



The United Nations  
University

**UNU-IIST**

International Institute for  
Software Technology

---

# A Pointer Logic for Object Diagrams

---

**Yifeng Chen and J W Sanders**

July 2007

## 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



The United Nations  
University

**UNU-IIST**

International Institute for  
Software Technology

P.O. Box 3058

Macau

---

# A Pointer Logic for Object Diagrams

---

**Yifeng Chen and J W Sanders**

## **Abstract**

Compositional reasoning about pointers and references is crucial to verification of contemporary software. This paper introduces a pointer logic that extends Separation Logic with a fixpoint operator and new compositions different from separating conjunction. Higher level of abstraction can be achieved if the right compositions are used in the right places. In particular, if a relation is a ‘full unique decomposition’ then the corresponding composition decomposes any given graph uniquely into two parts; an example is the separation between the non-garbage and garbage parts of memory. The logic is proved to be largely satisfaction-decidable in the sense that there is a finite procedure to determine whether or not a program state satisfies a formula. The main technical result of the paper is that if only full unique decompositions are used in compositions, then a rather general fragment becomes validity-decidable. The logic is axiomatized and, with the help of an infinitary inference rule, proved to be complete. Separation Logic without pointer arithmetic is shown to be a fragment of the logic.

**Yifeng Chen** is a lecturer in Computer Science at the University of Durham, England, with interests in imperative, parallel and object-oriented programming languages, including design, translation, static analysis, semantics, specifications and their support for decentralised software development.

**Jeff Sanders** is Principal Research Fellow at UNU-IIST, having recently joined from the Programming Research Group at Oxford. His interests lie largely in Formal Methods.

---

## Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Object diagram and heap representation</b>	<b>4</b>
<b>3</b>	<b>The pointer logic</b>	<b>7</b>
3.1	Syntax . . . . .	7
3.2	Semantics . . . . .	8
<b>4</b>	<b>Pragmatics</b>	<b>9</b>
4.1	More syntactical conventions . . . . .	9
4.2	Useful binary relations . . . . .	10
4.3	Reasoning about object diagrams . . . . .	12
4.4	Full unique decompositions . . . . .	14
<b>5</b>	<b>Separation Logic as a fragment</b>	<b>15</b>
<b>6</b>	<b>Decidability</b>	<b>16</b>
6.1	Defining decidability and normal-form reduction . . . . .	16
6.2	Satisfaction decidability . . . . .	17
6.3	A basic validity-decidable fragment . . . . .	20
6.4	Handling free variables . . . . .	22
6.5	An advanced validity-decidable fragment . . . . .	24
<b>7</b>	<b>Axiomatization and completeness</b>	<b>27</b>
<b>8</b>	<b>Conclusions</b>	<b>29</b>



## 1 Introduction

Separation Logic (SL) [25, 26, 28] is currently state-of-the-art in pointer analysis. Instead of handling individual pointer structures as program states like [20], SL supports reasoning about sets of pointer structures (*i.e.* *heaps*). Separating conjunction, like many parallel compositions [9, 10, 17, 19], combines two sets of heaps by merging individual heaps, provided they are mergeable and pass some consistency check for domain disjointness. Many interesting properties can be specified in SL and reasoned about by embedding SL in Hoare Logic. Future developments facing SL include lifting reasoning to higher level of abstraction, designing general but decidable fragments of SL and identifying complete axiom systems.

In this paper, we propose a pointer logic that contains merge and relational bond (filling a role like SL's separating conjunction), quantifiers, fixpoints (*e.g.* for expressing reachability) and projection (similar to separating implication). SL without pointer arithmetic then becomes a sub-logical fragment of the new logic (see Section 5).

Each property describes a set of object diagrams (under some name assignment of variables, similar to *store* in SL). An object diagram is a snapshot of the object diagram at a particular time. In this paper, we simply represent an object diagram as a graph, *i.e.* a finite set of edges. Each edge is a tuple of a source name, a label name and a target name. The source can be viewed as the “address” of an object, the label as an attribute of the object, and the target as the object (or constant value) stored for that attribute. Such graph-based representation reflects the view of higher-level OO languages such as Java. In terms of expressiveness, object diagrams and heap representation are essentially equivalent. Section 2 compares the two representation methods in more detail. Three main contributions result from the design of this logic.

Firstly, the logic allows creation of new compositions other than separating conjunction between properties. In some sense, it appears like a “dynamic SL”. Binary relations can be defined with connectives and quantifiers and used to check consistency between object diagrams, which are mergeable to form bigger graphs if they pass consistency checking. This means abstract properties without free variables can be combined to form a more advanced abstract property with the help of user-defined compositions. Section 4 shows an example in which two abstract distinct acyclic edges *without free variables* can be composed to form a loop using a composition that can not be replaced by separating conjunction unless the acyclic edges keep their source, label and target observable as free variables. This bears some similarity to the distinction between (binary) relational semantics and operational semantics of sequential programs. In relational semantics, hiding happens immediately after every sequential composition, leaving no observable intermediate state, while operational semantics leaves the hiding of all intermediate states to the end, achieving a lower level of abstraction.

More importantly, we are now able to create compositions that satisfy some desirable properties. The concept of *unique decomposition* has been introduced by Chen and Sanders [11]. An important operation in compositional reasoning of pointers is merge, *i.e.* the set union of two graphs' edges (or set union of partial-function heaps). In general, there is implicit internal choice made by a merge: to obtain a set  $A$  by union  $B \cup C$ , we may choose various pairs  $B$  and  $C$ .

Such a merge operator distributes over disjunction but not conjunction. Suppose that a binary relation is designed so that every graph  $A$  that is mergeable from two pairs  $(B_1, C_1)$  and  $(B_2, C_2)$  of related graphs (*i.e.*  $A = B_1 \cup C_1 = B_2 \cup C_2$ ) must satisfy  $B_1 = B_2$  and  $C_1 = C_2$ . In other words, the decomposition of every graph into sub-graphs related by the relation is unique. Then the relation is called a unique decomposition. It has been shown that this condition corresponds to conjunctivity of the composition [11]. A typical example is the separation between non-garbage and garbage parts of the memory. For any program state, the non-garbage part includes all memory cells reachable from some root access point of the memory, leaving the garbage part uniquely fixed. In this paper we mostly use a variant notion called *full unique decomposition* (or fud for short): a unique decomposition is a fud iff every graph can be uniquely decomposed into related sub-graphs.

Any unique decomposition can be converted into a fud; and negation distributes into a fud. A composition that merges graphs after checking their consistency with a fud is ‘passive’ and ‘transparent’ in the sense that connectives and quantifiers can move freely in or out. A logical fragment using only fuds before merges behaves much better in terms of compositionality and decidability. In particular, we have obtained the following technical results.

1. The logical fragment **S** without projection is satisfaction-decidable in the sense that there is a decision procedure to check whether or not a given graph satisfies a formula. The fragment’s validity problem is semi-decidable: given an invalid input formula, there exists a procedure to identify a counterexample in finitely-many steps.
2. The fragment **K** with atom equality, singleton graph properties and arbitrarily expanded edges closed under logical connectives is validity-decidable.
3. The fragment **T** containing **K** as well as Cartesian product (a special kind of relational bond), fud compositions, quantifiers and projection is validity-decidable. That result, the most important of the paper, assumes that every composition is a merge after consistency checking by a fud.

Note that although SL’s separating conjunction  $*$  is not a fud, it is often used in a context in which it essentially corresponds to a fud. Such predicates are known as ‘precise predicates’ [26]. For example, if the property  $P$  of a separating conjunction  $P * Q$  is a fixed singleton property (*i.e.* allowing only one heap), then the combination of  $P$  and  $*$  corresponds to a fud. The decidability of **T** can be obtained if all compositions are fud-based (see Section 6). Graph decompositions are also discussed extensively in various graph logics [12] and graph theory [13]. This paper focuses on user-created unique decompositions at the level of graph properties.

The final major contribution is a *complete axiomatization* of the logic. We are able to obtain completeness because of three characteristics of the logic. The first is that the most basic composition of our pointer logic is no longer separating conjunction but the merge operator without any consistency checking. Any graph can be represented syntactically as the merge of some edges. Such a singleton property is never false, even if we try to merge the same edge with itself (while separating conjunction of the same cell would lead to inconsistency false in SL). This makes such singleton graphs ideal candidates for *nominals*. Modal logics with nominals are called hybrid logics [2] whose fundamental theory was developed by

Sofia School in the 1980's [15]. Nominals are syntactical denotations of single-semantic possible worlds in modal logic. Logics using nominals behave much better for completeness. This would be equivalent to directly incorporating heaps inside SL.

The first decidability result for satisfaction shows that there is a decision procedure to determine whether or not a given graph satisfies a given formula. We simply need to mimic what this decision procedure does with axioms; then we will be able to prove in finitely-many steps whether or not a graph nominal implies a given formula. If a formula is semantically valid, then satisfaction for every graph nominal can be proved using an infinitary inference rule.

The closest to our results are of the decidable fragments of SL [6, 7]. Both do not allow quantifiers. Berdine et al. [4] have studied a restrictive decidable fragment with linked lists but without disjunction or quantifiers.

Monadic Second-Order Logics [14] use the simple merge operator without consistency checking and allow quantifiers over graphs but do not permit creation of new compositions. The decidability results for monadic second-order graph logic include low complexity associated with roughly tree-like behavior, and high complexity with the containment of large grids [16]. Specific shape, and sometimes data-dependent, properties of mutable data structures, however, have been captured by a corpus of material using first-order logic. We mention the decidable logic for a restricted family of structures (lists and trees) by the BRICS group [21]; the decidable logic  $L_r$  of reachable expressions in arbitrary *individual* graphs (instead of composition of graph properties) by Benedikt *et al* [3]; the SL-influenced decidable spatial logic by Calcagno *et al* [5]; Graph Logic [8] with one merge operator; and the decidable logic TLL expressing reachability in singly-linked lists by Ranise and Zarba [27] without separating conjunction (and hence requiring more inequalities to avoid disjointness and circles).

Work that is similarly second-order, and also for graph types, is PALE (the pointer assertion logic engine) by the BRICS group [24]. Structural invariants are described in a pointer assertion language PAL, a monadic second-order logic, and verified with PALE using the tool MONA to evaluate assertions in Hoare logic. PALE is appropriate for a moderately intermediate level of abstraction which still requires explicitly-stated loop invariants. The quite different TVLA [23], on the other hand, executes fixpoint iterations rather than capturing programs in PAL and so does not require explicit invariants. The more general propositional logic over reachability constraints,  $LRP$ , and its fragment for mutable data structures,  $LRP_2$ , by Yorsh *et al* is also expressive enough to capture loop invariants and some data-type invariants and decides satisfiability using a decision procedure for MSO logic on trees and reduction to normal form.  $LRP_2$  is in some ways complementary to our work, being focused at the pointer level (and extending the language  $L_r$ ).

Kuncak and Rinard [22] also point out that many spatial conjunctions are possible and have established an equivalence between first-order logic with a particular spatial conjunction and full second-order logic that preserves satisfiability of formulae, although concrete compositions like that between non-garbage and garbage are not on the agenda, neither is better decidability result benefiting from choosing compositions with the desirable property (of unique decomposition). An interesting higher-order logic, HTT (for Hoare type theory), has been explored by Birkedal *et al* [1]. It permits quantification over abstract predicates at the level of terms, types and assertions and hence seems an ideal candidate to bridge concepts expressed

in the more specification-oriented notation of the present paper with the more low level concerns of the first-order work.

Comparison with further work is made as appropriate through the paper.

## 2 Object diagram and heap representation

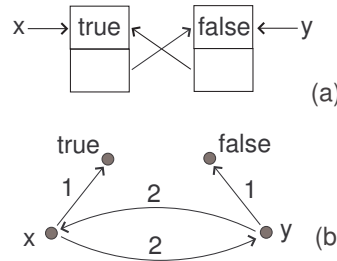
In UML, an object diagram is a snapshot of the object structure at a particular time. In this paper, we simply represent an object diagram as a graph, *i.e.* a finite set of edges. Each edge is a tuple consisting of a source name (an object's address), a label name (an attribute) and a target name (the value stored for that attribute). Such a graph-based representation reflects the view of states taken by higher-level OO languages such as Java.

Heap representation, on the other hand, reflects the language C's view of states. A heap is a (finite) partial function from an (integer) address space to values. The domain of such a function represents the allocated memory cells and the function maps each allocated address to a constant value or another address. An object is stored in a heap using pointer arithmetic: the first attribute is stored at an address  $x$  and other attributes are then stored at  $x+1$ ,  $x+2$ ,  $\dots$ . The value stored for each attribute of an object is unique. SL's separating conjunction  $*$  merges (*i.e.* with the union of partial functions) two heaps into a bigger heap if they are domain disjoint so that the merged heap still maintains the uniqueness of value at each address. Note that SL's store is simply an assignment of program variables to constants and addresses in the heap.

In some sense, the heap representation reflects the memory model after compilation when the attribute names of objects are transformed into  $+0, +1, +2, \dots$ , while a graph representation reflects the memory model of higher-level languages like Java before compilation (or at UML level during software development).

Despite the significant superficial difference in style, in expressiveness the object-diagram representation is essentially equivalent to the heap representation; neither is more abstract than the other. For example any heap can be transformed into an object diagram whose labels are always  $1, 2, \dots, n$  and the edges are *deterministic* (a term derived from graph representation of labeled-transition systems) in the sense that if two edges in a graph share the same source and label, then their targets differ.

The following figures compare the object diagram and heap of objects  $x$  and  $y$ .



Each object has two attributes. The first attribute of object  $x$  is the constant boolean value true and the second attribute is the object  $y$ . The object  $y$  instead stores false and the object  $x$ . This object structure is represented as a graph:

$$\{(x, 1, \text{true}), (x, 2, y), (y, 1, \text{false}), (y, 2, x)\}.$$

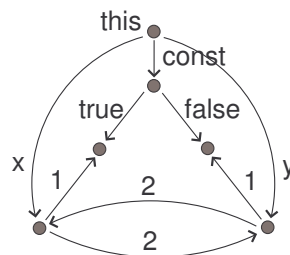
Two graphs can be merged into a bigger graph. Such merge is different from SL's separating conjunction as it does not include domain-disjointness. Any two graphs are mergeable using set union. Some other logics such as Graph Logic [8] use multi-sets instead.

To define merges that include consistency checking (like domain disjointness), we need binary relations between graphs. For example, the following relates two graphs iff they are disjoint at source or label:

$$\{(A, B) \mid \forall xy z_1 z_2 \cdot (x, y, z_1) \in A \wedge (x, y, z_2) \in B \rightarrow z_1 \neq z_2\}.$$

This is exactly SL's domain disjointness relation that ensures consistency (*i.e.* determinacy) of the union of two deterministic object diagrams.

In a more advanced representation of the same example, we can almost entirely eliminate the role of naming for sources and targets and use only the labels and the structure to represent all the information. The root 'this' is the only identifiable name indicating the point of access for the whole memory. Global variables (as attributes of the root) are immediate edges from the root. The label 'const' marks a special edge that is linked to all constant values. Each constant becomes the label of an edge. The value stored at an attribute of an object is a specific constant iff the edge for that attribute shares the same target with the edge of the constant. For example, we no longer need to distinguish constant values and addresses; they are already distinguished by structure. Such a representation is simpler to handle from a theoretical point of view and convenient for OO program semantics in which unboundedly many program variables may be needed in invoking recursive methods.



The set theory of object diagrams and their operators has been explored in [11] to which this paper is related. Note that if we regard  $x$ ,  $y$  and  $\text{const}$  as variables, the root ‘this’ can be removed too.

On the other hand, the two representation methods are indeed different in style. The heap representation naturally includes pointer arithmetic, while names in an object diagram are naturally independent (either equal or not equal). This does not mean that the heap must have pointer arithmetic. Indeed, many fragments with reasonable decidability results are free of pointer arithmetic [6, 7] (*i.e.* by assuming that each object has exactly two values and only the starting address of the object is accessible). Neither does it mean that object diagrams cannot have arithmetic. Arithmetic relations (*e.g.* either Peano or Presburger arithmetics) between source and target names (representing addresses) can always be added to the graph representation later on.

Caution is needed when incorporating pointer arithmetics, since they confer unequal status on addresses. Suppose that we store objects of varied lengths in memory. Then *non-deterministic* allocation (an assumption of most formalisms including SL) of memory cells for a new object becomes problematic. Where do we allocate the starting address of the object? If it is four cells down the first available address, then henceforth those four cells will be unable to be used for a larger object. Thus all unallocated addresses are distinct. The decision to choose a fresh address bears consequences for later allocation. This can be further complicated by object release and modern languages’ internal memory operation that moves objects around to vacate enough room for a new object when memory almost runs up. In summary, the concrete addresses of objects should not be subject to observation and may indeed be modified unexpectedly by the language system.

Another serious problem with relying on pointer arithmetic for representing objects of different sizes is that it makes quantifiers undecidable. Consider the statement that ‘there exists an address  $x$  from which the numbers 7, 8 and 9 are stored’. A decision procedure that analyzes the validity of a formula containing the above statement, must test it against every free address, as all free addresses are different. If the address space is infinite (a reasonable assumption of SL, as the memory bound is hard to predict), then the procedure would not terminate.

This will not be a problem in object diagrams in which all names naturally have equal status, and no name is better or worse than any other. That means in a validity decision procedure, a universal (or existential) quantifier can be reduced to a finite conjunction (or disjunction) of substitutions with all existing names as well as a fresh name. All fresh names have equal status. Finding a fresh name is the same as creating a new object. Testing something with a particular fresh name has no essential difference from testing it with another fresh name. That is exactly the reason quantifiers over an infinite set of names (*i.e.* the address space) are decidable in this paper but not in SL. That means *objects of varied sizes are not a source of undecidability*. This becomes apparent only in the object-diagram representation. After pointer arithmetic (*i.e.* name arithmetic in graphs) is added, undecidability of the arithmetic may still arise; but we are able to isolate the source of undecidability to the arithmetic itself (rather than blaming the pointer logic).

### 3 The pointer logic

#### 3.1 Syntax

Let  $\mathbf{N}$  be a countable infinite set of names and  $\mathbf{X}$  be a countable infinite set (disjoint with  $\mathbf{N}$ ) of variables, so that fresh names and variables are always available to a formula. The pointer logic has the following syntax:

$$\begin{aligned} \mathbf{N} &::= a \mid b \mid c \mid a_1, \mid \dots \\ \mathbf{X} &::= x \mid y \mid z \mid x_1 \mid \dots \\ \mathbf{A} &::= \mathbf{N} \mid \mathbf{X} \\ \mathbf{R} &::= \top^n \mid \emptyset \mid (\mathbf{A}, \mathbf{A}, \mathbf{A}) \mid \mathbf{A} = \mathbf{A} \mid \neg \mathbf{R} \mid \mathbf{R} \vee \mathbf{R} \\ &\quad \mid [\mathbf{R}] \mid \mathbf{R}(\mathbf{R}, \dots, \mathbf{R}) \mid \exists \mathbf{X} \cdot \mathbf{R} \mid \mathbf{R} \infty \mathbf{R} \mid \mathbf{R} @ i \end{aligned}$$

where  $\mathbf{A}$  is the set of atoms (each as either a name or a variable), and  $\mathbf{R}$  is the set of relations (among graphs). We use  $u, v, w, u_1, \dots$  to denote individual atoms,  $R, R_1, \dots$  to denote individual relations and  $P, P_1, \dots$  to denote properties (*i.e.* unary relations). A subset of  $\mathbf{R}$  is also called a *fragment*. For example,  $\overline{\mathbf{R}}$  denotes the fragment of relations without free variables.

Negation, conjunction, and existential quantification have the standard meanings.  $\top^n$  denotes the  $n$ -ary full relation. The empty graph is denoted  $\emptyset$ . Each edge  $(u, v, w)$  is a triple of atoms including a source  $u$ , a label  $v$  and a target  $w$ . Each atom equality  $(v = u)$  is either a true property satisfied by all graphs or a false property satisfied by none, depending on whether  $v$  is equal to  $u$ . Every relation can be viewed as a relation among graphs (under some name assignment to variables). The merge operator  $[\mathbf{R}]$  merges the graphs related by the relation into a larger graph as a set union. Relational bond  $R(R_1, \dots, R_m)$  is a certain kind of relational composition based on the merge operator. The result of a relational bond is a relation with more arguments. The first arguments of the bond are actually the arguments of  $R_1$ . The merge (*i.e.* set union) of the graphs related by  $R_1$  will become the first argument of  $R$ . All (merged graph) arguments of  $R$  must be related by  $R$ . Thus the whole relational bond is a relation among arguments of all  $R_i$  whose merged graphs satisfy  $R$ . For example,  $\top^2(P_1, P_2)$  corresponds to the Cartesian product of two properties, yielding a binary relation. The fixpoint  $R_1 \infty R_2$  expands a relation  $R_1$  recursively with another relation  $R_2$ . We assume that the first relation is a property while the second is a binary relation. Projection  $R @ i$  projects a relation to its  $i$ -th argument to form a property where  $i$  is bounded by the arity of the relation.

Let  $|R|$  denote the arity of a relation. We assume that both arguments in a conjunction have matching arity. In addition, the first argument of the fixpoint is a property, and its second is a binary relation. The arity of a relation can be identified by  $|\emptyset| = |(u, v, w)| = |v = u| = |[R]| = |P \infty R| = |R @ i| = 1$ ,  $|\top^n| = n$ ,  $|\neg R| = |\exists x \cdot R| = |R|$ ,  $|R(R_1, \dots, R_m)| = \sum_i^m |R_i|$  and  $|R_1 \vee R_2| = |R_1| = |R_2|$ .

Substitution  $R[v/u]$  replaces every free occurrence of  $u$  with  $v$  (avoiding clashes). Note that  $u$  (or  $v$ ) can be either a variable or a name. Substitution is also recursively definable. We use  $\alpha(R)$  to denote the set of names and  $\phi(R)$  to denote the set of free variables. Both are recursively definable. A relation is *variable-less* if  $\phi(R) = \{ \}$ . We use overlined symbols *e.g.*  $\overline{R}, \overline{R}_1, \dots$  to denote variable-less relations. In

the following sections, we shall discuss several fragments:

---

<b>G</b>	fragment of all singleton graph properties
<b>S</b>	satisfacton-decidable fragment without projection
<b>K</b>	basic validity-decidable fragment for properties
<b>F</b>	full unique decompositions in <b>T</b>
<b>T</b>	advanced validity-decidable fragment for relations.

---

The simple syntax of the pointer logic is designed for the convenience of semantic studies. In real applications we need more syntactical conventions, including the standard definitions of conjunction  $\wedge$ , implication  $\rightarrow$ , equivalence  $\leftrightarrow$  and universal quantification  $\forall$ .

### 3.2 Semantics

Let the set of all edges be  $\mathbf{Edge} \hat{=} (\mathbf{N} \times \mathbf{N} \times \mathbf{N})$ , the set of all (semantical) graphs be  $\mathbf{Graph} \hat{=} \wp^{fin}(\mathbf{Edge})$ , and the set of all  $n$ -ary graph relations be  $\mathbf{Relation}_n \hat{=} \wp(\mathbf{Graph}^n)$ . Under some name assignment to variables, the semantics of a relation formula is a set of tuples of graphs. Each graph is a finite set of edges. Each edge is a triple of names. An assignment is denoted by a mapping  $\varepsilon: \mathbf{A} \rightarrow \mathbf{N}$  from atoms to names, which we assume preserves names:  $\varepsilon(a) = a$ . We write  $d \models_{\varepsilon} R$  if a tuple  $d$  of graphs satisfies a relation  $R$  under name assignment  $\varepsilon$ . The  $i$ -th graph of a tuple  $d$  is denoted  $(d)_i$ . We assume that tuples are mergeable:  $(A_1, (A_2, A_3)) = (A_1, A_2, A_3)$ .

#### Definition 1 (Semantics)

$$d \models_{\varepsilon} \top^n.$$

$$A \models_{\varepsilon} \emptyset \text{ iff } A = \{ \}.$$

$$A \models_{\varepsilon} (u, v, w) \text{ iff } A = \{ (\varepsilon(u), \varepsilon(v), \varepsilon(w)) \}.$$

$$A \models_{\varepsilon} (v = u) \text{ iff } \varepsilon(v) = \varepsilon(u).$$

$$d \models_{\varepsilon} \neg R \text{ iff } d \not\models_{\varepsilon} R.$$

$$d \models_{\varepsilon} R_1 \vee R_2 \text{ iff } d \models_{\varepsilon} R_1 \text{ or } d \models_{\varepsilon} R_2.$$

$$A \models_{\varepsilon} [R] \text{ iff there exist } A_1, \dots, A_n \text{ such that } A = \bigcup_i^n A_i \text{ and } (A_1, \dots, A_n) \models_{\varepsilon} R.$$

$$d \models_{\varepsilon} R(R_1, \dots, R_m) \text{ iff for any } i \leq m \text{ we have } d_i \models_{\varepsilon} R_i \text{ and } (\bigcup_j^{|R_1|} (d_1)_j, \dots, \bigcup_j^{|R_m|} (d_m)_j) \models_{\varepsilon} R \text{ where } d = (d_1, \dots, d_m).$$

$$d \models_{\varepsilon} \exists x \cdot R \text{ iff there exists } a \in \mathbf{N} \text{ such that } d \models_{\varepsilon \dagger \{x \rightarrow a\}} R \text{ wherein the assignment } \varepsilon \text{ is overwritten to map } x \text{ to } a.$$

$d \models_{\varepsilon} P \infty R$  iff there exists  $k$  such that  $d \models_{\varepsilon} P \infty_k R$  where  $P \infty_0 R \hat{=} P$  and  $P \infty_{k+1} R \hat{=} P \infty_k R \vee [R(P \infty_k R, \top)]$ .

$A \models_{\varepsilon} R @ i$  iff there exists  $d$  such that  $A = (d)_i$  and  $d \models_{\varepsilon} R$ .

The empty graph  $\emptyset$  is a singleton property satisfied by only the empty set (of edges). A syntactical edge is a singleton edge property under the name assignment. Atom equality also depends on name assignment. A graph satisfies a merge if it is the union of graphs related by the relation. The fixpoint operator repeatedly extends a property with a binary relation. Any graph satisfying the fixpoint must satisfy some finite iteration (see Law 2(5)). A graph satisfies a projection if it is among the graphs of a tuple that satisfies the relation. Separating implication can be defined as a projected relational bond (see Section 5). If a relation  $\bar{P}$  has no free variables, then we write  $A \models \bar{P}$  for  $A \models_{\varepsilon} \bar{P}$ , as the name assignment no longer has any effect.

Two relations are *semantically equal*, denoted  $R_1 \equiv R_2$ , when  $d \models_{\varepsilon} R_1$  iff  $d \models_{\varepsilon} R_2$ . An  $n$ -ary relation  $R$  is *valid*, denoted  $\models R$ , if  $d \models_{\varepsilon} R$  holds for any assignment  $\varepsilon$  and  $n$ -tuple  $d$  of graphs. Two relations are *equivolid*, denoted  $R_1 \equiv_{\delta} R_2$ , if they are either both valid or both invalid.

## 4 Pragmatics

### 4.1 More syntactical conventions

Let  $\top \hat{=} \top^n$  be the true property, let  $\perp \hat{=} \neg \top$  be the false property and let  $\perp^n \hat{=} \neg \top^n$  be the  $n$ -ary empty relation. The (syntactical) Cartesian product  $P \times Q \hat{=} \top^2(P, Q)$  creates a binary relation from two properties. We use a shorthand  $P : Q \hat{=} [P \times Q]$  to denote the merge of two properties (without consistency checking).

A (syntactical) graph is the merge of a finite number of (syntactical) edges. A singleton graph property is a property allowing only one possible graph (just like a singleton set). Syntactical graphs form the fragment

$$\mathbf{G} ::= \emptyset \mid (\mathbf{A}, \mathbf{A}, \mathbf{A}) \mid \mathbf{G} : \mathbf{G}.$$

We use  $G, H, G_1, \dots$  to denote individual graph properties. For example, the property

$$(a_1, b_1, c_1) : (a_2, b_2, c_2)$$

allows only a graph containing exactly the two edges. Any two graphs can be merged (just like with set union). Unlike SL, there is no disjointness restriction; thus the property  $(a, b, c) : (a, b, c)$  is semantically equal to  $(a, b, c)$ . Graph properties are never false, which allows them to be the nominals of the logic. We use  $\bar{\mathbf{G}}$  to denote the fragment of variable-less graphs and  $\gamma : \bar{\mathbf{G}} \rightarrow \mathbf{Graph}$  to extract the semantical graph from each syntactical one:  $\gamma(\emptyset) \hat{=} \{ \}$ ,  $\gamma((a, b, c)) \hat{=} \{(a, b, c)\}$ , and  $\gamma(\bar{G}_1 : \bar{G}_2) \hat{=} \gamma(\bar{G}_1) \cup \gamma(\bar{G}_2)$ .

Let  $(u, v, w)^{\circ} \hat{=} [(u, v, w) \times \top]$  denote the property that allows just graphs containing the edge  $(u, v, w)$ . For example, the property  $(a_1, b_1, c_1)^{\circ} \wedge (a_2, b_2, c_2)^{\circ}$  allows just graphs containing the two edges. We

use shorthand  $(x, \cdot, y) \hat{=} \exists z \cdot (x, z, y)$  to denote an edge with source  $x$  and target  $y$ ,  $cyc \hat{=} \exists x \cdot (x, \cdot, x)$  to denote a cyclic edge, and

$$acyc \hat{=} (\cdot, \cdot, \cdot) \wedge \neg cyc$$

to denote an acyclic edge. Note that  $(x, y, \cdot)^\circ$  denotes  $\exists z \cdot ((x, y, z)^\circ)$  instead of  $(\exists z \cdot (x, y, z))^\circ$ . When no confusion is caused, we use a convention called *relational composition*,  $P_1 R P_2$ , to denote the merge of a relational bond  $[R(P_1, P_2)]$ . We introduce two adjoint operators [18] involving properties  $P_1$ ,  $P_2$  and a binary relation  $R$ : the right factorization

$$P_1 /_R P_2 \hat{=} (P_1 \wedge (\top R P_2)) @ 1$$

and the left factorization

$$P_1 \backslash_R P_2 \hat{=} (P_1 \wedge (P_2 R \top)) @ 2.$$

If  $R$  is symmetric then the two factorization operators are equal. For example, separating implication  $P_2 \multimap P_1$  is expressed as either  $P_1 /_* P_2$  or  $P_1 \backslash_* P_2$ .

## 4.2 Useful binary relations

Why do we want to create various binary relations between graphs? There are 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 and stronger decidability results (refer to Section 6).

We begin with some useful binary relations defined with quantifiers. For example, the following relation represents source or label disjointness:

$$* \hat{=} \neg \exists xy \cdot (x, y, \cdot)^\circ \times (x, y, \cdot)^\circ.$$

Two graphs are related by  $*$  if they do not have identical source and label. The relation corresponds to the domain-disjointness checking of SL's heap representation. If the target of every edge is unique for given source and label (*i.e.* determinism, a property definable within the logic, see Section 4.3), then SL's domain disjointness also coincides with edge disjointness:

$$\diamond \hat{=} \neg \exists xyz \cdot (x, y, z)^\circ \times (x, y, z)^\circ.$$

Two graphs are related by  $\diamond$  if they have no common edge.

Many more interesting relations are definable. For example, two graphs are related by

$$\ominus \hat{=} \neg \exists x \cdot (x, \cdot, \cdot)^\circ \times (\cdot, \cdot, x)^\circ$$

if the first graph has no source as a target of the second. The following table lists some useful composi-

tions.

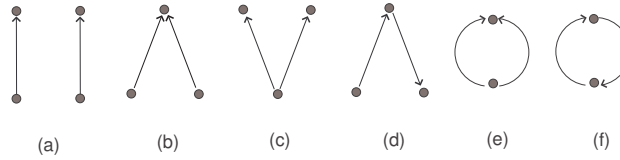
$\neg\exists x.(\times)$	$(x, \cdot, \cdot)^\circ$	$(\cdot, \cdot, x)^\circ$	$\neg(x, \cdot, \cdot)^\circ$	$\neg(\cdot, \cdot, x)^\circ$
$(x, \cdot, \cdot)^\circ$	$\circlearrowleft$	$\circlearrowright$	$\overleftarrow{\nabla}$	$\overrightarrow{\triangleleft}$
$(\cdot, \cdot, x)^\circ$	$\circlearrowright$	$\circlearrowleft$	$\overrightarrow{\triangleright}$	$\overleftarrow{\triangleleft}$
$\neg(x, \cdot, \cdot)^\circ$	$\overleftarrow{\nabla}$	$\overrightarrow{\triangleright}$	$\perp^2$	$\perp^2$
$\neg(\cdot, \cdot, x)^\circ$	$\overrightarrow{\triangleright}$	$\overleftarrow{\triangleleft}$	$\perp^2$	$\perp^2$

The following table lists different combinations of the above binary relations. Relations placed together are deemed to be in conjunction.

(a)	(b)	(c)	(d)	(e)	(f)
$\circlearrowleft \circlearrowright \circlearrowleft \circlearrowright$	$\circlearrowleft \overleftarrow{\triangleleft}$	$\circlearrowright \overrightarrow{\triangleright}$	$\circlearrowright \overrightarrow{\triangleright}$	$\circlearrowleft \overleftarrow{\triangleleft} \overrightarrow{\triangleright}$	$\circlearrowright \overrightarrow{\triangleright} \overleftarrow{\triangleleft}$

For example, the composition  $acyc (\overleftarrow{\triangleleft} \overrightarrow{\triangleright}) acyc$  requires that the distinct acyclic edges form a loop, because the source of the right-hand edge must be a target of the left-hand edge, and the target of the right-hand edge must be a source of the left.

The following figure demonstrates that the created binary relations can be used to combine two distinct acyclic edges in all possible layouts.



Here, the property  $acyc$  is abstract and has no free variables. The created binary relation combines such variable-less abstract properties, while the separating conjunction  $*$  of SL is not applicable in this circumstance unless the source, label and target of the edges are observable from the outside as free variables; but that lowers the level of reasoning. The mechanism of creating new binary relations helps promote the level of reasoning by hiding at the right time (just like the denotational semantics of sequential programs hides intermediate program states, hence increasing the level of abstraction of a sequential composition) while operational semantics leaves the hiding to a later point. Of course, in the pointer logic we may choose to make the source, label or target of an acyclic edge observable as free variables.

In practice, we often need to reason about deadends (*i.e.* targets without outgoing edges) and deadheads (*i.e.* sources without incoming edges). For example, to extend a linked list, we may add an edge to the end (or symmetrically to the head) of the list so that the deadend of the list is the same as the deadhead of the edge. The property that a target  $v$  is a deadend can be captured as  $de(v) \hat{=} (\cdot, \cdot, v)^\circ \wedge \neg(v, \cdot, \cdot)^\circ$ . Deadhead  $v$  is represented as  $dh(v) \hat{=} \neg(\cdot, \cdot, v)^\circ \wedge (v, \cdot, \cdot)^\circ$ . The concatenation relation between two linked list segments can be defined as the existence of a common join point as the deadhead of the LHS

and the deadend of the RHS, and the targets of the RHS cannot reach into the sources of the LHS:

$$\bowtie \hat{=} \exists x \cdot \odot (de(x), dh(x)).$$

This binary relation is associative. For example, a linked list of length four is represented

$$acyc \bowtie acyc \bowtie acyc \bowtie acyc.$$

Bracketing is irrelevant here because of associativity.

This definition is arguably more abstract and simpler than the corresponding expression in SL. Because separating conjunction does not insist on the RHS not reaching into the LHS, in SL that property would require ‘enough inequalities’ [4] between address variables in conjunction to prevent edges from forming a circle. 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.

A general property on linked lists is either an empty graph or comprises recursively  $\bowtie$ -concatenated acyclic edges:

$$list \hat{=} \emptyset \vee acyc \infty \bowtie (\top, acyc).$$

This definition is so general that it contains no names or free variables. Obviously, we have  $list \bowtie list = list$ . A linked list from some atom  $v$  is simply defined as  $list \wedge dh(v)$ .

### 4.3 Reasoning about object diagrams

The pointer logic itself does not assume determinism, but determinism can be specified in the logic as a condition for edges from the same source and label to have distinct targets:

$$determinism \hat{=} \forall xyz_1z_2 \cdot (x, y, z_1)^\circ \wedge (x, y, z_2)^\circ \rightarrow \neg(z_1 = z_2).$$

In a context that admits this assumption, we simply add the corresponding property as an axiom (or invariant). The separating conjunction between two deterministic object diagrams is also deterministic:  $determinism * determinism \equiv determinism$ . In the following discussions, we simply assume *determinism* to be an invariant for every state of the program and do not mention it explicitly.

SL has a ‘global rule’ for pointer assignment, reflecting the programming language C’s view of the memory state:

$$\{(x_1, a, \cdot) * P\} \quad x_1.a := x_2 \quad \{(x_1, a, x_2) * P\}. \quad (1)$$

If  $P$  has the free variable  $x_1$  as a source, the rule is then reduced to  $\{\perp\} \quad x_1.a := x_2 \quad \{\perp\}$ , which is still valid. *Unfortunately, this rule fails in Java-like object-oriented languages* with automatic garbage collection. For example, in Java, the virtual machine may non-deterministically call the garbage collector during an assignment. What an assignment in Java actually does is the pointer swing together with a nondeterministic choice between doing nothing and garbage collection:

$$x_1.a := x_2; (\text{skip} \sqcap \text{gc}).$$

The global rule (1) is invalid unless the property  $P$  does not specifically talk about what happens to the part that turns from non-garbage to garbage by the pointer swing. To reveal what exactly happens in Java-like assignments, we need to reason about reachability within the logic.

Reachability requires fixpoints. Unlike SL, here we have more binary relations to choose. That allows us to express the *general* reachability property without having to identify concrete local structures of memory. Any graph of edges reachable from  $v$  can be generated from an edge from  $v$  by repeatedly adding new edges whose sources are the targets of the existing part:

$$reach(v) \hat{=} \emptyset \vee (v, \cdot, \cdot) \overset{\leftarrow}{\triangleright}.$$

The fixpoint part can be comprehended as a universal disjunction:

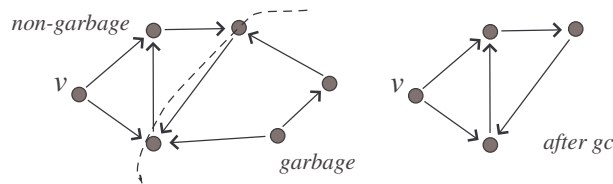
$$(v, \cdot, \cdot) \vee \left( (v, \cdot, \cdot) \overset{\leftarrow}{\triangleright} \top \right) \vee \left( \left( (v, \cdot, \cdot) \overset{\leftarrow}{\triangleright} \top \right) \overset{\leftarrow}{\triangleright} \top \right) \vee \dots$$

This abstract property of reachability is not definable in SL or SL's extension with fixpoints [29], as the relation  $\overset{\leftarrow}{\triangleright}$  is not able to be replaced by  $*$ . Reachability becomes definable in SL only when we know *a priori* the specific local structure of each object, for example, in a binary tree.

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:

$$\ll_v \hat{=} \ominus (reach(v), \neg(v, \cdot, \cdot)^\circ).$$

Note that the relation  $\ll_v$  implies source disjointness  $\ominus$ , as the only source that may not be a target in the  $v$ -reachable non-garbage part is the root  $v$ , which cannot be a source of the garbage part. The following figure shows the separation between the non-garbage part on the left and the garbage part on the right of an object diagram.



If we have several program variables  $x_1, \dots, x_n$ , we may define a relation for reachability from all of them:

$$\ll \hat{=} \ominus (reach(x_1) * \dots * reach(x_n), \bigwedge_i^n \neg(x_i, \cdot, \cdot)^\circ).$$

On the left-hand side, we have merged parts reachable from all program pointers, while the right-hand part does not contain any program pointers as sources. The variables  $x_1, \dots, x_n$  are now free in the relation.

If we choose not to specify the garbage part and assume automatic garbage collection, then we obtain the global rule:

$$\begin{aligned} & \{(x_1, a, \cdot) * (P \ll \top)\} \\ & x_1.a := x_2; (\text{skip} \sqcap \text{gc}) \\ & \{(x_1, a, x_2) * P\} \end{aligned} \quad (2)$$

where  $P$  is an *arbitrary property* (without any syntactical restrictions). The corresponding *frame condition* would allow any arbitrary  $P'$  to be conjoined with  $P$  on both sides simultaneously.

The rule appears pleasingly simple but requires a couple of cases to understand. If a subgraph satisfying  $P$  is reachable from the program variables, then it can be part of a graph that satisfies the precondition, given a non-reachable garbage part after the removal of the edge from  $x_1$  via  $a$ . Note that although we do not represent the garbage part, there is a ‘garbage-to-be part’ that will become garbage if we perform the assignment. In fact, that is exactly the part reachable *only* from the target of the edge from  $x_1$  via  $a$ . The part is specified as  $\top$ . If, on the other hand, a subgraph satisfying  $P$  is not entirely reachable, then the part is ignored (i.e. negated) by  $\ll$ , and the rule still stands.

The original global rule and frame condition of SL would be incorrect here, as the garbage part (the RHS of the composition  $\ll$ ) is removed, and any property about this part before the assignment is not preserved. The only property-preserving part is the one left after removing both the edge from  $x_1$  via  $a$  and the garbage-to-be part. Note that the precondition is subject to arbitrary strengthening, for example, by replacing  $\top$  with any property. Since the object diagram is assumed to be deterministic, the separating conjunction decomposes the graph uniquely, as there is at most one edge from  $x_1$  with label  $a$ . Backward reasoning is still possible using adjoints (Section 4.1):

$$\mathbf{wp}.(x_1.a := x_2; (\text{skip} \sqcap \text{gc})).Q = (x_1, a, \cdot) * (Q /_* (x_1, a, x_2) \ll \top).$$

#### 4.4 Full unique decompositions

The relation  $\ll$  decomposes any graph into a unique pair of disjoint non-garbage subgraph and garbage subgraph, and every graph can be obtained by merging two subgraphs related by the relation (*i.e.* fullness). Formally this notion is defined:

##### Definition 2 (Fud)

1. An  $n$ -ary relation  $R$  is a unique decomposition *if for any assignment  $\varepsilon$  and*

$$A, A_1, \dots, A_n, A'_1, \dots, A'_n \in \mathbf{Graph}$$

*such that  $A = \bigcup_i^n A_n = \bigcup_i^n A'_n$  and  $(A_1, \dots, A_n) \models_\varepsilon R$  and  $(A'_1, \dots, A'_n) \models_\varepsilon R$ , we have  $A_i = A'_i$  (for any  $i \leq n$ ).*

2. *The relation is a full unique decomposition (or fud), if in addition it satisfies  $[R] \equiv \top$ .*

There is no simple syntax to generates fuds; the distinction between fuds and non-fuds is semantical. Let  $F, F_1, \dots$  denote individual fuds. Any unique decomposition  $R$  can be transformed into the fud  $R \vee (\neg[R] \times \emptyset)$ . A property  $P$  is a fud iff it is valid:  $P \equiv \top$ .

A non-fud relation may be used in a context that it essentially becomes a fud. For example, the disjointness relation  $*$  is not a fud, but if either of its arguments is a fixed edge or empty, and the other argument does not contain this edge, then it becomes a fud:

$$\prec_{u,v,w} \hat{=} ((u, v, w) \vee \emptyset) \times \neg(u, v, w)^\circ.$$

Referring to the previous section, we now have

$$(x_1, a, x_2) * P = \top \prec_{x_1, a, x_2} P.$$

The reachability decompositions  $\ll_v$  and  $\ll$  are fuds too. This reflects the fact that the separation between garbage and non-garbage parts in memory is unique. Although  $\bowtie$  is not a fud, it is used in the context for linked lists as a fud:  $\bowtie(list, acyc) \vee (\emptyset \times \emptyset) \vee (\neg list \times \emptyset)$ , reflecting the fact that any non-empty linked list can be uniquely decomposed as a shorter list and the last acyclic edge.

For any  $n$ -ary fud  $F$  and fuds  $F_1, \dots, F_n$ , the relation  $F(F_1, \dots, F_n)$  is also a fud; it corresponds to the further decomposition of each of  $F$ 's argument by  $F_i$ . If  $F$  is a fud, so is the substitution  $F[v/x]$ .

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 quantifier.

- Law 1**
- (1)  $\neg[F \wedge R] \equiv [F \wedge \neg R]$
  - (2)  $[F \wedge R_1] \wedge [F \wedge R_2] \equiv [F \wedge R_1 \wedge R_2]$
  - (3)  $[\forall x. (F \wedge R)] \equiv \forall x. [F \wedge R]$

## 5 Separation Logic as a fragment

SL without pointer arithmetic has the syntax:

$$\mathbf{SL} ::= \emptyset \mid \mathbf{A} \mapsto \mathbf{A}, \mathbf{A} \mid \mathbf{A} = \mathbf{A} \mid \neg \mathbf{SL} \mid \mathbf{SL} \vee \mathbf{SL} \mid \exists \mathbf{X}. \mathbf{SL} \mid \mathbf{SL} * \mathbf{SL} \mid \mathbf{SL} \multimap \mathbf{SL}$$

We assume, following standard practice in selecting decidable fragments [6, 7], that there are always exactly two values stored at each memory cell.

**SL** can be embedded in the pointer logic as a fragment. Variables, equality, negation, disjunction and existential quantifier are essentially identical in both logics. We restrict the names of the pointer logic to be integers and apply the translation rules:

$$\begin{aligned} emp &\hat{=} \emptyset \\ u \mapsto v, w &\hat{=} (u, 1, v) : (u, 2, w) \\ P_1 * P_2 &\hat{=} [* (P_1, P_2)] \\ P_1 \multimap P_2 &\hat{=} P_2 /_* P_1 \wedge \text{determinism} \end{aligned}$$

where determinism must be enforced when taking the adjoint. The following lemma shows that the translated fragment of SL is closed for deterministic object diagrams:

**Lemma 1** *Let  $P$  be a formula in **SL**. Then the translated property  $P'$  implies determinism:  $\models P' \rightarrow \text{determinism}$ .*

**Proof.** By standard laws of predicate calculus and the fact that disjoint union of deterministic object diagrams is still deterministic.  $\square$

Now the following result follows from the relevant definitions.

**Theorem 1 (Separation-Logic Fragment)** *Let  $P$  be a formula in **SL**. Then  $P$  is valid in **SL** iff the translated property  $P'$  is valid in **R**.*  $\square$

## 6 Decidability

The decidability results of this paper extensively use the technique of normal forms. In this section the relevant notions are recalled.

### 6.1 Defining decidability and normal-form reduction

A fragment of the logic is *decidable* if there is a decision procedure to transform each of its relations to a equivalid normal form and hence decide its validity, in finitely-many steps.

**Proposition 1** *Two relations are semantically equivalent,  $R_1 \equiv R_2$ , iff  $\models R_1 \leftrightarrow R_2$ . An  $n$ -ary relation  $R$  is valid iff it is semantically equivalent to  $\top^n$ , i.e.  $R \equiv \top^n$ .*

Let  $\mathbf{W}, \mathbf{W}_1, \dots$  denote arbitrary fragments of the logic **R**.

**Definition 3 (Semantical reduction)** *A fragment  $\mathbf{W}_1$  can be semantically reduced to another fragment  $\mathbf{W}_2$ , i.e.  $\mathbf{W}_1 \Rightarrow \mathbf{W}_2$ , if for any relation  $W_1 \in \mathbf{W}_1$  there exists  $W_2 \in \mathbf{W}_2$  such that  $W_1 \equiv W_2$ , and the syntactical transformation takes finitely many steps.*

Note that if  $\mathbf{W}'_1 \subseteq \mathbf{W}_1$  and  $\mathbf{W}_2 \subseteq \mathbf{W}'_2$ , then  $\mathbf{W}_1 \Rightarrow \mathbf{W}_2$  implies  $\mathbf{W}'_1 \Rightarrow \mathbf{W}'_2$ . We further introduce  $\bigwedge \mathbf{W}$  to denote the fragment containing all  $\bigwedge_i^n W_i$  where  $W_i \in \mathbf{W}$ . For example, we have

$$\bigwedge \bigvee \mathbf{W} \Rightarrow \bigvee \bigwedge \mathbf{W} \Rightarrow \bigwedge \bigvee \mathbf{W},$$

according to the mutual distributivity laws of disjunction and conjunction. (Of course  $\bigwedge \bigvee \mathbf{W}$  there denotes all  $\bigwedge_i^n \bigvee_j^{m_i} W_{ij}$  where the range of each inner index depends on the outer index.) We use  $\prod \mathbf{W} \hat{=} \prod_i^n W_i$  to denote Cartesian product. Conjunction of Cartesian products can be merged:

$$\bigwedge \prod \mathbf{W} \Rightarrow \prod \bigwedge \mathbf{W} \Rightarrow \bigwedge \prod \mathbf{W},$$

although that fails for disjunction where only one direction still holds:

$$\prod \bigvee \mathbf{W} \Rightarrow \bigvee \prod \mathbf{W}.$$

As  $W = \bigvee_i^1 W = W$ , we always have  $\mathbf{W} \Rightarrow \bigvee \mathbf{W}$ . Similarly  $\mathbf{W} \Rightarrow \bigwedge \mathbf{W}$  and  $\mathbf{W} \Rightarrow \prod \mathbf{W}$ . Because  $W \equiv \exists x \cdot W$  if  $x$  is free in  $W$ , we also have  $\mathbf{W} \Rightarrow \exists x \cdot \mathbf{W}$ .

**Definition 4 (Validity reduction)** A fragment  $\mathbf{W}_1$  is validity-reducible to a fragment  $\mathbf{W}_2$ ,  $\mathbf{W}_1 \Rightarrow_\delta \mathbf{W}_2$ , if for any relation  $R_1 \in \mathbf{W}_1$  there exists a relation  $R_2 \in \mathbf{W}_2$  such that  $R_1 \equiv_\delta R_2$ , and the syntactical transformation takes finitely many steps.

Obviously semantical reduction implies validity reduction.

## 6.2 Satisfaction decidability

Satisfaction decidability tests whether or not a graph tuple satisfies a given relation formula. In this subsection, we consider the fragment  $\mathbf{S}$  without projection. Satisfaction decidability can be viewed as validity decidability of the fragment  $\prod \mathbf{G} \rightarrow \mathbf{S}$ .

We first identify some key laws of validity equivalence. Universal quantifiers are assumed over all variables for validity, which is defined to be for satisfaction of all graphs under all assignments. One fresh name is no better or worse than any other. Thus validity of quantifiers is reduced to checking a finite con/disjunction.

### Law 2

- (1)  $R \equiv_\delta \forall x \cdot R$
- (2)  $\forall x \cdot R \equiv_\delta R[c/x] \wedge \bigwedge_{a \in \alpha(R)} R[a/x] \quad (c \notin \alpha(R))$
- (3)  $\exists x \cdot R \equiv_\delta R[c/x] \vee \bigvee_{a \in \alpha(R)} R[a/x] \quad (c \notin \alpha(R))$
- (4)  $R[a/x] \equiv_\delta R[b/x] \quad (a, b \notin \alpha(R))$
- (5)  $\overline{G} \rightarrow (\overline{P} \infty \overline{R}) \equiv_\delta \models \overline{G} \rightarrow (\overline{P} \infty_k \overline{R}) \quad (k = |\gamma(\overline{G})|).$

**Proof.** For the nontrivial half of (5), suppose that  $\models \overline{G} \rightarrow (\overline{P} \infty_m \overline{R})$  but not  $\models \overline{G} \rightarrow (\overline{P} \infty_{m-1} \overline{R})$ . Then we obtain semantical graphs  $A_0 \subseteq A_1 \subseteq \dots \subseteq A_{m-1} \subset A_m = \gamma(G_1)$  such that  $A_i \models (\overline{P} \infty_i \overline{R})$ . There exist  $B_0, \dots, B_{m-1}$  such that  $A_i \cup B_i = A_{i+1}$  and  $(A_i, B_i) \models \overline{R}$ . Suppose  $i < m-1$  is such that  $A_i = A_{i+1}$ . Then  $A_i \cup B_{i+1} = A_{i+2}$  and hence  $A_{i+2} \models \overline{P} \infty_{i+1} \overline{R}$ ,  $A_{i+3} \models \overline{P} \infty_{i+2} \overline{R}, \dots$  and finally  $A_m \models \overline{P} \infty_{m-1} \overline{R}$ , which contradicts the assumption. Thus  $\models \overline{G} \rightarrow (\overline{P} \infty_m \overline{R})$  implies  $A_0 \subset A_1 \subset \dots \subset A_m = \gamma(G_1)$ . At least one new edge is added during each iteration. That means if a graph satisfies a fixpoint, then it must also satisfy  $\overline{P} \infty_m \overline{R}$  where  $m = |\gamma(G)|$ .  $\square$

The following lemma shows the satisfaction decidability of the variable-less fragment  $\overline{S}$ . That means we are always able to check whether a program state satisfies a property expressed in  $\overline{S}$ . The proof has taken advantage of the facts that each graph has a finite size (for merge), fresh names have equal status which makes quantifiers reducible to connectives, and graph composition expands a graph by at least one edge a time, rendering the necessary iteration of fixpoint finite for each given graph.

**Lemma 2** *The fragment  $\prod \overline{G} \rightarrow \overline{S}$  is decidable.*

**Proof.** Consider the satisfaction of an  $n$ -ary relation:  $\prod_i^n \overline{G}_i \rightarrow \overline{S}$ . We have

$$\models \prod_i^n \overline{G}_i \rightarrow \overline{S}$$

iff  $(\gamma(\overline{G}_1), \dots, \gamma(\overline{G}_n)) \models \overline{S}$  and so prove decidability by induction over the structure of  $S$ .

1.  $\models \overline{G}_1 \rightarrow \overline{G}_2$  iff  $\gamma(\overline{G}_1) = \gamma(\overline{G}_2)$ .
2.  $\models \overline{G}_1 \rightarrow (a=a)$  but  $\not\models \overline{G}_1 \rightarrow \neg(a=b)$ .
3.  $\models \prod_i^n \overline{G}_i \rightarrow \neg \overline{S}$  iff  $\not\models \prod_i^n \overline{G}_i \rightarrow \overline{S}$ .
4.  $\models \prod_i^n \overline{G}_i \rightarrow \overline{S}_1 \vee \overline{S}_2$  iff  $\models \prod_i^n \overline{G}_i \rightarrow \overline{S}_1$  or  $\models \prod_i^n \overline{G}_i \rightarrow \overline{S}_2$ .
5.  $\models \overline{G}_1 \rightarrow [\overline{S}]$  iff there exist subgraphs  $A_1, \dots, A_n$  such that  $\bigcup_i^n A_i = \gamma(\overline{G}_1)$  and  $(A_1, \dots, A_n) \models \overline{S}$ . The number of such subgraph tuples is finite.
6.  $\models \prod_i^n \prod_j^{m_i} \overline{G}_{ij} \rightarrow \overline{S}(\overline{S}_1, \dots, \overline{S}_n)$  iff for any  $i$  we have  $\models \prod_j^{m_i} \overline{G}_{ij} \rightarrow \overline{S}_i$ , and  $\models \prod_i^n (\overline{G}_{i1} : \overline{G}_{i2} : \dots : \overline{G}_{im_i}) \rightarrow \overline{S}$  where  $m_i \hat{=} |\overline{S}_i|$ .
7.  $\prod_i^n \overline{G}_i \rightarrow \exists x \cdot S \equiv \exists x \cdot (\prod_i^n \overline{G}_i \rightarrow S)$  as  $x$  is not free in  $\prod_i^n \overline{G}_i$ . We therefore have

$$\exists x \cdot (\prod_i^n \overline{G}_i \rightarrow S) \equiv_{\delta} \prod_i^n \overline{G}_i \rightarrow \bigvee_{a \in \mathbf{M}} S[a/x]$$

where  $\mathbf{M}$  contains  $\alpha(\prod_i^n \overline{G}_i \rightarrow S)$  as well as a fresh name.  $S$  has at most has free variable  $x$ . Thus the case for existential quantifier is reducible to the case 4.

8.  $\models \overline{G}_1 \rightarrow (\overline{P} \infty \overline{S})$  iff  $\models \overline{G}_1 \rightarrow (\overline{P} \infty_n \overline{S})$  where  $n = |\gamma(\overline{G}_1)|$ . Thus this case is also reducible to case 4 according to Law 2(5).

Thus the variable-less fragment is decidable.  $\square$

The more general fragment  $\mathbf{S}$  with free variables is also satisfaction-decidable as quantification is reducible to a finite con/disjunction. Fragments containing projection are discussed in Section 6.4.

**Theorem 2 (Satisfaction decidability)** *The fragment  $\prod \mathbf{G} \rightarrow \mathbf{S}$  is decidable.*

**Proof.** Assume that  $x_1, x_2, \dots, x_k$  are all the free variables appearing in  $\prod_i^n G_i \rightarrow S$ .

$$\begin{aligned}
 & \prod_i^n G_i \rightarrow S \\
 & \equiv_{\delta} && \text{Law 2(1)} \\
 & \forall x_1 x_2 \dots x_k \cdot (\prod_i^n G_i \rightarrow S) \\
 & \equiv_{\delta} && \mathbf{M} \text{ contains } \alpha(\prod_i^n G_i \rightarrow S) \text{ as well as } k \text{ fresh names} \\
 & \bigwedge_{a_1 \in \mathbf{M}} \dots \bigwedge_{a_k \in \mathbf{M}} (\prod_i^n G_i \rightarrow S)[a_1/x_1, \dots, a_k/x_k]
 \end{aligned}$$

After substitution, every conjunct now has the form  $\prod_i^n \overline{G}_i \rightarrow \overline{S}$ . So Lemma 2 is applicable, and every transformation step can be automated.  $\square$

The following corollary shows that if the sizes of graphs are known to be bounded (as in bounded memory), then the fragment  $\mathbf{S}$  is validity decidable. The decision procedure essentially model-checks all possible structures.

**Corollary 1** *The fragment  $\bigvee \prod \mathbf{G} \rightarrow \mathbf{S}$  is decidable.*

**Proof.** Validity of conjunction requires checking the conjuncts.  $\square$

Satisfaction decidability leads to semi-decidability for validity:

**Corollary 2 (Validity semi-decidability)** *There exists a procedure that terminates on every input relation and outputs a counterexample if the relation is invalid, but does not terminate if the relation is valid.*

**Proof.** We first consider the (semi) decidability of properties. Let  $P$  be the input property with free variables  $x_1, \dots, x_k$ . Consider instead a variable-free property  $\overline{P}_0 \hat{=} \neg \forall x_1 \dots x_k \cdot P$ . Then  $P$  is invalid iff there exists an example  $\overline{G}$  such that  $\models \overline{G} \rightarrow \overline{P}_0$ . Let  $a_1, a_2, \dots, a_n, \dots$  be the fresh names not in  $\overline{P}_0$ . Variable-less minimal graphs (syntactical graphs without duplicated edges) can be enumerated and

ordered by size. For each size  $n$ , we only need  $3n$  fresh names and just consider graphs using only names in  $\alpha(\overline{P}_0) \cup \{a_1, \dots, a_{3n}\}$ . If  $\models \overline{G}_1 \rightarrow \overline{P}_0$  and  $\overline{G}_1$  is of size  $n$ , then we may substitute fresh names from  $\{a_1, \dots, a_{3n}\}$  and obtain an enumerated graph  $\overline{G}_2$  such that  $\models \overline{G}_2 \rightarrow \overline{P}_0$ . Thus the procedure can identify a counterexample of  $P$  if such a counterexample does exist. The case for multi-arity relations is similar and involves enumerating tuples of minimal graphs.  $\square$

### 6.3 A basic validity-decidable fragment

We already know that the general fragment  $\mathbf{S}$  is semi-decidable and now study the decidability of a simple fragment  $\mathbf{K}$  with true, singleton graphs and edge extension closed under connectives:

$$\begin{aligned} \mathbf{L} &::= \top \mid \mathbf{G} \mid \mathbf{A} = \mathbf{A} \mid (\mathbf{A}, \mathbf{A}, \mathbf{A})^\circ \mid \neg \mathbf{L} \\ \mathbf{K} &::= \mathbf{L} \mid \mathbf{K} \vee \mathbf{K} \mid \mathbf{K} \wedge \mathbf{K}. \end{aligned}$$

For example, we have  $\neg(\mathbf{L} \wedge \mathbf{L}) \Rightarrow (\mathbf{L} \vee \mathbf{L}) \Rightarrow \mathbf{K}$ , which means that any relation  $\neg(L_1 \wedge L_2)$  in the fragment  $\neg(\mathbf{L} \wedge \mathbf{L})$  is semantically equivalent to  $\neg L_1 \vee \neg L_2$  in the fragment  $\mathbf{L} \vee \mathbf{L}$  (by De Morgan's law), which is further reducible to  $\mathbf{K}$  by definition. Similarly we also have  $\neg \mathbf{K} \Rightarrow \mathbf{K}$ . The fragment is also closed under substitution:  $\mathbf{K}[\mathbf{A}/\mathbf{A}] \Rightarrow \mathbf{K}$ .

The following lemma forms the basis of induction of the most important validity result of this paper. The proof consists essentially of establishing a bounded-model property: in order to find a counterexample to an invalid property in the fragment  $\mathbf{K}$ , we need search only a finite number of graphs of bounded size.

**Theorem 3 (Simple validity decidability)** *The fragment  $\mathbf{K}$  is decidable.*

**Proof.** We first consider the decidability of  $\bigvee \overline{\mathbf{L}}$ . Variable-less equality is reducible to either  $\top$  or  $\perp$ . If  $\top$  is one of  $\overline{L}_i$ , then  $\overline{K}$  is valid. Ignore  $\perp$ 's as they do not affect validity unless every  $\overline{L}_i$  is  $\perp$  in which case  $\overline{K}$  becomes invalid, and any graph is a counterexample. Let  $\neg \overline{G}_1, \dots, \neg \overline{G}_k$  be all negated graph properties in  $\overline{L}_i$ . Semantical equality  $\gamma(\neg \overline{G}_i) = \gamma(\neg \overline{G}_j)$  between them can be decided in finitely many steps. If all of them are equal then we consider  $\neg \overline{G}_1$ ; otherwise  $\overline{K}$  is valid and has no counterexample. With  $\neg \overline{G}_1$  being in disjunction, a graph  $\overline{G}$  either validates  $\overline{K}$  if  $\overline{G} \equiv \overline{G}_1$  or is ignorable otherwise; edge extension  $(a, b, c)^\circ$  either validates  $\overline{K}$  if  $\overline{G}_1$  contains the edge  $(a, b, c)$  or the edge extension is ignorable otherwise; negated edge extension  $\neg(a, b, c)^\circ$  validates  $\overline{K}$  if  $\overline{G}_1$  does not contain  $(a, b, c)$ . If on the other hand, there is no negated singleton graph property, then  $\overline{K}$  is valid iff there exist a pair of  $(a, b, c)^\circ$  and  $\neg(a, b, c)^\circ$ . Suppose that such a pair does not exist. Let  $\overline{G}_1, \dots, \overline{G}_k$  be all singleton graph properties in  $\overline{L}_i$  and  $m$  be their maximum size (*i.e.* maximum number of edges). Construct a counterexample graph of size  $\max(m, n) + 1$  so that it contains all edges  $(a, b, c)$  of negated edge extensions  $\neg(a, b, c)^\circ$  in all  $\overline{L}_i$  but does not contain edges  $(a, b, c)$  of edge extensions  $(a, b, c)^\circ$ . Such a graph must exist as there are infinitely many names and hence infinitely many edges to choose from. Then this counterexample graph must be larger than any  $\overline{G}_1, \dots, \overline{G}_k$  and does not satisfy the edge extensions or negated ones. Finally, we have  $\overline{\mathbf{K}} \Rightarrow \bigwedge \bigvee \overline{\mathbf{L}}$ . Every  $\bigwedge_i^n \bigvee_j^m \overline{L}_{ij}$  is valid iff  $\bigvee_j^m \overline{L}_{ij}$  is valid for every  $i \leq n$ . Since  $\bigvee \overline{\mathbf{L}}$  is decidable, so is  $\overline{\mathbf{K}}$ .

Now we are ready to consider  $\mathbf{K}$ . For any  $K \in \mathbf{K}$ , assume that  $x_1, \dots, x_k$  are all free variables of  $K$ . Then  $K \equiv_{\delta} \bigwedge_{a_1 \in \mathbf{M}} \dots \bigwedge_{a_k \in \mathbf{M}} K[a_1/x_1, \dots, a_k/x_k]$  where  $\mathbf{M}$  contains  $\alpha(K)$  and  $k$  fresh names. Each conjunct  $K[a_1/x_1, \dots, a_k/x_k]$  is a variable-less property  $\overline{\mathbf{K}}$ . Thus  $\mathbf{K}$  is decidable.  $\square$

The following law will be used in Lemma 3. It shows that the negation of Cartesian product is reducible to disjunction of Cartesian products and so leads to the reduction rule  $\neg \prod \mathbf{K} \Rightarrow \bigvee \prod \mathbf{K}$ . Cartesian product is conjunctive, which leads to the reduction rule  $\bigwedge \prod \mathbf{K} \Rightarrow \prod \mathbf{K}$ .

$$\mathbf{Law\ 3} \quad (1) \neg(R_1 \times R_2) \equiv (\top^{|R_1|} \times \neg R_2) \vee (\neg R_1 \times \top^{|R_2|})$$

$$(2) \prod_i^n P_i \wedge \prod_i^n P'_i \equiv \prod_i^n (P_i \wedge P'_i)$$

The following lemma shows that the fragment  $\overline{\mathbf{K}}$  is closed under fud-based decomposition. The main step takes advantage of the fact that a negated edge extension is reducible to the fud-composition of its duplicates.

**Lemma 3** *A variable-less fud  $\overline{F}$  can be used to decompose any variable-less simple properties:*

$$\overline{\mathbf{K}} \Rightarrow [\overline{F} \wedge \bigvee \prod \overline{\mathbf{K}}]$$

.

**Proof.** We argue by induction on the structure of  $K \in \mathbf{K}$ .

1.  $\top \Rightarrow [\overline{F} \wedge \prod \top]$  as a property of fud:  $\top \equiv [\overline{F}]$ .
2.  $\overline{G} \Rightarrow [\overline{F} \wedge \prod \overline{G}]$  as for any  $\overline{G}$ , there exist  $\overline{G}_1, \dots, \overline{G}_n$  such that  $\overline{G} \equiv [\overline{F}(\overline{G}_1, \dots, \overline{G}_n)]$  (i.e. fullness).
3. Name equality is reducible to  $\top$  or  $\neg \top$ .
4. Any graph  $\overline{G}$  that does not contain an edge  $(a, b, c)$ , has a unique decomposition  $\overline{G}_1, \dots, \overline{G}_n$  by  $\overline{F}$  such that  $\overline{G} \equiv \overline{G}_1 : \dots : \overline{G}_n$ . Obviously every  $\overline{G}_i$  must not contain  $(a, b, c)$ . That means  $\models \neg(a, b, c)^\circ \rightarrow [\overline{F} \wedge \prod_i^n \neg(a, b, c)^\circ]$ . On the other hand, we have  $\neg(a, b, c)^\circ \equiv [\prod_i^n \neg(a, b, c)^\circ]$ . Thus  $\neg(a, b, c)^\circ \equiv [\overline{F} \wedge \prod_i^n \neg(a, b, c)^\circ]$ . This shows that  $\neg(\mathbf{N}, \mathbf{N}, \mathbf{N})^\circ \Rightarrow [\overline{F} \wedge \prod \neg(\mathbf{N}, \mathbf{N}, \mathbf{N})^\circ]$ .
5.  $\neg[\overline{F} \wedge \bigvee \prod \overline{\mathbf{K}}] \Rightarrow [\overline{F} \wedge \neg \bigvee \prod \overline{\mathbf{K}}] \Rightarrow [\overline{F} \wedge \bigvee \prod \overline{\mathbf{K}}]$  according to Law 1(1).
6.  $\bigwedge \bigvee [\overline{F} \wedge \bigvee \prod \overline{\mathbf{K}}] \Rightarrow [\overline{F} \wedge \bigwedge \bigvee \prod \overline{\mathbf{K}}] \Rightarrow [\overline{F} \wedge \bigvee \prod \overline{\mathbf{K}}]$  according to Law 1(2) and disjunctivity.

Thus the reduction is complete.  $\square$

## 6.4 Handling free variables

In the previous subsection, we have obtained some results for variable-less fragments. To handle free variables, we exploit the fact that there are only two possible relations between free variables: equality or inequality. Each variable equals either one of the names in the context or a fresh name.

We now introduce a notation  $\partial_x^c R$  that represents a relation  $R$  under all possible cases of name assignment on  $x$  after identifying a fresh name  $c$ .

**Definition 5**  $\partial_x^c R \hat{=} \bigvee_{a \in \alpha(R)} (x = a \wedge R[a/x]) \vee \left( \bigwedge_{a \in \alpha(R)} \neg(x = a) \wedge R[c/x] \right)$   
 where  $c \notin \alpha(R)$  is a fresh name.

Thus the notation contains an additional name  $c$ :  $\alpha(\partial_x^c R) = \alpha(R) \cup \{c\}$ . This notation has the following ‘restoration’ property. The proof follows by induction on  $R$  and the definition of  $\sigma$ .

**Law 4**  $R \equiv (\partial_x^c R)[x/c]$ . □

Thus if  $\phi(R) = \{x_1, \dots, x_n\}$  and  $c_1, \dots, c_n \notin \alpha(R)$  are distinct fresh names, then all relations  $R$  with substitution in  $\partial_{x_n}^{c_n} \dots \partial_{x_1}^{c_1} R$  are variable-less and

$$R \equiv (\partial_{x_n}^{c_n} \dots \partial_{x_1}^{c_1} R)[x_1/c_1, \dots, x_n/c_n].$$

Using the above technique we are now able to establish the reduction rule of Lemma 3 for fragments allowing free variables.

**Theorem 4 (Fud decomposition)** *A fud  $F$  can be used to decompose any simple properties:*

$$\mathbf{K} \Rightarrow \left[ F \wedge \bigvee \prod \mathbf{K} \right]$$

.

**Proof.** Consider any  $K \in \mathbf{K}$  with any  $n$ -ary fud  $F$ . If  $\phi(K) \cup \phi(F) = \{x_1, \dots, x_m\}$  and  $c_1, \dots, c_m \notin \alpha(K)$  are distinct fresh names, then all relations  $K$  with substitution in  $\partial_{x_m}^{c_m} \dots \partial_{x_1}^{c_1} K$  are variable-less.

Note that every fud with variables substituted for concrete names is still a fud. Among all variables, some are fixed on existing names for each conjunct in  $\partial_{x_n}^{c_n} \dots \partial_{x_1}^{c_1} K$ , while other names are fresh. Consider a conjunct with name-fixing atomic equalities:

$$\bigwedge_i^m x_i = a_i \wedge K[c_1/x_1, \dots, c_m/x_m, a_1/x_{m+1}, \dots, a_{n-m}/x_n].$$

This is reducible to

$$\bigwedge_i^m x_i = a_i \wedge [F[c_1/x_1, \dots, a_{n-m}/x_n] \wedge \bigvee_j \prod_i^n \bar{K}_{ij}]$$

according to Lemma 3. The variables substituted for fixed names can be restored in the environment provided by the atomic equalities:

$$\bigwedge_i^m x_i = a_i \wedge [F[c_1/x_1, \dots, c_m/x_m] \wedge \bigvee_j \prod_i^n \bar{K}_{ij}].$$

Thus  $K \equiv (\partial_{x_m}^{c_m} \dots \partial_{x_1}^{c_1} K)[x_1/c_1, \dots, x_m/c_m]$  is decomposed with  $F$ , and merges with the same fud can be unified and closed in the fragment.  $\square$

We already know that compositions of the same fud are mergeable (Law 1). In fact, compositions of different fuds are mergeable too. This is because the relational bond  $F(F_1, \dots, F_n)$  of fuds  $F, F_1, \dots, F_n$  is also a fud. We can decompose any fud  $F$  into  $F(F', \dots, F')$  with another fud  $F'$ . This is similar to applying more cuts to an already decomposed graph. The following law shows that the cuts of fuds can be apply in either order. That means a finite number of fud-based compositions can be unified into compositions with the same fud (but which could have a very high arity).

$$\mathbf{Law\ 5} \quad [F_1(F_2, \dots, F_2) \wedge \prod_i^{|F_1|} \prod_j^{|F_2|} P_{ij}] \equiv [F_2(F_1, \dots, F_1) \wedge \prod_j^{|F_2|} \prod_i^{|F_1|} P_{ij}]$$

The following proposition uses the above law and unifies fud-based compositions into a single fud composition. Recall that the fragment  $\bigwedge \bigvee [\mathbf{F} \wedge \bigvee \prod \mathbf{K}]$  contains all relations

$$\bigwedge_i \bigvee_j [F_{ij} \wedge \bigvee_k \prod_l K_{ijkl}]$$

with different fuds before unification.

$$\mathbf{Corollary\ 3} \quad \bigwedge \bigvee [\mathbf{F} \wedge \bigvee \prod \mathbf{K}] \Rightarrow [\mathbf{F} \wedge \bigvee \prod \mathbf{K}].$$

**Proof.** We first show that  $\bigwedge [\mathbf{F} \wedge \bigvee \prod \mathbf{K}] \Rightarrow [\mathbf{F} \wedge \bigvee \prod \mathbf{K}]$ . Without loss of generality, we consider a simple case of binary conjunction between two different binary fuds:

$$\begin{aligned} & [F \wedge \bigvee_i (K_{i1} \times K_{i2})] \wedge [F' \wedge \bigvee_j (K'_{j1} \times K'_{j2})] \\ & \equiv \\ & \bigvee_i \bigvee_j [F(F', F') \wedge (K_{i11} \times K_{i12} \times K_{i21} \times K_{i22})] \wedge [F(F', F') \wedge (K'_{j11} \times K'_{j21} \times K'_{j12} \times K'_{j22})] \\ & \Rightarrow \\ & [\mathbf{F} \wedge \bigvee \prod \mathbf{K}] \end{aligned}$$

Similar reasoning is applicable to disjunction-led reduction.  $\square$

## 6.5 An advanced validity-decidable fragment

We are now ready to study a more comprehensive fragment containing Cartesian product, fud-based relational bonds, quantifiers and projection:

$$\mathbf{T} ::= \mathbf{K} \mid \neg \mathbf{T} \mid \mathbf{T} \vee \mathbf{T} \mid \mathbf{T} \times \mathbf{T} \mid [\mathbf{F} \wedge \mathbf{T}] \mid \exists \mathbf{X} \cdot \mathbf{T} \mid \mathbf{T} @ i$$

where  $\mathbf{F}$  is the fragment of all fuds in  $\mathbf{T}$ . In a fud-based relation, every merge must be applied to a fud conjunction. In normal forms, we use the notation  $\Sigma \mathbf{Q}$  to represent several quantifiers in sequence:

$$\begin{aligned} \mathbf{Q} &::= \forall \mathbf{X} \mid \exists \mathbf{X} \\ \Sigma \mathbf{Q} &::= \mathbf{Q} \mid \Sigma \mathbf{Q} \Sigma \mathbf{Q}. \end{aligned}$$

The following lemma reduces  $\mathbf{T}$  to a normal form.

**Lemma 4**  $\mathbf{T} \Rightarrow \Sigma \mathbf{Q} \cdot \forall \Pi [\mathbf{F} \wedge \forall \Pi \mathbf{K}]$

**Proof.** We argue by induction over the structure of any  $\mathbf{T}$ .

1.  $\mathbf{K} \Rightarrow \Sigma \mathbf{Q} \cdot \forall \Pi [\mathbf{F} \wedge \forall \Pi \mathbf{K}]$  as  $\top \in \mathbf{F}$ .
2. Negation of the normal form:

$$\neg \Sigma \mathbf{Q} \cdot \forall \Pi [\mathbf{F} \wedge \forall \Pi \mathbf{K}]$$

$$\Rightarrow$$

laws of quantifier negation and De Morgan's law

$$\Sigma \mathbf{Q} \cdot \wedge \neg \Pi [\mathbf{F} \wedge \forall \Pi \mathbf{K}]$$

$$\Rightarrow$$

Laws 1(1) and 3(1)

$$\Sigma \mathbf{Q} \cdot \wedge \forall \Pi [\mathbf{F} \wedge \neg \forall \Pi \mathbf{K}]$$

$$\Rightarrow$$

reduction rules of  $\mathbf{K}$

$$\Sigma \mathbf{Q} \cdot \forall \Pi [\mathbf{F} \wedge \forall \Pi \mathbf{K}].$$

3. Quantifiers distribute over disjunction if the bound variables are fresh, avoiding name conflict.
4. Cartesian product of normal forms:

$$\Pi \Sigma \mathbf{Q} \cdot \forall \Pi [\mathbf{F} \wedge \forall \Pi \mathbf{K}]$$

$$\Rightarrow$$

quantifiers distribute over Cartesian product

$$\Sigma \mathbf{Q} \cdot \Pi \forall \Pi [\mathbf{F} \wedge \forall \Pi \mathbf{K}]$$

$$\Rightarrow$$

disjunctivity

$$\Sigma \mathbf{Q} \cdot \forall \Pi [\mathbf{F} \wedge \forall \Pi \mathbf{K}].$$

5. Fud composition:

$$\begin{aligned}
& [\mathbf{F} \wedge \Sigma \mathbf{Q} \cdot \vee \Pi [\mathbf{F} \wedge \vee \Pi \mathbf{K}]] \\
& \Rightarrow \text{Law 1(3), disjunctivity and distributivity of } \exists \\
& \Sigma \mathbf{Q} \cdot \vee [\mathbf{F} \wedge \Pi [\mathbf{F} \wedge \vee \Pi \mathbf{K}]] \\
& \Rightarrow \text{disjunctivity and fud decomposition} \\
& \Sigma \mathbf{Q} \cdot [\mathbf{F} \wedge \vee \Pi \mathbf{K}] \\
& \Rightarrow \text{reduced as a special case} \\
& \Sigma \mathbf{Q} \cdot \vee \Pi [\mathbf{F} \wedge \vee \Pi \mathbf{K}].
\end{aligned}$$

6. Quantifiers over the normal form are directly mergeable.

7. If  $\mathbf{W}$  is a fragment of properties, then it can be shown that

$$(\Sigma \mathbf{Q} \cdot \vee \Pi \mathbf{W}) @ i \Rightarrow \Sigma \mathbf{Q} \cdot \vee \mathbf{W}.$$

This guarantees the reduction for projection. □

The following lemma is one of the critical steps of Theorem 5 and handles the decidability of Cartesian products in disjunction. Recall that a relation is valid iff it is a full relation. Thus every graph tuple must satisfy one of the disjuncts. The key to the proof is to generate a finite number of test graph tuples for checking satisfaction.

**Lemma 5** *Let  $\overline{\mathbf{W}}$  be a variable-less fragment of properties closed under negation,  $\neg \overline{\mathbf{W}} \Rightarrow \overline{\mathbf{W}}$ , and disjunction,  $\vee \overline{\mathbf{W}} \Rightarrow \overline{\mathbf{W}}$ . If the fragment  $\overline{\mathbf{W}}$  is decidable, then so is  $\vee \Pi \overline{\mathbf{W}}$ .*

**Proof.** Consider the validity of  $\vee_j \Pi_i \overline{W}_{ij}$ . A graph tuple  $(A_1, \dots, A_n)$  is a counterexample iff for every  $j \leq m$  there exists some  $i \leq n$  such that  $A_i \not\models \overline{W}_{ij}$ . We show that there exist no more than  $2^{mn}$  tuples of graphs such that the relation is valid iff it is satisfied by all the tuples.

The properties  $W_{i1}, \dots, W_{im}$  partition the graph space into  $2^m$  classes. Let  $b_1, \dots, b_m$  be Booleans. We use  $b_j \bullet W_{ij}$  to denote  $W_{ij}$  if  $\neg b_j$  and  $\neg W_{ij}$  if  $b_j$ . Each group of Booleans determine a class of graphs:  $\bigwedge_j b_j \bullet W_{ij}$ . All graphs in each class have the same satisfaction for all  $W_{i1}, \dots, W_{im}$ , and thus we need find only one representative. Because  $\mathbf{W}$  is decidable and closed under negation and disjunction, we consider the validity of  $\neg \bigwedge_j b_j \bullet W_{ij}$ . If it is valid, the class is empty and we do not need to choose any representative; otherwise we can find a counterexample, which will be used to represent the class. For each  $i$ , there are at most  $2^m$  classes and representatives. Thus we generate no more than  $2^{mn}$  tuples. If there is any counterexample of  $\vee_j \Pi_i \overline{W}_{ij}$ , there must be one among the generated tuples. Thus the fragment  $\vee \Pi \overline{\mathbf{W}}$  is also decidable. □

The following law provides a means to decide the validity of a fud composition. A fud composition  $[F \wedge R]$  is valid if every graph satisfies it. As every graph has a unique decomposition into a tuple of graphs by a fud, all the tuples allowed by  $F$  must be allowed by  $R$  too. The proof follows from the definitions.

**Law 6**  $[F \wedge R] \equiv_{\delta} \neg F \vee R$  □

We are now ready to prove the main theorem of this paper.

**Theorem 5** [Advanced validity decidability] *The fragment  $\mathbf{T}$  is decidable.*

**Proof.** A fud in  $\mathbf{T}$  may be defined using other fuds in  $\mathbf{T}$ . We reason by induction on the depth of such nesting and define a sequence of increasingly larger fragments  $\mathbf{T}_k$  and  $\mathbf{F}_k \hat{=} \mathbf{F} \cap \mathbf{T}_k$ :

$$\begin{aligned}\mathbf{T}_0 &\hat{=} \Sigma \mathbf{Q} \cdot \forall \Pi \mathbf{K} \\ \mathbf{T}_{k+1} &\hat{=} \Sigma \mathbf{Q} \cdot \forall \Pi [\mathbf{F}_k \wedge \forall \Pi \mathbf{K}].\end{aligned}$$

Thus  $\mathbf{T} = \bigcup_k \mathbf{T}_k$ . We first consider the validity of  $\mathbf{T}_0$ .

$$\begin{aligned}\mathbf{T}_0 & \\ = & \text{definition} \\ \Sigma \mathbf{Q} \cdot \forall \Pi \mathbf{K} & \\ \Rightarrow_{\delta} & \text{Law 2(2)} \\ \wedge \forall \forall \Pi \bar{\mathbf{K}} & \\ \Rightarrow & \text{distributivity and closure of } \mathbf{K} \\ \forall \Pi \bar{\mathbf{K}} & \end{aligned}$$

Thus, according to Lemma 5,  $\mathbf{T}_0$  is decidable. Secondly, we assume the validity of  $\mathbf{T}_k$  and consider the validity of  $\mathbf{T}_{k+1}$ .

$$\begin{aligned}\mathbf{T}_{k+1} & \\ = & \text{definition} \\ \Sigma \mathbf{Q} \cdot \forall \Pi [\mathbf{F}_k \wedge \forall \Pi \mathbf{K}] & \\ \Rightarrow_{\delta} & \text{Law 2(2)(3)} \\ \wedge \forall \forall \Pi [\bar{\mathbf{F}}_k \wedge \forall \Pi \bar{\mathbf{K}}] & \\ \Rightarrow & \text{laws of predicate calculus and closure of } \mathbf{K} \\ \forall \Pi [\bar{\mathbf{F}}_k \wedge \forall \Pi \bar{\mathbf{K}}] & \end{aligned}$$

For each fud composition, we have:

$$\begin{aligned}[\bar{\mathbf{F}}_k \wedge \forall \Pi \bar{\mathbf{K}}] & \\ \Rightarrow_{\delta} & \text{Law 6} \\ \neg \bar{\mathbf{F}}_k \vee \forall \Pi \bar{\mathbf{K}} & \end{aligned}$$

$\Rightarrow \delta$   
 $\mathbf{T}_k$ .

As  $\mathbf{T}_k$  is assumed to be decidable, the fragment  $\bigvee \Pi \mathbf{T}$  is decidable, and so is  $\mathbf{T}_{k+1}$ . Thus the whole fragment  $\mathbf{T}$  is decidable.  $\square$

## 7 Axiomatization and completeness

All the (syntactical) graphs  $\overline{\mathbf{G}}$  form the nominals of the logic and can thus significantly simplify its completeness proof.

We already know that the satisfaction problem for the pointer logic is decidable. Thus if we are able to use an axiom system to mimic the decision procedure, then any valid formula in the form  $\prod_i G_i \rightarrow R$  will be provable. If a formula  $R$  is valid, then all  $\prod_i G_i \rightarrow R$  are valid and hence provable. Thus  $R$  becomes provable if we have an infinitary inference rule that concludes so from the provability of all  $\prod_i G_i \rightarrow R$ .

The axioms and inference rules are identified as follows.

**Axiom 1** *Standard axioms and inference rules of predicate logic.*

An axiomatic inference rule  $R_1 \vdash R_2$  should be understood as “the provability of  $R_1$  implies the the provability of  $R_2$ ”. The following inference rules state that the operators are *semantical* in the sense that if formulas are semantically equal (or monotonically ordered), so are semantically equal operator-applied formulas. In the proof theory, they ensure that if two formulas are already proved to be equivalent, then one can be replaced with another inside an operator.

**Axiom 2** *Let  $i \leq |R|$ .*

- (1)  $R \rightarrow R' \vdash [R] \rightarrow [R']$
- (2)  $R \rightarrow R' \vdash R(R_1, \dots, R_m) \rightarrow R'(R_1, \dots, R_m)$
- (3)  $R_i \rightarrow R'_i \vdash R(R_1, \dots, R_i, \dots, R_m) \rightarrow R(R_1, \dots, R'_i, \dots, R_m)$
- (4)  $P \rightarrow P' \vdash (P \infty R) \rightarrow (P' \infty R)$
- (5)  $R \rightarrow R' \vdash (P \infty R) \rightarrow (P \infty R')$
- (6)  $R \rightarrow R' \vdash (R @ i) \rightarrow (R' @ i)$ .

The following axioms help establish the relations between individual graphs and the truth value of name equality:

- Axiom 3**
- (1)  $\vdash \top$
  - (2)  $\vdash \overline{G}_1 \rightarrow \overline{G}_2 \quad (\gamma(\overline{G}_1) = \gamma(\overline{G}_2))$
  - (3)  $\vdash \overline{G}_1 \rightarrow \neg \overline{G}_2 \quad (\gamma(\overline{G}_1) \neq \gamma(\overline{G}_2))$
  - (4)  $\vdash (a = a)$
  - (5)  $\vdash \neg(a = b).$

The following inference rules handle relational bond:

- Axiom 4**
- $$\begin{array}{l} \prod_j^{m_1} \overline{G}_{1j} \rightarrow \overline{R}_1, \dots, \prod_j^{m_n} \overline{G}_{nj} \rightarrow \overline{R}_n, \\ \prod_i^n \overline{G}_{i1} : \dots : \overline{G}_{im_i} \rightarrow \overline{R} \\ \dashv\vdash \prod_i^n \prod_j^{m_i} \overline{G}_{ij} \rightarrow \overline{R}(\overline{R}_1, \dots, \overline{R}_n). \end{array}$$

The following axiom states that an existential quantifier is provable iff the disjunction of various name substitutions including one fresh name are provable:

- Axiom 5** *If  $c \notin \alpha(R)$  then  $R[c/x] \vee \bigvee_{a \in \alpha(R)} R[a/x] \dashv\vdash \exists x \cdot R.$*

The following axiom states that provability for satisfaction of fixpoint can be reduced to provability for a finite approximation bounded by the size of the given graph:

- Axiom 6**  $\overline{G} \rightarrow (\overline{P} \infty_n \overline{R}) \dashv\vdash \overline{G} \rightarrow (\overline{P} \infty \overline{R})$  where  $n = |\gamma(\overline{G})|.$

The following axioms characterize some aspects of projection.

- Axiom 7**
- (1)  $\vdash (R_1 \vee R_2) @ i \rightarrow (R_1 @ i \vee R_2 @ i)$
  - (2)  $\vdash (\exists x \cdot R) @ i \rightarrow \exists x \cdot (R @ i)$
  - (3)  $\vdash (\prod_j^n P_j) @ i \leftrightarrow P_i$

Finally an infinitary rule links nominals with provability:

- Axiom 8** *If for any graph tuple  $\prod_i^n \overline{G}_i$  we have  $\vdash \prod_i^n \overline{G}_i \rightarrow \overline{R}$ , then  $\vdash \overline{R}.$*

The soundness of the axioms is easy to establish from the definitions. Completeness is also obtained readily:

**Theorem 6 (Completeness)** *If a relation  $S$  without projection is valid,  $\models S$ , then it is provable:  $\vdash S.$*

**Proof.** Let  $x_1, \dots, x_k$  be the free variables of an  $n$ -ary relation  $S$ . Then  $\models S$  implies  $\models \forall x_1 \dots \forall x_k \cdot S$  by definition. That means for every variable-less graph tuple  $\prod_i^n \overline{G}_i$  we have  $\models \prod_i^n \overline{G}_i \rightarrow \forall x_1 \dots \forall x_k \cdot S$ . Thus  $\vdash \prod_i^n \overline{G}_i \rightarrow \forall x_1 \dots \forall x_k \cdot S$ . Hence  $\vdash S$  according to Axiom 8 and predicate logic.  $\square$

Note that many common axioms are not present here, because they are provable using the infinitary rule. We are able to obtain completeness, because of three key characteristics of the logic: the presence of nominals that help us pinpoint individual semantic denotations within the logic, the absence of pointer arithmetic that makes the testing for quantifiers finite, and the infinitary rule.

## 8 Conclusions

This work is to be viewed in the context of Separation Logic although the syntax is designed differently to reflect object diagrams as the underlying semantics and the need to present the decidability results as succinctly as possible. More conventional shorthands are needed for applications.

The separating conjunction of SL implicitly includes two operations: consistency checking and merging. We have divided it into the merge operator and user-definable relation that checks consistency before merge. Such a division allows us to create new consistency-checking relations for new compositions. A higher level of abstraction can be achieved by using the right compositions at the right places. For example, the binary relation  $\bowtie$  in Section 4 can be used to connect acyclic edges to form a list without a circle. In SL, a large number of variable inequalities would be needed to avoid circles. In the pointer logic, hiding with existential quantifiers is often applied as locally as possible. This helps increase the level of abstraction.

The decidability results of this paper benefit from the design of the logic and makes extensive use of normal-form reduction and the finite-model property. The most interesting fragment **T** relies on the assumption that the compositions are unique decompositions and all names have equal status. Generally, a fragment without such restrictions is undecidable [7]. That corresponds to the requirement that all separating conjunctions are used in precise predicates (a notion studied in [26]).

Fixpoints are undecidable in general. It is still an open problem to design a restricted decidable fragment that includes fixpoints. Berdine et. al. [4] have studied a restrictive decidable fragment with linked lists but without disjunction or quantifiers. It is unclear to what extent their technique will be useful in the general fragment **T** closed under disjunction and quantifiers. Other known decidable fragments such as [6, 7] do not allow quantifiers and are special fragments of **T**.

A major benefit of the approach promoted here is to raise the level of abstraction for reasoning about mutable data structures. It might be expected to benefit in particular the active topic of concurrency. For example, a garbage collector will not interfere with a process if the two work on different parts of the memory partitioned by a fud.

## References

- [1] G. Morrisett, A. Nanevski, A. Ahmed and L. Birkdal. Abstract predicates and mutable ADTs in hoare type theory. <http://www.eecs.harvard.edu/~aleks/papers/hoarelogic/htthol.pdf>.
- [2] C. Areces, P. Blackburn, and M. Marx. Hybrid logics: Characterization, interpolation and complexity. *Journal of Symbolic Logic*, 66(3):977–1010, 2001.
- [3] M. Benedikt, T.W. Reps, and S. Sagiv. A decidable logic for describing linked data structures. In *ESOP*, pages 2–19, 1999.
- [4] J. Berdine, C. Calcagno, and P. O’Hearn. A decidable fragment of separation logic. In *FSTTCS*, pages 97–109, 2004.
- [5] C. Calcagno, L. Cardelli, and A.D. Gordon. Deciding validity in a spatial logic for trees. *Journal of Functional Programming*, 15(4):543–572, 2005.
- [6] C. Calcagno, P. Gardner, and M. Hague. From separation logic to first-order logic. In *FoSSaCS*, pages 395–409, 2005.
- [7] C. Calcagno, H. Yang, and P.W. O’Hearn. Computability and complexity results for a spatial assertion language for data structures. In *APLAS*, pages 289–300, 2001.
- [8] L. Cardelli, P. Gardner, and G. Ghelli. A spatial logic for querying graphs. In *ICALP’02*, volume 2380 of *LNCS*, pages 597–610. Springer, 2002.
- [9] Y. Chen. Generic composition. *Formal Aspects of Computing*, 14(2):108–122, 2002.
- [10] Y. Chen and J.W. Sanders. Logic of global synchrony. *ACM Transactions on Programming Languages and Systems*, 26(2):221–262, 2004.
- [11] Y. Chen and J.W. Sanders. Compositional reasoning for pointer structures. In *8th International Conference on Mathematics of Program Construction (MPC’06)*, volume 4014 of *LNCS*, pages 115–139. Springer, 2006.
- [12] B. Courcelle. Graph decompositions definable in monadic second-order logic. *Electronic Notes in Discrete Mathematics*, 22(15):13–19, 2005. 7th International Colloquium on Graph Theory.
- [13] W. Cunningham. Decomposition of directed graphs. *SIAM Journal Algebraic Discrete Methods*, 3:214–228, 1982.
- [14] A. Dawar, P. Gardiner, and G. Ghelli. Expressiveness and complexity of graph logic. *Information and Computation*, 205:263–310, 2006.
- [15] V. Goranko and S. Passy. Using the universal modality: gains and questions. *Journal of Logical Computation*, 2(1):5–30, 1992.
- [16] P. Hlineny and D. Seese. Trees, grids and mso decidability: From graphs to matroids. *Theoretical Computer Science*, 351(3):372–393, 2006.

- [17] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
- [18] C.A.R. Hoare and J. He. Weakest prespecifications, I,II. *Fundamenta Informatica*, 9:51–84, 217–252, 1986.
- [19] C.A.R. Hoare and J. He. *Unifying Theories of Programming*. Prentice Hall, 1998.
- [20] C.A.R. Hoare and J. He. A trace model for pointers and objects. In *ECOOP'99*, volume 1628 of *LNCS*, pages 1–17, 1999.
- [21] N. Klarland J.L. Jensen, M.E. Joergensen and M.I. Schwartzbach. Automatic verification of pointer programs using monadic second-order logic. In *SIGPLAN Conference on PLDI*, 1997.
- [22] V. Kuncak and M. Rinard. On spatial conjunction as second-order logic. Technical Report 970, MIT CSAIL technical report, 2004.
- [23] T. Lev-Ami and M. Sagiv. Tvla: A system for implementing static analyses. In *Seventh International Static Analysis Symposium, SAS'00*, 2000.
- [24] A. Møller and M.I. Schwartzbach. The pointer assertion logic engine. In *Programming Language Design and Implementation'01*, pages 221–231, 2001.
- [25] P.W. O'Hearn, J. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *Proceedings of Computer Science Logic*, volume 2142, pages 1–19. Springer, 2001.
- [26] P.W. O'Hearn, J. Reynolds, and H. Yang. Separation and information hiding. In *POPL'04*, volume 2142, pages 268–280. ACM, 2004.
- [27] S. Ranise and C. Zarba. A theory of singly-linked lists and its extensible decision procedure. In *Software Engineering and Formal Methods*, pages 206–215, 2006.
- [28] J.C. Reynolds. Separation logic: a logic for shared mutable data structures. In *LICS'02*, pages 55–74. IEEE Computer Society, 2002.
- [29] E.J. Sims. Extending separation logic with fixpoints and postponed substitution. *Theoretical Computer Science*, 351(2):258–275, 2006.