( g_1 \in g \oplus_{\equiv y} x, \) \( g' = g_1 \sqcup (x \equiv y) \).
   
   • \( \lambda(x) = \perp, \lambda(y) = \perp, \) there are signatures \( g_1, g_2 \) such that \( g_1 \in g \oplus_{\neq y} y, \) \( g_2 \in g_1 \oplus_{\equiv y} x, \) \( g' = g_2 \sqcup (x \equiv y) \).

   This \( \text{op} \) is enabled when the values of \( x \)-cell and \( y \)-cell are the same. If one (or both) of them is missing, then we add them to the signature (with the restriction that they have possibly the same value).

10. \( \text{op} \) is of the form \( x.num < y.num \) and one of the following conditions is satisfied:

   • \( \lambda(x) \in M, \lambda(y) \in M, \lambda(x) < \lambda(y), \) and \( g' = g \).
   
   • \( \lambda(x) \in M, \lambda(y) \in M, \text{Ord}(x, y) = \perp, \) and \( g' = g \sqcup (x < y) \).
   
   • \( \lambda(x) \in M, \lambda(y) = \perp, \) there is a signature \( g_1 \) such that \( g_1 \in g \oplus_{x <} y, \) \( g' = g_1 \sqcup (x < y) \).
   
   • \( \lambda(x) = \perp, \lambda(y) \in M, \) there is signature \( g_1 \) such that \( g_1 \in g \oplus_{\equiv y} x, \) \( g' = g_1 \sqcup (x < y) \).
   
   • \( \lambda(x) = \perp, \lambda(y) = \perp, \) there are signatures \( g_1, g_2 \) such that \( g_1 \in g \oplus_{\neq y} y, \) \( g_2 \in g_1 \oplus_{\equiv y} x, \) \( g' = g_2 \sqcup (x < y) \).

   We proceed as in case 9, but now under the restriction that \( x \)-cell has smaller value than \( y \)-cell (rather than to the same value).

11. \( \text{op} \) is of the form \( x.num := y.num \) and one of the following conditions is satisfied:

   • \( \lambda(x) \in M, \lambda(y) \in M, \lambda(x) \equiv \lambda(y), \) and \( g' = g \sqcup_{\text{Ord}} \lambda(x) \).
• $\lambda(x) \in M$, $\lambda(y) \in M$, $\text{Ord}(x, y) = \bot$, and $g' = g \sqcup_{\text{Ord}} \lambda(x)$.

• $\lambda(x) \in M$, $\lambda(y) = \bot$, and there is a signature $g_1$ such that $g_1 \in g \oplus_{\equiv x} y$, $g' = g_1 \sqcup_{\text{Ord}} \lambda(x)$.

• $\lambda(x) = \bot$, $\lambda(y) \in M$, and $g' = g$.

• $\lambda(x) = \bot$, $\lambda(y) = \bot$, and $g' \in g \oplus_{\neq} y$.

The difference compared to case 9 is that the $x$-cell may have had any value before performing the assignment. Therefore, we remove the ordering relation of $x$-cell from the signature.

12. $\text{op}$ is of the form $x.\text{num} < y.\text{num}$ and one of the following conditions is satisfied:

• $\lambda(x) \in M$, $\lambda(y) \in M$, $\lambda(x) < \lambda(y)$, and $g' = g \sqcup_{\text{Ord}} \lambda(x)$.

• $\lambda(x) \in M$, $\lambda(y) \in M$, $\text{Ord}(x, y) = \bot$, and $g' = g \sqcup_{\text{Ord}} \lambda(x)$.

• $\lambda(x) \in M$, $\lambda(y) = \bot$, and there is a signature $g_1$ such that $g_1 \in g \oplus_{x < y}$, $g' = g_1 \sqcup_{\text{Ord}} \lambda(x)$.

• $\lambda(x) = \bot$, $\lambda(y) \in M$, and $g' = g$.

• $\lambda(x) = \bot$, $\lambda(y) = \bot$, and $g' \in g \oplus_{\neq} y$.

We proceed as in case 11, but now under the restriction that $x$-cell has smaller value than $y$-cell.

13. $\text{op}$ is of the form $x.\text{num} > y.\text{num}$ and one of the following conditions is satisfied:

• $\lambda(x) \in M$, $\lambda(y) \in M$, $\lambda(x) < \lambda(y)$, and $g' = g \sqcup_{\text{Ord}} \lambda(x)$.

• $\lambda(x) \in M$, $\lambda(y) \in M$, $\text{Ord}(x, y) = \bot$, and $g' = g \sqcup_{\text{Ord}} \lambda(x)$.

• $\lambda(x) \in M$, $\lambda(y) = \bot$, and there is a signature $g_1$ such that $g_1 \in g \oplus_{x > y}$, $g' = g_1 \sqcup_{\text{Ord}} \lambda(x)$.

• $\lambda(x) = \bot$, $\lambda(y) \in M$, and $g' = g$.

• $\lambda(x) = \bot$, $\lambda(y) = \bot$, and $g' \in g \oplus_{\neq} y$.

We proceed as in case 11, but now under the restriction that $x$-cell has bigger value than $y$-cell.
5.3 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 [6], and augment the program with an initialization phase. The program starts from an empty heap (denoted $h_e$) 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 $h_e$. Figure 5.1 shows the procedure of generating a well-formed linear list.

```plaintext
1  new(x)
2  read(x)
3  x.next:=#
4  while(NonDet) {
5      new(temp)
6      if(NonDet) {
7          temp.num:=x.num
8      }
9      else {
10         temp.num:<x.num
11      }
12      temp.next:=x
13      x:=temp
14  }
```

Figure 5.1: The procedure of generating a linear list

5.4 Checking Safety Properties

5.4.1 Reachability Algorithm

To check the safety properties, 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 characterizes all heaps from which we can reach a bad heap through the approximate transition relation $\rightarrow_A$. The program satisfies the safety properties if $g \neq \text{sig}(h_e)$ for all $g \in G$. 

32
Input: A set of bad signatures $G_{Bad}$

Output: $\text{sig}(h_e)$ is reachable some $g \in G_{Bad}$?

1. $\text{WorkingSet} := G_{Bad}$
2. $\text{VisitedSet} := G_{Bad}$
3. while $\text{WorkingSet} \neq \emptyset$
   1. remove some $g$ from $\text{WorkingSet}$
   2. if $g = \text{sig}(h_e)$ then
      1. generate a sequence $g_0, \ldots, g_n$
      2. return $\text{TRUE}$
   3. else if $\exists g' \in \text{VisitedSet}. g' \sqsubseteq g$ then
      1. discard $g$
   4. else
      1. $\text{WorkingSet} := \text{WorkingSet} \cup \text{Pre}(g)$
      2. $\text{VisitedSet} := \{g\} \cup \{g' | g' \in \text{VisitedSet} \land (g \not\sqsubseteq g')\}$
   5. return $\text{FALSE}$

end

The safety properties are satisfied if the reachability algorithm returns $\text{FALSE}$. When $\text{TRUE}$ is returned, the safety properties are NOT satisfied, and a sequence of signatures $g_0, \ldots, g_n$ is generated where $g_0 \in G_{Bad}$, $g_n = \text{sig}(h_e)$ and $g_{i+1} \in \text{Pre}(g_i)$ for $i : 0 \leq i < n$. Such sequence can be viewed as an error trace represented in signatures.

5.4.2 Checking Entailment

In the reachability algorithm, we need to check that for two signatures $g$ and $g'$, if $g \sqsubseteq g'$ (or $g' \sqsubseteq g$) holds or not. This procedure is called checking entailment. We introduce a brute-force algorithm with some simple pruning strategies. First of all, we compare $g$ and $g'$ the program counters, labels, the number of cells, the number of edges (the successor relation), the number of dashed single-headed arrows (the ordering relation $\prec$) and the number of dashed double-headed arrows (the ordering relation $\equiv$), in order to make pruning. Secondly, we match $\#$, $*$ and labeled cells from $g$ to $g'$. Then, all possible matchings of other cells from $g$ to $g'$ are generated. Here we also use signature to denote a possible matching in the sense that if $g''$ is a matching from $g$ to $g'$, then $g''$ is isomorphic with $g'$ but with some cells tightly matched by all cells of $g$. In the third step, we remove some matching $g''$ and check the consistency between $g$ and $g''$ w.r.t. the edges, the dashed single-headed arrows and the dashed double-headed arrows. For instance, if there are cells $m_1, m_2 \in M$ matched to $m'_1, m'_2 \in M''$ with an edge between $m_1$ and $m_2$ in $g$ but no edge between
If $m_1''$ and $m_2''$ in $g''$, then $g$ and $g''$ are not consistent w.r.t. edges, therefore $g''$ should be discarded. Finally, if all possible matchings are discarded then $g \not\subseteq g'$, otherwise $g \subseteq g'$.

Input: Two signatures $g$ and $g'$

Output: $g \subseteq g'$?

if $g$ and $g'$ have the same program counter then
    return FALSE
else if $g$ contains more labels than $g'$ then
    return FALSE
else if $g$ contains more cells than $g'$ then
    return FALSE
else if $g$ contains more edges than $g'$ then
    return FALSE
else if $g$ contains more dashed single-headed arrows than $g'$ then
    return FALSE
else if $g$ contains more dashed double-headed arrows than $g'$ then
    return FALSE
else
    match #, * and labeled cells from $g$ to $g'$
    generate all possible matchings of other cells from $g$ to $g'$, named their set PossibleMatchingSet
    while PossibleMatchingSet $\neq \emptyset$
        remove some $g''$ from PossibleMatchingSet
        if $g$ and $g''$ are not consistent w.r.t. edges then
            discard $g''$
        else if $g$ and $g''$ are not consistent w.r.t. dashed single-headed arrows then
            discard $g''$
        else if $g$ and $g''$ are not consistent w.r.t. dashed double-headed arrows then
            discard $g''$
        else
            return TRUE
    return FALSE
end

Notice that the worst-case time complexity of this algorithm is $O(n!/(n - m)!)$, where $m$ is the number of cells in $g$ and $n$ is the number of cells in $g'$. So it will take more time if more signatures with large number of cells are computed. To make the reachability algorithm more efficient, we should generate as few as possible signatures of large sizes in the procedure, which depends on the searching strategy we choose.
5.4.3 Searching Strategies

In the reachability algorithm, we need to remove some $g$ from $\text{WorkingSet}$ and compute $\text{Pre}(g)$.

A common method is based on the principle of breadth-first search. In each iteration, we take all signatures in the $\text{WorkingSet}$, compute their predecessors and put them into a set called $\text{NewWorkingSet}$, then this $\text{NewWorkingSet}$ will be the new $\text{WorkingSet}$ for the next iteration. This method is quite straightforward and easy to implement. However, when the safety properties are not satisfied and an error-trace is generated, it will take a long time since we compute all the signatures of the same depth w.r.t. $G_{Bad}$ and an error-trace is usually found after performing a number of $\text{Pre}$ operations. Moreover, even if the safety properties are satisfied, we may compute a lot of predecessors and some of them have large sizes of cells, which also makes the method time consuming as discussed in Section 5.4.2.

To overcome these, we propose a new method based on the principle of depth-first search. Actually, an error-trace is usually a sequence of signatures in which every signature consists a small number of cells, which is observed from our experimental testings. Based on this observation, instead of taking all signatures in the $\text{WorkingSet}$ in each iteration, we take the signatures which have the same smallest size of cells in the $\text{WorkingSet}$ and name the set of them $\text{SmallestSizeWorkingSet}$. We compute the predecessors of all signatures in the $\text{SmallestSizeWorkingSet}$ and put them into the set $\text{NewWorkingSet}$, then the new $\text{WorkingSet}$ for the next iteration will be $\text{NewWorkingSet} \cup (\text{WorkingSet} - \text{SmallestSizeWorkingSet})$. This depth-first method provides a way to find the error-trace much faster than the breadth-first method. And because every time just the signatures of smallest size are taken, we get as few as possible signatures of very large sizes in the procedure, which also makes this method more efficient.

In the prototype we implemented, the depth-first method is used.
Chapter 6

Experimental Results

In this chapter, we give some example programs and show the experimental results.

6.1 Example Programs

6.1.1 Insert

In Figure 6.1, the insert program is illustrated. It inserts the elem-cell into the sorted linear list pointed to by x, such that the returning linear list is again sorted. The initialization phase (lines 1 - 16) first creates a non-empty sorted linear list, and then an additional node to be inserted. The set of bad configurations is described in Section 4.1.4.

6.1.2 Insert(bug)

The faulty version of the insert program is identical to the one in Figure 6.1, except that t1:=x.next is substituted for t1:=x on line 22. It is intended to work the same way as the correct version, but the faulty initialization of t1 makes it fail for the case of the elem-cell containing a smaller value than any other value in the list. This program also uses the set of bad configurations described in Section 4.1.4, and we get an error-trace to the bad state b11 as expected.

6.1.3 Merge

In Figure 6.2, the merge program is shown. It takes as input two sorted lists x and y, and merges them into one sorted list x. The set of bad configurations is described in Section 4.1.4.
new(x)
read(x)
x.next:=#
while(NonDet) {
  new(temp)
  if(NonDet) {
    temp.num:=x.num
  }
  else {
    temp.num:<x.num
  }
  temp.next:=x
  x:=temp
}
new(elem)
read(elem)
if(x.num>elem.num) {
elem.next:=x
x:=elem
return x
}
t1:=x.next
t2:=x
while(t1=/=#) {
  if(t1.num<elem.num) {
    t2:=t1
t1:=t2.next
  }
  else {
    t2.next:=elem
t2.next:=t1
    return x
  }
}
t2.next:=elem
t2.next:=#
return x

if(x=#) {
h:=y
return h
}
if(y=#) {
h:=x
return h
}
if(x.num<y.num) {
h:=x
x:=h.next
}
else {
h:=y
y:=h.next
}
t:=h
while(x=/=#&&y=/=#) {
  if(x.num<y.num) {
    t.next:=x
t:=x
    x:=x.next
  }
  else {
    t.next:=y
t:=y
    y:=y.next
  }
}
if(x=/=#) {
t.next:=x
}
if(y=/=#) {
t.next:=y
}
return h

Figure 6.1: The insert program including initialization phase
Figure 6.2: The merge program without initialization phase
6.1.4 Reverse

In Figure 6.3, the reverse program is shown. It reverses a sorted linear list $x$, and returns an inversely sorted linear list $y$. The set of bad configurations is described in Section 4.4.1, except that the variable $x$ should be replaced by $y$ wherever it occurs.

```
1  y:=#
2  while(x=/=#) {
3    t:=y
4    y:=x
5    x:=x.next
6    y.next:=t
7  }
8  return y
```

Figure 6.3: The reverse program without initialization phase

6.1.5 ReverseCyclic

In Figure 6.4, the reverseCyclic program is shown. It takes a sorted cyclic list $x$ as input, and returns an inversely sorted cyclic list $y$. The initialization phase (lines 1 - 23) creates a non-empty sorted cyclic list. The set of bad configurations is described in Section 4.4.2, except that the variable $x$ should be replaced by $y$ wherever it occurs.

6.1.6 Partition

In Figure 6.5, the partition program is shown. It takes a linear list $b$ as input, and returns two linear lists $b$ and $s$ such that whenever a cell $m_1 \in M$ is visible from the $b$-cell then $Val(\lambda(b)) \leq Val(m_1)$, and whenever a cell $m_2 \in M$ is visible from the $s$-cell then $Val(m_2) < Val(\lambda(b))$. The initialization phase (lines 1 - 9) creates a non-empty linear list $b$. The set of bad configurations is described in Section 4.3.5, except that the variables $x$ and $y$ should be replaced by $b$ and $s$ wherever they occur.
1 new(x)
2 read(x)
3 new(tail)
4 if(NonDet) {
5    tail.num:=x.num
6 }
7 else {
8    tail.num:=x.num
9 }
10 x.next:=tail
11 tail.next:=x
12 while(NonDet) {
13    new(temp)
14    if(NonDet) {
15        temp.num:=x.num
16    }
17    else {
18        temp.num:=x.num
19    }
20    temp.next:=x
21    tail.next:=temp
22    x:=temp
23 }
24 t:=x.next
25 while(t=/=x) {
26    k:=t
27    t:=t.next
28 }
29 y:=k
30 while(x=/=y) {
31    t:=k
32    k:=x
33    x:=x.next
34    k.next:=t
35 }
36 y.next:=k
37 return y

1 new(b)
2 read(b)
3 b.next:=#
4 while(NonDet) {
5    new(temp)
6    read(temp)
7    temp.next:=b
8    b:=temp
9 }
10 t:=b.next
11 b.next:=#
12 s:=#
13 while(t=/=#) {
14    tt:=t.next
15    if(t.num<b.num) {
16        t.next:=s
17        s:=t
18    }
19    else {
20        tb:=b.next
21        b.next:=t
22        t.next:=tb
23    }
24    t:=tt
25 }
26 return b,s

Figure 6.4: The reverseCyclic program including initialization phase

Figure 6.5: The partition program including initialization phase
6.1.7 BubbleSort

In Figure 6.6, the bubbleSort program is shown. It takes a linear list $x$ as input, and returns a sorted linear list $x$. The set of bad configurations is described in Section 4.1.4.

```plaintext
1  if(x=#) {
2    return x
3  }
4  change:=TRUE
5  while(change) {
6    p:=#
7    change:=FALSE
8    y:=x
9    yn:=y.next
10   while(yn=/=#) {
11      if(y.num>yn.num) {
12        t:=yn.next
13        change:=TRUE
14        y.next:=t
15        yn.next:=y
16        if(p=#) {
17          x:=yn
18        }
19      } else {
20        p.next:=yn
21      }
22    p:=yn
23    y:=t
24  }
25  else {
26    p:=y
27    y:=yn
28    yn:=y.next
29  }
30 }
31 return x
```

Figure 6.6: The bubbleSort program without initialization phase
6.1.8 BubbleSort(bug)

The faulty version of the program bubbleSort is identical to the one in Figure 6.6, except that \( y := x \) is substituted for \( y := x.\text{next} \) on line 8. It is intended to work the same way as the correct version, but the faulty initialization of \( y \) makes it fail for the case of the \( y \)-cell containing a smaller value than any other value in the list. This program also uses the set of bad configurations described in Section 4.1.4, and we get an error-trace to the bad state \( b_{11} \) as expected.

6.1.9 BubbleSortCyclic

In Figure 6.7, the bubbleSortCyclic program is shown. It takes a cyclic list \( x \) as input, and returns a sorted cyclic list \( x \). The initialization phase (lines 1 - 13) creates a non-empty cyclic list \( x \). The set of bad configurations is described in Section 4.2.4.

```plaintext
1 new(x) 23 while(yn=/=#) { 24 if(y.num>yn.num) { 25 t:=yn.next 26 change:=TRUE 27 y.next:=t 28 yn.next:=y 29 if(p=#) { 30 x:=yn 31 } 32 else { 33 p.next:=yn 34 } 35 p:=yn 36 yn:=t 37 } 38 else { 39 p:=y 40 y:=yn 41 yn:=y.next 42 } 43 } 44 y.next:=x 45 return x
```

Figure 6.7: The bubbleSortCyclic program including initialization phase
6.1.10 InsertionSort

In Figure 6.8, the insertionSort program is shown. It takes a linear list \( x \) as input, and returns a sorted linear list \( x \). The set of bad configurations is described in Section 4.1.4.

```plaintext
1   while(t=/=#) {
2       t1:=x
3       t2:=x.next
4       while(t2=/=#&&t2.val<t.val) {
5           t1:=t2
6           t2:=t2.next
7       }
8       e:=t
9       t:=t.next
10      t1.next:=e
11      e.next:=t2
12   }
13  return x
```

Figure 6.8: The insertionSort program without initialization phase

6.2 Experimental Results

We have implemented a prototype written in Java. All the programs shown in Section 6.1 are verified w.r.t. their own safety properties. For instance, we verified well-formedness, absence of garbage and sortedness for insert problem, by using the set of bad configurations described in Section 4.1.4.

<table>
<thead>
<tr>
<th>Prog.</th>
<th>Time</th>
<th>#Sig.</th>
<th>#Iter.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert</td>
<td>2.9 s</td>
<td>1601</td>
<td>82</td>
</tr>
<tr>
<td>Insert (bug)</td>
<td>0.9 s</td>
<td>267</td>
<td>28</td>
</tr>
<tr>
<td>Merge</td>
<td>22.4 s</td>
<td>5830</td>
<td>183</td>
</tr>
<tr>
<td>Reverse</td>
<td>1.2 s</td>
<td>311</td>
<td>69</td>
</tr>
<tr>
<td>ReverseCyclic</td>
<td>1.5 s</td>
<td>574</td>
<td>85</td>
</tr>
<tr>
<td>Partition</td>
<td>1501 s</td>
<td>32944</td>
<td>150</td>
</tr>
<tr>
<td>BubbleSort</td>
<td>35.7 s</td>
<td>10034</td>
<td>142</td>
</tr>
<tr>
<td>BubbleSortCyclic</td>
<td>36.1 s</td>
<td>10143</td>
<td>150</td>
</tr>
<tr>
<td>BubbleSort (bug)</td>
<td>0.9 s</td>
<td>181</td>
<td>22</td>
</tr>
<tr>
<td>InsertionSort</td>
<td>1510 s</td>
<td>39267</td>
<td>212</td>
</tr>
</tbody>
</table>

Table 6.1: Experimental results
Table 6.1 shows the time and number of iterations it took for the run to complete. The column “#Sig.” shows the total number of signatures that were computed throughout the analysis. All experiments were performed on a 2.2 GHz Intel Core 2 Duo with 4 GB of RAM.

**Remark on Spurious Errors** Since the transition relation used in the analysis is an over-approximation, there is a risk of generating false counter-examples. In our experiments, this arises only in one example, namely a version of Merge (different from the one included in Table 6.1) in which one pointer $x$ follows another pointer $y$, such that there is one cell between $x$ and $y$. If $y$ moves first followed by $x$ then, according to our approximation, the pointers may now point to the same cell. This kind of counter-examples can be dealt with by refining the approximation so that contraction is allowed only if the size of the resulting list segment will not become smaller than some given $k \geq 1$. For the above counter-example, it is sufficient to take $k = 2$. Notice that in the current approximation we have $k = 1$.
Chapter 7

Conclusions and Future Work

We have presented a method for automatic verification of safety properties for programs which manipulate heaps containing data. The main ingredient is a compact data structure, called a signature, for symbolic representation of infinite sets of heaps. Signatures are used as forbidden patterns which characterize heaps violating safety properties such as absence of garbage, well-formedness, and sortedness. A given safety property can be checked by performing backward reachability analysis on the set of signatures. Although the procedure is not guaranteed to terminate in general, our prototype implementation terminates on all the given examples. 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 are currently using a simple but inefficient (brute-force) algorithm for checking entailment. We believe that adapting specialized algorithms such as the one described in [17] will substantially improve performances of the tool.

Several extensions of our framework can be carried out by refining the considered preorder (and the abstraction it induces). For instance, it could be possible to consider integer program variables whose values are related to the lengths of paths in the heap. Such an extension with arithmetical reasoning can be done in our framework by considering preorders involving for instance gap-order constraints [14]. 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.
Bibliography


