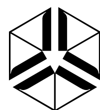


The United Nations University

**UNU-IIST**

**International Institute for Software Technology**

# Abstract Object Graphs for Program Verification

**Yifeng Chen and J. W. Sanders**

**May 2009**

**UNU-IIST Report No. 413**

R

# UNU-IIST and UNU-IIST Reports

UNU-IIST (United Nations University International Institute for Software Technology) is a Research and Training Centre of the United Nations University (UNU). It is based in Macao, and was founded in 1991. It started operations in July 1992. UNU-IIST is jointly funded by the government of Macao and the governments of the People's Republic of China and Portugal through a contribution to the UNU Endowment Fund. As well as providing two-thirds of the endowment fund, the Macao authorities also supply UNU-IIST with its office premises and furniture and subsidise fellow accommodation.

The mission of UNU-IIST is to assist developing countries in the application and development of software technology.

UNU-IIST contributes through its programmatic activities:

1. Advanced development projects, in which software techniques supported by tools are applied,
2. Research projects, in which new techniques for software development are investigated,
3. Curriculum development projects, in which courses of software technology for universities in developing countries are developed,
4. University development projects, which complement the curriculum development projects by aiming to strengthen all aspects of computer science teaching in universities in developing countries,
5. Schools and Courses, which typically teach advanced software development techniques,
6. Events, in which conferences and workshops are organised or supported by UNU-IIST, and
7. Dissemination, in which UNU-IIST regularly distributes to developing countries information on international progress of software technology.

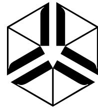
Fellows, who are young scientists and engineers from developing countries, are invited to actively participate in all these projects. By doing the projects they are trained.

At present, the technical focus of UNU-IIST is on formal methods for software development. UNU-IIST is an internationally recognised center in the area of formal methods. However, no software technique is universally applicable. We are prepared to choose complementary techniques for our projects, if necessary.

UNU-IIST produces a report series. Reports are either Research R, Technical T, Compendia C or Administrative A. They are records of UNU-IIST activities and research and development achievements. Many of the reports are also published in conference proceedings and journals.

Please write to UNU-IIST at P.O. Box 3058, Macao or visit UNU-IIST's home page: http://www.iist.unu.edu, if you would like to know more about UNU-IIST and its report series.

G. M. Reed, Director




**UNU-IIST**

**International Institute for Software Technology**

P.O. Box 3058
Macao

# Abstract Object Graphs for Program Verification

## Yifeng Chen and J. W. Sanders

### Abstract

A set-theoretic formalism, *AOG*, is introduced to support automated verification of pointer programs. *AOG* targets pointer reasoning for source programs before compilation (*i.e.* before removal of field names). Pointer structures are represented as object graphs instead of heaps. Each property in *AOG* is a relation between object graphs and name assignments of program variables, and specifications result from composing properties. *AOG* extends Separation Logic's compositions of address-disjoint separating conjunction to more restrictive compositions with different disjointness conditions; the extension is shown to be strict when fixpoints are present. A composition that is a "unique decomposition" decomposes any given graph uniquely into two parts. An example is the separation between the non-garbage and garbage parts of memory. Although *AOG* is in general undecidable, it is used to define the semantics of specific decidable logics that support automated program verification of specific topologies of pointer structure. One logic studied in this paper describes pointer variables located on multiple parallel linked lists. The logic contains quantifiers and fixpoints but is nonetheless decidable; it is applied to the example of in-place list reversal for automated verification. The Schorr-Waite marking algorithm is also considered. The technique of unique decomposition is particularly useful for establishing laws of such logics.

**Yifeng Chen** is a Research Professor at the HCST Key Lab at the School of EECS, Peking University, China. Previously he spent several years in the U.K., as Senior Lecturer at the University of Durham and Lecturer at the University of Leicester, after completing a *DPhil.* at the University of Oxford. His interests include imperative, parallel and object-oriented programming languages, including design, translation, static analysis, semantics, specications and their support for decentralised software development.

**Jeff Sanders** is Principal Research Fellow at UNU-IIST. His interests lie largely in Formal Methods.

This work was supported by the Macao Science and Technology Development Fund, under the PEARL project, grant number 041/2007/A3.

**Copyright © 2009** by UNU-IIST

# Contents

**1 Introduction** **1**

**2 Abstract Object Graphs** **2**
2.1 Definitions . . . 3
2.2 Simple examples . . . 4
2.3 Useful compositions . . . 5
2.4 List, parallel lists and reachability . . . 6
2.5 Unique decomposition . . . 8

**3 Logic-Based Automated Verification I: List Reversal** **9**
3.1 Logic for parallel lists . . . 9
3.2 Normal form reduction . . . 12
3.3 Automated program verification . . . 12
3.4 Automated verification of list reversal . . . 14

**4 Automated Verification II: Schorr-Waite Graph Marking** **16**

**5 Related work** **17**
5.1 Models . . . 18
5.2 Formalisms . . . 19
5.3 Logics and decidability . . . 19

**6 Conclusions** **20**

**A Proof outlines** **22**

# 1 Introduction

In the standard approach of Formal Methods, in order to achieve accountable programs the programmer is expected to specify and verify code. Even aided by automation, for example in the context of Hoare's verifying compiler [19], that is a challenging task requiring expertise not normally attributed to programmers. Is it inevitable for accountable programs?

In this paper an alternative approach is considered, in which the programmer is expected to know enough about the properties of the program to be able to issue (meta-level) guidance to a smart type checker that consequently uses library routines to check properties of the program. That general idea has been considered by Chen [11]. Here we concentrate on the case of programs that act on mutable data structures, and support the programmer by considering the design of useful shape-related properties for which there is a decision procedure that can as a result be incorporated in a smart type checker performing static analysis.

Suppose the programmer is occupied with a program involving pointers — say that it is to perform in-place list reversal. Some of the most common errors in pointer programs are related to incorrectly ordered pointer assignments; yet common type systems do not detect such errors. However, if the programmer can identify the overall topology of the pointer structures (of which the programmer is normally aware), the compiler will be able to perform in-depth verification to identify or confirm the absence of such errors.

Now, in this case the programmer is fully aware that all pointer program variables are located on parallel lists and should be free of loops and aliasing. That knowledge is not exploited by the standard C or Java type systems; but it could be. The programmer seeks confirmation (that pointer variables are indeed constrained in parallel lists and free of loops and aliasing) and also that no objects are lost as a result of pointer assignments. If those properties are formalised as annotated invariants, the standard two-line program for in-place list reversal becomes

$$
\begin{array}{l}
\texttt{inv ParLists}(a)\ \texttt{in} \\
\quad y := \mathrm{null};\ z := \mathrm{null}; \\
\quad \texttt{inv NoMemoryLeak in} \\
\quad\quad \texttt{while}\ (x \neq \mathrm{null})\ \texttt{do}\ \ (z := x.a;\ x.a := y;\ y := x;\ x := z).
\end{array}
$$

In real applications, pointer structures have diversified overall topologies such as parallel lists, rings, trees and even general graphs. With some assistance from the program to identify the overall topology, a compiler can perform much more precise analysis. Naturally the applications programmer is not expected to write the routines invoked by the compiler. That is the task of the systems programmer who in turn is not expected to establish decision procedures for properties of interest. That is our task. The present paper makes a start.

What support is already available for the programmer, and what more might be expected? Currently the most popular tool for reasoning about pointers is Separation Logic (SL) [27, 24]. At the level of explicit memory locations, spatial conjunction provides an expressive power equivalent to a second-order

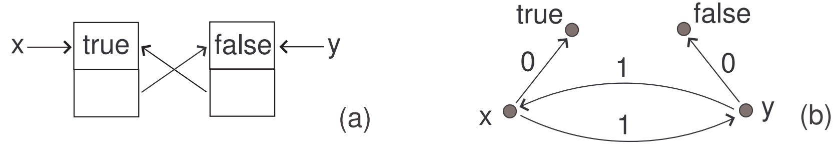
logic, as shown by Kuncak and Rinard [20]; indeed there it is shown that the addition of (unrestricted) spatial conjunction to a decidable logic can lead to an undecidable logic. So automating support for the programmer to reason about pointers using SL must be done carefully. One success, due to Calcagno, Yang and O'Hearn [5], is the decidability of a quantifier-free language, containing separating conjunction and its adjoint, for expressing shape of pointer structures (without properties of data). Another is a custom-crafted logic, due to Rakamarić, Bingham and Hu [26], with a decision procedure for verification of heap-manipulation programs (for further work, see the literature review in [26]); the kinds of property decided include reachability (a node reachable from the root initially is also reachable from the root on termination — referred to above as `NoMemoryLeak`), cyclicity, acyclicity and double-linkedness of a list. Further work is mentioned in Section 5. We conclude that there is scope for the design of logics that at once express properties of interest concerning shape, and are decidable.

The approach taken here is to consider a range of such spatial logics. A set-theoretic formalism, *AOG*, (for *Abstract Object Graphs*) is given in which pointer structures are represented, like several formalisms studied by the denotational and algebraic-semantic community (for details, see Section 5) as object graphs (with the presence of pre-compilation field names). Unlike those formalisms that directly manipulate individual graphs, *AOG* handles *sets* of object graphs. That facilitates a higher level of abstraction by constraining only certain areas of the pointer structure (allowing others to remain flexible). *AOG* contains 'properties' and 'constructs'. Each property describes a set of object graphs (under certain node-name assignment to variables). Constructs combine properties to generate more sophisticated properties, and include the fixpoint operator to express transitively-closed compositions. Two examples provided to illustrate its application: in-place list reversal (as above), and the Schorr-Waite marking algorithm. Both are standards and so facilitate comparison of our approach with that of others.

## 2 Abstract Object Graphs

A graph is a finite set of edges, each a triple containing a source name, a label name and a target name. An *object graph* is a graph whose nodes represent objects and whose directed edges (also called *labels*) represent fields. The label of an edge in every object is unique, reflecting the fact that the object stored at each field is unique.

Such graph-based representation reflects the states taken by higher-level OO languages such as Java *before compilation*. The heap representation of SL, on the other hand, reflects the pointer structures *after compilation*. An object is stored in a heap using pointer arithmetic: the first field is stored at an address $x$ and other fields are stored following at $x+1$, $x+2$, $\cdots$. The figure below (from our earlier paper [10]) compares the object graph and heap of two mutually referenced objects $x$ and $y$. Heap representation naturally includes pointer arithmetic, although many decidable SL fragments are free of pointer arithmetic (*i.e.* by assuming that each object has exactly two values and only the starting address is accessible); see for example those due to Calcagno, Gardner and Hague [4] and Calcagno, Yang and O'Hearn [5]. Arithmetic relations between numeric labels can always be added to object graphs.




Each *AOG* property is a relation between name assignments of the pointer variables (*i.e.* each is a mapping from variables to node names) and object graphs. This is very similar to SL, whose formulae correspond essentially to relations between name assignments (*i.e.* stacks) and heaps.

## 2.1 Definitions

Let $\mathbf{N} = \{a, b, c, a_1, \cdots\}$ be a countably infinite set of *names* and $\mathbf{X} = \{x, y, z, x_1, \cdots\}$ be a countably infinite set (disjoint from $\mathbf{N}$) of *variables*. Let the symbols $u, v, w, u_1, \cdots$ denote *atoms*, each either a name or a variable, and $\mathbf{A} := \mathbf{N} \cup \mathbf{X}$ be the set of all atoms. An *edge* is a triple $(a, b, c)$ of names where $a$ is called the source, $b$ the label and $c$ the target. An *object graph*, denoted $G, H, G_1, \cdots$, is a set of edges such that for all $(a, b, c)$ and $(a, b, d)$ in the set, we have $c = d$. Let $\mathbf{G}$ denote the set of all object graphs. An (atom) assignment function $\varepsilon : \mathbf{A} \to \mathbf{N}$ maps variables to names, fixing names: $\varepsilon(a) = a$ for every $a : \mathbf{N}$.

A *property* (denoted $P, Q, P_1, \cdots$) is a binary relation between an assignment and an object graph, while a *composition* (denoted $r, s, r_1, \cdots$) corresponds to a triadic relation between an assignment and two object graphs.

Let $\top$ denote the largest property: $\top(\varepsilon, G)$ is true for every $\varepsilon : \mathbf{A} \to \mathbf{N}$ and $G \in \mathbf{G}$. Let $u = v$ denote variable equality: $(u = v)(\varepsilon, G)$ iff $\varepsilon(u) = \varepsilon(v)$. For example, $a = a$ equals $\top$. Let $u \xrightarrow{v} w$ denote the edge property: $(u \xrightarrow{v} w)(\varepsilon, G)$ iff $G = \{(\varepsilon(u), \varepsilon(v), \varepsilon(w))\}$. The empty-graph property $\varnothing(\varepsilon, G)$ is true iff $G = \{\}$. If $P$ and $Q$ are properties and $r$ is a composition, the following constructs exhaust all properties:

$$\neg P \mid P \vee Q \mid P\; r\; Q \mid P /_r\, Q \mid P \backslash_r Q \mid \exists x \cdot P \mid P \propto_r Q\,.$$

The complement property $\neg P(\varepsilon, G)$ is true iff $P(\varepsilon, G)$ is not true, and $(P \vee Q)(\varepsilon, G)$ is true iff either $P(\varepsilon, G)$ or $Q(\varepsilon, G)$ is true. The merge operator $(P\; r\; Q)(\varepsilon, G)$ is true iff there exist $G_1, G_2 \in \mathbf{G}$ such that $G = G_1 \cup G_2$ and $P(\varepsilon, G_1)$, $Q(\varepsilon, G_2)$ and $r(\varepsilon, G_1, G_2)$ are true. It merges the $r$-related graphs from $P$ and $Q$ as union collection of edges under the same assignment. The left weak inverse of $r$, characterised with a Galois connection, is the weakest property $X$ such that $X\; r\; Q \subseteq P$. The right weak inverse of $r$ is the weakest $X$ such that $Q\; r\; X \subseteq P$. The existential quantifier $(\exists x \cdot P)(\varepsilon, G)$ is true iff there exist $a \in \mathbf{N}$ and $\varepsilon' = \varepsilon \dagger \{x \mapsto a\}$ such that $P(\varepsilon', G)$ is true (where $\varepsilon \dagger \{x \mapsto a\}$ overwrites $\varepsilon$ to take the value $a$ at $x$). The fixpoint recursion $P \propto_r Q$ is the transitive closure: $P \propto_r Q := \bigvee_k P \propto_r^k Q$ where $P \propto_r^0 Q := P$ and $P \propto_r^{k+1} Q := (P \propto_r^k Q)\; r\; Q$.

To ensure that any composition of two properties is still a property, every relation must imply the

largest edge disjointness composition $*$ where $*(\varepsilon, G, H)$ is true iff there do not exist $(a,b,c) \in G$ and $(a,b,d) \in H$ with the same source and label. Compositions can be constructed as follows:

$$P \times Q \mid \overline{r} \mid r \cup s \mid r|x$$

where the Cartesian product $(P \times Q)(\varepsilon, G, H)$ is true iff $P(\varepsilon, G)$, $Q(\varepsilon, H)$ and $*(\varepsilon, G, H)$ are true. The complement composition $\overline{r}(\varepsilon, G, H)$ is true iff $r(\varepsilon, G, H)$ is not true but $*(\varepsilon, G, H)$ is true. The union composition $(r \cup s)(\varepsilon, G, H)$ is true iff either $r(\varepsilon, G, H)$ or $s(\varepsilon, G, H)$ is true. The hiding $(r|x)(\varepsilon, G, H)$ is true iff there exist $a \in \mathbf{N}$ and $\varepsilon' = \varepsilon \dagger \{x \mapsto a\}$ such that $r(\varepsilon', G, H)$ is true.

The following proposition guarantees the closure of the space of properties (*i.e.* the soundness of *AOG*).

**Proposition 1** *Every composition r in AOG is a sub-relation of $*$, and every merge P r Q of properties P and Q with composition r is again a property.*

## 2.2 Simple examples

Let $\perp := \neg\top$ be the false property. We shall use $P \wedge Q := \neg(\neg P \vee \neg Q)$ and $r \cap s := \overline{\overline{r} \cup \overline{s}}$ to represent conjunction and intersection respectively and let $[r]$ denote the property $(\top\ r\ \top)$. Let $P^2$ denote $P \times P$. A (syntactical) graph is the merge of a finite number of edge properties. For example, the property $(a_1 \xrightarrow{b_1} c_1) * (a_2 \xrightarrow{b_2} c_2)$ allows only a graph containing exactly the two edges. However, the merge $(a \xrightarrow{b} c) * (a \xrightarrow{b} d)$ is equal to $\perp$, since the two sides have edges sharing the same source and label.

Let $P^\top := P * \top$ denote the arbitrary extension of $P$. For example, the property $(a_1 \xrightarrow{b_1} c_1)^\top \wedge (a_2 \xrightarrow{b_2} c_2)^\top$ allows just graphs containing at least the two edges. We use shorthand $(x \xrightarrow{\bullet} y) := \exists z \cdot (x \xrightarrow{z} y)$ to denote an edge with source $x$ and target $y$, $cyc := \exists x \cdot (x \xrightarrow{\bullet} x)$ to denote a cyclic edge, and $acyc := (\bullet \xrightarrow{\bullet} \bullet) \wedge \neg cyc$ to denote an acyclic edge (with the obvious extension of bullets to sources and targets!). Let $acyc(a)$ denote an acyclic edge with label $a$. A name is a *node* if it is either a source or a target $node(v) := (v \xrightarrow{\bullet} \bullet)^\top \vee (\bullet \xrightarrow{\bullet} v)^\top$.

The operators of *AOG* satisfy various algebraic laws, most of which can be found in [9]. For example, the properties $\top$, $\perp$, $\vee$, $\wedge$, $\neg$ and $\exists$ satisfy the formation of a Boolean complete lattice and the laws of predicate calculus; so do the operators $*$, $\perp^2$, $\cup$, $\cap$, $\overline{(\ )}$ and $|$. The top composition is $* = \top^2$ which relation contains all other compositions. The bottom composition is $\overline{*} = \perp \times P = P \times \perp$. The restrictions imposed by the properties $P$ and $Q$ in a composition $P\ r\ Q$ can be integrated into the composition: $P\ r\ Q = [r \cap (P \times Q)]$ as a Cartesian product. Compositions distribute over set union: $P\ (r \cup s)\ Q = (P\ r\ Q) \vee (P\ s\ Q)$ and hiding: $[r|x] = \exists x \cdot [r]$. Both $P$ and $(P \propto_r Q)\ r\ Q$ are included (as sub-relations) in the fixpoint $P \propto_r Q$. Other distributivity laws of the Cartesian product are listed below:

**Law 1** (1) $(P_1 \vee P_2) \times Q \;=\; (P_1 \times Q) \cup (P_2 \times Q)$
(2) $P \times (Q_1 \vee Q_2) \;=\; (P \times Q_1) \cup (P \times Q_2)$
(3) $(P_1 \times Q_1) \cap (P_2 \times Q_2) \;=\; (P_1 \wedge P_2) \times (Q_1 \wedge Q_2)$
(4) $\overline{P \times Q} \;=\; (\neg P \times \top) \cup (\top \times \neg Q)$
(5) $(\exists x \cdot P) \times (\exists x \cdot Q) \;=\; (P \times \exists x \cdot Q)|x \;=\; (\exists x \cdot P \times Q)|x$

*AOG* extends SL without pointer arithmetic and is thus undecidable [4]. In Sections 3 and 4 we use it to define specific logics that support automated verification of certain pointer programs.

## 2.3 Useful compositions

*AOG* properties can be combined into more sophisticated properties using various compositions. A composition is a triadic relation among name assignments, left properties and right properties. The resulting property relates name assignments with the merged (*i.e.* by set union) object graphs from the left and right properties respectively (under the same name assignment). Notice that the set union of two object graphs may not necessarily be an object graph. Thus certain consistency checks must be performed before the merge. In graph-based representation, SL's address disjointness corresponds to the condition that requires the object graphs not to contain edges sharing the same source and label. Besides SL's spatial conjunction, there exist many different useful compositions that satisfy this condition. When fixpoints are present, having multiple different compositions is strictly more expressive than having only spatial conjunction.

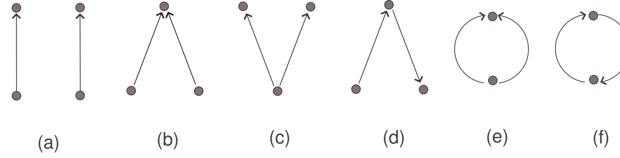
Novel user-defined compositions, other than separating conjunction, can be created in *AOG* using operators like union, complement and quantifiers. In some sense, *AOG* is like a 'dynamic SL' similar to dynamic modal logic and the technique of parallel-by-merge from Hoare and He's UTP [17]. We intend to create various binary relations between graphs for two reasons. The first is to build up the level of abstraction so that hiding (by quantifiers) is applied at the right places. The second is to create advanced operators that hold desirable properties and lead to better compositionality of reasoning. For example, two graphs are related by $\olessthan := \neg((x \stackrel{\bullet}{\rightarrow} \bullet)^\top \times (\bullet \stackrel{\bullet}{\rightarrow} x)^\top |x)$ if the first graph has no source as a target of the second. The following table lists some useful compositions.

| $\neg(? \times ?\,\|x)$ | $(x \stackrel{\bullet}{\rightarrow} \bullet)^\top$ | $(\bullet \stackrel{\bullet}{\rightarrow} x)^\top$ | $\neg((x \stackrel{\bullet}{\rightarrow} \bullet)^\top)$ | $\neg((\bullet \stackrel{\bullet}{\rightarrow} x)^\top)$ |
|---|---|---|---|---|
| $(x \stackrel{\bullet}{\rightarrow} \bullet)^\top$ | $\ovee$ | $\olessthan$ | $\overrightarrow{\triangledown}$ | $\overrightarrow{\triangleleft}$ |
| $(\bullet \stackrel{\bullet}{\rightarrow} x)^\top$ | $\ogreaterthan$ | $\owedge$ | $\overrightarrow{\triangleright}$ | $\overrightarrow{\triangle}$ |
| $\neg((x \stackrel{\bullet}{\rightarrow} \bullet)^\top)$ | $\overleftarrow{\triangledown}$ | $\overleftarrow{\triangleleft}$ | $\bot^2$ | $\bot^2$ |
| $\neg((\bullet \stackrel{\bullet}{\rightarrow} x)^\top)$ | $\overleftarrow{\triangleright}$ | $\overleftarrow{\triangle}$ | $\bot^2$ | $\bot^2$ |

The following table lists different important combinations of the above binary relations.

| (a) | (b) | (c) | (d) | (e) | (f) |
|---|---|---|---|---|---|
| $(\ovee \cap \owedge \cap \olessthan \cap \ogreaterthan)$ | $(\ovee \cap \overleftarrow{\triangle})$ | $(\owedge \cap \overleftarrow{\triangledown})$ | $(\olessthan \cap \overleftarrow{\triangleright})$ | $(\diamond \cap \overleftarrow{\triangle} \cap \overleftarrow{\triangledown})$ | $(\diamond \cap \overleftarrow{\triangleright} \cap \overleftarrow{\triangleleft})$ |

For example, the composition $acyc\ (\diamond \cap \overleftarrow{\rhd} \cap \overleftarrow{\lhd})\ acyc$ requires that the distinct acyclic edges form a loop. Note that all compositions imply the source-label disjointness condition. Thus $acyc\ (\bigcirc\!\!\!\!\!\vartriangle \cap \overleftarrow{\triangledown})\ acyc$ will require the two source-sharing edges to have different labels. The following figure (again from [10]) demonstrates that the created binary relations can be used to combine two distinct acyclic edges in all possible layouts (to within obvious symmetry).




Here, the property $acyc$ is abstract, having no free variables. Similar structures can be specified in SL if the source, label and target of the edges are observable from the outside as free variables with inequalities between them and existential quantifiers applied outside.

## 2.4 List, parallel lists and reachability

In practice, we often reason about deadends (*i.e.* targets without outgoing edges) and deadheads (*i.e.* sources without incoming edges). For example, to extend a list, we may add an edge to the end (or symmetrically to the head) of the list so that the deadend of the list meets the source of the edge. Formally, a node $v$ is a deadend if it is the unique non-source target:

$$de(v)\ :=\ \neg\,\exists x\cdot((\bullet \overset{\bullet}{\rightarrow} x)^{\top} \wedge \neg(x \overset{\bullet}{\rightarrow} \bullet)^{\top} \wedge x \neq v)$$

where $x$ is a fresh variable different from $v$. That definition stipulates that there does not exist a node $x$ different than $v$ such that $x$ is the target of some edge but not the source of any edge. Deadhead $v$ is the unique non-target source:

$$dh(v)\ :=\ \neg\,\exists x\cdot(\neg(\bullet \overset{\bullet}{\rightarrow} x)^{\top} \wedge (x \overset{\bullet}{\rightarrow} \bullet)^{\top} \wedge x \neq v).$$

Let $\diamond := \exists x\cdot(de(x) \times dh(x))$ denote the composition that joins, under some assignment, the deadend of one graph with the deadhead of another; if either the deadend or the deadhead does not exists (*e.g.* in $\varnothing$ and $cyc$ etc.), then $\diamond$ behaves the same as $*$.

The concatenation relation between two list segments can be defined as the existence of a common join point as the deadend of the LHS and the deadhead of the RHS, and the targets of the RHS cannot reach into the sources of the LHS: $\bowtie\ :=\ (\bigcirc\!\!\!\!\!\lhd \cap \diamond)$. This binary relation is associative. For example, a list of length four is defined:

$$acyc^4\ :=\ acyc \bowtie acyc \bowtie acyc \bowtie acyc\,.$$

Because separating conjunction does not insist on the RHS not reaching into the LHS, in SL that property would require "enough inequalities" [1] between variables in conjunction to prevent edges from forming a cycle. Those variables are then hidden with the same number of existential quantifiers at the outermost

layer of the formula. The existential hiding in our representation is applied locally, making the representation more abstract. Using separating conjunction $*$ (note that exactly four edges should be spatially conjoined with a designated label):

$$acyc(a)^4 = \exists x_1x_2x_3x_4x_5 \cdot (x_1 \xrightarrow{a} x_2) * (x_2 \xrightarrow{a} x_3) * (x_3 \xrightarrow{a} x_4) * (x_4 \xrightarrow{a} x_5) \wedge \bigwedge_{i=1}^{4} x_i \neq x_5 .$$

Without the inequalities, the list could form a cycle via $x_5$. The definition using $\bowtie$ is arguably more abstract and simpler than the corresponding expression using $*$ in SL. The hiding of the outermost free variables reflects the fact that the composition $*$ is not abstract enough. This problem becomes more significant when fixpoints are present.

Let us first define a general property on lists comprising recursively $\bowtie$-concatenated acyclic edges: $list \ := \ \varnothing \propto_{\bowtie} acyc$, which can be comprehended as a universal disjunction:

$$list \ = \ \varnothing \vee acyc \vee (acyc \bowtie acyc) \vee \cdots .$$

The definition is so general that it contains no names or free variables. A list segment $list(u,v)$ from some atom $u$ to another atom $v$ is simply defined:

$$list(u,v) \ := \ (list \wedge dh(u) \wedge de(v)) \vee (u = v \wedge \varnothing).$$

However, the following two properties are very different:

$$list(u,v) * list(v,w) \qquad \text{and} \qquad list(u,v) \circledcirc list(v,w).$$

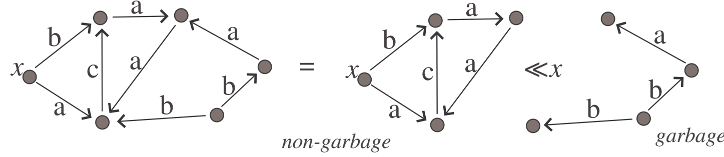
The property on the left connects a list segment from $u$ to $v$ with a list segment from $v$ to $w$, but the merge operator does not disallow the formation of a loop from $w$ into some intermediate node in the list segment from $u$ to $v$; while the property on the right not only links them but also disallows the end node $w$ to reach any nodes between $u$ and $v$ (hence forming a longer list segment). All free variables can be avoided using the $\bowtie$ composition: $list \bowtie list = list$. It is impossible to connect two general list segments to form a longer segment in SL, as this would require an unbounded number of free variables to be placed in inequalities and existentially quantified. Thus the mechanism to create new compositions is strictly more expressive than SL's separating conjunction when extended to include fixpoints, as done by Sims [31].

Although separating conjunction is not abstract enough for merging list segments, it is just right for forming multiple parallel lists which meet only at the end: $plists \ := \ \varnothing \propto_{*} list$. A node $u$ can reach another node $v$ if there exists a path (*i.e.* list) from $u$ to $v$: $reach(u,v) := list(u,v)^{\top}$. Graphs reachable from a node $v$ form a property such that no node in the graph is not path-reachable from $v$: $reachable(v) \ := \ \neg \exists x \cdot (node(x) \wedge \neg reach(v,x))$ where $x$ is a fresh variable. Unlike the extension of SL with fixpoints, the general reachability property here does not identify the concrete local pointer structure.

General reachability can be used to define the decomposition between the garbage and non-garbage parts of memory. The garbage part contains all edges whose sources are not reachable from a given root node, which represents the access point of the memory; the non-garbage part contains all edges reachable from the root. The relation between non-garbage and garbage is defined:

$$\ll_v \ := \ \circleddash \cap \ (reachable(v) \times \neg (v \xrightarrow{\bullet} \bullet)^{\top}).$$

The following figure illustrates the garbage-non-garbage decomposition.




For more program variables $x_1, \cdots, x_n$, we define reachability from all of them:

$$\ll_{x_1 \cdots x_n} \; := \; \ogreaterthan \cap \; (\mathit{reachable}(x_1) * \cdots * \mathit{reachable}(x_n) \times \textstyle\bigwedge_i^n \neg (x_i \xrightarrow{\bullet} \bullet)^{\top}).$$

The reachable part from multiple variables are mergeable, and the garbage part must not contain these variables as sources.

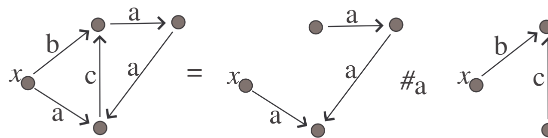
## 2.5 Unique decomposition

The formation of new compositions leads to the important concept of 'unique decompositions' [9]. A composition is a *unique decomposition* if the decomposition of any graph into sub-graphs related by the relation is unique. It has been shown that this condition corresponds to the conjunctivity law [9]. A typical example is the separation between non-garbage and garbage parts of the memory. A logic using only unique decompositions behaves well compositionally, satisfying strong laws. Although spatial conjunction is not a unique decomposition, if one of its arguments is fixed as a 'precise predicate', as defined by O'Hearn, Reynolds and Yang [24], then it is essentially strengthened into a unique decomposition.

Any given graph's decomposition into a non-garbage part (reachable from some node) and a garbage part is unique. Another simple example of unique decomposition is the separation between edges labeled as $a$ and those with other labels:

$$\#_a \; := \; (\neg \exists x \cdot (\bullet \xrightarrow{x} \bullet)^{\top} \wedge x \neq a) \times \neg (\bullet \xrightarrow{a} \bullet)^{\top}.$$

The following figure illustrates how a graph is uniquely decomposed into two parts.




**Def 1** *A composition r is a* unique decomposition *if for any assignment* $\varepsilon$ *and object graphs* $G, G', H, H'$ *such that* $G \cup H = G' \cup H'$ *and both* $r(\varepsilon, G, H)$ *and* $r(\varepsilon, G', H')$ *and true, we have* $G = G'$ *and* $H = H'$. *The relation is a* full unique decomposition *(or* fud*), if in addition it satisfies* $[r] = \top$.

The false composition $\overline{*}$ is a unique decomposition (but not a fud). The intersection of two unique decompositions is also a unique decomposition. Any unique decomposition $r$ can be transformed into the fud $r \cup (\neg[r] \times \varnothing)$ (recall that $[r]$ is defined to be $(\top\ r\ \top)$).

A fud is always "passive and transparent" in the sense that it distributes over all connectives and quantifiers, while a normal composition like $*$ distributes over only disjunction and existential quantification. Here we list only laws specific to fuds $r$:

**Law 2** (1) $|r \cap \overline{s}| = \neg |r \cap s|$
(2) $|r \cap s_1 \cap s_2| = |r \cap s_1| \wedge |r \cap s_2|$

A *precise predicate* corresponds, in *AOG*, to a property $P_0(\varepsilon, G)$ such that for any assignment $\varepsilon$ is related to a unique object graph $G$. Let $P_0, Q_0, \cdots$ denote precise predicates. For example, the empty-graph property is precise, so is any edge property. The composition of two precise predicates is either $\perp$ or a precise predicate of the merged graph.

A non-fud relation may be used in a context in which it essentially corresponds to a fud (strengthened by its arguments). The disjointness relation $*$ is not a fud, but if either of its arguments is precise and not an empty-graph property, and the other argument does not contain the subgraph, then it becomes a fud: $\prec_{P_0}\ :=\ (P_0 \vee \varnothing) \times \neg P_0^{\top}$. The arbitrary graph extension $P_0^{\top}$ is representable as $\neg\varnothing \prec_{P_0} \top$ (or equally as $P_0 \prec_{P_0} \top$). This suggests that compositions of precise predicates (and $\top$) are special cases of unique decompositions. Although $\bowtie$ itself is not a fud, it corresponds to a fud in the context of lists:

$$\bowtie\ =\ (\circledcirc \cap \diamond \cap list \times acyc) \cup (\neg list \times \varnothing) \cup (\varnothing \times \varnothing).$$

# 3 Logic-Based Automated Verification I: List Reversal

## 3.1 Logic for parallel lists

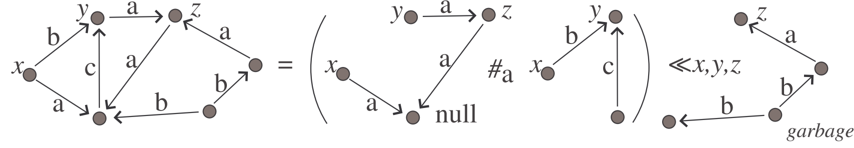
Many pointer algorithms such as in-place list reversal maintain an overall topological structure of several linked lists meeting at the deadend null. To check this invariant property, a compiler must eliminate all pointer alias and pointer loops.

The in-place list reversal algorithm uses three pointer variables. Each state is an object graph. We assume that the pointer variables always point to objects that contain a field $a$. In each state of the program, the $a$-edges reachable from the variables form parallel lists sharing the deadend null. Other non-$a$ fields can form arbitrary pointer structures. Each state may also contain arbitrary garbage.

Let $x$, $y$ and $z$ be the pointer variables used in the program. The following function maps a local shape of parallel lists to a property of program state:

$$\ell(P)\ :=\ ((P \wedge plists \wedge de(\text{null}))\ \#_a\ \top) \ll_{\mathbf{x}} \top$$

where $\mathbf{X} = \{x, y, z\}$. The following figure shows how a graph is uniquely decomposed by $\ll_{x,y,z}$ and then $\#_a$.




Notice that the definition $\ell(P)$ contains quantifiers, fixpoints and compositions like $\bowtie$ and $\ll_{x,y,z}$. Although *AOG* containing these is not decidable, the logic for parallel lists is much more tractable. Once we have established the laws of the logic, we can then forget about the definition and use the logic directly.

The function $\ell(P)$ satisfies some elegant distributivity laws, which are conveniently proved using the laws of unique decomposition.

**Law 3** (1) $\ell(\neg P) \;=\; \neg\ell(P) \wedge \ell(\top)$
(2) $\ell(P_1 \vee P_2) = \ell(P_1) \vee \ell(P_2)$
(3) $\ell(P_1 \wedge P_2) = \ell(P_1) \wedge \ell(P_2)$.

The syntax of the logic for parallel lists is:

$$\begin{aligned}
&\mathbf{X} ::= x \mid y \mid x_1 \mid \cdots \mid z\\
&\mathbf{V} ::= \text{null} \mid \mathbf{X}\\
&\mathbf{L} ::= \mathit{true} \mid \mathbf{V} \xrightarrow{I} \mathbf{V} \mid \sim\mathbf{L} \mid \mathbf{L} \vee \mathbf{L} \mid \mathbf{L} * \mathbf{L} \mid \exists \mathbf{X} \cdot \mathbf{L}
\end{aligned}$$

where $\mathbf{X}$ is finite, null is the deadend of all linked lists, and $\mathit{true} := \ell(\top)$. The interesting property $u \xrightarrow{I} v$ describes the distance from atom $u$ to atom $v$ on the parallel lists and should not be confused with the edge property. The set $I$ is a set of natural numbers. For example, the property $x \xrightarrow{\{1,2\}} y$ states that the variables $x$ and $y$ are on the same linked $a$-list, and we have either $x.a = y$ or $x.a.a = y$. If the two variables are not on the same linked list, we write $x \xrightarrow{\{\infty\}} y$, or $x \xrightarrow{\infty} y$ for short (a similar shorthand applies to other singleton sets). If the distance between the two variables on the same linked list is unknown, their relation is represented as $x \xrightarrow{[0,\infty)} y$ or $x \xrightarrow{*} y$ for short. Atom equality is captured with $u \xrightarrow{0} v$ (or $v \xrightarrow{0} u$):

$$\begin{aligned}
u \xrightarrow{0} v &:= \ell(u = v)\\
u \xrightarrow{n} v &:= \ell(\mathit{acyc}^n \wedge \mathit{dh}(u) \wedge \mathit{de}(v))\\
u \xrightarrow{\infty} v &:= \ell(\neg \mathit{reach}(x, v))\,.
\end{aligned}$$

For example, the full object graph in the previous figure satisfies the property

$$(x \xrightarrow{1} \text{null} \wedge y \xrightarrow{1} z \xrightarrow{1} \text{null} \wedge x \xleftrightarrow{\infty} y)$$

where $x \xleftrightarrow{\infty} y \;:=\; x \xrightarrow{\infty} y \wedge y \xrightarrow{\infty} x$.

The definitions of disjunction, conjunction and quantification in the logic are the same as in *AOG*. Negation in the logic must be enforced by *true*: $\sim P \;:=\; \neg P \wedge true$. False is the negation of true: $false \;:=\; \sim true = \bot = u \xrightarrow{\emptyset} v$. The propositional laws for $\vee$ and $\sim$ are established using the laws of fuds.

Set union of $I$ corresponds to logical disjunction: $u \xrightarrow{I_1 \cup I_2} v = u \xrightarrow{I_1} v \vee u \xrightarrow{I_2} v$, intersection to conjunction, and complement to negation: $u \xrightarrow{[0,\infty]\setminus I} v = \sim(u \xrightarrow{I} v)$. The distances between atoms satisfy simple arithmetic laws. Every atom in $\mathbf{X} \cup \{\mathrm{null}\}$ can reach the end node null. Any distance ranges within $[0,\infty]$ (an interval including $\infty$). The distance sets are addable. The deadend null does not reach any non-null variable.

**Law 4** (1) $u \xrightarrow{*} \mathrm{null} \;\equiv\; true$
(2) $u \xrightarrow{[0,\infty]} v \;\equiv\; true$
(3) $u \xrightarrow{m} v \wedge v \xrightarrow{n} w \Rightarrow u \xrightarrow{m+n} w$
(4) $u \xrightarrow{0} \mathrm{null} \vee \mathrm{null} \xrightarrow{\infty} u \;\equiv\; true$

The above laws, as properties of *plists*, eliminate negation and transform any formula without quantifiers to a disjunctive or conjunctive normal form. To handle quantifiers, we need a constructive normal form:

$$\bigvee_i \prod_j chain(X_{ij1}, I_{ij1}, \cdots, X_{ijk}, I_{ijk}, X_i) \tag{1}$$

where $\prod$ denotes universal separating conjunction, $k$ depends on $i$ and $j$, each $X_{ijt}$ or $X_i$ denotes a non-empty set of aliased pointer variables, each $I_{ijt}$ is either $\{n\}$ or $[n,\infty)$ where $n>0$, and the deadend node null is always in every $X_i$. Each property $chain(Y_1, I_1, \cdots, Y_k, I_k, Y_{k+1})$ describes a linked list with $k$ segments. All pointer variables in each $Y_t$ ($1 \leqslant t \leqslant k+1$) are equal and aliased. The head of the list is referenced by all pointer variables in $Y_1$. The list ends at the node $\mathrm{null} \in Y_{k+1}$. All variables in $Y_{k+1}$ are equal to null. Chains are defined as follows:

$$chain(Y_1, I_1, \cdots, Y_k, I_k, Y_{k+1}) \;:=\; \bigwedge_{t \leqslant k} \bigwedge_{u,u' \in Y_t;\, v,v' \in Y_{t+1}} u \xrightarrow{0} u' \xrightarrow{I_t} v \xrightarrow{0} v'$$

where we assume $\mathrm{null} \in Y_{k+1}$. The definition implies that two pointer variables are aliased iff they are in the same $Y_t$. Parallel chains conjoined by $*$ imply disjointness between variable sets from different chains except for the endnode.

For example, if there are only two pointer variables $x$ and $y$, the property *true* can be decomposed to a constructive normal form with seven disjuncts:

$$\begin{aligned} true \;\equiv\; & chain(\{x\}, 1^+, \{y\}, 1^+, \{\mathrm{null}\}) \vee chain(\{y\}, 1^+, \{x\}, 1^+, \{\mathrm{null}\}) \\ & \vee\; chain(\{x\}, 1^+, \{\mathrm{null}\}) * chain(\{y\}, 1^+, \{\mathrm{null}\}) \\ & \vee\; chain(\{x\}, 1^+, \{y, \mathrm{null}\}) \vee chain(\{y\}, 1^+, \{x, \mathrm{null}\}) \\ & \vee\; chain(\{x, y\}, 1^+, \{\mathrm{null}\}) \vee chain(\{x, y, \mathrm{null}\}). \end{aligned}$$

Note that the normal form is not unique: each range $[1,\infty)$ can be further decomposed into $\{1\} \cup [2,\infty)$. However each property has a simplest normal form.

### 3.2 Normal form reduction

Every property can be transformed to the normal form (1). To show this, we need laws to eliminate negations, conjunctions, quantifiers and separating conjunction. The previous laws are already capable of removing negations and conjunctions. Two chains

$$c_1 \;:=\; chain(X_1, I_1, \ldots, X_k, I_k, X) \quad \text{and} \quad c_2 \;:=\; chain(X_1', I_1', \ldots, X_l', I_l', X')$$

conjoined with separating conjunction is false iff some variable sets from the two chains are not disjoint. However, if they are indeed disjoint, then they do not reach each other:

$$c_1 * c_2 \;\equiv\; c_1 \wedge c_2 \wedge \bigwedge_{y \in I_1 \cup \cdots \cup I_k} \bigwedge_{y' \in I_1' \cup \cdots \cup I_l'} y \overset{\infty}{\leftrightarrow} y'.$$

The following laws help eliminate quantifiers.

**Law 5** (1) $\exists x \cdot chain(X_1, I_1, \cdots, X_k, I_k, X)$
$\equiv\; chain(X_1, I_1, \cdots, X_i \setminus \{x\}, I_i, \cdots, X_k, I_k, X) \qquad (x \in X_i \neq \{x\})$
(2) $\exists x \cdot chain(X_1, I_1, \cdots, I_i, \{x\}, I_{i+1}, , \cdots X_k, I_k, X)$
$\equiv\; chain(X_1, I_1, \cdots, X_i, I_i + I_{i+1}, X_{i+1}, \cdots, X_k, I_k, X)$
(3) $\exists x \cdot chain(X_1, I_1, \cdots, X_k, I_k, X \cup \{x\}) \;\equiv\; chain(X_1, I_1, \cdots, X_k, I_k, X \setminus \{x\})$.

**Theorem 1 (Normal form and validity)** *Every property in the logic is semantically equal to a property in constructive normal form, and validity of the normal form is decidable.*

### 3.3 Automated program verification

We now apply static symbolic execution over logical formulas as abstract states. The language that we consider is similar to the Guarded Command Language containing pointer assignments, conditionals (as well as nondeterminism), sequential composition and loops, though a different syntax is adopted. Pointer assignments have four forms. Let **pre**.$U$ denote the condition under which an assignment $U$ can proceed safely without generating errors or destroying the overall structure of parallel lists ($v$ is either a variable or null):

$$\begin{aligned}
\mathbf{pre}.(x := v) \;&:=\; true \\
\mathbf{pre}.(x := y.a) \;&:=\; y \overset{1^+}{\rightarrow} \text{null} \\
\mathbf{pre}.(x.a := y.a) \;&:=\; (x \overset{0}{\rightarrow} y \vee y \overset{1}{\rightarrow} \text{null}) \wedge x \overset{1^+}{\rightarrow} \text{null} \\
\mathbf{pre}.(x.a := v) \;&:=\; x \overset{1^+}{\rightarrow} \text{null} \wedge v \overset{\infty}{\rightarrow} x \wedge \textstyle\bigwedge_z (z \overset{0}{\rightarrow} v \vee z \overset{\infty}{\rightarrow} v \vee v \overset{0}{\rightarrow} \text{null} \vee z \overset{*}{\rightarrow} x).
\end{aligned}$$

The program *skip*, that does not change program state, is the same as $x := x$. Non-pointer arithmetic assignments are regarded as *skip*. More precise analysis that takes arithmetic variables into account is possible but requires extension of the logic (see, for example, Chang and Rival's paper [7]). Note that $v$ can be either a variable or the null pointer. Direct pointer assignment $x := v$ does not change

the pointer structure and hence requires no checking. The assignment $x := y.a$ requires $y$ not to be the null pointer. The assignment $x.a := y.a$ requires not only $x \neq \text{null}$ and $y \neq \text{null}$ but also requires $x = y$ unless $y.a = \text{null}$, because null is the only name that can be aliased. The assignment $x.a := v$ requires that $x \neq \text{null}$, $v$ is not path-reachable to $x$ (forming a loop), and a non-null pointer $v$ should be either at the start of a different chain or reachable from $x$.

We use **pre** to eliminate pointer-related dynamic errors statically. Non-pointer errors are not handled by this logic. The evaluation of expressions may also generate errors. Let **pre**.$e$ denote the condition for an expression $e$ to be safely evaluated: **pre**.$e$ := $\bigwedge_{x \in V(e)} x \xrightarrow{1^+} \text{null}$ where $V(e)$ is the set of free variables $x$ such that $x.a$ appear in $e$.

Let **sp**.$S.P$ denote the strongest postcondition restricted to properties $P$ in the logic. In other words, **sp**.$S.P$ holds at just those states in which computation $S$ is sure to terminate when started in a state satisfying property $P$ (in the logic). Errors detectable by **pre** render **sp**.$S.P$ unimportant:

$$
\begin{aligned}
&\mathbf{sp}.(x := v).P \ := \ x \xrightarrow{0} v \wedge \exists x \cdot P \\
&\mathbf{sp}.(x := y.a).(P \wedge y \xrightarrow{1^+} \text{null}) \ := \ y \xrightarrow{1} x \wedge \exists x \cdot P \\
&\mathbf{sp}.(x.a := y.a).(P \wedge y \xrightarrow{1} \text{null}) \ := \ y \xrightarrow{1} \text{null} \wedge \mathbf{sp}.(x.a := \text{null}).P \\
&\mathbf{sp}.(x.a := y.a).(P \wedge x = y \neq \text{null}) \ := \ P \wedge x = y \neq \text{null}.
\end{aligned}
$$

The assignment $(x.a := \text{null})$ is a special case of $x.a := v$. The effect of the assignment $x.a := v$ depends on the pointer structure before the assignment. If $v \neq \text{null}$ points to the start of another non-empty chain then the chain containing $x$ will break into two parts with the first segment to $x$ connected to the chain starting from $y$. If $v$ happens to be the next from $x$ (with no variables between them) then the assignment shortens the chain so that $x$ directly points to $v$. If $v$ is null then the chain breaks into two segments; and if $v$ is null and no variables are between $x$ and $v$ then the chain is shortened at the end so that $x$'s $a$-field is null. Other initial pointer structures are prohibited by **pre**:

$$
\begin{aligned}
&\mathbf{sp}.(x.a := v).(chain(X_1, I_1, \cdots, X_i \cup \{x\}, I_i, \cdots, X) * chain(X'_1 \cup \{v\}, I'_1, \cdots, X'_k, I'_l, X) * P) \\
&:= \ chain(X_{i+1}, I_{i+1}, \cdots, X) * chain(X_1, I_1, \cdots, X_i \cup \{x\}, 1, X'_1 \cup \{v\}, I'_1, \cdots, X'_l, I'_l, X) * P \\
&\mathbf{sp}.(x.a := v).(chain(X_1, I_1, \cdots, X_i \cup \{x\}, I_i, , \cdots, X_j \cup \{v\}, I_j, \cdots, X_k, I_k, X) * P) \\
&:= \ chain(X_1, I_1, \cdots, X_i \cup \{x\}, 1, X_j \cup \{v\}, I_j, \cdots, X_k, I_k, X) * P \\
&\mathbf{sp}.(x.a := v).(chain(X_1, I_1, \cdots, X_i \cup \{x\}, I_i, \cdots, X_k, I_k, X \cup \{v\}) * P) \\
&:= \ chain(X_{i+1}, I_{i+1}, \cdots X_k, I_k, X \cup \{v\}) * chain(X_1, I_1, \cdots, X_i \cup \{x\}, 1, X \cup \{v\}) * P \\
&\mathbf{sp}.(x.a := v).(chain(X_1, I_1, \cdots, X_k \cup \{x\}, I_k, X \cup \{v\}) * P) \\
&:= \ chain(X_1, I_1, \cdots, X_i \cup \{x\}, 1, X \cup \{v\}) * P.
\end{aligned}
$$

A program Boolean expression $x.a = y$ corresponds to a formula: $|x.a = y| := x \xrightarrow{1} y$. Some Boolean expressions may contain sub-expressions not expressible in the logic. For example, the expression $(x.a = y \wedge n > 1)$ contains an arithmetic expression $n > 1$. This is handled by substituting every unknown proposition with *true* and *false* respectively and take the overall disjunction:

$$|x.a = y \vee n > 1| \ = \ (x \xrightarrow{1} y \vee true) \vee (x \xrightarrow{1} y \vee false) \ = \ true.$$

Note that $|\sim b| \neq \sim |b|$ in general, although we do have $|b| \vee |\sim b| = true$. For example

$$|x.a \neq y \wedge n \leqslant 1| \ = \ x \xrightarrow{0} y \vee x \xrightarrow{[2,\infty]} y.$$

A conditional statement `if` $(n>1)$ `then` $S$ `else` $T$ essentially becomes a nondeterministic choice. The proof obligation for `if` $b$ `then` $S$ `else` $T$ is in the annotational style:

$$
\begin{array}{l}
\{P \quad (\text{check } P \subseteq \mathbf{pre}.b)\} \\
\texttt{if } b \texttt{ then } \{P\wedge|b|\} \quad S \quad \{\mathbf{sp}.S.(P\wedge|b|)\} \\
\qquad \texttt{else } \{P\wedge|\sim b|\} \quad T \quad \{\mathbf{sp}.T.(P\wedge|\sim b|)\} \\
\{\mathbf{sp}.(\texttt{if } b \texttt{ then } S \texttt{ else } T).P \quad (:= \ \mathbf{sp}.S.(P\wedge|b|)\vee \mathbf{sp}.T.(P\wedge|\sim b|))\}
\end{array}
$$

where the final abstract state is defined to be the disjunction of the final states of the two branches.

Program loop $($`while` $b$ `do` $S)$ is handled by statically simulating its iteration in abstract states and checking the structure at every step. Here we are using a widening operator $P\!\uparrow$. The space of abstract states is infinite. We use the technique of widening to force the static iteration to reach a fixpoint in finitely-many steps. In the constructive normal form, the widening operator lifts every singleton range $\{n\}$ to $[n,\infty)$ (or written as $n$ and $n^+$ respectively). For example, $(y\xrightarrow{2}x\xrightarrow{0}\text{null})\uparrow = (y\xrightarrow{2^+}x\xrightarrow{0}\text{null})$:

$$
\begin{array}{l}
\{P_0 \quad (\text{check } P \subseteq \mathbf{pre}.b)\} \\
//\{P_1 \quad (:= \ P_0'\uparrow\wedge\sim P_0, \quad \text{check } P_1 \subseteq \mathbf{pre}.b \text{ and } P_1 \not\subseteq |b|)\} \\
\quad \cdots\cdots \\
//\{P_{m+1} \quad (:= \ P_m'\uparrow\wedge\sim P_m\wedge\cdots\wedge\sim P_0, \text{check } P_{m+1} \subseteq \mathbf{pre}.b \text{ and } P_{m+1} \subseteq \sim|b|)\} \\
\texttt{while } b \texttt{ do} \quad \{P_0\wedge|b|\}\,//\,\{P_1\wedge|b|\}\,//\,\cdots\,//\,\{P_m\wedge|b|\} \\
\qquad S \qquad \{P_0'\}\,//\,\{P_1'\}\,//\,\cdots\,//\,\{P_m'\} \\
\{\mathbf{sp}.(\texttt{while } b \texttt{ do } S).P_0 \quad (:= \ (P_0\vee\cdots\vee P_m)\wedge|\sim b|)\}.
\end{array}
$$

where $P_i' := \mathbf{sp}.S.(P_i\wedge|b|)$. The initial abstract state $P_0$ must allow $b$ to be evaluated. The abstract state $P_1$ at the beginning of the second iteration is the widened result of the first iteration negating $P_0$. Previous checking does not need to be repeated. If $P_1$ is inconsistent with $|b|$ (*i.e.* $P_1' \subseteq \sim|b|$)) then in the real execution the program will terminate at this point, which requires no further static iteration. If $P_1\wedge|b|$ is not invalid, the iteration continues until the abstract state $P_{m+1}$ is inconsistent with $|b|$, and the final abstract state is the disjunction of all $P_i$ conjoined with $|\sim b|$.

The following theorem guarantees termination of the verification:

**Theorem 2** *For any program, the parallel-list verification terminates in finitely-many steps.*

### 3.4 Automated verification of list reversal

The logic can be used to verify that a program maintains the structure of parallel lists and to detect errors like dereferencing a null pointer. See Figure 1. Note that we show the abstract states in the restrictive form succinctly, although they should be represented in the constructive normal form for automated verification.

As pointed out in the Introduction, some of the most common errors in pointer programs relate to incorrectly ordered pointer assignments and our aim here is to provide compiler support to pick up such errors. For example, if the last two assignments in the loop body are wrongly swapped, the compiler

$$
\begin{array}{l}
\texttt{inv ParLists}(a)\ \texttt{in} \\
\quad \{\mathit{true}\} \\
\quad y := \text{null};\, z := \text{null}; \\
\quad \left\{ y \xrightarrow{0} z \xrightarrow{0} \text{null} \right\} /\!/ \left\{ y \xrightarrow{1^+} x \xrightarrow{0} z \xrightarrow{0} \text{null} \vee\ y \xrightarrow{1^+} \text{null} \wedge x \xrightarrow{0} z \xrightarrow{1^+} \text{null} \wedge x \xleftrightarrow{\infty} y \xleftrightarrow{\infty} z \right\} \\
\quad \texttt{while}\ (x \neq \text{null})\ \texttt{do} \quad \left\{ x \xrightarrow{1^+} y \xrightarrow{0} z \xrightarrow{0} \text{null} \right\} /\!/ \left\{ y \xrightarrow{1^+} \text{null} \wedge x \xrightarrow{0} z \xrightarrow{*} \text{null} \wedge x \xleftrightarrow{\infty} y \xleftrightarrow{\infty} z \right\} \\
\qquad z := x.a; \quad \left\{ x \xrightarrow{1} z \xrightarrow{*} y \xrightarrow{0} \text{null} \right\} \\
\qquad /\!/ \left\{ y \xrightarrow{1^+} \text{null} \wedge x \xrightarrow{1} z \xrightarrow{0} \text{null} \wedge x \xleftrightarrow{\infty} y\ \vee\ y \xrightarrow{1^+} \text{null} \wedge x \xrightarrow{1} z \xrightarrow{1^+} \text{null} \wedge x \xleftrightarrow{\infty} y \xleftrightarrow{\infty} z \right\} \\
\qquad x.a := y; \quad \left\{ x \xrightarrow{1} y \xrightarrow{0} z \xrightarrow{0} \text{null}\ \vee\ x \xrightarrow{1} y \xrightarrow{0} \text{null} \wedge z \xrightarrow{1^+} \text{null} \wedge x \xleftrightarrow{\infty} z \right\} \\
\qquad /\!/ \left\{ x \xrightarrow{1} y \xrightarrow{1^+} z \xrightarrow{0} \text{null}\ \vee\ x \xrightarrow{1} y \xrightarrow{1^+} \text{null} \wedge z \xrightarrow{1^+} \text{null} \wedge x \xleftrightarrow{\infty} z \xleftrightarrow{\infty} y \right\} \\
\qquad y := x; \quad \left\{ x \xrightarrow{0} y \xrightarrow{1} z \xrightarrow{0} \text{null}\ \vee\ x \xrightarrow{0} y \xrightarrow{1} \text{null} \wedge z \xrightarrow{1^+} \text{null} \wedge x \xleftrightarrow{\infty} z \xleftrightarrow{\infty} y \right\} \\
\qquad /\!/ \left\{ x \xrightarrow{0} y \xrightarrow{2^+} z \xrightarrow{0} \text{null}\ \vee\ x \xrightarrow{0} y \xrightarrow{2^+} \text{null} \wedge z \xrightarrow{1^+} \text{null} \wedge x \xleftrightarrow{\infty} z \xleftrightarrow{\infty} y \right\} \\
\qquad x := z; \quad \left\{ y \xrightarrow{1} x \xrightarrow{0} z \xrightarrow{0} \text{null}\ \vee\ y \xrightarrow{1} \text{null} \wedge x \xrightarrow{0} z \xrightarrow{1^+} \text{null} \wedge x \xleftrightarrow{\infty} y \xleftrightarrow{\infty} z \right\} \\
\qquad /\!/ \left\{ y \xrightarrow{2^+} x \xrightarrow{0} z \xrightarrow{0} \text{null}\ \vee\ y \xrightarrow{2^+} \text{null} \wedge x \xrightarrow{0} z \xrightarrow{1^+} \text{null} \wedge x \xleftrightarrow{\infty} y \xleftrightarrow{\infty} z \right\} \\
\quad \left\{ x \xrightarrow{0} z \xrightarrow{0} \text{null} \right\}
\end{array}
$$

Figure 1: Verification of properties for the program performing in-place list reversal.

will pick up this error in the second static iteration when the program may form a pointer loop from $y$ to itself:

$$
\begin{array}{l}
\texttt{inv ParLists}(a)\ \texttt{in} \quad \{\mathit{true}\} \\
\quad y := \text{null};\, z := \text{null}; \quad \left\{ y \xrightarrow{0} z \xrightarrow{0} \text{null} \right\} /\!/ \left\{ x \xrightarrow{0} y \xrightarrow{0} z \xrightarrow{1^+} \text{null} \right\} \\
\quad \texttt{while}\ (x \neq \text{null})\ \texttt{do} \quad \left\{ x \xrightarrow{1^+} y \xrightarrow{0} z \xrightarrow{0} \text{null} \right\} /\!/ \left\{ x \xrightarrow{0} y \xrightarrow{0} z \xrightarrow{1^+} \text{null} \right\} \\
\qquad z := x.a; \quad \left\{ x \xrightarrow{1} z \xrightarrow{*} y \xrightarrow{0} \text{null} \right\} /\!/ \left\{ x \xrightarrow{0} y \xrightarrow{1} z \xrightarrow{*} \text{null} \right\} \\
\qquad x.a := y; \quad \left\{ x \xrightarrow{1} y \xrightarrow{0} z \xrightarrow{0} \text{null}\ \vee\ x \xrightarrow{1} y \xrightarrow{0} \text{null} \wedge z \xrightarrow{1^+} \text{null} \wedge x \xleftrightarrow{\infty} z \right\} /\!/ \textbf{Error: loop} \\
\qquad x := z; \quad \left\{ x \xrightarrow{0} z \xrightarrow{*} y \xrightarrow{0} \text{null} \right\} \\
\qquad y := x; \quad \left\{ x \xrightarrow{0} y \xrightarrow{0} z \xrightarrow{*} \text{null} \right\}
\end{array}
$$

Another feature that we can verify using the logic concerns memory leakage. Even if a pointer swing $x.a := v$ maintains the structure of parallel lists, it may still lose objects unless the object $x.a$ is null or referenced by another pointer variable (a condition represented as $\bigvee_u x \xrightarrow{1} u$). The assignment to a variable $x$, on the other hand, must maintain the structure and check that the object initially referenced

by $x$ is still reachable from other variables:

$$\begin{aligned}
\mathbf{pre}'.(x:=v) &:= v \xrightarrow{0} x \vee \bigvee_{u \notin \{x\}} u \xrightarrow{*} x \\
\mathbf{pre}'.(x:=y.a) &:= \mathbf{pre}.(x:=y.a) \wedge \bigvee_{u \notin \{x\}} u \xrightarrow{*} x \\
\mathbf{pre}'.(x.a:=y.a) &:= \mathbf{pre}.(x.a:=y.a) \wedge \bigvee_{u} x \xrightarrow{1} u \\
\mathbf{pre}'.(x.a:=v) &:= \mathbf{pre}.(x.a:=v) \vee \bigvee_{u} x \xrightarrow{1} u\,.
\end{aligned}$$

For in-place list reversal, if the first two assignments in the loop body are wrongly swapped, then the assignment $x.a:=y$ will shortcut the linked list from $x$ and set the object $x.a$ to be null immediately. If $x.a$ is initially a non-null object, then the shortcut assignment will lose the reference to that object. This error is detectable in the first static iteration:

```
inv ParLists(a) in     {true}
  y := null; z := null;   {y -0-> z -0-> null}
  inv NoMemoryLeak in
    while (x ≠ null) do   {x -1+-> y -0-> z -0-> null}     Error: loss of objects
      x.a := y; z := x.a; y := x; x := z.
```

# 4 Automated Verification II: Schorr-Waite Graph Marking

Abstraction of general object graphs can be achieved with different levels of precision. Here, we consider the abstraction that describes the rough distances between objects (*e.g.* $x, y, \cdots$) and their immediate fields (*e.g.* $x.l, y.r, \cdots$). There are three possibilities: $u \xrightarrow{0} v$ for pointer equality, $u \xrightarrow{+} v$ for unequal reachability and $u \xrightarrow{\infty} v$ for non-reachability in the logic. Here we discuss the logic more briefly.

$$\begin{aligned}
&\mathbf{N} ::= \text{null} \mid a \mid b \mid c \mid a_1 \mid \cdots \\
&\mathbf{X} ::= x \mid y \mid z \\
&\mathbf{V} ::= \mathbf{N} \mid \mathbf{X} \mid \mathbf{X}.l \mid \mathbf{X}.r \\
&\mathbf{L} ::= \mathit{true} \mid \mathbf{V}=\mathbf{V} \mid \mathbf{V} \xrightarrow{+} \mathbf{V} \mid \sim\!\mathbf{L} \mid \mathbf{L} \vee \mathbf{L}.
\end{aligned}$$

For simplicity, we consider only three variables and assume that every object has two fields $l$ and $r$. The following function maps a shape property to a graph-abstraction property:

$$\begin{aligned}
\ell(P) := (\ &\exists x_0 y_0 z_0 x_1 y_1 z_1 \cdot P[x_0, x_1, y_0, y_1, z_0.z_1 / x.l, x.r, y.l, y.r, z.l, z.r] \\
&\wedge x \xrightarrow{l} x_0 \wedge x \xrightarrow{r} x_1 \wedge y \xrightarrow{l} y_0 \wedge y \xrightarrow{r} y_1 \wedge z \xrightarrow{l} z_0 \wedge z \xrightarrow{r} z_1) \ll_{x,y,z} \top.
\end{aligned}$$

The quantified variables are fresh and point to the $l/r$-fields of the objects $x$, $y$ and $z$. Objects not reachable from $x$, $y$ and $z$ are subject to garbage collection. The property $\mathit{true} := \ell(\top)$ is the largest property. Equality $u=v$ corresponds to $\ell(u=v)$. Unequal reachability is

$$u \xrightarrow{+} v := \ell(\neg(u=v) \wedge \mathit{reach}(u,v)).$$

Non-reachability is the negation of equality and unequal reachability:

$$u \xrightarrow{\infty} v \ := \ \sim(u = v) \wedge \sim(u \xrightarrow{+} v).$$

It is easy to see that negation is closed in the normal form $\bigvee_i \bigwedge_j u_{ij} \xrightarrow{I_{ij}} v_{ij}$ .

Note that in this particular logic, the equality between $x.l.r$ and $y$ is represented as $x.l \xrightarrow{+} y$ , which means the latter is not equal to but reachable from the former. It turns out that such abstraction is already precise enough to verify the safety property of Gries's Schorr-Waite code [15]: the pointer manipulations do not cause memory leakage. See Figure 2.

Assume that at the beginning of the algorithm, the $l$-field of $z$ is $x$, and $y = z$. In the first round of iteration, the predicate-abstract state reaches $X_1$ after pointer assignment and conditional test of the counter $x.m$ being 3 or 0. Note that the logic does not handle arithmetic variables. The conditional essentially becomes a nondeterministic choice under such logical analysis. The other conditional branch reaches the abstract state $X_1'$ instead. The disjunctively accumulated predicate-abstract states reach the fixpoint after four iterations (separated by double backslashes). The invariant `GraphAnalysis` instructs the compiler to perform the analysis according to the above logic, while the other invariant `NoMemoryLeak` insists that no command converts to garbage that is reachable from $x$, $y$ or $z$.

Interestingly, although the two branches of the conditional are very different, the resulting assertions are entirely symmetric (with the positions of $x$ and $y$ swapped).

The compiler does not verify the functional correctness of the code. It checks the safety property that useful contents are not lost. This will significantly increase the confidence in its correctness.

# 5 Related work

The work presented in this paper uses a pointer-graph *model* as the basis for the *formalism AOG* in which to define *logics* that are appropriately expressible yet still *decidable*. In this section we consider work related to each of those italicised points.

A productive vehicle for formalising various aspects of object orientation and pointers has been Hoare and He's UTP [17]. UTP provides a framework in which the healthiness conditions satisfied by a binary-relational model (initial/final state) are imposed *seriatim*, enabling features in a relatively complex computational paradigm to be appreciated incrementally. Thus novel features pertaining to object-orientation are combined in a structured manner with standard features. Such an approach yields a model, but not necessarily explicit laws (although healthiness conditions can usually be read as laws) nor logics or a decision procedure.

$\{\, z.l = x \wedge y = z \quad (X_0) \,\}$

`inv GraphAnalysis in   inv NoMemoryLeak in`

$\quad \{X_1 \vee X'_1\} \;\; /\!/ \;\; \{X_2 \vee X'_2\} \;\; /\!/ \;\; \{X_3 \vee X'_3 \vee X_4 \vee X'_4\} \;\; /\!/ \;\; \{X_5 \vee X'_5\}$

$\quad$ `while` $(x \neq z)$ `and` $(x \neq \text{null})$ `do`

$\qquad \{\, y = z \neq z.l = x \neq \text{null} \,\} /\!/ \{\, z.l = y \neq y.r = z \neq x \neq \text{null} \vee z.l = x \neq x.r = z \,\}$

$\qquad x.m := x.m + 1\,;$

$\qquad$ `if` $(x.m = 3$ `or` $x.m = 0)$

$\qquad$ `then` $x, x.l, x.r, y := x.l, x.r, y, x \qquad \{\, z.l = y \neq y.r = z \quad (X_1) \,\}$

$$/\!/ \Big\{\, z \neq y.r = z.l \overset{+}{\to} z \neq y \vee z.l = y \neq y.l = z \qquad (X_2) \,\Big\}$$

$$/\!/ \left\{ \begin{array}{ll} \quad y.r \overset{+}{\to} z.l \overset{+}{\to} z \neq z.l \wedge y \neq z \vee z \neq y.l = z.l \overset{+}{\to} z \neq y & (X_3) \\ \vee\; x = z \neq z.l = y \neq \text{null} & (X_4) \end{array} \right\}$$

$$/\!/ \Big\{\, X_3 \vee X_1 \vee y.l \overset{+}{\to} z.l \overset{+}{\to} z \neq z.l \wedge y \neq z \qquad (X_5) \,\Big\}$$

$\qquad$ `else` $x.l, x.r, y := x.r, y, x.l \quad . \quad \{\, z.l = x \neq x.r = z \quad (X'_1) \,\}$

$$/\!/ \Big\{\, z \neq x.r = z.l \overset{+}{\to} z \neq x \vee z.l = x \neq x.l = z \qquad (X'_2) \,\Big\}$$

$$/\!/ \left\{ \begin{array}{ll} \quad x.r \overset{+}{\to} z.l \overset{+}{\to} z \neq z.l \wedge x \neq z \vee z \neq x.l = z.l \overset{+}{\to} z \neq x & (X'_3) \\ \vee\; x = z \neq z.l = x \neq \text{null} & (X'_4) \end{array} \right\}$$

$$/\!/ \Big\{\, X'_3 \vee X'_1 \vee x.l \overset{+}{\to} z.l \overset{+}{\to} z \neq z.l \wedge x \neq z \qquad (X'_5) \,\Big\}$$

$$\Big\{\, (x = z \vee x = \text{null}) \wedge (X_0 \vee \textstyle\bigvee_{i=1}^{5} X_i \vee X'_i) \,\Big\}.$$

Figure 2: Verification of properties for the Schorr-Waite marking algorithm.

## 5.1 Models

The model we use derives from the traces model of Hoare and He [18] whose benefit is that it avoids explicit enumeration of memory (a feature of earlier, and some current low-level, models but which is inappropriate for current object-oriented programming). There, rooted edge-labeled directed graphs are used as a formalism for reasoning about pointers (amongst other things). A canonical model is constructed in which each node is the set of all 'traces', or sequences of labels on paths to it from the root. Then traces replace explicit memory enumeration.

The use of abstract object graphs, properties and combinators was presented by the authors in [9]. The same model forms the basis of the Abstract Location Trace Graph (ALTG) used by Smith and Gibbons [33] who give healthiness conditions and achieve a UTP treatment of locations. They model not just shareable locations (those able to be dereference by a pointer) but also containable locations, reasoning that both are necessary in a theory sufficient to reason about C# for example. Their extension attributes meaning to pointer values, so that a value of a location can be a pointer to a shareable location and hence to a 'handle' (or pointer to a pointer).

An abstraction of the traces model is used by Paige and Ostroff [25] as a basis of a refinement calculus for Eiffel. They introduce an 'entity group' to be a partitioning of path names into sets of path names that

access the same value. 'Entity groups' are called 'path groups' by Cavalcanti, Harwood and Woodcock [16] and used in a carefully justified model of object-orientated programming that models shareable pointers without being tied to explicit memory enumeration, and being unconcerned with classes or visibility. A UTP approach is used to unify that work with termination.

The approach of UTP relies on binary-relation semantics. Predicate-transformer models of object orientation have been considered by Naumann [23] and Cavalcanti and Naumann [6] and have been progressively developed by Sampaio and Borba *et al.*; see for example [2] and [3].

A model that uses traces for navigation is also used by Liu, Liu and Zhao [21] in analysing refinement of UML-inspired directed labelled graphs. They use graph surgery to formalise the relationship between changes in class declarations and method definitions. Subsequently Ke, Liu, Wang and Zhao have used the model to give a small-step operational semantics of object-orientated programs [34].

## 5.2 Formalisms

We have concentrated on the *verification* of decidable shape-related properties. But there is a substantial branch of Formal Methods that takes the more comprehensive route of providing laws powerful enough to facilitate the development of code by a sequence of refinements from its specification. Exploitation of that approach requires much more of the programmer than specification and verification of code: development is demanded.

In that setting, treatments of pointers have been given by Borba, Sampaio and Cornélio [2], and Sampaio and Borba [28], where the language ROOL is introduced to model sequential Java with a copy semantics. In [30] Silva, Sampaio and Liu consider aliasing with a reference semantics and in [29] Santos, Cavalcanti and Sampaio take a UTP approach to the semantics of object orientation.

In [32] Smith and Gibbons model the object calculus of Abadi and Cardelli relationally in the style of UTP.

## 5.3 Logics and decidability

Graph decompositions are discussed extensively in various graph logics; see for example the work of Courcelle [12]. This paper focuses on user-created unique decompositions. Results related to ours include the decidable fragments of SL studied by Calcagno, Gardner and Hague [4], Calcagno, Yang and O'Hearn [5], and Distefano, O'Hearn and Yang [14] in which quantifiers are not allowed. Berdine, Calcagno and O'Hearn [1] have studied a restrictive decidable fragment with linked lists but without disjunction or quantifiers. Monadic Second-Order Logics (see Dawar, Gardiner and Ghelli's work [13]) use the simple merge operator without consistency checking and allow quantifiers over graphs but do not permit creation of new compositions. Work that is similarly second-order, and also for graph types, is PALE (the pointer assertion logic engine), due to Møller and Schwartzbach [22]. Chang and Rival [7] introduced inductive types for shape analysis. Unfolding (*e.g.* a list) of a structure may occur at different

positions (near head or tail) and need different rules to handle different points of breaking in the structure. By comparison, *AOG* does not rely on a specific inductive formation for lists.

Compared with the approaches related to SL, *AOG* is different not only for its set-theoretic and algebraic presentation but also for two important aspects: firstly, it describes properties about object graphs instead of post-compilation heaps (*i.e.* each of which is a mapping from addresses to addresses/values); secondly, SL uses just one composition (the separating conjunction of address disjointness), while *AOG* allows creation of more restrictive compositions that are strictly more expressive.

# 6 Conclusions

A key challenge for pointer analysis and verification is the design of useful decidable logical fragments. Such fragments should be general enough to suit a range of pointer algorithms and, at the same time, intuitive enough for programmers to understand. Designing a logic with sophisticated syntactical restrictions (to ensure decidability) or relying on uncertain feedback from a theorem prover can undesirably complicate the programmer's understanding of what are expressible and verifiable. Our suggestion is not to design one single pointer logic, but to define a collection of logics, each handling a specific class of pointer algorithms. The programmer inserts compilation commands in the source program to choose appropriate logics and their corresponding analysis and verification algorithms. The laws of new logical operators are proved using those of *AOG*.

Automated verification of legacy pointer code is restricted by the lack of information about the role of pointer variables in source programs. In a standard C/Java program, the type of a pointer variable determines only its object type and does not indicate whether it points to a tree or some position in a cycle; but that information can be extremely useful for a compiler in conducting the most appropriate static analysis. On the other hand, manual reasoning can establish the entire correctness of a program with respect to some formal specification, but the general formalism used is often undecidable.

We have adopted an alternative tradeoff approach to require the source programmer to provide a small amount of information about how the variables are used in the program by identifying the overall topology of the pointer structures. With such information, the compiler can then perform much more in-depth analysis. Instead of designing a large expressive logic, we propose to design many small logics and organise the verification procedures in a type library.

Singly-linked lists have been studied intensively in the literature. In fact the success of some previous approaches already relies on compositions different from the spatial conjunction of SL. For example, the SL fragment studied by Distefano, O'Hearn and Yang [14] uses $\mathtt{ls}(E,F)$ to denote a recursively defined list segment from address $E$ to $F$ and the spatial conjunction for concatenation: $\mathtt{ls}(E,F) * \mathtt{ls}(F,G)$. However, what is intended to be applied here is actually $\circledcirc$ or $\bowtie$ instead of $*$, because $*$ does not prevent $G$ from forming a loop with some address intermediate between $E$ and $F$. Similar phenomena occur to $\mathtt{ls}(E,F) * (F \mapsto G)$. Such unsoundness is not a problem if all composition and decomposition occur only when the first segment is a single address cell $(E \mapsto F) * \mathtt{ls}(F,G)$ in which case an additional inequality $E \neq G$ can prevent loop formation. To facilitate correct composition and decomposition from varied

positions, we need $\olessthan$ (or $\bowtie$, which is slightly more abstract than the former for not identifying the end nodes of concatenation). Using these new compositions will also save the existentially quantified auxiliary variables used in added inequalities [14] that prevent loop formation.

We have used strongest-postcondition-style forward reasoning. A backward weakest-precondition-style reasoning scheme could make the inverse operators useful. Assertions in our examples are generated, but they can be inserted by programmers too. An interesting future direction is study of the interactions between various verification methods based on *AOG* as well as their interaction with other logical methods on arithmetics. One obvious advantage of running two analysis methods at the same time is to improve the precision of static evaluation of Boolean expressions in conditional statements.

# A Proof outlines

**Theorem 1** Every property in the logic is semantically equal to a property in constructive normal form, and there is a decision procedure to determine the validity of the normal form.

**Proof outline.** Let $\mathbf{X} := \{x_1, \ldots, x_n\}$ be the set of variables. For the property *true*, we need to enumerate all possible layouts of these variables on parallel lists. This can be achieved by enumerating all partitions of $\mathbf{X}$ and, for each part in each partition, identifying all possible (total) orderings among variables, assigning $0$ or $n^+$ to the distances between adjacent variables, creating multiple chains, and finally using disjunction and separating conjunction appropriately to construct a normal form. The number of disjuncts is estimated to be of order $O(2^n)$ (by Rademacher's series expansion for the partition function).

Law 4 guarantees that any quantifier-free sublogical property has a negation-free non-constructive normal form:

$$\bigvee_i \bigwedge_j u_{ij} \xrightarrow{I_{ij}} v_{ij} \,. \tag{2}$$

This transformation takes $O(2^m)$ steps where $m$ is the maximum of the numbers of negations and disjunctions. Each disjunct

$$\bigwedge_j u_{ij} \xrightarrow{I_{ij}} v_{ij} \tag{3}$$

is logically conjoined with the constructive normal form of *true*. Only those (constructive) disjuncts of *true* that are consistent with all conjuncts $u_{ij} \xrightarrow{I_{ij}} v_{ij}$ are collected to form the constructive normal form of (3). The constructive normal form of (2) is the disjunction of the constructive normal forms obtained above. This phase is estimated to have $O(2^{n+m})$ steps.

If the formula is $\exists x \cdot P$, with a single is one outermost existential quantifier, then Law 5 can be used to eliminate the quantifier over the constructive normal form of $P$. Thus the overall reduction takes $O(2^{n+m})$ steps where $m$ is the maximum of the numbers of negations, disjunctions and existential quantifiers.

A property in the normal form is valid if it is semantically equal to *true*. To check its validity, we conjoin the property with the canonical normal form of *true* in which the intersection between every pair of distinct disjuncts is false, reflecting distinct relative positions of variables. The disjuncts of the given normal form are grouped by their relative positions, with their distance intervals combined by union. The given normal form is valid if and only if for every position layout, the union renders every non-zero distance $1^+$ (like *true* itself). This validity checking takes $O(2^n)$ steps. □

**Law 3** (1) $\ell(\sim P) = \sim\ell(P) \wedge \ell(\top)$
(2) $\ell(P_1 \vee P_2) = \ell(P_1) \vee \ell(P_2)$
(3) $\ell(P_1 \wedge P_2) = \ell(P_1) \wedge \ell(P_2)$.

**Proof.** Notice that $\#_a$ and $\ll_{\mathbf{X}}$ are fuds. Let $f(P) := (P \;\#_a\; \top) \ll_{\mathbf{X}} \top$. According to Law 2(1), we

have $f(\sim P) = \sim f(P)$. Then by Law 2(2), we have

$$\ell(P) \;=\; f(P) \wedge \ell(\top).$$

Again Law 2(2) guarantees $f(P_1 \wedge P_2) = f(P_1) \wedge f(P_2)$, which leads to $\ell(P_1 \wedge P_2) = \ell(P_1) \wedge \ell(P_2)$ as well as Law 3(1). □

**Theorem 2** For any program, the parallel-list verification terminates in finitely-many steps.

**Proof outline.** Law 1 guarantees that all validity proof obligations terminate. Other operators such as $|b|$ and $P\!\uparrow$ have finite evaluation. The widening operator forces static verification of any loop to terminate in 3 iterations. The overall verification complexity is estimated to be $O(m \cdot 2^n)$ where $n$ is the number of variables, and $m$ is the length of the code. □

# References


[1] J. Berdine, C. Calcagno and P. W. O'Hearn. A decidable fragment of separation logic. In *FSTTCS*, 97–109, 2004.

[2] P. Borba, A. Sampaio and M. Cornélio. A refinement algebra for object-oriented programming. *ECOOP 2003*, 457–482, 2003.

[3] P. Borba, A. Sampaio, A. Cavalcanti and M.,Cornélio. Algebraic reasoning for object-oriented programming. *Science of Computer Programmming*, **52**:53–100, 2004.

[4] C. Calcagno, P. Gardner and M. Hague. From separation logic to first-order logic. In *FoSSaCS*, 395–409, 2005.

[5] C. Calcagno, H. Yang and P. W. O'Hearn. Computability and complexity results for a spatial assertion language for data structures. In *APLAS*, 289–300, 2001.

[6] A. L. C. Cavalcanti and D. A. Naumann. A weakest precondition semantics for renement of object-oriented programs. *IEEE Transactions on Software Engineering*, **26**(8):713–728, 2000.

[7] B. Y. E. Chang and X. Rival. Relational inductive shape analysis. In *POPL '08: Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 247–260, ACM, 2008.

[8] Y. Chen and J. W. Sanders. Logic of global synchrony. *ACM Transactions on Programming Languages and Systems*, **26**(2):221–262, 2004.

[9] Y. Chen and J. W. Sanders. Compositional reasoning for pointer structures. In *8th International Conference on Mathematics of Program Construction (MPC'06)*, *LNCS* **4014**:115–139. Springer, 2006.

[10] Y. Chen and J. W. Sanders. A pointer logic for object diagrams. *UNU-IIST Technical Report* **379**, July, 2007.

[11] Y. Chen. Seminar presented at UNU-IIST, Macao, 11 April, 2008.

[12] B. Courcelle. Graph decompositions definable in monadic second-order logic. *Electronic Notes in Discrete Mathematics*, 7th International Colloquium on Graph Theory, **22**(15):13–19, 2005.

[13] A. Dawar, P. Gardiner and G. Ghelli. Expressiveness and complexity of graph logic. *Information and Computation*, **205**:263–310, 2006.

[14] D. Distefano, P. W. O'Hearn and H. Yang. A local shape analysis based on separation logic. In *TACAS*, 287–302, 2006.

[15] D. Gries. The Schorr-Waite graph marking algorithm. *Acta Informatica*, **11**:223–232, 1979.

[16] W. Harwood, A. Cavalcanti and J. Woodcock. A theory of pointers for the utp. In *ICTAC*, 141–155, 2008.

[17] C. A. R. Hoare and J. He. *Unifying Theories of Programming*. Prentice Hall, 1998.

[18] C. A. R. Hoare and J. He. A trace model for pointers and objects. In *ECOOP'99*, *LNCS* **1628**:1–17, 1999.

[19] C. A. R. Hoare. The verifying compiler: A grand challenge for computing research. *Journal of the ACM*, **50**(1):63–69, 2003.

[20] V. Kuncak and M. C. Rinard. On spatial conjunction as second-order logic. *MIT CSAIL Technical Report* **970**, October, 2004.

[21] X. Liu, Z. Liu and L. Zhao. Object-oriented structure renement A graph transformational approach. *UNU-IIST Technical Report* **340**, July 2006.

[22] A. Møller and M. I. Schwartzbach. The pointer assertion logic engine. In *Programming Language Design and Implementation'01*, 221–231, 2001.

[23] D. A. Naumann. Predicate transformer semantics of a higher order imperative language with record subtypes. *Science of Computer Programming*, **41**(1):1-51, 2001.

[24] P. W. O'Hearn, J. C. Reynolds and H. Yang. Separation and information hiding. In *POPL'04*, **2142**:268–280, ACM, 2004.

[25] R. F. Paige and J. S. Ostroff. Erc: an object-oriented renement calculus for Eiffel. *Formal Aspects of Computing*, **16**(1):51-79, April 2004.

[26] Z. Rakamarić, J. Bingham and A. J. Hu. An inference-rule-based decision procedure for verication of heap-manipulating programs with mutable data and cyclic data structures. In B. Cook and A. Podelski (Editors): *VMCAI 2007*, LNCS **4349**:106-121, Springer Verlag, 2007.

[27] J. C. Reynolds. Separation logic: a logic for shared mutable data structures. In *LICS'02*, 55–74, IEEE Computer Society, 2002.

[28] A. Sampaio and P. Borba. Transformation laws for sequential object-oriented programming. *PSSE 2004*, 18–63, 2004.

[29] T. L. V. L. Santos, A. Cavalcanti and A. Sampaio. Object-orientation in the UTP. *UTP 2006*, 18–37, 2006.

[30] L. Silva, A. Sampaio and Z. Liu. Laws of object-orientation with reference semantics. In *SEFM 2008*, 217–226, 2008.

[31] E.-J. Sims. Extending separation logic with fixpoints and postponed substitution. *Theoretical Computer Science*, **351**(2):258–275, 2006.

[32] M. A. Smith and J. Gibbons. Unifying theories of objects. In Jim Davies and Jeremy Gibbons, editors, Integrated Formal Methods, *LNCS* **4591**:599–618, Springer-Verlag, 2007.

[33] M. A. Smith and J. Gibbons. Unifying theories of locations. In A. Butterfield (editor), *Unifying Theories of Programming*, September, Dublin, 2008.

[34] W. Ke, Z. Liu, S. Wang and L. Zhao. A graph-based operational semantics of OO programs. May, 2009.