

The Complexity of Abduction for Separated Heap Abstractions

Nikos Gorogiannis, Max Kanovich, and Peter W. O’Hearn

Queen Mary University of London

Abstract. Abduction, the problem of discovering hypotheses that support a conclusion, has mainly been studied in the context of philosophical logic and Artificial Intelligence. Recently, it was used in a compositional program analysis based on separation logic that discovers (partial) pre/post specifications for un-annotated code which approximates memory requirements. Although promising practical results have been obtained, completeness issues and the computational hardness of the problem have not been studied. We consider a fragment of separation logic that is representative of applications in program analysis, and we study the complexity of searching for feasible solutions to abduction. We show that standard entailment is decidable in polynomial time, while abduction ranges from NP-complete to polynomial time for different sub-problems.

1 Introduction

Abductive inference is a mode of reasoning that concerns generation of new hypotheses [25]. Abduction has attracted attention in Artificial Intelligence (e.g., [24]), based on the idea that humans perform abduction when reasoning about the world, such as when a doctor looking at a collection of symptoms hypothesizes a cause which explains them.

Similarly, when a programmer tries to understand a piece of code, he or she makes hypotheses as well as deductions. If you look at the C code for traversing a cyclic linked list, you might hypothesize that an assertion describing a cyclic list should be part of the precondition, else one would obtain a memory error, and you might even discover this *from the code itself* rather than by communication from the program’s designer. In separation logic, a specialized logic for computer memory, the abduction problem – given A and B , find X where the separating conjunction $A * X$ is consistent and $A * X$ entails B – takes on a spatial connotation where X describes “new” or “missing” memory, not available in the part of memory described by A . Recent work has used abductive inference for separation logic to construct an automatic program analysis which partly mimics, for a restricted collection of assertions describing memory-usage requirements of procedures, the combined abductive-deductive-inductive reasoning that programmers employ when approaching bare code [6]. Abduction is used in the generation of preconditions, after which forwards analysis can be used to obtain a postcondition and, hence, a true Hoare triple for the procedure, without consulting the procedure’s calling context, resulting in a compositional analysis.

Compositionality – that the analysis result of a whole is computed from the analysis results of its parts – has well-known benefits in program analysis [9],

including the ability to analyze incomplete programs (e.g., programs as they are being written) and increased potential to scale. Abductive inference has enabled a boost in the level of automation in shape analysis (e.g., [26,13]) – an expensive “deep memory” analysis which involves discovering data structures of unbounded depth in the heap. The ABDUCTOR academic prototype tool has been applied to several open-source projects in the hundreds of thousands of LOC [11], and INFER is an industrial tool which incorporates these and other ideas [5].

Other applications of abduction for separation logic include the analysis of concurrent programs [7], memory leaks [12], abduction for functional correctness rather than just memory safety [17], and discovering specifications of unknown procedures [22]. The latter is somewhat reminiscent of the (to our knowledge) first use of abduction in program analysis, a top-down method that infers constraints on literals in a logic program starting from a specification of a top-level program [16]. In contrast, ABDUCTOR works bottom-up, obtaining specs for procedures from specs of their callees (it would evidently be valuable to mix the two approaches). A seemingly unrelated use of abduction in program analysis is in under-approximation of logical operators such as conjunction and disjunction in abstract domains with quantification [18].

While the potential applications are perhaps encouraging, the abduction problem for separated heap abstractions has not been investigated thoroughly from a theoretical point of view. The proof procedures used are pragmatically motivated and sound but demonstrably incomplete, and questions concerning complexity or the existence of complete procedures have not been addressed. Our purpose in this paper is to consider complexity questions (taking completeness as a requirement) for the abduction problem for a fragment of logic representative of that used in program analysis.

In the context of classical logic, abduction has been studied extensively and there are several results about its algorithmic properties when using, for example, different fragments of propositional logic as the base language [14,10]. However, the results do not carry over to our problem because the special abstract domains used in shape analyzers are different in flavour from propositional logic. For example, the use of variables and equalities and disequalities in separated heap abstractions raise particular problems. Furthermore, understanding the interaction between heap-reachability and separation is subtle but essential.

The contents of the paper are as follows. In Section 2 we define the restricted separation logic formulae we use, called ‘symbolic heaps’ [2,13], which include a basic ‘points-to’ predicate, and an inductive predicate for describing linked-list segments. The separated abduction problem is defined in Section 3 along with a ‘relaxed’ version of the problem, often used in program analysis. Section 4 shows that entailment is in PTIME and contains an interpolation-like result which bounds the sizes of solutions that must be considered in abduction. Section 5 establishes that when lists are present both the general and relaxed problems are NP-complete. Section 6 shows that the abduction problem is NP-complete when the formulae have only points-to predicates, and that the ‘relaxed’ version of the problem can be solved in polynomial time.

2 Preliminaries

This section records background material on separated heap abstractions [2,13].

2.1 Syntax of Separated Heap Abstractions

Let \mathbf{Var} be a countable set of variables. The set of terms is simply $\mathbf{Terms} = \mathbf{Var} \cup \{\mathbf{nil}\}$ where $\mathbf{nil} \notin \mathbf{Var}$. Spatial predicates P , pure formulae Π and spatial formulae Σ are defined as follows, where x, y are terms.

$$\begin{aligned} P(x, y) &::= x \mapsto y \mid \mathbf{ls}(x, y) \\ \Pi &::= \Pi \wedge \Pi \mid x = y \mid x \neq y \\ \Sigma &::= \Sigma * \Sigma \mid P(x, y) \mid \mathbf{emp} \mid \mathbf{true} \end{aligned}$$

A formula in one of the forms: $\Pi \wedge \Sigma$, Π , or Σ is called a *symbolic heap*. We employ \equiv to denote syntactic equality of two expressions modulo commutativity of \wedge , $*$, and symmetry of $=$ and \neq . We will say that the term x is an *L-value* in a formula A if there is a term y such that the spatial predicate $P(x, y)$ is in A .

The separating conjunction of two symbolic heaps is defined as follows.

$$(\Pi_A \wedge \Sigma_A) * (\Pi_B \wedge \Sigma_B) = (\Pi_A \wedge \Pi_B) \wedge (\Sigma_A * \Sigma_B)$$

This definition is, in fact, an equivalence in general separation logic.

The formula $\mathbf{ls}(y, z)$ expresses that the heap consists of a non-empty acyclic path from y to z , and that z is not an allocated cell in the heap. The formula $x \mapsto y$ describes a singleton heap in which x is allocated and has contents y . Cycles can be expressed with compound formulae. For instance, $\mathbf{ls}(y, x) * \mathbf{ls}(x, y)$ describes two acyclic non-empty linked lists which together form a cycle: this is the kind of structure sometimes used in cyclic buffer programs.

As always in program analysis, what *cannot* be said is important for allowing efficient algorithms for consistency and entailment checking. General separation logic, which allows $*$ and its adjoint \multimap to be combined with all boolean connectives, is undecidable even at the propositional level [4]. In the fragment here we cannot express, e.g., that there are not two separate lists in the heap, or that the data elements held in a list are sorted. We can describe memory safety properties of linked-list programs (that data structures are well formed) but not functional correctness (e.g., that a list insertion procedure actually inserts). As we will see, we obtain a polynomial-time algorithm for entailment (hence, consistency). In more expressive decidable heap logics (e.g., [23,27,21,3]) these problems can range from PSPACE to even non-elementary complexity.

2.2 Semantics

The logic is based on a model of *heap partitioning*.

Definition 2.1 *Stack-and-heap models* are defined as pairs (s, h) , where s (the *stack*) is a mapping from variables \mathbf{Var} to values \mathbf{Val} , and h (the *heap*) is a finite partial function from an infinite L to RV (L-values to R-values in Strachey's terminology). Here we take RV to be $L \cup \{\mathbf{nil}\}$ where $\mathbf{nil} \notin L$. The composition $h_1 \circ h_2$ is the union of h_1 and h_2 if $\text{dom}(h_1) \cap \text{dom}(h_2) = \emptyset$, else undefined.

The value `nil` plays the role of a special never-to-be allocated pointer which is useful, e.g., for terminating linked lists. The heaps here have at most one successor for any node, reflecting our focus on linked lists rather than trees or graphs in the symbolic heaps that we study. (Eventually, we would like to consider general inductive definitions; they are easy to consider semantically, but extremely challenging, and beyond our scope, for decidability of entailment.)

We will use stacks as functions and write $s(x)$ for the value of a variable x . We extend this notation to include `nil`: $s(\text{nil}) = \text{nil}$. Whenever $s(x)=a$ and $s(y)=b$, the *spatial* predicate $x \mapsto y$ is true in the one-cell heap of the form $a \rightarrow \boxed{b}$, or $\bullet \xrightarrow{a} \bullet^b$, which depicts that the ‘location’ a contains the value b .

The formal definition of the semantics is as follows.

Definition 2.2 Given any (s, h) and formula A , we define the forcing relation $(s, h) \vDash A$ by induction on A (see [2]):

- $(s, h) \vDash \text{emp}$ iff $h = []$ is the empty heap
- $(s, h) \vDash A * B$ iff $\exists h_1, h_2. h = h_1 \circ h_2$ and $(s, h_1) \vDash A$ and $(s, h_2) \vDash B$,
- $(s, h) \vDash A \wedge B$ iff $(s, h) \vDash A$ and $(s, h) \vDash B$,
- $(s, h) \vDash \text{true}$ always,
- $(s, h) \vDash (x = y)$ iff $s(x) = s(y)$,
- $(s, h) \vDash (x \neq y)$ iff $s(x) \neq s(y)$,
- $(s, h) \vDash x \mapsto y$ iff $\text{dom}(h) = \{s(x)\}$ and $h(s(x)) = s(y)$,
- $(s, h) \vDash \text{ls}(x, y)$ iff for some $n \geq 1$, $(s, h) \vDash \text{ls}^{(n)}(x, y)$,
- $(s, h) \vDash \text{ls}^{(n)}(x, y)$ iff $|\text{dom}(h)| = n$, and there is a chain a_0, \dots, a_n ,
with no repetitions, such that $h(a_0) = a_1, \dots, h(a_{n-1}) = a_n$,
where $a_0 = s(x)$, $a_n = s(y)$ and $a_n \neq a_0$ (notice that $s(y) \notin \text{dom}(h)$).

Remark 2.3 A *precise* formula [8] cuts out a unique piece of heap, making the non-deterministic \exists -selection in the semantics of $*$ become deterministic. For instance, if $(s, h) \vDash \text{ls}(x, y) * B$, then h is *uniquely* split into h_1, h_2 so that $(s, h_1) \vDash \text{ls}(x, y)$ and $(s, h_2) \vDash B$, where h_1 is defined as an acyclic path from $s(x)$ to $s(y)$. In this fragment, any formula not containing `true` is precise.

As usual, we say that a formula A is *consistent* if $(s, h) \vDash A$ for some (s, h) . A sequent $A \vDash B$ is called *valid* if for any model (s, h) such that $(s, h) \vDash A$ we have $(s, h) \vDash B$. Finally, we call a formula *explicit* if it syntactically contains all equalities and disequalities it entails.

We shall use the inference rules below [2].

$$A \vDash B \implies A * C \vDash B * C \quad (*\text{-Intr})$$

$$x = y \wedge A \vDash B \iff A[x/y] \vDash B[x/y] \quad (\text{Subst})$$

Rule $*\text{-Intr}$ expresses the monotonicity of $*$ w.r.t \vDash . Rule Subst is a standard substitution principle.

SAMPLE TRUE AND FALSE ENTAILMENTS

$$\begin{array}{ll} x \mapsto x' * y \mapsto y' \vDash x \neq y & x \mapsto x' * \text{ls}(x', y) \not\vDash \text{ls}(x, y) \\ x \mapsto x' * x \mapsto y' \vDash x \neq x & x \mapsto x' * \text{ls}(x', y) * y \mapsto z \vDash \text{ls}(x, y) * y \mapsto z \\ \text{ls}(x, x') * \text{ls}(y, y') \vDash x \neq y & \end{array}$$

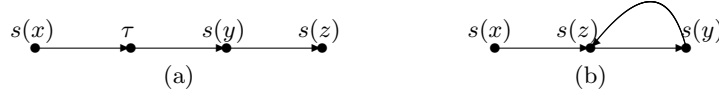


Fig. 1. (a) The fully acyclic heap. (b) The scorpion-like heap which destroys the validity of $(z \neq x) \wedge (z \neq y) \wedge \text{ls}(x, y) * y \mapsto z \models \text{ls}(x, z)$.

The three examples on the left illustrate the anti-aliasing or separating properties of $*$, where the two on the right illustrate issues to be taken into account in rules for appending onto the end of list segments (a point which is essential in completeness considerations [2]). Appending a cell to the head of a list is always valid, as long as we make sure the end-points are distinct:

$$(z \neq x) \wedge x \mapsto y * \text{ls}(y, z) \models \text{ls}(x, z),$$

whereas appending a cell to the tail of a list generally is not valid. Although $s(z) \neq s(x)$ and $s(z) \neq s(y)$ in Fig 1(b), the dangling z stings an intermediate point τ in $\text{ls}(x, y)$, so that

$$(z \neq x) \wedge (z \neq y) \wedge \text{ls}(x, y) * y \mapsto z \not\models \text{ls}(x, z).$$

To provide validity, we have to ‘freeze’ the dangling z , for instance, with $\text{ls}(z, v)$:

$$\text{ls}(x, y) * \text{ls}(y, z) * \text{ls}(z, v) \models \text{ls}(x, z) * \text{ls}(z, v)$$

3 Separated Abduction Problems

Before giving the main definitions, it will be helpful to provide some context by a brief discussion of how abduction can be used in program analysis.

We consider a situation where each program operation has preconditions that must be met for the operation to succeed. A pointer dereferencing statement $x \rightarrow \text{next} = y$ might have the assertion $x \mapsto x'$ as a precondition. A larger procedure might have preconditions tabulated as part of a ‘procedure summary’. Abduction is used during a forwards-running analysis to find out what is missing from the current abstract state at a program point, compared to what is required by the operation, and this abducted information is used to build up an overall precondition for a body of code.

For example, suppose that the assertion $\text{ls}(x, \text{nil}) * \text{ls}(y, \text{nil})$ is the precondition for a procedure (it might be a procedure to merge lists) and assume that we have the assertion $x \mapsto \text{nil}$ at the call site of the procedure. Then solving

$$x \mapsto \text{nil} * Y? \models \text{ls}(x, \text{nil}) * \text{ls}(y, \text{nil})$$

would tell us information we could add to the current state, in order to abstractly execute the procedure. An evident answer is $Y = \text{ls}(y, \text{nil})$ in this case.

Sometimes, there is additional material in the current state, not needed by the procedure; this unused portion of state is called the frame ([20], after the frame problem from Artificial Intelligence). Suppose $x \mapsto \text{nil} * z \mapsto w * w \mapsto z$ is the current state and $\text{ls}(x, \text{nil}) * \text{ls}(y, \text{nil})$ is again the procedure precondition, then $z \mapsto w * w \mapsto z$ is the leftover part. In order to cater for leftovers, abduction is performed with **true** as a $*$ -conjunct on the right. To see why, consider that

$$x \mapsto \text{nil} * z \mapsto w * w \mapsto z * Y? \models \text{ls}(x, \text{nil}) * \text{ls}(y, \text{nil}).$$

has no consistent solution, since the cycle between z and w cannot form a part of a list from y to `nil`. However, $Y = \text{ls}(y, \text{nil})$ is indeed a solution for

$$x \mapsto \text{nil} * z \mapsto w * w \mapsto z * Y? \models \text{ls}(x, \text{nil}) * \text{ls}(y, \text{nil}) * \text{true}.$$

In the approach of [6] a separate mechanism, frame inference, is used after abduction, to percolate the frame from the precondition to the postcondition of an operation. We will refer to this special case of the abduction problem, with `true` on the right, as the *relaxed* abduction problem.

With this as background, we now give the definition of the main problems studied in the paper.

Definition 3.1 (1) We say that X is a solution to the abduction problem $A * X? \models B$ if $A * X$ is consistent and the sequent $A * X \models B$ is valid.

(2) We use $\text{SAP}(\mapsto)$ to refer to the abduction problem restricted to formulae that have no `ls` predicate, and $\text{SAP}(\mapsto, \text{ls})$ for the general problem. The inputs are two symbolic heaps A and B . The output is ‘yes’ if there is a solution to the problem $A * X? \models B$, ‘no’ otherwise. $\text{rSAP}(\mapsto)$ and $\text{rSAP}(\mapsto, \text{ls})$ refer to the specializations of these problems when B is required to include `*true`.

We have formulated SAP as a decision problem, a yes/no problem. For applications to program analysis the analogous problem is a search problem: find a particular solution to the abduction problem $A * X? \models B$, or say that no solution exists. Clearly, the decision problem provides a lower bound for the search problem and as such, when considering NP-hardness we will focus on the decision problem. When considering upper bounds, we give algorithms for the search problem, and show membership of the decision problem in the appropriate class.

In general when defining abduction problems, the solutions are restricted to ‘abducible’ facts, which in classical logic are often conjunctions of literals. Here, the symbolic heaps are already in restricted form, which give us a separation logic analogue of the notion of abducible: `*`-conjunctions of points-to and list segment predicates, and \wedge -conjunctions of equalities and disequalities. Also, when studying abduction, one often makes a requirement that solutions be minimal in some sense. At least two criteria have been offered in the literature for minimality of SAP [6,22]. We do not study minimality here. Our principal results on NP-hardness apply as well to algorithms searching for minimal solutions as lower bounds, and we believe that our ‘easiness’ results concerning cases when polytime is achievable could carry over to minimality questions. We have concentrated on the more basic case of consistent solutions here, leaving minimality for the future.

4 Membership in NP

The main result of this section is the following, covering both $\text{SAP}(\mapsto, \text{ls})$ and $\text{rSAP}(\mapsto, \text{ls})$.

Theorem 4.1 (NP upper bound) *There is an algorithm, running in nondeterministic polynomial time, such that for any formulae A and B , it outputs a solution X to the abduction problem $A * X? \models B$, or says that no solution exists.*

Proof. Given A and B , let $\mathcal{Z} = \{z_1, z_2, \dots, z_n\}$ be the set of all variables and constants occurring in A and B . We define a set of *candidates* X in the following way. The spatial part Σ_X of X is defined as

$$x_1 \mapsto z_{i_1} * x_2 \mapsto z_{i_2} * \dots * x_m \mapsto z_{i_m}$$

where *distinct* x_1, x_2, \dots, x_m are taken from \mathcal{Z} and $\{z_{i_1}, z_{i_2}, \dots, z_{i_m}\} \subseteq \mathcal{Z}$ (for the sake of consistency we take x_1, x_2, \dots, x_m that are not L -values in A).

The pure part Π_X of X is defined as an \wedge -conjunction of formula of the form $(z_i = z_j)^{\varepsilon_{ij}}$, where $(z_i = z_j)^1$ stands for $(z_i = z_j)$, and $(z_i = z_j)^0$ stands for $(z_i \neq z_j)$.

The size of any *candidate* X is $\mathcal{O}(n^2)$ is quadratic in the size of A and B . Since consistency and entailment are in PTIME (Theorem 4.3 below), each *candidate* X can be checked in polynomial time as to whether it is a solution or not.

The Interpolation Theorem (Theorem 4.4 below) guarantees the completeness of our procedure. \square

The gist of this argument is that we can check a candidate solution in polynomial time, and only polynomial-sized solutions need be considered (by Interpolation). The entailment procedure we use to check solutions relies essentially on our use of *necessarily non-empty* list segments (as in [13]). For formulae with *possibly-empty* list segments, which are sometimes used (e.g., [19]), we do not know if entailment can be decided in polynomial time; indeed, this has been an open question since symbolic heaps were introduced [2].

However, we can still find an NP upper bound for abduction, even if we consider possibly empty list segment predicates. The key idea is to consider *saturated solutions* only, i.e., solutions which, for any two variables contain either an equality or disequality between them. For candidate saturated solutions there is no empty/non-empty ambiguity, and we can fall back on the polytime entailment procedure below. Furthermore, a saturation can be guessed by an NP algorithm.

This remark is fleshed out in the following theorem.

Theorem 4.2 (NP upper bound) *There is an algorithm, running in nondeterministic polynomial time, such that for any formulae A and B in the language extended with possibly-empty list segments, it outputs a particular solution X to the abduction problem $A * X? \vDash B$, or says that no solution exists.*

We now turn to the two results used in the proof of the above theorems.

Theorem 4.3 (Entailment/Consistency) *There is a sound and complete algorithm that decides $A \vDash B$ in polynomial time. As a consequence, consistency of symbolic heaps can also be decided in polynomial time.*

Proof Sketch. The main idea behind the algorithm is to turn the sequent $A \vDash B$ into an equi-valid sequent $A' \vDash B$ where all **ls** predicates in A have been converted to \mapsto predicates in A' . A sequent of this form can be decided using “subtraction” rules, i.e., rules that produce new equi-valid sequents whose antecedent and consequent have shorter spatial parts. E.g., it is the case that

$$C * x \mapsto y \vDash D * x \mapsto y \iff C \vDash D.$$

The procedure terminates when an axiomatically valid sequent is produced, e.g., $\text{emp} \vDash \text{emp}$ or $C \vDash \text{true}$.

The completeness of the algorithm rests on the fact that for a valid sequent $A \vDash \text{ls}(x_1, y_1) * \dots * \text{ls}(x_n, y_n) * T$ (where T is emp or true) we can uniquely partition A into n sets A_i that form non-empty paths from x_i to y_i . \square

Next we establish that the solutions to abduction can be obtained using variables only appearing in the antecedent and consequent. This, together with the fact that in consistent formulae there are *no repetitions of L-values* in different $*$ -conjuncts (thus the formula $x \mapsto y * x \mapsto y'$ is inconsistent), allows us to conclude that only *polynomial-sized* candidate solutions need be considered.

Theorem 4.4 (Interpolation Theorem) *Let X be a solution to the abduction problem: $A * X \vDash B$. Then there is an \widehat{X} , a solution to the same abduction problem, such that \widehat{X} uses only variables and constants mentioned in A or in B , and the spatial part of \widehat{X} consists only of ‘points-to’ subformulae.*

Proof Sketch. We sketch the main ideas with an example X .

First, note that we can get rid of all ls -subformulae in X , since X will be still a solution to the problem, even if we replace each $\text{ls}(x, y)$ occurring in X with the formula $(x \neq y) \wedge x \mapsto y$.

Suppose X of the form $X' * x_1 \mapsto z * \dots * x_k \mapsto z * z \mapsto y$ is a solution to the abduction problem $A * X \vDash B$, and z does not occur in A , or B , or X' .

In order to eliminate such an extra z , we replace X with \widehat{X}

$$\widehat{X} = X' * x_1 \mapsto y * \dots * x_k \mapsto y.$$

To check that such an \widehat{X} is a solution - that is, $(s, h) \vDash A * \widehat{X}$ implies $(s, h) \vDash B$, we construct a specific model (s', h_z) so that $(s', h_z) \vDash A * X$ with the original X .

Here we take advantage of the fact that z does not participate in h , and modify s with $s'(z) = c$ where c is *fresh*. To make h_z from h , we substitute the c for all occurrences of $s(y)$ in h related to $s(x_i)$ and then add the cell $c \mapsto \boxed{s(y)}$.

Being a solution, the original X provides that $(s', h_z) \vDash B$.

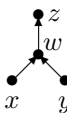
To complete the proof, it suffices to show that, because of our specific choice of the modified (s', h_z) , the poorer (s, h) is a model for B as well. \square

5 NP-completeness

We now show that the general separated abduction problem is NP-complete by reducing from 3-SAT.

The obstruction to a direct reduction from 3-SAT is that the Boolean disjunction \vee , which is a core ingredient of NP-hardness of the problem, is *not* expressible in our language. However, the following example illustrates how disjunctions over equalities and disequalities can be emulated through abduction.

Example 5.1 Define a formula A as follows, presented graphically to the right.

$$A \equiv x \mapsto w * y \mapsto w * w \mapsto z$$


Now, let X be an arbitrary solution to the abduction problem:

$$A * z \mapsto z' * X? \models u \neq v \wedge z \mapsto z' * \mathbf{ls}(x, u) * \mathbf{ls}(y, v) * \mathbf{true}$$

Then we can prove the following disjunction:

$$A * z \mapsto z' * X \models ((u=z) \wedge (v=w)) \vee ((v=z) \wedge (u=w)).$$

Thus any solution X provides either $\mathbf{ls}(x, z) * \mathbf{ls}(y, w)$, i.e. the path from x to z and the path from y to w , or $\mathbf{ls}(y, z) * \mathbf{ls}(x, w)$, i.e. the path from the leaf y to the root z and the path from the leaf x to the non-terminal vertex w . \square

This mechanism, which utilizes the semantics of lists and separation in tandem, achieves emulation of disjunction over pure formulae. We generalize this in the combinatorial lemma 5.2 and then put it to use in our reduction from 3-SAT.

Lemma 5.2 *The tree in Fig. 2(a) has exactly eight non-overlapping (having no common edges) paths from its leaves to distinct non-terminal vertices. The disjunction we will emulate is provided by the fact that each path leading from a leaf to the root is realizable within such a partition into non-overlapping paths.*

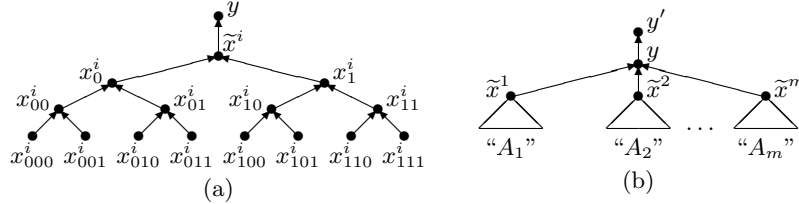


Fig. 2. (a) The graph presents A_i associated with a given clause C_i . (b) The graph presents the “whole” $A_0 * A_1 * A_2 * \dots * A_m$.

Now we can state our reduction. In essence, for each clause C_i we use a tree such as the one in Fig. 2a, and add appropriate disequalities so that any solution to the abduction problem selects a propositional valuation that satisfies all clauses.

Definition 5.3 (reduction) Given a set of 3-SAT clauses C_1, C_2, \dots, C_m , the problem we consider is to find an X , a solution to the abduction problem

$$A_0 * A_1 * \dots * A_m * X \models \Pi \wedge A_0 * B_1 * \dots * B_m * \mathbf{true} \quad (\text{P1})$$

where A_0 is $y \mapsto y'$, and each A_i is defined as a $*$ -conjunction of all formulae of the form (see Fig. 2(a)):

$$x_{\varepsilon_1 \varepsilon_2 \varepsilon_3}^i \mapsto x_{\varepsilon_1 \varepsilon_2}^i, \quad x_{\varepsilon_1 \varepsilon_2}^i \mapsto x_{\varepsilon_1}^i, \quad x_{\varepsilon_1}^i \mapsto \tilde{x}^i, \quad \tilde{x}^i \mapsto y$$

where $\varepsilon_1, \varepsilon_2, \varepsilon_3$ range over zeros and ones, and B_i is defined as a $*$ -conjunction of the form:

$$\mathbf{ls}(x_{000}^i, z_{000}^i) * \dots * \mathbf{ls}(x_{\varepsilon_1 \varepsilon_2 \varepsilon_3}^i, z_{\varepsilon_1 \varepsilon_2 \varepsilon_3}^i) * \dots * \mathbf{ls}(x_{111}^i, z_{111}^i)$$

For each i , the non-spatial part Π includes disequalities:

$$\begin{aligned} & (z_{\varepsilon_1 \varepsilon_2 \varepsilon_3}^i \neq x_{\varepsilon_1 \varepsilon_2 \varepsilon_3}^i), \\ & (z_{\varepsilon_1 \varepsilon_2 \varepsilon_3}^i \neq z_{\delta_1 \delta_2 \delta_3}^i), \text{ for } (\varepsilon_1, \varepsilon_2, \varepsilon_3) \neq (\delta_1, \delta_2, \delta_3), \\ & (z_{\varepsilon_1 \varepsilon_2 \varepsilon_3}^i \neq y), \text{ if } C_i(\varepsilon_1, \varepsilon_2, \varepsilon_3) \text{ is false.} \end{aligned} \quad (1)$$

For distinct i and j , Π also includes disequalities of the form

$$(z_{\varepsilon_1 \varepsilon_2 \varepsilon_3}^i \neq z_{\delta_1 \delta_2 \delta_3}^j) \quad (2)$$

whenever $(\varepsilon_1, \varepsilon_2, \varepsilon_3)$ and $(\delta_1, \delta_2, \delta_3)$ are *incompatible* - that is, they assign contradictory Boolean values to a common variable u from C_i and C_j . \square

5.1 From 3-SAT to the abduction problem (P1)

Let $(\alpha_1, \alpha_2, \dots, \alpha_n)$ be an assignment of a value 0(false) or 1(true) to each of the Boolean variables such that it makes all clauses C_1, \dots, C_m true. Then we can find a solution \tilde{X} to our abduction problem in the following way.

By $(\beta_1^i, \beta_2^i, \beta_3^i)$ we denote the part of the assignment related to the variables used in C_i , so that $C_i(\beta_1^i, \beta_2^i, \beta_3^i)$ is true, and for all i and j , $(\beta_1^i, \beta_2^i, \beta_3^i)$ and $(\beta_1^j, \beta_2^j, \beta_3^j)$ are compatible.

Let $v_1^i, v_2^i, v_3^i, v_4^i, v_5^i, v_6^i, v_7^i, v_8^i$ denote $y, \tilde{x}^i, x_0^i, x_1^i, x_{00}^i, x_{01}^i, x_{10}^i, x_{11}^i$, the non-terminal vertices in Fig. 2. As in Lemma 5.2, we construct eight non-overlapping paths leading from $x_{000}^i, x_{001}^i, \dots, x_{111}^i$ to distinct $v_{k_1}^i, v_{k_2}^i, \dots, v_{k_8}^i$, respectively, so that one path leads from $x_{\beta_1^i \beta_2^i \beta_3^i}^i$ to v_1^i , where $(\beta_1^i, \beta_2^i, \beta_3^i)$ is specified above.

The part X_i is defined as a set of the following equalities

$$\begin{aligned} (z_{000}^i = v_{k_1}^i), (z_{001}^i = v_{k_2}^i), (z_{010}^i = v_{k_3}^i), (z_{011}^i = v_{k_4}^i), \\ (z_{100}^i = v_{k_5}^i), (z_{101}^i = v_{k_6}^i), (z_{110}^i = v_{k_7}^i), (z_{111}^i = v_{k_8}^i) \end{aligned}$$

which contains, in particular, the equality $(z_{\beta_1^i \beta_2^i \beta_3^i}^i = y)$.

Example 5.4 Fig. 3 yields the following X_i :

$$\begin{aligned} (z_{000}^i = x_{00}^i), (z_{001}^i = x_0^i), (z_{010}^i = x_{01}^i), (z_{011}^i = y), \\ (z_{100}^i = x_{10}^i), (z_{101}^i = x_1^i), (z_{110}^i = x_{11}^i), (z_{111}^i = \tilde{x}^i) \end{aligned}$$

Lemma 5.5 With $\tilde{X} = \Pi \wedge X_1 \wedge X_2 \wedge \dots \wedge X_m$ we get a solution to (the non-relaxed version of) the abduction problem (P1):

$$\tilde{X} \wedge A_0 * A_1 * A_2 * \dots * A_m \models \Pi \wedge A_0 * B_1 * B_2 * \dots * B_m$$

Proof. It suffices to show that $X_i \wedge A_i \models B_i$ is valid for each i . \square

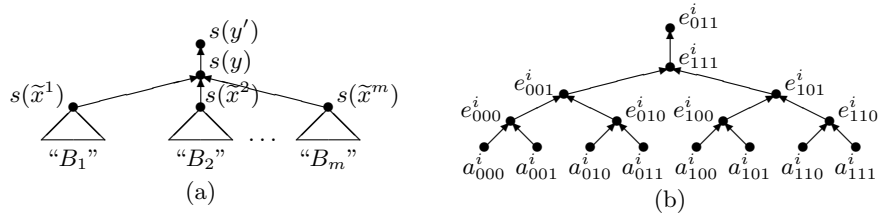


Fig. 3. (a) The graph depicts (s, h) , a model for $A_0 * B_1 * B_2 * \dots * B_m$. (b) The graph depicts the part (s, h_i) such that $(s, h_i) \models B_i$. Here, the following properties hold, $a_{\varepsilon_1 \varepsilon_2 \varepsilon_3}^i = s(x_{\varepsilon_1 \varepsilon_2 \varepsilon_3}^i)$, and $e_{\varepsilon_1 \varepsilon_2 \varepsilon_3}^i = s(z_{\varepsilon_1 \varepsilon_2 \varepsilon_3}^i)$, and $s(z_{011}^i) = s(y)$.

5.2 From the abduction problem (P1) to 3-SAT

Here we prove that our encoding is faithful.

Lemma 5.6 Given an X , a solution to the abduction problem (P1), we can construct an assignment $(\alpha_1, \alpha_2, \dots, \alpha_n)$ that makes all clauses C_1, \dots, C_m true.

Proof. Assume $(s, h') \models A_0 * A_1 * A_2 * \dots * A_m * X$.

Then $(s, h) \models A_0 * B_1 * B_2 * \dots * B_m$ for a ‘sub-heap’ h , and h can be split in heaps \hat{h} and h_1, h_2, \dots, h_m , so that $(s, \hat{h}) \models y \mapsto y'$, and

$$(s, h_1) \models B_1, (s, h_2) \models B_2, \dots, (s, h_m) \models B_m,$$

respectively. The non-overlapping conditions provide that any path in each of the h_i is blocked by the ‘bottleneck’ A_0 and hence cannot go beyond $s(y)$. Therefore, the whole h must be of the form shown in Fig. 3(a), and each of the h_i must be of the form shown in Fig. 3(b).

For every B_i , to comply with Π , these eight values $s(z_{\varepsilon_1 \varepsilon_2 \varepsilon_3}^i)$ must be one-to-one assigned to the eight non-terminal vertices in the tree in Fig. 3(b) (see Lemma 5.2).

Hence for each of the “non-terminal” variables $v_1^i, v_2^i, \dots, v_8^i$ in Fig. 2(a), X must impose equalities of the form

$$s(z_{\varepsilon_1 \varepsilon_2 \varepsilon_3}^i) = s(v_{k_\varepsilon}^i).$$

In particular, $s(z_{\delta_1^i \delta_2^i \delta_3^i}^i) = s(y)$ for some $(\delta_1^i, \delta_2^i, \delta_3^i)$. To be consistent with the third line of (1), $C_i(\delta_1^i, \delta_2^i, \delta_3^i)$ must be true. In addition to that, for distinct i and j , we get

$$s(z_{\delta_1^i \delta_2^i \delta_3^i}^i) = s(y) = s(z_{\delta_1^j \delta_2^j \delta_3^j}^j),$$

which makes these $(\delta_1^i, \delta_2^i, \delta_3^i)$ and $(\delta_1^j, \delta_2^j, \delta_3^j)$ compatible, in accordance with (2).

We assemble the desired assignment $(\alpha_1, \alpha_2, \dots, \alpha_n)$ in the following way.

For any Boolean variable u occurring in some clause C_i we take the triple $(\delta_1^i, \delta_2^i, \delta_3^i)$ specified above and assign the corresponding δ_ℓ^i to u . The fact that all $(\delta_1^i, \delta_2^i, \delta_3^i)$ and $(\delta_1^j, \delta_2^j, \delta_3^j)$ are compatible provides that our assignment procedure is well-defined. \square

Theorem 5.7 *The following problems are NP-complete, given formulae A, B :*

- (a) *Determine if there is a solution X to the problem $A * X? \models B$.*
- (b) *Determine if there is a solution X to the problem $A * X? \models B * \mathbf{true}$.*
- (c) *Determine if there is a solution X to the problem $A * X? \models B * \mathbf{true}$*

in the case where the spatial part of A uses only \mapsto -subformulae, and the models are confined to the heaps the length of any acyclic path in which is bounded by 5 (even if the spatial part of X is supposed to be the trivial \mathbf{emp}).

Proof. It follows from Lemmas 5.5 and 5.6, Theorem 4.1, and the fact that the tree height in Fig. 3(a) is bounded by 5. \square

Remark 5.8 It might seem that NP-hardness for $\mathbf{RSAP}(\mapsto, \mathbf{ls})$ should necessarily use lists of unbounded length. But, our encoding exploits only list segments of length no more than 5.

Remark 5.9 By Theorem 5.7, any solving algorithm running in polynomial time is likely to be *incomplete*. Consider, for instance, the abduction problem

$$x \mapsto y * y \mapsto z * w \mapsto y * X? \models \mathbf{ls}(x, a) * \mathbf{ls}(w, a) * \mathbf{true}.$$

There is a solution, namely, $y = a \wedge \mathbf{emp}$. However, the polynomial-time algorithm presented in [6] would stop without producing a solution, hence the incompleteness of that algorithm.

6 NP-completeness and PTIME results for \mapsto fragments

We have seen that the abduction problem for symbolic heaps is **NP**-complete in general. In this section we consider a restricted collection of formulae, which contain \mapsto but not **ls**. Such formulae occur often in program analysis, and form an important sub-case of the general problem. Here we find a perhaps surprising phenomenon: the general problem $\text{SAP}(\mapsto)$ is **NP**-complete, but the ‘relaxed’ problem $\text{RSAP}(\mapsto)$ can be solved in polynomial time. The relaxed problem, which has ***true** in the consequent, is relevant to program analysis (and is used in the **ABDUCTOR** tool), and thus the polynomial case is of practical importance.

6.1 $\text{SAP}(\mapsto)$ is **NP**-complete

Here, we show **NP**-hardness of $\text{SAP}(\mapsto)$ by reducing from the 3-partition problem [15]. The intuitions behind this reduction are as follows. (a) We coerce the abduction solution, if it exists, to be a conjunction of equalities, with **emp** as the spatial part; here the absence of **true** in the consequent is crucial. (b) The separating conjunction enables us to specify that distinct parts of the antecedent must be matched, via equalities, to distinct parts of the consequent.

Definition 6.1 (reduction) Given the 3-partition problem:

Given an integer bound b and a set of $3m$ integers s_1, s_2, \dots, s_{3m} , strictly between $b/4$ and $b/2$, decide if these numbers can be partitioned into triples $(s_{i_1}, s_{i_2}, s_{i_3})$ so that $s_{i_1} + s_{i_2} + s_{i_3} = b$.

the problem we consider is to find an X , a solution to the abduction problem

$$A_1 * \dots * A_m * X? \models \Pi \wedge B_1 * \dots * B_{3m} * C_1 * \dots * C_m \quad (\text{P2})$$

where A_j, B_i, C_k , and Π are defined as follows, here j and k range from 1 to m , and i ranges from 1 to $3m$ (cf. Fig. 4):

$$A_j = x_1^j \mapsto \tilde{x}^j * x_2^j \mapsto \tilde{x}^j * \dots * x_b^j \mapsto \tilde{x}^j * \tilde{x}^j \mapsto y$$

$$B_i = u_1^i \mapsto \tilde{u}^i * u_2^i \mapsto \tilde{u}^i * \dots * u_{s_i}^i \mapsto \tilde{u}^i$$

$C_k = w^k \mapsto y$, and Π consists of all disequalities of the form $y \neq x_\ell^j$, $y \neq \tilde{x}^j$, $\tilde{u}^i \neq x_\ell^j$, $\tilde{u}^i \neq y$, and $w^k \neq x_\ell^j$. \square

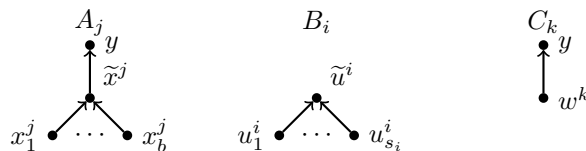


Fig. 4. The formulae used in Definition 6.1.

6.1.1 From 3-partition to the abduction problem (P2)

Here we transform any solution to the 3-partition problem into a solution \tilde{X} to the abduction problem (P2).

Suppose that for any j , $s_{3j-2} + s_{3j-1} + s_{3j} = b$.

First, we take Π as \tilde{X} and add to it all equalities of the form

$$\tilde{u}^{3j-2} = \tilde{u}^{3j-1} = \tilde{u}^{3j} = \tilde{x}^j = w^j.$$

For each j , we include in \tilde{X} all equalities of the form

$$(x_1^j = u_1^{3j-2}), \dots, (x_\ell^j = t_j(x_\ell^j)), \dots, (x_b^j = u_{s_{3j}}^{3j})$$

where t_j is a bijection t_j between the sets $\{x_1^j, \dots, x_b^j\}$ and $\{u_1^{3j-2}, \dots, u_{s_{3j-2}}^{3j-2}, u_1^{3j-1}, \dots, u_{s_{3j-1}}^{3j-1}, u_1^{3j}, \dots, u_{s_{3j}}^{3j}\}$. \square

Lemma 6.2 $\tilde{X} \wedge A_j \models B_{3j-2} * B_{3j-1} * B_{3j} * C_j$ is valid for every j . Hence \tilde{X} is a solution to the problem (P2).

6.1.2 From the abduction problem (P2) to 3-partition

Here we prove that our encoding is faithful.

Lemma 6.3 Given an X , a solution to (P2), we can construct a solution to the related 3-partition problem.

Proof. We suppose that $\sum_{i=1}^{3m} s_i = mb$.

Assume $(s, h) \models A_1 * A_2 * \dots * A_m * X$.

Then h can be split in heaps h_1, h_2, \dots, h_m , and h' , so that $(s, h_1) \models A_1$, $(s, h_2) \models A_2, \dots, (s, h_m) \models A_m$, and $(s, h') \models X$, respectively.

More precisely, each h_j is of the form

$$\begin{array}{ccccccc} \boxed{b^j} & \boxed{b^j} & \dots & \boxed{b^j} & \boxed{s(y)} \\ \uparrow & \uparrow & \uparrow\uparrow\uparrow & \uparrow & \uparrow \\ s(x_1^j) & s(x_2^j) & \dots & s(x_b^j) & b^j \end{array}$$

where $b^j = s(\tilde{x}^j)$. Notice that h_j can be uniquely identified by b^j .

Because of (P2), $(s, h) \models B_1 * \dots * B_{3m} * C_1 * \dots * C_m$.

The left-hand side of (P2) indicates the size of h as $m(b+1)$ plus the size of h' . The right-hand side of (P2) shows that the size of h is exactly $m(b+1)$. Bringing all together, we conclude that h' is the empty heap.

Let f_i be a part of h such that $(s, f_i) \models C_i$, and g_i be a part of h such that $(s, g_i) \models B_i$. To comply with Π , every $s(w^k)$ must be one-to-one assigned to one of these m points b^1, \dots, b^m , and every $s(\tilde{u}^i)$ must be assigned to one of these b^1, \dots, b^m , as well. The effect is that each h_j is the exact composition of a one-cell heap $f_{i'}$ and heaps $g_{j_1}, g_{j_2}, \dots, g_{j_\ell}$, whose domains are disjoint, and hence $1 + s_{j_1} + s_{j_2} + \dots + s_{j_\ell} = b+1$, which provides that $\ell=3$ and thereby the desired instance of the 3-partition problem. \square

Theorem 6.4 The following problems are NP-complete:

(a) Given formulae A and B whose spatial parts use only \mapsto -subformulae, determine if there is a solution X to the abduction problem $A * X? \models B$.

(b) Given formulae A and B whose spatial parts use only \mapsto -subformulae, determine if there is a solution X to the abduction problem $A * X? \models B$ in the case where the models are confined to the heaps the length of any acyclic path in which is bounded by 2 (even if the spatial part of X is supposed to be **emp**).

Proof. It follows from Lemmas 6.2 and 6.3 and Theorem 4.1, and the fact that the longest path in h_j is of length 2. \square

6.2 RSAP(\mapsto) is in PTIME

We will assume that all formulae in this subsection contain no list predicates, and make no more mention of this fact. We will also use the notation $\text{alloc}(A)$ to denote the set of variables x such that there is an L-value y in A and $A \models x = y$.

An algorithm which constructs a solution for $A * X? \models B * \text{true}$ if one exists or fails if there is no solution, is as follows. We check the consistency of the pure consequences of A and B and return false if they are not consistent. Then we transform the problem into one of the form $A' * X? \models \Sigma * \text{true}$, i.e., the consequent has no pure part, while guaranteeing that the two problem are in a precise sense equivalent. Next we subtract \mapsto -predicates from A' and Σ that have the same L-value. This step may also generate requirements (equalities) that the solution must entail. Finally we show that if this step cannot be repeated any more, then $A' * \Sigma$ is consistent and therefore Σ is a solution. We then transform this solution back to one for the original abduction problem.

Lemma 6.5 *The abduction problem $A * X? \models \Pi \wedge \Sigma$ has a solution if and only if $(\Pi \wedge A) * X? \models \Sigma$ has a solution. Moreover, if X is a solution for $(\Pi \wedge A) * X? \models \Sigma$, then $\Pi \wedge X$ is a solution for $A * X? \models \Pi \wedge \Sigma$.*

Thus we can concentrate on instances where the consequent has no pure part.

Lemma 6.6 *The following conjunctions are equivalent:*

$$\left\{ \begin{array}{l} A \text{ consistent} \\ A \models x = y \wedge w = z \wedge B \\ x \notin \text{alloc}(A) \end{array} \right\} \iff \left\{ \begin{array}{l} A \models x = y \\ A * x \mapsto w \text{ consistent} \\ A * x \mapsto w \models B * y \mapsto z \end{array} \right\}$$

Proof. Left-to-right: The entailment $A * x \mapsto w \models B * y \mapsto z$ follows by $*$ -introduction. The consistency of $A * x \mapsto w$ follows easily by an argument that whenever $x \notin \text{alloc}(C)$ for some consistent C , there exists a model (s, h) of C that does not include $s(x)$ in the domain of h .

Right-to-left: since $A * x \mapsto w$ is consistent then so is A . It is also easy to see that if $A \not\models w = z$ then $A \wedge w \neq z$ is consistent, thus there exist a countermodel for $A * x \mapsto w \models B * y \mapsto z$ where the cell at address $s(x)$ contains a value $c \neq s(z)$. Also, from the assumption of consistency trivially follows that $x \notin \text{alloc}(A)$.

Let $A \equiv E \wedge A'$ where E are the equalities appearing in A . Then,

$$A * x \mapsto w \models B * y \mapsto z \implies A'[E] * (x \mapsto w)[E] \models B[E] * (y \mapsto z)[E]$$

where the $[E]$ notation indicates substitution through the equalities in E . Let $a \equiv x[E]$ and $b \equiv w[E]$. Then, we can derive

$$A'[E] * a \mapsto b \models B[E] * a \mapsto b.$$

Let $A'' * a \mapsto b$ be the explicit, equivalent, version of $A'[E] * a \mapsto b$. Then it can be shown that $A'' * a \mapsto b \models B[E] * a \mapsto b$ implies $A'' \models B[E]$. Thus, $E \wedge A'' \models E \wedge B[E]$ and, by the substitution rule, $A \models B$. \square

Lemma 6.7 *Suppose A and Σ are such that (a) there are variables x, y such that $x \in \text{alloc}(A)$, $y \in \text{alloc}(\Sigma)$ and $A \models x = w$, and (b) there are no distinct predicates $x \mapsto y$, $w \mapsto z$ in Σ for which $A \models x = w$. Then $A * \Sigma$ is consistent.*

Theorem 6.8 (RSAP(\mapsto) is in PTIME) *There is a polytime algorithm that finds a solution for $A * X? \models B * \mathbf{true}$, or answers no otherwise.*

Proof. Let $B \equiv \Pi \wedge \Sigma$. If $\Pi \wedge A$ is inconsistent then clearly there is no solution, and we can check this in polynomial time.

The abduction problem $(\Pi \wedge A) * X? \models \Sigma * \mathbf{true}$ has a solution iff the original one has and we know how to transform a solution of the former to one of the latter through Lemma 6.5. Thus from now on we solve $(\Pi \wedge A) * X? \models \Sigma * \mathbf{true}$.

We repeatedly subtract the \mapsto predicates from antecedent and consequent for which the rule in Lemma 6.6 applies, to obtain an abduction problem with the side condition that certain variables may not appear as L-values in the solution.

When this is no longer possible we check the antecedent for consistency, as it is possible that the equalities introduced through this step contradict some disequality. For example, $y \neq w \wedge x \mapsto y * X? \models x \mapsto w * \mathbf{true}$.

Ultimately, we will arrive at an abduction problem $(\Pi' \wedge \Sigma') * X? \models \Sigma'' * \mathbf{true}$ which satisfies the conditions of Lemma 6.7. In this case, Σ'' is a solution as $(\Pi' \wedge \Sigma') * \Sigma''$ is consistent, and trivially, $(\Pi' \wedge \Sigma') * \Sigma'' \models \Sigma'' * \mathbf{true}$.

Thus we return $\Pi' \wedge \Sigma''$ as the solution for the original problem. \square

Remark 6.9 This result may seem surprising as RSAP(\mapsto , $\mathbf{1s}$) and SAP(\mapsto) are both NP-complete.

To illustrate the differences between RSAP(\mapsto) and RSAP(\mapsto , $\mathbf{1s}$) that allow this divergence, we consider the two abduction problems.

$$A * x \mapsto a_1 * a_1 \mapsto a_2 * \dots * a_{k-1} \mapsto a_k * X? \models x \mapsto y * B * \mathbf{true} \quad (3)$$

$$A * x \mapsto a_1 * a_1 \mapsto a_2 * \dots * a_{k-1} \mapsto a_k * X? \models \mathbf{1s}(x, y) * B * \mathbf{true} \quad (4)$$

One of the ingredients that makes the algorithm in Theorem 6.8 polynomial-time is the fact that for any solution X to the abduction problem (3), there is no other choice but to take y as a_1 . On the other hand, in problem (4), y can be made equal to a_1 , to a_2, \dots , to a_k , or something else. *It is this phenomenon we exploit (even with $k \leq 5$) to achieve NP-hardness in Theorem 5.7.*

In the case of RSAP(\mapsto) and SAP(\mapsto) another factor is at play. Consider, e.g., the two abduction problems, where x_1, \dots, x_k are not L-values in the LHS:

$$A * a_1 \mapsto b_1 * \dots * a_k \mapsto b_k * X? \models B * x_1 \mapsto y_1 * \dots * x_k \mapsto y_k \quad (5)$$

$$A * a_1 \mapsto b_1 * \dots * a_k \mapsto b_k * X? \models B * x_1 \mapsto y_1 * \dots * x_k \mapsto y_k * \mathbf{true} \quad (6)$$

To find a solution to (5), because of precision (Remark 2.3), x_1, \dots, x_k must be assigned to L-values in the left-hand side. We have at least $k!$ possibilities and each may need to be checked (*this point lies behind the NP-hardness in Theorem 6.4*).

In contrast, to find a solution to (6) we can opt for the most “conservative” candidate solution leaving x_1, \dots, x_k unassigned, or in other words, we can include $x_1 \mapsto y_1 * \dots * x_k \mapsto y_k$ as a part of a candidate solution X , since

$$a_1 \mapsto b_1 * \dots * a_k \mapsto b_k * X \models x_1 \mapsto y_1 * \dots * x_k \mapsto y_k * \mathbf{true}.$$

If such an X is *not* a solution then problem (6) has no solution at all.

These two observations are among the crucial points behind the design of the polynomial-time algorithm in Theorem 6.8. \square

7 Conclusion

Our results are summarized in the table below.

	SAP	RSAP
$\{\mapsto\}$	NP-complete	in PTIME
$\{\mapsto, \mathbf{ls}\}$	NP-complete	NP-complete

We have studied the complexity of abduction for certain logical formulae representative of those used in program analysis for the heap. Naturally, practical program analysis tools (e.g., ABDUCTOR, SLAYER, INFER, XISA) use more complicated predicates in order to be able to deal with examples arising in practice. For example, to analyze a particular device driver a formula was needed corresponding to ‘five cyclic linked lists sharing a common header node, where three of the cyclic lists have nested acyclic sublists’ [1].

The fragment we consider is a basic core used in program analysis. All of the separation logic-based analyses use at least a points-to and a list segment predicate. So our lower bounds likely carry over to richer languages used in tools.

Furthermore, the work here could have practical applications. The tools use abduction procedures that are incomplete but the ideas here can be used to immediately obtain procedures that are less incomplete (even when the fragment of logic they are using is not known to be decidable). Further, the polynomial-time sub-cases we identified correspond to cases that do frequently arise in practice. For example, when the ABDUCTOR tool was run on an IMAP server of around 230k LOC [11], it found consistent pre/post specs for 1150 out of 1654 procedures, and only 37 (or around 3%) of the successfully analyzed procedures had specifications involving list segments. The remainder (97% of the successfully analyzed procedures, or 67% of all procedures) had specs lying within the \mapsto -fragment which has a polynomial-time relaxed abduction problem (and the tool uses the relaxed problem). Furthermore, the 37 specifications involving list segments did not include any assertions with more than one list predicate. Indeed, we might hypothesize that in real-world programs one would only rarely encounter a single procedure that traverses a large number of distinct data structures, and having a variable number of list segment predicates was crucial in our NP-hardness argument. Because of these considerations, we expect that the worst cases of the separated abduction problem can often be avoided in practice.

ACKNOWLEDGEMENTS. This research was supported by funding from the EPSRC. O’Hearn also acknowledges the support of a Royal Society Wolfson Research Merit award and a gift from Microsoft Research.

References

1. J. Berdine, C. Calcagno, B. Cook, D. Distefano, P.W. O’Hearn, T. Wies, and H. Yang. Shape analysis of composite data structures. In *19th CAV*, 2007.
2. J. Berdine, C. Calcagno, and P.W. O’Hearn. Symbolic execution with separation logic. In *3rd APLAS, Springer LNCS 3780*, 2005.
3. N. Bjørner and J. Hendrix. Linear functional fixed-points. In *21st CAV, Springer LNCS 5643*, pages 124–139, 2009.

4. J. Brotherston and M.I. Kanovich. Undecidability of propositional separation logic and its neighbours. In *LICS*, pages 130–139. IEEE Computer Society, 2010.
5. C. Calcagno and D. Distefano. Infer: an automatic program verifier for memory safety of C programs. To appear in 3rd NASA Formal Methods Symposium, 2011.
6. C. Calcagno, D. Distefano, P.W. O’Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. In *36th POPL*, pages 289–300, 2009.
7. C. Calcagno, D. Distefano, and V. Vafeiadis. Bi-abductive resource invariant synthesis. In *APLAS*, pages 259–274, 2009.
8. C. Calcagno, P.W. O’Hearn, and H. Yang. Local action and abstract separation logic. In *LICS*, 2007.
9. P. Cousot and R. Cousot. Modular static program analysis. *11th Conference of Compiler Construction, Springer LNCS 2304*. pp159-178, 2002.
10. N. Creignou and B. Zanuttini. A complete classification of the complexity of propositional abduction. *SIAM J. Comput.*, 36(1):207–229, 2006.
11. D. Distefano. Attacking large industrial code with bi-abductive inference. In *14th FMICS, Springer LNCS 5825*, pages 1–8, 2009.
12. D. Distefano and I.Filipovic. Memory leaks detection in java by bi-abductive inference. In *13th FASE, Springer LNCS 6013*, pages 278–292, 2010.
13. D. Distefano, P. O’Hearn, and H. Yang. A local shape analysis based on separation logic. In *TACAS’06*, 2006.
14. T. Eiter and G. Gottlob. The complexity of logic-based abduction. *J. ACM*, 42(1):3–42, 1995.
15. M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.
16. R. Giacobazzi. Abductive analysis of modular logic programs. In *Proc. of the 1994 International Logic Prog. Symp.*, pages 377–392. The MIT Press, 1994.
17. B. Gulavani, S. Chakraborty, G. Ramalingam, and A. Nori. Bottom-up shape analysis. In *Static Analysis*, volume 5673 of *LNCS*, pages 188–204. Springer, 2009.
18. S. Gulwani, B. McCloskey, and A. Tiwari. Lifting abstract interpreters to quantified logical domains. In *35th POPL*, pages 235–246, 2008.
19. H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. O’Hearn. Scalable shape analysis for systems code. In *20th CAV, Springer LNCS 5123*, 2008.
20. S. Ishtiaq and P.W. O’Hearn. BI as an assertion language for mutable data structures. In *Proceedings of the 28th POPL*, pages 14–26, 2001.
21. S.K. Lahiri and S. Qadeer. Back to the future: revisiting precise program verification using SMT solvers. In *35th POPL*, pages 171–182, 2008.
22. C. Luo, F. Craciun, S. Qin, G. He, and W.-N. Chin. Verifying pointer safety for programs with unknown calls. *Journal of Symbolic Computation*, 45(11):1163–1183, 2010.
23. A. Möller and M.I. Schwartzbach. The pointer assertion logic engine. In *22nd PLDI*, pages 221–231, 2001.
24. G. Paul. Approaches to abductive reasoning: an overview. *Artif. Intell. Rev.*, 7(2):109–152, 1993.
25. C.S. Peirce. *The collected papers of Charles Sanders Peirce*. Harvard University Press, 1958.
26. M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM TOPLAS*, 20(1):1–50, 1998.
27. G. Yorsh, A.M. Rabinovich, M. Sagiv, A. Meyer, and A. Bouajjani. A logic of reachable patterns in linked data-structures. *J. Log. Algebr. Program.*, 73(1-2):111–142, 2007.