A Shape Analysis for Non-linear Data Structures

Renato Cherini, Lucas Rearte, and Javier Blanco

FaMAF, Universidad Nacional de Córdoba, 5000 Córdoba, Argentina

Abstract. We present a terminating shape analysis based on Separation Logic for programs that manipulate non-linear data structures such as trees and graphs. The analysis automatically calculates concise invariants for loops, with a level of precision depending on the manipulations applied on each program variable. We report experimental results obtained from running a prototype that implements our analysis on a variety of examples.

1 Introduction

Shape Analysis is a form of static code analysis for imperative programs using dynamic memory that attempts to discover and verify properties of linked data structures such as lists, trees, heaps, etc. A shape analysis is intended not only to report null-pointer dereferences or memory leaks, but also to analyze the validity of non trivial properties about the shape of the dynamic structures in memory.

There exists in the literature a variety of shape analysis based on different theories, for instance: 3-valued logic ([22,28]), monadic second order logic ([24,20]), and Separation Logic ([12,4,25,11]). In particular the latter received recently great attention due to the advantages of local reasoning enabled by their logical framework ([27,26]), which leads to modular analysis that can be relatively easily extended to support large-scale programs ([7,3,6,33,15]), concurrent programming ([16,31,9,8]) and object-oriented programming ([13,29]).

This family of shape analysis is based on a symbolic execution of the program over abstract states specified by formulæ of a restricted subset of Separation Logic, including predicates describing linear data structures, possibly combined in intricate ways, such as linked lists, doubly linked lists, etc. In this article we explore the possibility of extending the shape analysis of [12] to support non-linear data structures such as trees and graphs. At a first glance, we can foresee some difficulties:

- The use of a naive predicate to describe the structures, like the usual tree predicate of Separation Logic, would lead to a unmanageable proliferation of occurrences of predicates in formulæ, caused by the inherent multiple recursion. In the case of graphs, there is no well established predicate in the literature which adequately specifies them.
- Algorithms over trees and graphs are often more complex than their counterparts on lists. Usually they involve intensive pointer manipulation and nested control structures. Algorithms like the Schorr-Waite graph traversal

R. Cousot and M. Martel (Eds.): SAS 2010, LNCS 6337, pp. 201–217, 2010.

are usually proposed as challenges for any new verification methodology. Their complexity imposes strong requirements for precision in calculating the loop invariants needed to verify interesting properties.
– Every particular path in the traverse of a structure is usually relevant to the validity of interesting properties satisfied by a given algorithm. However the multiple links of non-linear structure nodes triggers an exponential growth of the number of paths in traversing such structures. This raises the need for a balance between precision and abstraction to prevent an excessive growth in the number of formulæ composing an abstract state.

The contribution of this article is a terminating shape analysis which, given a precondition for a program, automatically computes a postcondition and invariants for each program loop. This analysis adjusts the level of abstraction in the computation of invariants, taking into account the information requirements for the the manipulation applied to each variable. Thus, the calculated invariants turn out to be compact and accurate enough in most practical cases. In order to define the analysis we have introduced a linear recursive predicate for the description of (families of) binary trees, leading to simple specifications of partial data structures ocurring at intermediate points in the execution of an iterative algorithm. This predicate can be easily adapted to deal with other data structures, such as threaded trees, balanced trees, graphs, etc.

The article is organized as follows. In section 2 we adapt the semantic setting from [12] to our purposes. First, we introduce the concrete memory model and semantics that defines the programming language for algorithms manipulating binary trees. Then we present an executable intermediate semantics that considers programs as abstract state transformers. Each abstract state consists of a set of restricted formulæ of Separation Logic called Symbolic Heaps, representing all concrete states which satisfy any of these formulæ. In section 3 we extend this semantics by introducing an abstraction phase for computing loop invariants, defining an executable and terminating abstract semantics. In section 4 we briefly present an extension of our framework to the domain of graphs. In section 5 we show the experimental results of our analysis on a variety of examples. Finally, in section 6 we discuss the conclusions and related work.

2 Semantic Settings

2.1 Concrete Semantics

We assume the existence of a *finite* set Vars of program variables $x, y, \ldots$, with values in $\mathsf{Locations} \cup \{\mathbf{nil}\}$. A state is a pair consisting of a (total) function Stack from program variables to values, and a partial function Heap mapping allocated memory adresses to triples of values (l, r, v), representing a binary tree node with value v and links to left and right subtrees l and r respectively.

$$\mathsf{Values} \supseteq \mathsf{Locations} \cup \{\mathbf{nil}\} \qquad \mathsf{Heaps} \doteq \mathsf{Locations} \rightharpoonup_f (\mathsf{Values}, \mathsf{Values}, \mathsf{Values})$$
$$\mathsf{Stacks} \doteq \mathsf{Vars} \to \mathsf{Values} \qquad \mathsf{States} \doteq \mathsf{Stacks} \times \mathsf{Heaps}$$

$$[x]_e.s \doteq s.x \qquad [\mathbf{nil}]_e.s \doteq \mathbf{nil}$$

$$\frac{[e]_e.s = v}{(s,h),\ x := e \ \leadsto\ (s|x \to v, h)} \qquad \frac{l \notin dom.h}{(s,h),\ \mathbf{new}(x) \ \leadsto\ (s|x \to l, h|l \to (v_1, v_2, v_3)}$$

$$\frac{[y]_e.s = l \quad h.l.i = v}{(s,h),\ x := y.i \ \leadsto\ (s|x \to v, h)} \qquad \frac{[x]_e.s = l \quad l \in dom.h}{(s,h),\ \mathbf{free}(x) \ \leadsto\ (s, h - l)}$$

$$\frac{[y]_e.s = l \quad [e]_e.s = v \quad l \in dom.h}{(s,h),\ y.i := e \ \leadsto\ (s, h|l.i \to v)} \qquad \frac{[x]_e.s \notin dom.h}{(s,h),\ a(x) \ \leadsto\ \top}$$

Fig. 1. Concrete semantics of expressions and atomic commands

We use a simple imperative programming language with explicit heap manipulation commands for lookup, mutation, allocation and disposing, given by the following grammar, where x is a program variable, and $0 \leq i \leq 2$:

$$\begin{array}{rll} e & ::= x \mid \mathbf{nil} \mid 0 \mid 1 \ldots & \text{Expressions} \\ eq & ::= e = e \mid e \neq e & \text{(In)Equalities} \\ b & ::= (eq \wedge \ldots \wedge eq) \vee \ldots \vee (eq \wedge \ldots \wedge eq) & \text{DNF Boolean Expressions} \\ a & ::= x := e \mid x := x.i \mid x.i := e \mid \mathbf{new}(x) \mid \mathbf{free}(x) & \text{Atomic Commands} \\ p & ::= a \mid p; p \mid \mathbf{while}\ b\ \mathbf{do}\ p \mid \mathbf{if}\ b\ \mathbf{then}\ p\ \mathbf{else}\ p & \text{Compound Commands} \end{array}$$

The concrete semantics is given by continuous functions $[\![p]\!]_c : \mathsf{D}_c \to \mathsf{D}_c$ for each program p on the complete lattice D_c defined as the topped powerset of States, i.e. $\mathsf{D}_c \doteq \mathcal{P}(\mathsf{States} \cup \{\top\})$. The distinguished element $\top$ represents an execution that terminates abnormally with a memory fault. The order of the lattice is given by set inclusion $\subseteq$, taking $\top$ as the top element and equating all sets that contain it.

For each atomic command a we present a relation $\leadsto \subseteq \mathsf{State} \times \mathsf{State} \cup \{\top\}$ in figure 1. *Notation: We use $a(x)$ to denote an atomic command that accesses the heap cell x, period (.) for function application, $f|x \to v$ for the update of function at x with new value v, $dom.f$ and $img.f$ for the domain and the image of a function f respectively, and $f - x$ for the elimination of x from the partial function f.* The relation $\leadsto$ mimics the standard operational semantics for Separation Logic [27], and can be easily extended to a function $a^{\dagger} : \mathsf{D}_c \to \mathsf{D}_c$:

$$a^{\dagger}.S \doteq \{\sigma' \mid \exists \sigma \in S \cdot \sigma, a \ \leadsto\ \sigma' \text{ or } (\sigma = \sigma' = \top)\}$$

Thus concrete semantics for atomic command a is defined as $[\![a]\!]_c = a^{\dagger}$. This semantics is extended to compound commands as usual:

$$\begin{array}{rl} [\![p; p']\!]_c & \doteq [\![p']\!]_c \circ [\![p]\!]_c \\ [\![\mathbf{if}\ b\ \mathbf{then}\ p\ \mathbf{else}\ p']\!]_c & \doteq ([\![p]\!]_c \circ \mathsf{filter}.b) \cup ([\![p']\!]_c \circ \mathsf{filter}.\neg b) \\ [\![\mathbf{while}\ b\ \mathbf{do}\ p]\!]_c & \doteq \lambda S \cdot \mathsf{filter}.\neg b \circ (\mu S' \cdot S \cup ([\![p]\!]_c \circ \mathsf{filter}.b).S') \end{array}$$

where we use μ to denote the minimal fixed point operator, $\circ$ the functional composition, and $\neg$ a meta-operation that transforms a boolean expression b in its negation in disjunctive normal form. Function filter removes states that are inconsistent with the truth value of boolean expression b of guarded commands.

$s \models \mathbf{true}$	always
$s \models eq$	iff $[eq]_e.s$
$s \models \Pi_1 \wedge \Pi_2$	iff $s \models \Pi_1$ and $s \models \Pi_2$
$(s,h) \models \mathbf{emp}$	iff $h = \emptyset$
$(s,h) \models \mathbf{junk}$	iff $h \neq \emptyset$
$(s,h) \models \mathbf{true}$	always
$(s,h) \models x \mapsto l,r,v$	iff $h = \{([x]_e.s, ([l]_e.s, [r]_e.s, [v]_e.s))\}$
$(s,h) \models \Sigma_1 * \Sigma_2$	iff there exist h_1, h_2 such that $h_1 \cap h_2 = \emptyset$ and $h = h_1 \cup h_2$ and $(s,h_1) \models \Sigma_1$ and $(s,h_2) \models \Sigma_2$
$(s,h) \models \Pi \mid \Sigma$	iff there exists $\overline{v}'$ such that $s[\overline{x}' \rightarrow \overline{v}' \models \Pi$ and $(s[\overline{x}' \rightarrow \overline{v}', h) \models \Sigma$ where $\overline{x}'$ is the (sequence of) primed variable(s) in $\Pi \mid \Sigma$ and $\overline{v}'$ is a (sequence of) fresh value(s).

Fig. 2. Semantics of Symbolic Heaps

2.2 Symbolic Heaps and Intermediate Semantics

In order to define intermediate semantics, it is necessary to extend the concrete state model, assuming the existence of an infinite set Vars' of primed variables. These variables are implicitly existentially quantified in the semantics and are intended to be used only in formulæ and not within the program text.

A symbolic heap $\Pi \mid \Sigma$ consists of a formula Π of *pure* predicates about equalities and inequalities of variables, and a formula Σ of *spatial* predicates about the heap, according to the following grammar:

$$
\begin{array}{rll}
e & ::= x \mid x' \mid \mathbf{nil} \mid 0 \mid 1 \ldots & \text{Expressions} \\
ms & ::= \{\!|e, e, \ldots, e|\!\} & \text{Multiset Expressions} \\
\Pi & ::= \mathbf{true} \mid eq \mid \Pi \wedge \Pi & \text{Pure Predicates} \\
\Sigma & ::= \mathbf{emp} \mid \mathbf{true} \mid \mathbf{junk} \mid v \mapsto e,e,e \mid \mathbf{trees}.ms.ms \mid \Sigma * \Sigma & \text{Heap Predicates} \\
SH & ::= \Pi \mid \Sigma & \text{Symbolic Heaps}
\end{array}
$$

where $x \in \mathsf{Vars}$, $x' \in \mathsf{Vars}'$, $v \in \mathsf{Vars} \cup \mathsf{Vars}'$, eq is defined as in the previous section, and the expressions ms represent constant multisets of expressions. *Notation: we use metavariables $\mathcal{C}, \mathcal{D}, \mathcal{E}, \mathcal{F}, \ldots$ to denote multisets, $\uplus$ for the sum of multisets, $\emptyset$ for the empty multiset, $\oplus$ for insertion of an item instance, $\ominus$ for deletion of all item instances, $\Subset$ for membership, and $\Subset\!\!\!\!\!/$ for non-membership.*

As symbolic heaps represent a fragment of Separation Logic, its semantics with respect to concrete state model is mostly standard. We present it in figure 2, except for **trees** predicate. We briefly discuss the spatial component; interested reader should refer to [27] for more details. The predicate **emp** specifies that no dynamic memory is allocated, whereas **junk** states there is garbage, consisting in some allocated but inaccessible cell. The predicate **true** is valid in any heap and it will be heavily used to indicate the *possible* existence of garbage. 'Points-to' predicate $x \mapsto l,r,v$ specifies the heap consisting of a single memory cell with address x and value (l,r,v). The *spatial conjunction* $\Sigma_1 * \Sigma_2$ is valid if the heap can be divided into two disjoint subheaps satisfying Σ_1 and Σ_2 respectively.

The predicate **trees** is intended to define (a family of) binary trees. More generally, $\mathbf{trees}.\mathcal{C}.\mathcal{D}$ defines a family of possible partial trees with roots in $\mathcal{C}$ and dangling pointers in $\mathcal{D}$. On the one hand, a dangling pointer of a partial tree can point to an internal node of another tree. On the other hand, a dangling pointer can be shared by two or more partial trees. Thus, $\mathbf{trees}.\mathcal{C}.\mathcal{D}$ defines a binary node structure which allows the possibility of sharing, but only restricted to the heap cells mentioned by $\mathcal{D}$. More precisely, $\mathbf{trees}.\mathcal{C}.\mathcal{D}$ is defined by the least predicate satisfying the following equations of the standard Separation Logic:

$$\begin{aligned}
\mathbf{trees}.\emptyset.\emptyset &\doteq \mathbf{emp} \\
\mathbf{trees}.\emptyset.\mathbf{nil} \oplus \mathcal{D} &\doteq \mathbf{trees}.\emptyset.\mathcal{D} \\
\mathbf{trees}.x \oplus \mathcal{C}.\mathcal{D} &\doteq ((x = \mathbf{nil} \vee x \Subset \mathcal{D}) \wedge \mathbf{trees}.\mathcal{C}.(x \ominus \mathcal{D})) \vee \\
&\quad ((x \neq \mathbf{nil} \wedge x \notin \mathcal{D}) \wedge (\exists l, r, v \cdot x \mapsto l, r, v * \mathbf{trees}.l \oplus r \oplus \mathcal{C}.\mathcal{D}))
\end{aligned}$$

The predicate **trees** is useful to specify the intermediate structures that occur during loop iteration and it satisfies some nice syntactic properties. The possible sharing is manageable due to the kind of formulæ that usually specify the pre-conditions of interest. The relation to the standard predicate **tree**[1] occurring in the literature of Separation Logic, is quite straightforward:

Lemma 1. *The following is a valid formula in Separation Logic:*

$$\mathbf{trees}.\{\!|x, y, \ldots, z|\!\}.\emptyset \Leftrightarrow \mathbf{tree}.x * \mathbf{tree}.y * \ldots * \mathbf{tree}.z$$

Analogously to the concrete semantics, the intermediate semantics is defined on the lattice $\mathsf{D}_a \doteq \mathcal{P}(\mathsf{SH} \cup \top)$, where SH denotes the set of symbolic heaps. For each atomic command a transition relation $\rightsquigarrow \subseteq \mathsf{SH} \times \mathsf{SH} \cup \{\top\}$ is defined:

$$\Pi \mid \Sigma,\ x := e \ \rightsquigarrow\ \Pi' \wedge x = e_{/x \leftarrow x'} \mid \Sigma' \qquad \Pi \mid \Sigma,\ \mathbf{new}(x) \ \rightsquigarrow\ \Pi' \mid \Sigma' * x \mapsto \overline{e'}$$

$$\frac{\overline{g} = \overline{e}|i \rightarrow f}{\Pi \mid \Sigma * x \mapsto \overline{e},\ x.i := f \ \rightsquigarrow\ \Pi \mid \Sigma * x \mapsto \overline{g}} \qquad \Pi \mid \Sigma * x \mapsto \overline{e},\ \mathbf{free}(x) \ \rightsquigarrow\ \Pi \mid \Sigma$$

$$\frac{f = \overline{e}.i_{/x \leftarrow x'}}{\Pi \mid \Sigma * y \mapsto \overline{e}, x := y.i \ \rightsquigarrow\ \Pi' \wedge x = f \mid \Sigma' * (y \mapsto \overline{e})_{/x \leftarrow x'}}$$

Notation: we denote by $\overline{e}$ a triple of expressions, by $\overline{e'}$ a triple of quantified fresh variables, and by $P_{/x \leftarrow y}$ the syntactic substitution of y for x in P. In every case $\Pi' = \Pi_{/x \leftarrow x'}$, and $\Sigma' = \Sigma_{/x \leftarrow x'}$, where $x' \in \mathsf{Vars}'$ is a fresh variable.

Commands for mutation, lookup and disposing require that the pre-state explicitly indicates the existence of the cell to be dereferenced. To ensure this, we define a function $\mathsf{rearr}.x : \mathsf{D}_a \rightarrow \mathsf{D}_a$ that given a variable of interest x tries to reveal the memory cell pointed to by x in every symbolic heap in certain abstract

[1] This is defined as the least predicate that satisfies the equation

$$\mathbf{tree}.x \Leftrightarrow (x = \mathbf{nil} \wedge \mathbf{emp}) \vee (\exists l, r, v \cdot x \mapsto l, r, v * \mathbf{tree}.l * \mathbf{tree}.r)$$

$$
\begin{array}{ll}
\Pi \mid \Sigma \vdash x = \mathbf{nil} & \text{if } (x \equiv \mathbf{nil}) \text{ or } (\Pi = \Pi' \wedge x = y \text{ and } \Pi' \mid \Sigma \vdash y = \mathbf{nil}) \\
\Pi \mid \Sigma \vdash x = y & \text{if } (x \equiv y) \text{ or } (\Pi = \Pi' \wedge x = z \text{ and } \Pi' \mid \Sigma \vdash z = y) \\
\Pi \mid \Sigma \vdash x \neq \mathbf{nil} & \text{if } (\Pi = \Pi' \wedge x \neq \mathbf{nil}) \text{ or } (\Sigma = \Sigma' * y \mapsto \overline{e} \text{ and } \Pi \mid \Sigma \vdash x = y) \\
 & \text{or } (\Pi = \Pi' \wedge z = \mathbf{nil} \text{ and } \Pi' \mid \Sigma \vdash x \neq z) \\
 & \text{or } (\Pi = \Pi' \wedge x = y \text{ and } \Pi' \mid \Sigma \vdash y \neq \mathbf{nil}) \\
\Pi \mid \Sigma \vdash x \neq y & \text{if } (\Pi = \Pi' \wedge x \neq y) \text{ or } (\Pi = \Pi' \wedge x = z \text{ and } \Pi' \mid \Sigma \vdash z \neq y) \\
 & \text{or } (\Pi \mid \Sigma \vdash \mathit{Cell}(x) * \mathit{Cell}(y) \text{ and } \Pi \mid \Sigma \vdash x \neq \mathbf{nil} \vee y \neq \mathbf{nil}) \\
\Pi \mid \Sigma \vdash x \notin \emptyset & \text{never} \\
\Pi \mid \Sigma \vdash x \in \{\!|e_1, .., e_n|\!\} & \text{if } \Pi \mid \Sigma \vdash x = e_i \text{ for some } i \text{ such that } 1 \leq i \leq n \\
\Pi \mid \Sigma \vdash x \notin \emptyset & \text{always} \\
\Pi \mid \Sigma \vdash x \notin \{\!|e_1, .., e_n|\!\} & \text{if } \Pi \mid \Sigma \vdash x \neq e_i \text{ for every } i \text{ such that } 1 \leq i \leq n \\
\Pi \mid \Sigma \vdash eq_1 \wedge \ldots \wedge eq_n & \text{if } \Pi \mid \Sigma \vdash eq_1 \text{ and } \ldots \text{and } \Pi \mid \Sigma \vdash eq_n \\
\Pi \mid \Sigma \vdash eq_1 \vee \ldots \vee eq_n & \text{if } \Pi \mid \Sigma \vdash eq_1 \text{ or } \ldots \text{or } \Pi \mid \Sigma \vdash eq_n \\
\Pi \mid \Sigma \vdash \mathit{False} & \text{if } (\Sigma = \Sigma' * x \mapsto \overline{e} \text{ and } \Pi \mid \Sigma \vdash x = \mathbf{nil}) \\
 & \text{or } (\Sigma = \Sigma' * P(x) * P(y) \text{ and } \Pi \mid \Sigma \vdash x = y \wedge x \neq \mathbf{nil}) \\
\Pi \mid \Sigma \vdash \mathit{Cell}(x_1) * \ldots * \mathit{Cell}(x_n) & \text{if } \Sigma = \Sigma' * P(y_1) * \ldots * P(y_n) \\
 & \quad \text{and } \Pi \mid \Sigma' \vdash x_1 = y_1 \wedge \ldots \wedge x_n = y_n \\
\multicolumn{2}{r}{\text{where } P(x) \equiv x \mapsto \overline{e} \text{ or } (P(x) \equiv \mathbf{trees}.x \oplus \mathcal{C}.\mathcal{D} \text{ and } \Pi \mid \Sigma' \vdash x \notin \mathcal{D})}
\end{array}
$$

Fig. 3. Syntactic Theorem Prover $\vdash$

state. Function $\mathsf{rearr}.x$ applies rewrite rules to every $\Pi \mid \Sigma$ until $x \mapsto \overline{e}$ occurs in Σ; otherwise it returns $\{\top\}$ when no rule can be applied.

Rewrite rules try to reveal a memory cell through equalities of variables and the unfolding of **trees** predicates:

$$\frac{\Pi \mid \Sigma \vdash x = y}{\Pi \mid \Sigma * y \mapsto \overline{e} \implies \Pi \mid \Sigma * x \mapsto \overline{e}}\,[\mathsf{Eq}]$$

$$\frac{\Pi \mid \Sigma \vdash x = y \wedge y \neq \mathbf{nil} \wedge y \notin \mathcal{D}}{\Pi \mid \Sigma * \mathbf{trees}.y \oplus \mathcal{C}.\mathcal{D} \implies \Pi \mid \Sigma * x \mapsto l', r', v' * \mathbf{trees}.l' \oplus r' \oplus \mathcal{C}.\mathcal{D}}\,[\mathsf{Unfold}]$$

where l', r', v' are fresh variables. The conditions for application of these rules require the ability to decide on the validity of certain predicates, denoted by $\vdash$. In figure 3 we present a small and simple syntactic theorem prover that in some circumstances allows to derive the validity of predicates like (in)equality of expressions, (non-)membership in a multiset, etc. *Notation: we use* $\equiv$ *for syntactic equality;* x, y, z, x_i, y_i *are variables in* $\mathsf{Var} \cup \mathsf{Var}'$.

Taking $\_^{\dagger}$ defined analogously as in the previous section, we define the intermediate semantics for an atomic command a as:

$$[\![a(x)]\!]_i \doteq a(x)^{\dagger} \circ \mathsf{rearr}.x$$

The semantics of compound statements follows the same pattern as the concrete semantics, leaving only the function $\mathsf{filter}.b$ to be defined. Recall that a program guard b is a disjunction of terms eqs that are conjunctions of (in)equalities eq. If $b \doteq eqs_1 \vee \ldots \vee eqs_n$ we define the function $\mathsf{filter}.b$ as:

$$\mathsf{filter}.b.S \doteq \{\top \mid \top \in S\} \cup \bigcup_{1 \leq i \leq n} \{\Pi \wedge eqs_i \mid \Sigma \mid \Pi \mid \Sigma \in S \text{ and } \Pi \mid \Sigma \nvdash \neg eqs_i\}$$

Rewriting rules defining rearr represent valid semantic implications. But although the algorithm for $\vdash$ is consistent, clearly it is not complete. Therefore, on some circustamces rearr will not be able to reveal the existence of a particular memory cell with the consequence that certain executions of intermediate semantics finish in $\{\top\}$, even when there is no memory violation according to the concrete semantics. To establish this, we need to define a relation between both semantics using a concretization function $\gamma : \mathcal{P}(\mathsf{SH} \cup \{\top\}) \to \mathcal{P}(\mathsf{States} \cup \{\top\})$:

$$\gamma.S \doteq \begin{cases} \{\top\} & \text{if } \top \in S \\ \{(s,h) \mid \exists\, \Pi \mid \Sigma \in S \cdot (s,h) \models \Pi \mid \Sigma\} & \text{otherwise} \end{cases}$$

Extending the semantics $\models$ for any predicate P of figure 3 in the obvious way, we can prove the following results.

Lemma 2. (Soundness of $\Longrightarrow$)

1. *If $\Pi \mid \Sigma \vdash P$, then $(s,h) \models \Pi \mid \Sigma$ implies $(s,h) \models P$ for all s, h.*
2. *If $\Pi \mid \Sigma \Longrightarrow \Pi' \mid \Sigma'$, then $(s,h) \models \Pi \mid \Sigma$ implies $(s,h) \models \Pi' \mid \Sigma'$ for all s, h.*

Theorem 3. *The intermediate semantics is a sound over-approximation of the concrete semantics: $[\![p]\!]_c.(\gamma.S) \subseteq \gamma.([\![p]\!]_i.S)$ for all $S \in \mathcal{P}(\mathsf{SH} \cup \{\top\})$.*

3 Abstract Semantics: The Analysis

The intermediate semantics is executable but has no mechanism to facilitate the calculation of loop invariants and ensure termination. Generally, the execution of a loop dereferencing a variable x generates states with a large number of formulæ which contain an arbitrary number of terms of the form $x' \mapsto l', r', v'$. For example, the execution of the intermediate semantics on an algorithm that iteratively uses variable p to run through a binary search tree (BST) on the left link searching for its lowest value, starting from the expected precondition $\{\mathbf{true} \mid \mathbf{trees}.\{\!|x|\!\}.\emptyset\}$, generates statements of the form:

$$\begin{array}{l} \{p = x \mid \mathbf{trees}.\{\!|x|\!\}.\emptyset, \\ \quad \mathbf{true} \mid x \mapsto p, r'_1, v'_1 * \mathbf{trees}.\{\!|p, r'_1|\!\}.\emptyset, \\ \quad \mathbf{true} \mid x \mapsto l'_1, r'_1, v'_1 * l'_1 \mapsto p, r'_2, v'_2 * \mathbf{trees}.\{\!|p, r'_1, r'_2|\!\}.\emptyset, \\ \quad \mathbf{true} \mid x \mapsto l'_1, r'_1, v'_1 * l'_1 \mapsto l'_2, r'_2, v'_2 * l'_2 \mapsto p, r'_3, v'_3 * \mathbf{trees}.\{\!|p, r'_1, r'_2, r'_3|\!\}.\emptyset, \ldots \end{array}$$

To handle this situation, we define an abstraction function $\mathsf{abs} : \mathsf{D}_a \to \mathsf{D}_a$, which aims at simplifying the formulæ of a state, replacing concrete information about the shape of the heap by a more abstract but still useful one. The use of such function is intended to facilitate the convergence of fixed-point computation which represents the semantics of a loop. Our abstract semantics $[\![\ ]\!]_a$ applies the function abs at the entry point and after each iteration of a loop. More precisely, the only change with respect to the intermediate semantics is:

$$[\![\mathbf{while}\ b\ \mathbf{do}\ p]\!]_a \doteq (\lambda S \cdot \mathsf{filter}.\neg b \circ (\mu S' \cdot \mathsf{abs}.(S \cup ([\![p]\!]_a \circ \mathsf{filter}.b).S'))) \circ \mathsf{abs}$$

$$\frac{\Sigma \doteq \Sigma' * \mathbf{trees}.\mathcal{C}.x \oplus \mathcal{D} * \mathbf{trees}.y \oplus \mathcal{E}.\emptyset \qquad \Pi \mid \Sigma \vdash x = y}{\Pi \mid \Sigma \implies \Pi \mid \Sigma' * \mathbf{trees}.\mathcal{C}.\mathcal{D} * \mathbf{trees}.\mathcal{E}.\emptyset} \text{[AbsTree1]}$$

$$\frac{\Pi \mid \Sigma \vdash x = y \qquad \Pi \mid \Sigma \vdash \mathit{Cell}(e) \vee e = \mathbf{nil} \quad \text{for all } e \Subset \mathcal{F}}{\Pi \mid \Sigma * \mathbf{trees}.\mathcal{C}.x \oplus \mathcal{D} * T.y \oplus \mathcal{E}.\mathcal{F} \implies \Pi \mid \Sigma * \mathbf{trees}.\mathcal{C} \uplus \mathcal{E}.\mathcal{D} \uplus \mathcal{F}} \text{[AbsTree2]}$$

$$\Pi \mid \Sigma * \mathbf{trees}.\mathcal{C}.\mathcal{D} \implies \Pi \mid \Sigma * \mathbf{trees}.\mathbf{nil} \ominus \mathcal{C}.\mathbf{nil} \ominus \mathcal{D} \text{ [AbsTree3]}$$

$$\frac{\Pi \mid \Sigma \vdash x = y}{\Pi \mid \Sigma * \mathbf{trees}.x \oplus \mathcal{C}.y \oplus \mathcal{D} \implies \Pi \mid \Sigma * \mathbf{trees}.\mathcal{C}.\mathcal{D}} \text{[AbsTree4]}$$

$$\frac{\Sigma \doteq \Sigma' * x \mapsto l, r, v * \mathbf{trees}.y \oplus \mathcal{C}.\emptyset \qquad \Pi \mid \Sigma \vdash l = y \wedge x \neq r}{\Pi \mid \Sigma \implies \Pi \mid \Sigma' * \mathbf{trees}.\{\!|x|\!\}.\{\!|r|\!\} * \mathbf{trees}.\mathcal{C}.\emptyset} \text{[AbsArrow1]}$$

$$\frac{\Sigma \doteq \Sigma' * x \mapsto l, r, v * T.y \oplus \mathcal{C}.\mathcal{D} \qquad \Pi \mid \Sigma \vdash l = y \wedge x \notin r \oplus \mathcal{D} \wedge y \notin \mathcal{D}}{\Pi \mid \Sigma \implies \Pi \mid \Sigma' * \mathbf{trees}.x \oplus \mathcal{C}.r \oplus \mathcal{D}} \text{[AbsArrow2]}$$

Fig. 4. Abstraction rules (first stage)

In this way, although the domain of abstract semantics remains $\mathcal{P}(\mathsf{SH} \cup \top)$, the semantics of a cycle is calculated over a subset of states which, as we will see in the following sections, ensure the convergence of fixed-point calculation.

The function abs is given by a set of rewriting rules. The more interesting ones are presented in Figure 4. *Notation: we use $T.\mathcal{C}.\mathcal{D}$ to denote either a term* $\mathbf{trees}.\mathcal{C}.\mathcal{D}$ *or a term* $x \mapsto l, r, v$ *with* $\mathcal{C} = \{\!|x|\!\}$ *and* $\mathcal{D} = \{\!|l, r|\!\}$. The rules AbsArrow abstract predicates 'points-to' into predicates **trees**, forgetting the number of nodes that form the tree-like structure. The rules are presented for the left link being completely analogous for the right one. The rules AbsTree1-2 combine trees forgetting intermediate points between them. The conditions of application prevent the formation of cycles within the structure thus preserving the tree-like characteristic. The rules AbsTrees3-4 remove entry and outlet points which do not provide relevant information about the heap.

Recalling the example above, if we apply rules AbsArrow1, AbsTree2 and AbsTree4 in the second iteration of the fixed point calculation, we obtain the invariant:

$$\{p = x \mid \mathbf{trees}.\{\!|x|\!\}.\emptyset, \quad \mathbf{true} \mid \mathbf{trees}.\{\!|x|\!\}.\{\!|p|\!\} * \mathbf{trees}.\{\!|p|\!\}.\emptyset\}$$

3.1 Relevance of Variables and Abstraction

We begin by defining some terminology. We say that two terms $T_1.\mathcal{C}.\mathcal{D}$ and $T_2.\mathcal{E}.\mathcal{F}$ of a symbolic heap form a *chain link* if there exists an $x \Subset \mathcal{D}$ such that $x \Subset \mathcal{E}$. A sequence of terms $T_1, T_2, \ldots, T_n$ is a *chain* if T_i and T_{i+1} form a chain link for all $1 \leq i < n$. The application of abstraction rules involves losing two kinds of information about the heap: first, specific information on a cell referenced by some variable; second, the information about the variable that defines

a link between two terms. For example both cases ocurr in rule AbsArrow1, for x and y respectively. If an abstracted variable x (or y) is dereferenced at a later point of execution, this data loss can result in the impossibility of continuing a non-trivial analysis. This is either because it is not possible to reveal the cell pointed to by x, since application conditions for rearr rules are not granted anymore; or because the trace of the variable y as the midpoint of the tree-like structure is lost.

In general, the shape analysis derived from Separation Logic apply abstraction rules when the variable defining a link is quantified, but a similar approach is not adequate for our case. On the one side, it could become too strong as we have discussed before. On the other hand, it could be too weak, resulting in unnecessarily precise formulæ and therefore larger invariants. Keeping track of a chain of two, three or more links do not seem a problem in the case of linear structures. But in treating multilinked structures, given the many possible combinations, the number of formulæ needed to describe this chain grows exponentially. Although from the standpoint of correctness this is not a problem, to keep reduced abstract states speeds up the analysis and enables better understanding of the obtained invariants and postconditions.

The application of our abstraction rules is relative to a *relevance level* assigned to each variable. The level of a variable at a certain point of execution depends on the kind of commands involving it, which are intended to be executed after this point.

The very simple static analysis relev is performed on a program, and returns a function $f : \mathsf{Var} \to \mathsf{Nat}_0$ which assigns a value to each variable, encoding the foreseen needed information for it. Considering a program as a semicolon separated sequence of commands, relev updates a function f initialized in 0 for each variable, according to:

$$
\begin{aligned}
\mathsf{relev}.f.(x := e; ps) &= (\mathsf{relev}.f.ps \mid x \to 0)\uparrow y \to 2 \quad \text{forall } y \in e \\
\mathsf{relev}.f.(x := y.i; ps) &= (\mathsf{relev}.f.ps \mid x \to 0)\uparrow y \to 3 \\
\mathsf{relev}.f.(x.i := e; ps) &= (\mathsf{relev}.f.ps\uparrow x \to 4)\uparrow y \to 2 \quad \text{forall } y \in e \\
\mathsf{relev}.f.(\mathbf{free}(x); ps) &= \mathsf{relev}.f.ps\uparrow x \to 3 \\
\mathsf{relev}.f.(\mathbf{new}(x); ps) &= \mathsf{relev}.f.ps \mid x \to 0 \\
\mathsf{relev}.f.(\mathbf{if}\ b\ \mathbf{then}\ ps_1 \mathbf{else}\ ps_2; ps) &= \\
(\mathsf{relev}.f.(ps_1 +\!\!+ ps)\ \max\ \mathsf{relev}.f.(ps_2 +\!\!+ ps))&\uparrow y \to 2 \quad \text{forall } y \in b \\
\mathsf{relev}.f.(\mathbf{while}\ b\ \mathbf{do}\ ps_1; ps) &= \\
(\mathsf{relev}.f.(ps_1 +\!\!+ ps)\ \max\ \mathsf{relev}.f.ps)&\uparrow y \to 2 \quad \text{forall } y \in b \\
\mathsf{relev}.f.\epsilon &= f
\end{aligned}
$$

Notation: $f \mid x \to n$ is the function update previously presented, ϵ is the empty sequence, $+\!\!+$ denotes concatenation, and operators $\uparrow$ and max *are defined as:*

$$
(f\uparrow y \to n).x \doteq \begin{cases} f.x & \text{if } x \neq y \text{ or } f.x \geq n \\ n & \text{otherwise} \end{cases} \qquad (f \max g).x \doteq \begin{cases} f.x & \text{if } f.x \geq g.x \\ g.x & \text{otherwise} \end{cases}
$$

Thus, the relevance level of x in $\Pi \mid \Sigma$ is given by the highest value according to f within its equivalence class induced by terms $x = y$ in Π. Level 1 is reserved for variables of precondition and level -1 for quantified variables.

$\Pi \wedge x = e \mid \Sigma \implies (\Pi \mid \Sigma)_{/x \leftarrow e}$ [EqElim] when relev computes for x a value < 1

$\Pi \wedge e_1 \neq e_2 \mid \Sigma \implies \Pi \mid \Sigma$ [NeqElim] when no variable with relevance level greater than 0 appears in e_1, e_2

$$\Pi \mid \Sigma * \mathbf{trees}.x' \oplus \mathcal{C}.\emptyset \implies \Pi \mid \Sigma * \mathbf{trees}.\mathcal{C}.\emptyset * \mathbf{true}\ [\mathsf{Gb1}]$$

$$\Pi \mid \Sigma * T.x' \oplus \mathcal{C}.\mathcal{D} \implies \Pi \mid \Sigma * \mathbf{junk/true}\ [\mathsf{Gb2}]$$

$$\Pi \mid \Sigma * \mathbf{trees}.\mathcal{C}.x' \oplus \mathcal{D} \implies \Pi \mid \Sigma * \mathbf{true}\ [\mathsf{Gb3}]$$

$$\Pi \mid \Sigma * \mathbf{trees}.\mathcal{C}.\mathcal{D} * \mathbf{trees}.\mathcal{C}.\mathcal{D} \implies \Pi \mid \Sigma\ [\mathsf{Gb4}]$$

$$\Pi \mid \Sigma * x' \mapsto y', r, v * T.y' \oplus \mathcal{C}.\emptyset \implies \Pi \mid \Sigma * x' \mapsto y', r, v * T.\mathcal{C}.\emptyset * \mathbf{true}\ [\mathsf{ChBrk1}]$$

$$\Pi \mid \Sigma * x' \mapsto y', r, v * T.y' \oplus \mathcal{C}.\mathcal{D} \implies \Pi \mid \Sigma * x' \mapsto y', r, v * \mathbf{junk/true}\ [\mathsf{ChBrk2}]$$

$$\Pi \mid \Sigma * \mathbf{trees}.\mathcal{C}.y' \oplus \mathcal{D} * T.y' \oplus \mathcal{E}.\mathcal{F} \implies \Pi \mid \Sigma * \mathbf{trees}.\mathcal{C}.y' \oplus \mathcal{D} * \mathbf{true}\ [\mathsf{ChBrk3}]$$

$$\Pi \wedge \mathbf{true} \mid \Sigma \implies \Pi \mid \Sigma\ [\mathsf{Zero1}] \qquad \Pi \mid \Sigma * \mathbf{emp} \implies \Pi \mid \Sigma\ [\mathsf{Zero2}]$$

$$\Pi \wedge eq \wedge eq \mid \Sigma \implies \Pi \wedge eq \mid \Sigma\ [\mathsf{Idemp1}]$$

$$\Pi \mid \Sigma * \mathbf{true} * \mathbf{true} \implies \Pi \mid \Sigma * \mathbf{true}\ [\mathsf{Idemp2}]$$

$$\Pi \mid \Sigma * \mathbf{junk} * \mathbf{junk/true} \implies \Pi \mid \Sigma * \mathbf{junk}\ [\mathsf{Junk}]$$

Fig. 5. Abstraction rules: second stage (top), third stage (bottom)

In our analysis, the relevance level required in each rule is parameterized, allowing us to adjust the abstraction performed. Possibly different requirements are impossed on variables being abstracted, depending on whether they define a link or they occur only as an address of a 'points-to' term. Abstracting variables with relevance level up to 2 in both cases, resulted in a safe choice to effectively carry out our experiments. Some algoritms allow higher limits, allowing more compact invariants and shorter execution times.

3.2 Quantified Variables and Termination

Quantified variables play a fundamental role in the completeness of the analysis induced by $[\![\ ]\!]_a$, since they can be used to generate an infinite set of formulæ as the image of abs. The way to generate these formulæ is by forming chains of terms T_i of arbitrary length, for which it is necessary to have an arbitrary large number of links formed with quantified variables. In circumstances where $img.\mathsf{abs}$ is infinite, the computation of the minimum fixed point that represents the semantics of a cycle cannot be carried out effectively.

However, abs consists of a number of rules that limit the number of quantified variables and ensures the finiteness of its image and therefore the convergence of fixed point calculation. The analysis is organized in three stages, each of which involve the exhaustive application of rewriting rules.

The first stage is defined by the abstraction rules previously presented (Fig. 4). If the limit imposed over the relevance level of variables to be abstracted is

greater than zero, the application of these rules thereby limit the length of the chains by removing a significant number of quantified variables, when it is possible to ensure the absence of cycles in the structure.

In a second stage, the top rules of Figure 5 are applied. *Notation: with* **junk**/**true** *we mean* **junk** *in the case that term T is a predicate 'points-to', and* **true** *if it is a predicate* **trees**. Essentially, these rules simplify the situations in which it is impossible to ensure a tree-like structure, and therefore the rules of the first stage do not apply. The rules ChBrk have the application condition $x', y' \notin \Sigma$ and prevent the formation of arbitrarily long chains by removing links formed with irrelevant quantified variables. Every Gb rule has the application condition $x' \notin \Pi \nmid \Sigma$. The rules Gb1-2 remove all chains that do not start with program variables. Rules Gb3-4 bound chains starting with a term $\mathbf{trees}.\mathcal{C}.\mathcal{D}$, either by eliminating terms containing quantified variables among its outlets, or by eliminating multiples ocurrences of a term, since they do not lead to an inconsistency only if $\mathcal{C} = \mathcal{D}$ and therefore $\mathbf{trees}.\mathcal{C}.\mathcal{D} = \mathbf{emp}$.

Finally, in the third stage the bottom rules of Figure 5 are applied. They collect all garbage in one predicate **junk** or **true**, and delimit the number of pure formulæ (along with rules EqElim and NeqElim of the previous stage). Furthermore, at this stage two or more formulæ differing only in the name of quantified variables are reduced to one instance; and all inconsistent symbolic heaps that can be detected sintactically (denoted $\Pi \nmid \Sigma \vdash \mathit{False}$ in fig. 3) are eliminated. As a consecuence, the number of chains that begin with a term $x \mapsto l, r, v$ (x a program variable) is bounded. In this way it is possible to prove:

Lemma 4. (Termination of abs**)**

1. *img.*abs *is finite.*
2. *The set of abstraction rules is strongly normalizing, i.e. every sequence of rewrites eventually terminates.*

The abstraction rules are not confluent, i.e. the obtained normal form after a terminating sequence of rewrites is not unique. The application order of the first stage rules is relevant as AbsArrow rules loose information that could be necessary to derive the application conditions of other rules. Our implementation obey the presented order. However, we do not observe significant differences when changing the application order in each stage.

The soundness of the rewriting rules of the previous section can be extended to include the abstraction rules:

Lemma 5. (Soundness of $\Longrightarrow$)

If $\Pi \nmid \Sigma \implies \Pi' \nmid \Sigma'$, then $(s,h) \models \Pi \nmid \Sigma$ implies $(s,h) \models \Pi' \nmid \Sigma'$ for all s, h

Previous results guarantee that the analysis given by the execution of the abstract semantics is sound and always terminates.

Theorem 6. *The abstract semantics is a sound over-approximation of the concrete semantics: $[\![p]\!]_c.(\gamma.S) \subseteq \gamma.([\![p]\!]_a.S)$ for all $S \in \mathcal{P}(\mathsf{SH} \cup \{\top\})$. Moreover, the algorithm defined by $[\![\ ]\!]_a$ is terminating.*

4 Managing Graphs

It is possible to use similar ideas to those underlying the predicate **trees** to describe general graph structures that includes cycles and sharing. The new predicate $\mathbf{graph}.\mathcal{C}.\mathcal{D}$ specifies with $\mathcal{C}$ the entry points of the graph, but unlike **trees** uses the multiset $\mathcal{D}$ to account for the dangling pointers that could point to nodes within the structure. Thus, these parameters resemble the ideas normally used to reason about graph algorithms: while $\mathcal{C}$ represents the entry points of paths to traverse, $\mathcal{D}$ realizes 'already visited' nodes. For a binary node graph, the formal semantics of **graph** is given by the least predicate satisfying:

$$\begin{aligned}\mathbf{graph}.\emptyset.\mathcal{D} &\doteq \mathbf{emp}\\ \mathbf{graph}.x \oplus \mathcal{C}.\mathcal{D} &\doteq ((x = \mathbf{nil} \vee x \Mapsto \mathcal{D}) \wedge \mathbf{graph}.\mathcal{C}.\mathcal{D}) \vee\\ &\quad ((x \neq \mathbf{nil} \wedge x \not\Mapsto \mathcal{D}) \wedge (\exists l, r, v \cdot x \mapsto l, r, v * \mathbf{graph}.l \oplus r \oplus \mathcal{C}.x \oplus \mathcal{D}))\end{aligned}$$

To adapt our analysis to deal with graphs it is necessary to modify several rewrite rules that define it, while others will remain intact. In the rearr phase, rule Unfold must account for the differences in the definition of the predicate:

$$\frac{\Pi \mid \Sigma \vdash y = x \wedge y \neq \mathbf{nil} \wedge y \not\Mapsto \mathcal{D}}{\Pi \mid \Sigma * \mathbf{graph}.y \oplus \mathcal{C}.\mathcal{D} \implies \Pi \mid \Sigma * x \mapsto l', r', v' * \mathbf{graph}.l' \oplus r' \oplus \mathcal{C}.x \oplus \mathcal{D}}\,[\mathsf{Unfold}]$$

Given the possible existence of cycles, the condition $y \not\Mapsto \mathcal{D}$ could not always be derived. When rearr fail over a symbolic heap S because of this, it is replaced by the set of symbolic heaps obtained by adding the hypothesis $y \not\Mapsto \mathcal{D} \vee y \Mapsto \mathcal{D}$. More precisely, S is replaced by the semantically equivalent set $\mathsf{filter}.b.S$, where $b \doteq (y \neq e_1 \wedge \ldots \wedge y \neq e_n) \vee y = e_1 \vee \ldots \vee y = e_n$ given $\mathcal{D} = \{\!|e_1, \ldots, e_n|\!\}$. Then rearr is relaunched on the new (richer) state.

The rules for the abs phase should take into account the possibility of cycles and sharing, plus the fact that a dangling pointer in $\mathcal{D}$ is not necessarily reachable from some entry point anymore. The most relevant rewriting rules are presented below:

$$\frac{\Sigma \doteq \Sigma' * x \mapsto l, r, v * \mathbf{graph}.y \oplus z \oplus \mathcal{C}.\mathcal{D} \qquad \Pi \mid \Sigma \vdash l = y \wedge r = z}{\Pi \mid \Sigma \implies \Pi \mid \Sigma' * \mathbf{graph}.x \oplus \mathcal{C}.x \ominus \mathcal{D}}\,[\mathsf{AbsGraph1}]$$

$$\frac{\Sigma \doteq \Sigma' * x \mapsto l, r, v * \mathbf{graph}.y \oplus \mathcal{C}.\mathcal{D} \quad \Pi \mid \Sigma \vdash l = y \quad \Pi \mid \Sigma \vdash r = x \vee r = \mathbf{nil}}{\Pi \mid \Sigma \implies \Pi \mid \Sigma' * \mathbf{graph}.x \oplus \mathcal{C}.x \ominus \mathcal{D}}\,[\mathsf{AbsGraph2}]$$

$$\Pi \mid \Sigma * \mathbf{graph}.\mathcal{C}.\mathcal{D} \implies \Pi \mid \Sigma * \mathbf{graph}.\mathbf{nil} \ominus \mathcal{C}.\mathbf{nil} \ominus \mathcal{D}\ [\mathsf{AbsGraph3}]$$

$$\frac{\Pi \mid \Sigma \vdash x = y}{\Pi \mid \Sigma * \mathbf{graph}.x \oplus \mathcal{C}.y \oplus \mathcal{D} \implies \Pi \mid \Sigma * \mathbf{graph}.\mathcal{C}.y \oplus \mathcal{D}}\,[\mathsf{AbsGraph4}]$$

$$\frac{\Sigma \doteq \Sigma' * x \mapsto l, r, v * y \mapsto \overline{e} * z \mapsto \overline{f} \qquad \Pi \mid \Sigma \vdash l = y \wedge r = z}{\Pi \mid \Sigma \implies \Pi \mid \Sigma' * x \mapsto l, r, v * \mathbf{graph}.\{\!|y, z|\!\}.y \ominus z \ominus \{\!|\overline{e}, \overline{f}|\!\}}\,[\mathsf{AbsArrow1}]$$

$$\frac{\Sigma \doteq \Sigma' * x \mapsto l, r, v * y \mapsto \overline{e} \qquad \Pi \mid \Sigma \vdash l = y \quad \Pi \mid \Sigma \vdash r = x \vee r = \mathbf{nil}}{\Pi \mid \Sigma \implies \Pi \mid \Sigma' * x \mapsto l, r, v * \mathbf{graph}.\{\!|y|\!\}.y \ominus \{\!|\overline{e}|\!\}}\,[\mathsf{AbsArrow2}]$$

Table 1. Results of the shape analysis over several examples

Algorithm	Precondition / Postcondition	Link	Arrow	Inv.	Iter.	Time
`min/max`	$\mathbf{true} \mid \mathbf{trees}.\{\!\{x\}\!\}.\emptyset$	2	4	2	2	0.003
	$x = \mathbf{nil} \mid \mathbf{emp}$					
	$x \neq \mathbf{nil} \mid \mathbf{trees}.\{\!\{x\}\!\}.\emptyset$					
`destroy`	$\mathbf{true} \mid \mathbf{trees}.\{\!\{x\}\!\}.\emptyset$	4	4	4	3	0.005
	$x = \mathbf{nil} \mid \mathbf{emp}$					
	$x \neq \mathbf{nil} \mid \mathbf{emp}$					
`search`	$\mathbf{true} \mid \mathbf{trees}.\{\!\{x\}\!\}.\emptyset$	2	4	4	3	0.006
	$x = \mathbf{nil} \mid \mathbf{emp}$					
	$x \neq \mathbf{nil} \mid \mathbf{trees}.\{\!\{x\}\!\}.\emptyset$					
`insert`	$\mathbf{true} \mid t \mapsto x', 0, 0 * \mathbf{trees}.\{\!\{x\}\!\}.\emptyset$	3	3	12	4	0.036
	$\mathbf{true} \mid t \mapsto x', 0, 0 * x' \mapsto \mathbf{nil}, \mathbf{nil}, x''$					
	$\mathbf{true} \mid t \mapsto x', 0, 0 * \mathbf{trees}.\{\!\{x\}\!\}.\emptyset$					
`toVine`	$\mathbf{true} \mid t \mapsto x', 0, 0 * \mathbf{trees}.\{\!\{x\}\!\}.\emptyset$	3	4	8	6	0.061
	$\mathbf{true} \mid t \mapsto \mathbf{nil}, 0, 0$					
	$\mathbf{true} \mid t \mapsto x', 0, 0 * x' \mapsto \mathbf{nil}, \mathbf{nil}, x''$					
	$\mathbf{true} \mid t \mapsto x', 0, 0 * \mathbf{trees}.\{\!\{x\}\!\}.\emptyset$					
`delete`	$\mathbf{true} \mid t \mapsto x', 0, 0 * \mathbf{trees}.\{\!\{x\}\!\}.\emptyset$	3	2	14	5	0.142
	$\mathbf{true} \mid t \mapsto \mathbf{nil}, 0, 0$					
	$\mathbf{true} \mid t \mapsto x', 0, 0 * \mathbf{trees}.\{\!\{x\}\!\}.\emptyset$					
`Schorr-Waite`	$r = x \mid \mathbf{trees}.\{\!\{x\}\!\}.\emptyset$	4	4	8	4	0.061
	$\mathbf{true} \mid \mathbf{emp}$					
	$x \neq \mathbf{nil} \mid \mathbf{trees}.\{\!\{x\}\!\}.\emptyset$					
	$r \neq \mathbf{nil} \mid \mathbf{trees}.\{\!\{r\}\!\}.\emptyset$					
`Schorr-Waite`	$r = x \mid \mathbf{graph}.\{\!\{x\}\!\}.\emptyset$	4	4	17	4	0.185
	$x \neq \mathbf{nil} \mid \mathbf{graph}.\{\!\{x\}\!\}.\emptyset$					
	$r \neq \mathbf{nil} \mid \mathbf{graph}.\{\!\{r\}\!\}.\emptyset$					
	$\mathbf{true} \mid \mathbf{emp}$					

5 Experimental Results

We implemented our analysis in Haskell, and conducted experiments on an Intel Core Duo 1.86Ghz with 2GB RAM. In table 1 experimental results are presented for a variety of iterative algorithms on binary search trees adapted from GNU LibAVL [2], and an adaptation of Schorr-Waite traversal from [32].

Our implementation applies an abstraction phase on the final state to compact the postcondition, treating every variable not occurring in the precondition as a quantified one. Columns Link and Arrow represent the limits imposed over relevance levels to abstract a variable that forms a chain link, and a variable that occurs as an address of a "points-to" term, respectively. The column Inv. contains the number of states making up the invariant of the main cycle and the column Iter. the number of iterations needed to reach the fixed point. The execution time is measured in seconds. In all cases the used memory did not exceed the default allocation of 132 KB.

The `insert` and `delete` algorithms are the cases with a significant Arrow limit. This is because they present a deep pointer manipulation after the main loop, and therefore require a sufficiently precise invariant.

In the verification of the Schorr-Waite traversal over graphs, the computed invariant is really concise, consisting of seventeen symbolic heaps whose spatial part is some of the formulæ $\mathbf{graph}.\{\!|r, p|\!\}.\emptyset$, $\mathbf{graph}.\{\!|p|\!\}.\emptyset$, $\mathbf{graph}.\{\!|r|\!\}.\emptyset$, or $\mathbf{emp}$, where r and p are the variables used to traverse de graph. This is a good example of the precision that our shape analysis is capable of achieving. Running the analysis applying the abstraction rules on quantified variables only, gives an invariant consisting of more than 180 formulæ.

We extended even more the analysis to deal with arrays of values, which enables the verification of Cheney's copying garbage collector algorithm from [30]. This extension is not presented here due to lack of space. The prototype implementation, code of examples and full results are available online [1].

6 Conclusions

Contributions. In this paper we introduced an abstract semantics that implicitly define an analysis able to automatically verify interesting properties about the shape of manipulated data structures, calculating loop invariants and postconditions. This semantics is an over-approximation of operational semantics on a standard memory model, hence, it could report false memory faults or leaks. Despite that, experiments show fitness between the computed abstract states and those expected according to the concrete semantics. It would be desirable to have a precise characterization of this relationship.

In order to define the analysis we have introduced novel linear predicates **trees** and **graph** to describe graph-like structures with multiple entry and outlet points. Good syntactic properties enable a simple caracterization of the abstraction phase. Preliminary experiments, based on variation of **tree** predicate and abstraction rules, are promising in verifying sorting and balancing invariants on BST trees and AVL trees.

The use of variable relevance level, although a very simple idea, introduces a significant improvement in the compactness of loop invariants, without impairing the necessary precision to obtain relevant postconditions. In some cases, a dramatic reduction in the number of formulæ making up an abstract state is achieved, making the entire process of analysis faster, and helping in the computation of intuitively understandable and correct invariants. Also it enables us to foresee a good behavior of the analysis on large-scale code. It would be interesting to extend the concept of Bi-Abduction of [7] to our domain to support incomplete specifications and procedure calls.

As a final consideration, by basing our analysis in [12] we inherit all advantages of Separation Logic. Our rewrite rules are valid implications whose semantic verification is quite simple, while symbolic execution rules derive directly from Hoare triples. Thus, our analysis is intuitive and its correctness easily verifiable.

Related Work. The main related works are [7,3,33] that derive from [12]. Proposed extensions implemented in SpaceInvader tool, aim at verifying real large-scale code, supporting only linear structures possibly combined in intrincate ways. Our work widen the domain of applicability of these methods by supporting non-linear structures. The predecessor work [4] introduces Smallfoot tool, based in a different kind of symbolic execution, although it supports binary trees. The tool reduces the verification of Hoare triples to logical implications between symbolic heaps but it does not compute loop invariants. The usual tree predicates seems adequated to verified recursive programs. In [25,15] the analysis presented are very similar to the ones in [12] both, in their technical aspects and their limitations.

Xisa tool [11] also uses Separation Logic to describe abstract states, but it uses generalized inductive predicates in the form of structure invariant checkers supplied by users. The fold and unfold of predicates guide automatic strategies for materialization (rearr in this work) and abstraction. Extension of [10] adds expressive power to specify certain relationships among summarized regions of heaps, as ordering invariants, back and cross pointers, etc, and a case study of item insertion in a red-black Tree is presented. The lack of examples impair the assessment of complexity this tool can handle.

Verifast tool [18] is also based in Separation Logic, allowing the verification of richly specified Hoare triples using inductive predicates for structures and pure recursive functions over those datatypes. It generates verification conditions discharged by a SMT Solver. No abstraction mechanism is provided, but loop invariants and instruction guiding fold/unfold of predicates must be manually annotated. It is able to verify a recursive implementation of binary trees library and the composite pattern (with its underlying graph structure) [19].

PALE system [24,20] allows the verification of programs specified with assertions in Weak Second-order Monadic Logic of Graph Types. Loops must be annotated with invariants and verification conditions are discharged in MONA tool [17]. The extension [14] enables a more efficient verification of algorithms on tree-shaped structures.

The analysis of [21] uses shape graphs and grammar annotations to specify non-cyclic data structures, discovering automatically descriptions for structures occurring at intermediate execution points. The analysis verify Schorr-Waite traversal and disposal on trees, and the construction of a binomial heap.

The parametric shape analysis of [28,5] based on 3-valued logic, together with their implementation TVLA [22], is the most general, powerful and used framework in the verification of shape properties on programs that involve complex manipulations on dynamic structures. This framework should be instantiated for each particular case through user defined intrumentation predicates that specify the type of supported structures and forms of abstraction. The instance presented in [23] allows the verification of partial correctness (as our analysis) and also termination of the Schorr-Waite tree traversal. Our proposal is less ambitious since our analysis is specialized in trees and graphs. However our abstract domains allow for high efficiency and precision in a wide class of algorithms, and the analysis support local reasoning which has shown great potential to scale on real large-scale programs as demonstrated in [7,33].

Acknowledgements. We are grateful to Dino Distefano for his encouragement to write this article, to Miguel Pagano for many discussions, and to the anonymous reviewers for their helpful comments for improving this work.

References

1. Details of experiments, `http://cs.famaf.unc.edu.ar/~renato/seplogic.html`
2. GNU LibAVL, `http://www.stanford.edu/~blp/avl/`
3. Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P.W., Yang, H.: Shape analysis for composite data structures. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 178–192. Springer, Heidelberg (2007)
4. Berdine, J., Calcagno, C., O'Hearn, P.W.: Symbolic execution with separation logic. In: Yi, K. (ed.) APLAS 2005. LNCS, vol. 3780, pp. 52–68. Springer, Heidelberg (2005)
5. Bogudlov, I., Lev-Ami, T., Reps, T.W., Sagiv, M.: Revamping TVLA: Making parametric shape analysis competitive. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 221–225. Springer, Heidelberg (2007)
6. Calcagno, C., Distefano, D., O'Hearn, P.W., Yang, H.: Beyond reachability: Shape abstraction in the presence of pointer arithmetic. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, pp. 182–203. Springer, Heidelberg (2006)
7. Calcagno, C., Distefano, D., O'Hearn, P.W., Yang, H.: Compositional shape analysis by means of bi-abduction. In: Shao, Z., Pierce, B.C. (eds.) ACM SIGPLAN-SIGACT 2009 Symposium on Principles of Programming Languages, pp. 289–300. ACM, New York (2009)
8. Calcagno, C., Distefano, D., Vafeiadis, V.: Bi-abductive resource invariant synthesis. In: Hu, Z. (ed.) APLAS 2009. LNCS, vol. 5904, pp. 259–274. Springer, Heidelberg (2009)
9. Calcagno, C., Parkinson, M.J., Vafeiadis, V.: Modular safety checking for fine-grained concurrency. In: Riis Nielson, H., Filé, G. (eds.) SAS 2007. LNCS, vol. 4634, pp. 233–248. Springer, Heidelberg (2007)
10. Chang, B.-Y.E., Rival, X.: Relational inductive shape analysis. In: ACM SIGPLAN-SIGACT 2008 Symposium on Principles of Programming Languages, pp. 247–260. ACM, New York (2008)
11. Chang, B.-Y.E., Rival, X., Necula, G.C.: Shape analysis with structural invariant checkers. In: Nielson, H.R., Filé, G. (eds.) SAS 2007. LNCS, vol. 4634, pp. 384–401. Springer, Heidelberg (2007)
12. Distefano, D., O'Hearn, P.W., Yang, H.: A local shape analysis based on separation logic. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 287–302. Springer, Heidelberg (2006)
13. Distefano, D., Parkinson, M.J.: jStar: towards practical verification for java. In: Harris, G.E. (ed.) ACM SIGPLAN 2008 Conference on Object-Oriented Programming, Systems, Languages, and Applications, pp. 213–226. ACM, New York (2008)
14. Elgaard, J., Møller, A., Schwartzbach, M.I.: Compile-time debugging of C programs working on trees. In: Smolka, G. (ed.) ESOP 2000. LNCS, vol. 1782, pp. 182–194. Springer, Heidelberg (2000)
15. Gotsman, A., Berdine, J., Cook, B.: Interprocedural shape analysis with separated heap abstractions. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, pp. 240–260. Springer, Heidelberg (2006)

16. Gotsman, A., Berdine, J., Cook, B., Sagiv, M.: Thread-modular shape analysis. In: Ferrante, J., McKinley, K.S. (eds.) ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, pp. 266–277. ACM, New York (2007)
17. Henriksen, J.G., Jensen, J.L., Jørgensen, M.E., Klarlund, N., Paige, R., Rauhe, T., Sandholm, A.: Mona: Monadic second-order logic in practice. In: Brinksma, E., Steffen, B., Cleaveland, W.R., Larsen, K.G., Margaria, T. (eds.) TACAS 1995. LNCS, vol. 1019, pp. 89–110. Springer, Heidelberg (1995)
18. Jacobs, B., Piessens, F.: The verifast program verifier. Technical Report CW-520, Department of Computer Science, Katholieke Universiteit Leuven, Belgium (August. 2008)
19. Jacobs, B., Smans, J., Piessens, F.: Verifying the composite pattern using separation logic. In: SAVCBS Composite Pattern Challenge Track (2008)
20. Jensen, J.L., Jørgensen, M.E., Schwartzbach, M.I., Klarlund, N.: Automatic verification of pointer programs using monadic second-order logic. In: ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation, pp. 226–236. ACM, New York (1997)
21. Lee, O., Yang, H., Yi, K.: Automatic verification of pointer programs using grammar-based shape analysis. In: Sagiv, M. (ed.) ESOP 2005. LNCS, vol. 3444, pp. 124–140. Springer, Heidelberg (2005)
22. Lev-Ami, T., Sagiv, S.: TVLA: A system for implementing static analyses. In: Palsberg, J. (ed.) SAS 2000. LNCS, vol. 1824, pp. 280–301. Springer, Heidelberg (2000)
23. Loginov, A., Reps, T.W., Sagiv, M.: Automated verification of the deutsch-schorr-waite tree-traversal algorithm. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, pp. 261–279. Springer, Heidelberg (2006)
24. Møller, A., Schwartzbach, M.I.: The pointer assertion logic engine. In: ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation. ACM, New York (2001)
25. Nguyen, H.H., David, C., Qin, S., Chin, W.-N.: Automated verification of shape and size properties via separation logic. In: Cook, B., Podelski, A. (eds.) VMCAI 2007. LNCS, vol. 4349, pp. 251–266. Springer, Heidelberg (2007)
26. O'Hearn, P.W., Reynolds, J.C., Yang, H.: Local reasoning about programs that alter data structures. In: Fribourg, L. (ed.) CSL 2001. LNCS, vol. 2142, pp. 1–19. Springer, Heidelberg (2001)
27. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: 17th IEEE Symposium on Logic in Computer Science, pp. 55–74. IEEE Computer Society Press, Los Alamitos (2002)
28. Sagiv, S., Reps, T.W., Wilhelm, R.: Parametric shape analysis via 3-valued logic. ACM Trans. Program. Lang. Syst. 24(3), 217–298 (2002)
29. Smans, J., Jacobs, B., Piessens, F.: Implicit dynamic frames: Combining dynamic frames and separation logic. In: Drossopoulou, S. (ed.) ECOOP 2009. LNCS, vol. 5653, pp. 148–172. Springer, Heidelberg (2009)
30. Torp-Smith, N., Birkedal, L., Reynolds, J.C.: Local reasoning about a copying garbage collector. ACM Trans. Program. Lang. Syst. 30(4) (2008)
31. Villard, J., Lozes, É., Calcagno, C.: Proving copyless message passing. In: Hu, Z. (ed.) APLAS 2009. LNCS, vol. 5904, pp. 194–209. Springer, Heidelberg (2009)
32. Yang, H.: Local reasoning for stateful programs. PhD thesis, Champaign, IL, USA, Adviser-Uday S. Reddy (2001)
33. Yang, H., Lee, O., Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P.W.: Scalable shape analysis for systems code. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 385–398. Springer, Heidelberg (2008)