Compositional Shape Analysis by Means of Bi-Abduction

CRISTIANO CALCAGNO, Monoidics Ltd and Imperial College London
DINO DISTEFANO and PETER W. O'HEARN, Queen Mary University of London
HONGSEOK YANG, University of Oxford

The accurate and efficient treatment of mutable data structures is one of the outstanding problem areas in automatic program verification and analysis. Shape analysis is a form of program analysis that attempts to infer descriptions of the data structures in a program, and to prove that these structures are not misused or corrupted. It is one of the more challenging and expensive forms of program analysis, due to the complexity of aliasing and the need to look arbitrarily deeply into the program heap. This article describes a method of boosting shape analyses by defining a compositional method, where each procedure is analyzed independently of its callers. The analysis algorithm uses a restricted fragment of separation logic, and assigns a collection of Hoare triples to each procedure; the triples provide an over-approximation of data structure usage. Our method brings the usual benefits of compositionality—increased potential to scale, ability to deal with incomplete programs, graceful way to deal with imprecision—to shape analysis, for the first time.

The analysis rests on a generalized form of abduction (inference of explanatory hypotheses), which we call *bi-abduction*. Bi-abduction displays abduction as a kind of inverse to the frame problem: it jointly infers anti-frames (missing portions of state) and frames (portions of state not touched by an operation), and is the basis of a new analysis algorithm. We have implemented our analysis and we report case studies on smaller programs to evaluate the quality of discovered specifications, and larger code bases (e.g., sendmail, an imap server, a Linux distribution) to illustrate the level of automation and scalability that we obtain from our compositional method.

This article makes number of specific technical contributions on proof procedures and analysis algorithms, but in a sense its more important contribution is holistic: the explanation and demonstration of how a massive increase in automation is possible using abductive inference.

Categories and Subject Descriptors: D.2.4 [**Software Engineering**]: Software/Program Verification—*Correctness proofs*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*Mechanical verification; pre- and post-conditions*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*Denotational semantics; program analysis*

General Terms: Languages, Reliability, Theory, Verification

Additional Key Words and Phrases: Abstract interpretation, compositionality, static analysis, program proving, separation logic

This article reports work that was conducted over a number of years, made possible by significant support from the UK EPSRC: This includes the Smallfoot project, the Resource Reasoning Programme Grant, and Advanced Fellowships for C. Calcagno, P. W. O'Hearn, and H. Yang. Additionally, D. Distefano was supported by a Royal Academy of Engineering research fellowship and P. W. O'Hearn was supported by a Royal Society Wolfson Research Merit Award.

Authors' addresses: C. Calcagno, Department of Computing, Imperial College of Science, Technology, and Medicine, 180 Queen's Gate, London SW7 2BZ, United Kingdom; email: c.calcagno@imperial.ac.uk; D. Distefano and P. W. O'Hearn, School of Electronic Engineering and Computer Science, Queen Mary, University of London, London, E1 4NS, UK; email: ddino@dcs.qmul.ac.uk, ohearn@eecs.qmul.ac.uk; H. Yang, Computing Laboratory, University of Oxford, OUCL, Wolfson Building, Parks Road, Oxford, OX1 3QD, UK; email: hongseok00@gmail.com.

DOI 10.1145/2049697.2049700 http://doi.acm.org/10.1145/2049697.2049700

ACM Reference Format:
Calcagno, C., Distefano D., O'Hearn, P. W., and Yang, H. 2011. Compositional shape analysis by means of bi-abduction. J. ACM 58, 6, Article 26 (December 2011), 66 pages.
DOI = 10.1145/2049697.2049700 http://doi.acm.org/10.1145/2049697.2049700

1. INTRODUCTION

Abductive inference – inference of explanatory hypotheses – was introduced by Charles Peirce in his writings on the scientific process [Peirce 1958]. Abduction was formulated to distinguish the creation of new hypotheses from deductive and inductive (generalization) inference patterns. Abductive inference has been widely studied in philosophy, and it has been used in a number of ways in artificial intelligence, such as in diagnosis and in planning. (The survey articles [Kakas et al. 1992; Paul 1993] point to many formalizations and applications in the AI literature.)

This article is about the use of abductive inference in reasoning about computer programs. Just as abduction considered philosophically is thought of as a mechanism to *create hypotheses*, we propose that in reasoning about computer programs it can be used to *generate preconditions*, in the sense of pre/post specifications of program components. We state right away that the goal of abduction used in this way is not necessarily to supplant the human in the writing of specifications, but rather to complement him or her: we suggest that generated preconditions could be useful in program understanding or as a helpful guide during interactive proof efforts, and we demonstrate that they can be used to raise the scalability and level of automation in mechanized tools for program verification and analysis.

1.1. Context and Motivating Question

The context in which this research arose is verification of programs that access and mutate data structures in computer memory. We are interested in particular in the use of techniques from static program analysis to perform automatic verification. Program analyses can relieve the user of the burden of stating loop invariants and other annotations, and can in principle be applied more easily to existing code bases than can more manual techniques.

The last decade has seen a surge of interest in verification-by-static-analysis, with prominent success stories such as SLAM's application of proof technology to Microsoft device drivers [Ball et al. 2006] and ASTRÉE's proof of the absence of run-time errors in Airbus code [Cousot et al. 2005]. But, most practical tools for verification-oriented static analysis ignore pointer-based data structures, or use coarse models that are insufficient to prove basic properties of them; for example, SLAM *assumes* memory safety and ASTRÉE works only on input programs that do not use dynamic allocation. Data structures present one of the most pressing open problems in verification-oriented program analysis, progress on which would extend the reach of automatic program verification tools.

Although the general problem is not solved, there has been substantial prior work under the heading "shape analysis",[1] a kind of pointer analysis looks deeply into the heap in an attempt to discover invariants that describe the data structures in a program (e.g., Sagiv et al. [1998], Balaban et al. [2005], Podelski and Wies [2005], Guo et al. [2007], and Berdine et al. [2007]). A shape analysis can in principle prove that programs do not commit pointer-safety errors (dereferencing a null or dangling pointer,

[1]The data structure verification problem is broader and can concern more than shape, but we will persist with the term "shape analysis" in this article for backwards compatibility.

or leaking memory), without the user having to write loop invariants or even pre/post specifications for procedures; these are inferred during analysis. To do this, the analysis typically has to accurately identify (or distinguish) cyclic and acyclic linked structures, nested lists, and so on.

There are, though, some fundamental problems holding back the wider use of shape analyses in verification-by-static-analysis tools. First of all, shape analyses are notoriously expensive, and had mostly been applied to verification of codes in the tens or hundreds of LOC and, only recently, after much effort, to a restricted family of similar programs up to 10k LOC [Yang et al. 2008]. Second, in our prior work on SPACEINVADER attempting to apply shape analysis techniques to verification [Berdine et al. 2007; Calcagno et al. 2006; Yang et al. 2008], we were frustrated not only by the issue of scaling in terms of LOC, but even more by the amount of effort we had to put in *before* our analysis was run. We would have to write preconditions, or the verification analogue of a test harness, and this meant studying a program for days or weeks before "pushing the button". We speak of our personal experience with SPACEINVADER here, but believe that the same issue of "human time spent before pushing button" arises as well when attempting to apply any other existing shape analysis tools to verification problems. The final problem is that shape analyses have been formulated as whole-program analyses, meaning that it is difficult to apply them to a program as it is being written, before there is a complete program to analyze; this limits their potential to be used during the program-development process.

These "problems" all motivate the central question of this article, which is whether a *compositional* shape analysis can be formulated, which will help existing techniques be more easily applied to a greater variety of code.[2]

The term "compositionality" is used in different ways by different people, so we should be precise about our usage. We first recall Frege's original notion for language semantics: a semantic definition of a language is compositional if the meaning of a composite expression is defined in terms of the meanings of its parts. So, we say that a program analysis is compositional if the analysis result of a composite program (or program fragment) is computed from the analysis results of its parts. We can understand this definition at the level of granularity of procedures or groups of recursive procedures.

The most important feature of this definition of compositionality is that it presupposes that you can do the analysis of a part of a program without knowing the complete program.[3] If we can achieve compositionality, we are immediately in a position of being able to analyze incomplete programs. Further, compositional analysis, by its nature, easily yields an incremental algorithm. When an analysis is first run its results can be stored to disk. Then, if a procedure (or recursive procedure group) is changed, only it has to be re-analyzed; the old analysis results for independent procedures remain unchanged.

A second feature of compositionality is what we term *graceful imprecision*. There is no single existing shape domain that has yet been proposed which is appropriate to all of the kinds of data structures found in a large program (such as Linux): any known shape domain will deliver uselessly imprecise results at some point, after which (in a whole program analysis) meaningful results cease to be obtained even for portions of

[2]Note that we are not claiming that compositionality will "solve" the data structure verification problem. It can boost existing techniques, but is still limited by the limitations of the abstract domains in these techniques; for example, it remains difficult to precisely and efficiently analyze graph structures with significant internal sharing.

[3]Thus, in contrast to the usage in some papers that have appeared in the literature, a whole-program analysis that (say) uses procedure summaries to reduce work is not necessarily compositional in the sense we are using (or in Frege's original sense).

code which could be well treated, were a suitable precondition known. However, if a compositional analysis is unable to get precise results for one procedure, due to limitations of its abstract domain, it can still obtain precise results for other procedures.

So one could apply a compositional analysis partially, without waiting for the discovery of the "magical abstract domain" that covers all structures in all the large programs one might ever encounter. The human might then be brought into the loop by changing the domain, by supplying manual proofs for difficult parts of the code, etc. But, with compositionality, we are not prevented from getting started just because the magical domain doesn't yet exist.

Finally, compositional analyses are interesting for their potential to scale. With a compositional analysis, it becomes relatively easy to get meaningful (if partial) results for large code bases, as long as the analyses of the program parts are not prohibitive.

It is all well and good to wish for a compositional analysis with very many remarkable properties, but how might we go about obtaining one? Before we suggest our own approach, we must clarify what such an analysis should be aiming to do.

1.2. Compositionality, Overapproximation, Footprints

In formulating the compositional analysis, we take our lead from denotational semantics. For sequential programs, denotational semantics established that the meaning of a procedure can be taken to be a mathematical function from states to states which abstracts away from intermediate states of the code: for example, a function of type

$$\mathsf{ConcreteProcs} \stackrel{def}{=} \mathsf{States} \rightarrow \mathcal{P}(\mathsf{States}),$$

where the states can include input parameters as well as return values. What we would like to do in a compositional analysis is to over-approximate such a function space (what the Cousots call a "relational" analysis [Cousot and Cousot 2002]).

If we have a set of abstract states AbstractStates that overapproximate the concrete states (so that one abstract state may correspond to several concrete), then a naive way of overapproximating ConcreteProcs would be to use the function type

$$\mathsf{BadAbstractProcs} \stackrel{def}{=} \mathsf{AbstractStates} \rightarrow \mathcal{P}(\mathsf{AbstractStates}).$$

This would be impractical because BadAbstractProcs is far too large, especially in the case of shape analysis when there are very many abstract states. Something similarly unrealistic is obtained if one attempts to use the "most general client" for a procedure as a way to generate "procedure summaries," as that involves enumerating a great many abstract states. To be practical, we need to abstract the procedures more directly, without using a (total) function on the entirety of the abstract states, to get a more parsimonious abstraction.

Instead of an entire function table, we will use a collection of Hoare triples as our abstract domain. Although at first glance, it looks like this choice could be as bad as BadAbstractProcs, it is actually possible to get very compact representations of procedure meanings. The key to this is the "principle of local reasoning [O'Hearn et al. 2001]", which allows program specifications to concentrate on the *footprint*, the cells accessed by a procedure, and this allows for compact representations of procedure meanings. For example, a single Hoare triple

$$\{\text{the heap is an acyclic list rooted at } x\}\ \texttt{disposelist}(x)\ \{\text{the heap is empty}\}$$

is enough to specify a procedure to free all the elements in a linked list; a considerably smaller representation of the meaning than an element of BadAbstractProcs. (It might seem at first that this spec is not enough, because it does not talk heaps bigger than

the one containing only the list x, but an inference rule of separation logic, the frame rule, allows such specifications to be applied in wider contexts.)

In this example spec, the precondition describes exactly the cells accessed by the procedure (the footprint). This will be the aim of our compositional analysis: following on from the local reasoning idea, the analysis will aim to discover pre/post specs that talk about the footprint of a procedure only. Because of computability reasons, we know that we cannot make an algorithm that computes static descriptions of footprints perfectly, but we will use the footprint idea as a guide, to attempt to obtain compact over-approximations of procedure meanings.

Is this wishful thinking again? It is all well and good to wish for the footprint, but (quite apart from computability considerations) what might be an algorithmic technique to attempt to approximate it? Our answer is abduction, or rather a variant on the notion of abduction, and a program analysis algorithm which uses it in a new way.

1.3. Approach and Contributions

The basic idea of how our method works is this. We begin by trying to prove a program (either manually or automatically). During the proof attempt, we might find that we do not have a strong enough precondition to perform an operation – a procedure call, or a dereferencing. For example: we might need to know that a pointer is allocated, not dangling, before we attempt to free it; or, we might need to know that a variable points to an acyclic linked list, before we attempt to traverse it. We then perform abductive inference to infer what is missing, and hypothesize that this is part of the precondition that you need to describe the memory that the program uses. We abduce preconditions describing allocatedness of cells, or even entire linked lists, from the free operation or the code in a traversal procedure.

In fact, to treat procedure calls properly we will deal with a more general problem, which we call *bi-abduction*, that infers "frames" describing extra, unneeded portions of state (those cells that automatically remain unchanged) as well as the needed, missing portions (the "anti-frames").

We make two specific technical contributions which give us a way to leverage the above ideas in an automated tool: (i) we define an abductive proof procedure for separated heap abstractions, certain restricted separation logic formulas that have been used previously in automatic program analysis [Berdine et al. 2005b; Distefano et al. 2006]; (ii) we present a new heuristic algorithm, PREGEN, which uses bi-abduction to guide search for preconditions. After precondition generation, we use an existing forwards program analysis algorithm, POSTGEN, to generate postconditions (and to remove preconditions erroneously hypothesized by our heuristic search). The overall result is a compositional algorithm which finds pre/post specs approximating a procedure's memory usage, without knowledge or even existence of the procedure's calling context within a larger program.

Although these are new specific technical contributions of the article, the more significant contribution concerns how these ideas can be used together to increase automation in program verification. We have implemented our analysis algorithm in prototype tool, ABDUCTOR. We have done a case study on a number of small programs to examine the quality of synthesized specifications. To probe implications for scalability of automatic program verification techniques, we have also run ABDUCTOR on larger code-bases and open-source projects. The results we obtain there are partial (we are unable to prove anything about some procedures), but the significant point is that we are able to apply the analysis immediately, and get the partial information in the form of specs of some procedures without going through the weeks or months of prep work that was necessary previously. The larger code bases include a number

of open-source projects in the hundreds of thousands of LOC, and one of over three million LOC.

2. PRELIMINARY CONCEPTS AND EXAMPLES

The presentation in this article will consist of a gradual movement from logic towards program analysis. As we go along, we will use particular programs, and speak about program proofs or analysis in a semi-formal way, as a means to motivate and illustrate our techniques.

2.1. Abductive Inference

In standard logic, abduction can be set up as follows.

Given: assumption A and goal G.
To find: "missing" assumptions M making the entailment

$$A \wedge M \vdash G$$

true.

Constraints are often placed on what counts as a solution: that it be consistent, that it be expressed using a restricted collection of "abducible" facts, and sometimes that it be minimal in some sense. For example, in artificial intelligence it is often required that M be a conjunction of literals, rather than a general logical formula, thus ruling out the trivial $M = A \Rightarrow G$ solution. Minimality requirements rule out another trivial solution, $M = G$. In any case, the intuition of abduction is that, ideally, M should contain "just enough" information to conclude G, and even when this ideal cannot be reached perfectly the basic intuition can serve as a guide in the search for solutions.

In this article we will be solving a similar problem, but for separation logic [Ishtiaq and O'Hearn 2001; Reynolds 2002] rather than classical logic.

Given: assumption A and goal G.
To find: "missing" assumptions M making the entailment

$$A * M \vdash G$$

true.

Here, formulas describe sets of states, and the separating conjunction $A * M$ is true of a memory state just if A and M are true for disjoint partitions of the state: in words, A "and separately" M. (The formal semantics of our separation logic formulae will be given in Section 3.1.2).

In separation logic, the abduction question takes on a spatial connotation: In $A * M \vdash G$, we think of M as describing a missing portion of memory, which is in G but not in A. As in classical logic, there are immediate solutions to the question if expressive-enough logical machinery is present, such as by taking M to be the separating implication formula $A \mathrel{-\!\!*} G$. Motivated by program analysis, we will later constrain our solutions to be selected from certain special formulae called symbolic heaps, which will have the effect of ruling out such immediate solutions.

2.2. Generating Preconditions Using Abduction

Suppose that, during an attempted program verification we have an assertion A at a call site for a procedure, and the procedure has a precondition G. For example,

Procedure precondition: $G \stackrel{def}{=} \mathsf{list}(x) * \mathsf{list}(y)$,
Assertion at call site: $A \stackrel{def}{=} x \mapsto \mathsf{nil}$

Here the precondition says that x and y point to acyclic linked lists occupying separate memory, while the assertion A says that x is a pointer variable containing nil (so a list of length 1).

The difficulty we face in our attempted program proof is that the assertion A does not imply G; the call site and given precondition do not match up. One thing we can do is to give up on our verification, or our program analysis, at this point. But we might also realize

> if only we had the assertion $\mathsf{list}(y)$ as well, separately conjoined, then we would be able to meet the precondition.

The informal inference step expressed here is abductive in nature: it is about inferring an hypothesis concerning what is missing. Formally, it involves solving a question of the form

$$A * ?? \vdash G.$$

We can see the relevance of this question more clearly using a specific program. Suppose we are given a pre/post spec of a procedure `foo(x,y)`. This spec might have been computed previously in an automatic program analysis (where the jargon "procedure summary" is sometimes used for the spec), or it might have been supplied manually by a user. The $\mathsf{list}(\cdot)$ predicate describes acyclic, nil-terminated singly-linked lists. We have an enclosing procedure `p(y)` that calls `foo` as follows.

```
0  struct node {struct node* tl; };

1  struct node* p(struct node *y) { // Inferred Pre: list(y)
2    struct node *x;
3    x=malloc(sizeof(struct node)); x->tail =  0;
4    foo(x,y);  // Obtained Post: list(x)
5    return(x);
6  }  // Inferred Post: list(ret)
7
8  void foo(struct node *x,struct node *y){//SUMMARY ONLY
9     // Given Pre: list(x) * list(y)
10 }  // Given Post: list(x)
```

Here is how we can synthesize the pre and post described at lines 1 and 6. We begin by executing `p()` with starting symbolic heap emp, an assertion describing the empty memory. Just after line 3, we obtain the assertion $A = x{\mapsto}\mathsf{nil}$ that says that x is an allocated cell pointing to nil,[4] and for the following procedure call `foo(x,y)` we ask the abduction question

$$x{\mapsto}\mathsf{nil} * ?? \vdash \mathsf{list}(x) * \mathsf{list}(y).$$

A sensible answer is $?? = \mathsf{list}(y)$, so we infer that we should have started execution with $\mathsf{list}(y)$ rather than emp, and record $\mathsf{list}(y)$ as a missing precondition. We pretend that the analysis of the call at line 4 was successful, and continue the analysis of `p()` with the postcondition $\mathsf{list}(x)$ of `foo(x,y)`. At the end of procedure `p()`, we obtain $\mathsf{list}(\mathtt{ret})$ as the computed post, where `ret` is the value returned by the procedure.

Notice that the specification synthesized for `p()` is not at all random: the precondition describes the set of states on which the procedure can be safely run, presuming the given spec of `foo()`.

[4]For simplicity, in this example, we have ignored the possibility that `malloc()` might fail (returning 0); we will treat the failure case later (see Section 4.1).

2.3. Bi-Abduction

Abduction gives us a way to synthesize missing portions of state. We also have to synthesize additional, leftover portions of state (the frame) by solving a more general problem, which we term bi-abduction:

$$A * ?\textit{anti-frame} \vdash G * ?\textit{frame}.$$

We illustrate the use of bi-abduction with the following variation on the example above, using the same procedure summary as before for `foo()`.

```
1  struct node* q(struct node *y) { // Inferred Pre: list(y)
2    struct node *x, *z;
3    x=malloc(sizeof(struct node)); x->tail=0;
4    z=malloc(sizeof(struct node)); z->tail=0;
5    // Abduced: list(y), Framed: z|->0
6    foo(x,y);  // Obtained Post: list(x)*z|->0
7    // Abduced:emp, Framed: emp
8    foo(x,z);  // Obtained Post: list(x)
9    return(x);
10  }  // Inferred Post: list(ret)
```

This time we infer anti-frame $\mathsf{list}(y)$ as before, but using bi-abduction we also infer $z{\mapsto}\mathsf{nil}$ as the frame axiom that won't be needed by procedure call at line 6. That is, we obtain a solution to the bi-abduction question

$$x{\mapsto}\mathsf{nil} * z{\mapsto}\mathsf{nil} * ?\textit{anti-frame} \vdash \mathsf{list}(x) * \mathsf{list}(y) * ?\textit{frame}$$

where $?\textit{anti-frame} = \mathsf{list}(y)$, $?\textit{frame} = z{\mapsto}\mathsf{nil}$. We tack the frame on to the postcondition $\mathsf{list}(x)$ obtained from the procedure summary, continue execution at line 7, and this eventually gives us the indicated pre/post pair at lines 1 and 10.

Again, the inferred pre/post spec talks about only those cells that the procedure accesses. Such "small specifications" are useful to aim for when synthesizing pre- and postconditions, because (1) shape domains usually have an enormous number of states that might be used and (2) "small specifications" describe more general facts about procedures than "big specifications", so that they lead to a more precise analysis of callers of those procedures.

The more general point we wish to make is that bi-abduction gives us an inference technique to realize the principle of local reasoning stated earlier. Synthesis of frames allows us to *use* small specifications of memory portions by slotting them into larger states found at call sites, whereas abduction of anti-frames helps us to *find* the small specs.

2.4. Abstraction and Loops

Consider the simple program for freeing the elements of an acyclic linked list.

```
// start state: x=X /\ emp
// Inferred pre: list(x)
 1  void free_list(struct node *x) {
 2   while (x!=0) {
 3      t=x;
 4      x=x->tl;
 5      free(t);
 6   }
```

If we start symbolically executing the program from a starting state $x{=}X \wedge \mathsf{emp}$, then we will discover at line 4 that we cannot proceed without the assertion $x{\mapsto}X'$ that x is allocated, for some X'. It is convenient to express this fact using the additional variable X instead of x: the variable X does not appear in the program and will not be altered, where x will eventually be altered, so it is easier to use X to talk about pre-states. So we could update our precondition so that it changes from $x{=}X \wedge \mathsf{emp}$ to become to be $x{=}X \wedge X{\mapsto}X'$, and this gives us enough information to avoid memory faulting on the first iteration of the loop.

Then, on the second iteration, we run into the same problem again. Let us assume that our program analysis technique is smart enough to guess that $x{=}X \wedge (X{\mapsto}X' * X'{\mapsto}X'')$, for another new variable X'', rules out the memory faulting in *either* of the first or the second iteration of the loop. If we continue in this way, we get ever longer formulae to cover successive iterates

$$\begin{aligned} &x{=}X \wedge X{\mapsto}X' \\ &x{=}X \wedge (X{\mapsto}X' * X'{\mapsto}X'') \\ &x{=}X \wedge (X{\mapsto}X' * X'{\mapsto}X'' * X''{\mapsto}X''') \\ &\quad\vdots \end{aligned}$$

and if we do not do something we will obtain an infinite regress.

We can interrupt this regress using abstraction. At the second step, we can simplify our assertion by using the operation

$$\text{replace} \qquad x{=}X \wedge X{\mapsto}X' * X'{\mapsto}X'' \quad \text{with} \quad x = X \wedge \mathsf{ls}(X, X''),$$

where $\mathsf{ls}(X, X'')$ is a predicate that says that there is a linked list segment of indeterminate length from X to X'' (Our previous predicate $\mathsf{list}(x)$ is equivalent to $\mathsf{ls}(x, \mathsf{nil})$.) A similar step in the third iterate shows that our infinite regress has stopped (up to the names of non-program variables X'', X''', etc.)

before abstraction	after abstraction
$x{=}X \wedge X{\mapsto}X' * X'{\mapsto}X''$	$x{=}X \wedge \mathsf{ls}(X, X'')$
$x{=}X \wedge \mathsf{ls}(X, X'') * X''{\mapsto}X'''$	$x{=}X \wedge \mathsf{ls}(X, X''')$.

Quite apart from the technicalities concerning the additional variables X', X'' and so on, and the details of symbolic execution, it should be clear how abstraction can be used on candidate preconditions in this way.

Next, when we exit the loop we add the negation of the loop condition to the candidate precondition, as well as to the state being used in forwards symbolic execution. At that point it happens that $x{=}X''$ just before the loop condition on line 2, so the exit condition $X'' = 0$ and a bit of logic gives us precondition $x{=}X \wedge \mathsf{ls}(X, 0)$.

The key steps in this narrative involved abstraction, from two cells to a list segment or from a cell and a segment into one segment. Steps of this variety are commonly used in shape analyses in the calculation of loop invariants during forwards analysis, where they are deductively sound. In contrast, here our use of abstraction is tantamount to *weakening a precondition*, and so for the purpose of guessing preconditions is a potentially unsound step.[5] For this reason, we will use a standard forwards analysis to check our work after the fact.

[5]We think of what is happening here as an analogue of (scientific) induction: We have an hypothesis that the program works (does not commit a memory fault) for linked lists of length two as input, and we hazard a guess that it works for lists of arbitrary size.

The important point, though, is that in shape analyses for data structures it can often be the case that a reasonably general and sound assertion can be obtained from a specific concrete heap using an abstraction function. An informal explanation of why this sort of abstraction is not a completely unreasonable tactic is that it is unusual for a program to work safely on uninterrupted lists of length two or three, but not four or five. This explanation will be probed in experimental work in Section 5.

The description in this section has been oriented to showing the basic idea of how abductive and bi-abductive inference, and abstraction, can help generate preconditions for procedures. Extending the ideas to a whole programming language will involve some further heuristic choices, where we use abduction to guide guesses as to what good preconditions might be. The remainder of this article is devoted to filling out this basic idea, by working out theoretical and practical aspects of the inference questions, how they can be used within a synthesis algorithm that deals with standard programming constructs such as conditionals and loops, and by describing our experience with a tool implementing the ideas.

3. ABDUCTION AND BI-ABDUCTION FOR SEPARATED HEAP ABSTRACTIONS

In this section, we develop our theories of abduction and bi-abduction. After defining symbolic heaps, which are certain separation logic formulae, we describe a particular abductive procedure. The procedure is heuristic, in that it selects one out of a multitude of possible solutions. While limited by necessity, the heuristics are chosen based on programming examples, and by the requirement to apply the ideas to large code bases.

We then move on to investigate a concept of quality of solutions, and also explore the possibility of systematic (if expensive) algorithms in certain situations. Finally, we describe how to solve the bi-abductive inference question by pairing abduction with frame inference.

The reader most interested in applications to program analysis could read Sections 3.1 and 3.2 and then jump forward to the program analysis-oriented proof rules for programs in Section 3.5. Sections 3.3 and 3.4 contain more theoretical material on abduction itself, about the notion of quality of solutions, and should be relevant to readers interested in foundational questions surrounding the abduction notion.

3.1. Symbolic Heaps

3.1.1. Syntax. In this section, we define the special separation logic formulas that our abduction algorithm will work with. *Symbolic heaps* [Berdine et al. 2005b; Distefano et al. 2006] are formulae of the form

$$\exists \vec{X}.(P_1 \wedge \cdots \wedge P_n) \wedge (S_1 * \cdots * S_m),$$

where the P_i and S_j are primitive pure and spatial predicates, and $\vec{X}$ is a vector of logical variables (variables not used in programs). The special form of symbolic heaps does not allow, for instance, nesting of $*$ and $\wedge$, or Boolean negation $\neg$ around $*$, or the separating implication $-\!\!*$. This special form was chosen, originally, to match the idiomatic usage of separation logic in a number of by-hand proofs that had been done. Technically, it makes the problem of abduction and bi-abduction easier than it might otherwise be. For the purposes of this work, the restriction determines the collection of "abducible facts" that abduction must aim for: additively and separately conjoined collections of atomic predicates.

More formally, we assume two disjoint sets of variables: a finite set of program variables Var (ranged over by $x, y, z, \ldots$) and a countable set of logical variables LVar

(ranged over by $X, Y, Z, \ldots$). We assume a collection of constants (e.g., 0, 1, ...) ranged over by κ. The grammar for symbolic heaps H is as follows.

$$
\begin{array}{lll}
E & ::= & x \mid X \mid \kappa & \textit{Expressions} \\
P & ::= & \ldots & \textit{Basic pure predicates} \\
\Pi & ::= & P \mid \mathsf{true} \mid \Pi \wedge \Pi & \textit{Pure formulae} \\
S & ::= & \ldots & \textit{Basic spatial predicates} \\
\Sigma & ::= & S \mid \mathsf{true} \mid \mathsf{emp} \mid \Sigma * \Sigma & \textit{Spatial formulae} \\
\Delta & ::= & \Pi \wedge \Sigma & \textit{Quantifier-free symb. heaps} \\
H & ::= & \exists \vec{X}.\, \Delta & \textit{Symbolic heaps}
\end{array}
$$

The grammar for symbolic heaps has undetermined collections of basic pure and spatial predicates. Pure formulas are heap-independent, and describe properties of variables only, where the spatial formulas specify properties of the heap. One instantiation of these predicates is as follows.

SIMPLE LISTS INSTANTIATION

$$
\begin{array}{rcl}
P & ::= & E{=}E \mid E{\neq}E \\
S & ::= & E{\mapsto}E \mid \mathsf{ls}(E, E)
\end{array}
$$

Expressions E include program variables x, logical variables a, or constants κ (e.g., nil). Here, the *points-to* predicate $x{\mapsto}y$ denotes a heap with a single allocated cell at address x with content y, and $\mathsf{ls}(x, y)$ denotes a list segment from x to y (not included).

A more involved instantiation keeps P the same and replaces simple linked-lists by higher-order lists that are possibly empty or known to be nonempty [Berdine et al. 2007; Yang et al. 2008].

HIGHER-ORDER LISTS INSTANTIATION

$$
k ::= \mathsf{PE} \mid \mathsf{NE} \qquad S ::= (e{\mapsto}\vec{\mathtt{f}}\colon \vec{e}) \mid \mathsf{hls}\, k\, \phi\, e\, e
$$

Here, the points-to predicate $(e{\mapsto}\vec{\mathtt{f}}\colon \vec{e})$ is for records with fields $\vec{\mathtt{f}}$, and ϕ is a binary predicate that describes each node in a list. The definition of the nonempty list segment is

$$
\mathsf{hls}\,\mathsf{NE}\,\phi\,x\,y \iff \phi(x, y) \vee (\exists y'.\, \phi(x, y') * \mathsf{hls}\,\mathsf{NE}\,\phi\,y'\,y)
$$

and the ϕ predicate gives us a way to describe composite structures. The formula for possibly empty lists just replaces the left $\phi(e, f)$ disjunct by emp.

For examples of the higher-order predicates, let $\phi_{\mathtt{f}}(x, y)$ be the predicate $x{\mapsto}\mathtt{f}\colon y$. Then using $\phi_{\mathtt{f}}$ as ϕ, the formula $\mathsf{hls}\,\mathsf{NE}\,\phi\,x\,x$ describes lists linked by the f field. The formula

$$
(x{\mapsto}\,\mathtt{next}\colon y', \mathtt{tail}\colon z') \;*\; \mathsf{hls}\,\mathsf{PE}\,\phi_{\mathtt{next}}\,y'\,x \;*\; \mathsf{hls}\,\mathsf{PE}\,\phi_{\mathtt{tail}}\,z'\,x
$$

describes two circular linked lists sharing a common header, where one list uses next for linking and the other uses tail. Finally, if ϕ itself describes lists, such as when $\phi(x, y)$ is $\exists x'.\, (x{\mapsto}\,\mathtt{down}\colon x', \mathtt{next}\colon y) * \mathsf{hls}\,\mathsf{PE}\,\phi_{\mathtt{down}}\,x'\,0$, then $\mathsf{hls}\,\mathsf{NE}\,\phi\,x\,y$ describes a nonempty linked list where each node points to a possibly empty sublist, and where the sublists are disjoint. Combinations of these kinds of structures, nested lists and multiple lists with a common header node, are common in systems programs [Berdine et al. 2007].

The experiments in this article are done using a generalization of this second, more complex, instantiation, a generalization that accounts for doubly as well as singly

linked higher-order lists. For simplicity, we describe our algorithms and results mainly using the Simple Lists Instantiation, although they work equally well for many more sophisticated instantiations after slight or no modifications. Many other instantiations are possible, including predicates for various forms of tree [Chang et al. 2007].

The predicate emp holds in the empty heap where nothing is allocated. The formula $\Sigma_1 * \Sigma_2$ uses the separating conjunction of separation logic and holds in a heap h which can be split into two *disjoint parts* h_1 and h_2 such that Σ_1 holds in h_1 and Σ_2 in h_2. Symbolic heaps H have the form $\exists \vec{X}. \Pi \wedge \Sigma$, where only some (not necessarily all) logical variables in $\Pi \wedge \Sigma$ are existentially quantified. The set of all symbolic heaps is denoted by SH.

Logical variables are typically used to express limited forms of input-output relationships in procedure summaries. Typically, they are bound to initial values of input parameters or addresses of shared heap cells in a precondition. A postcondition, then, uses those variables to relate final states with values from initial states. For example, the pre/post spec $\{x{\mapsto}X * y{\mapsto}Y\} - -\{x{\mapsto}Y * y{\mapsto}X\}$ would be true of a procedure that exchanged the values of two pointers.

We overload the $*$ operator, so that it also works for Δ and H. For $i = 1, 2$, let $\Delta_i = \Pi_i \wedge \Sigma_i$ and $H_i = \exists \vec{X}_i. \Delta_i$ where all bound variables are distinct and they are different from free variables. We define $\Delta_1 * \Delta_2$ and $H_1 * H_2$ as follows:

$$\Delta_1 * \Delta_2 \stackrel{def}{=} (\Pi_1 \wedge \Pi_2) \wedge (\Sigma_1 * \Sigma_2), \quad H_1 * H_2 \stackrel{def}{=} \exists \vec{X}_1 \vec{X}_2. \Delta_1 * \Delta_2.$$

We overload $- * \Sigma$ and $\Pi \wedge -$ similarly:

$$\begin{array}{ll} \Delta_i * \Sigma \stackrel{def}{=} \Pi_i \wedge (\Sigma * \Sigma_i), & H_i * \Sigma \stackrel{def}{=} \exists \vec{X}_i. \Delta_i * \Sigma, \\ \Pi \wedge \Delta_i \stackrel{def}{=} (\Pi \wedge \Pi_i) \wedge \Sigma_i, & \Pi \wedge H_i \stackrel{def}{=} \exists \vec{X}_i. \Pi \wedge \Delta_i. \end{array}$$

A Convention. Sometimes, when writing symbolic heaps, we elide either the pure part or the spatial part when it is true; that is, we regard true $\wedge \Sigma$ as the same as Σ and Π as equivalent to $\Pi \wedge$ true. For example, in the next section, we write an entailment $x{\mapsto}x' * y{\mapsto}y' \models x \neq y$, which would more literally be rendered true $\wedge (x{\mapsto}x' * y{\mapsto}y') \models x \neq y \wedge$ true without this convention.

3.1.2. Semantics. We give a semantics whereby each symbolic heap determines a set of concrete states, those that satisfy it. Let Loc be a countably infinite set of locations, and let Val be a set of values that includes Loc. We use the following storage model:

$$\begin{array}{ll} \mathsf{Heap} & \stackrel{def}{=} \mathsf{Loc} \rightharpoonup_{\mathsf{fin}} \mathsf{Val} \\ \mathsf{Stack} & \stackrel{def}{=} (\mathsf{Var} \cup \mathsf{LVar}) \to \mathsf{Val} \\ \mathsf{States} & \stackrel{def}{=} \mathsf{Stack} \times \mathsf{Heap}. \end{array}$$

Heaps are finite partial functions mapping locations to values, and stacks are functions from both program and logical variables to values. States are pairs of stacks and heaps. Variations on this model are possible by adjusting the set Val: for instance, it could include records (tuples) that themselves could contain locations.

The semantics is given by a forcing relation $s, h \models A$ where $s \in$ Stack, $h \in$ Heap, and A is a pure assertion, spatial assertion, or symbolic heap. The partial binary operator $h_0 \uplus h_1$ on symbolic heaps is defined only if the domains of h_0 and h_1 are disjoint, and in that case the resulting partial function is their graph union. When we write $h = h_0 \uplus h_1$, this implies that $h_0 \uplus h_1$ is defined. We assume a semantics $[\![E]\!]s \in$ Val of pure expressions is given, where $[\![x]\!]s \stackrel{def}{=} s(x)$. Here are the generic semantic clauses,

followed by those for the basic predicates in the instantiation of symbolic heaps that we focus on in this article.

GENERIC SEMANTICS

$$
\begin{array}{lll}
s,h & \models \mathsf{true} & \text{always} \\
s,h & \models \Pi_0 \wedge \Pi_1 & \text{iff } s,h \models \Pi_0 \text{ and } s,h \models \Pi_1 \\
s,h & \models \mathsf{emp} & \text{iff } h = \emptyset \\
s,h & \models \Sigma_0 * \Sigma_1 & \text{iff } \exists h_0 h_1.\, h = h_0 \uplus h_1 \text{ and } s,h_0 \models \Sigma_0 \text{ and } s,h_1 \models \Sigma_1 \\
s,h & \models \Pi \wedge \Sigma & \text{iff } s,h \models \Pi \text{ and } s,h \models \Sigma \\
s,h & \models \exists \vec{X}.\Delta & \text{iff there is } \vec{v} \text{ where } (s|\vec{X} := \vec{v}), h \models \Delta
\end{array}
$$

BASIC PREDICATES

$$
\begin{array}{lll}
s,h & \models E \mapsto E' & \text{iff } h = [[\![E]\!]s \mapsto [\![E']\!]s] \\
s,h & \models E = E' & \text{iff } [\![E]\!]s = [\![E']\!]s \\
s,h & \models E \neq E' & \text{iff } [\![E]\!]s \neq [\![E']\!]s
\end{array}
$$

The list segment predicate is the least satisfying

$$\mathsf{ls}(E, E') \iff \mathsf{emp} \ \vee \ \exists y.E \mapsto y * \mathsf{ls}(y, E')\ .$$

Although $\vee$ is not officially in our fragment of symbolic heaps, the meaning should be clear. Note that this list segment predicate allows for cycles, so that $\mathsf{ls}(x, x)$ can describe either an empty heap, or a nonempty cyclic heap.

As bi-abduction is related to entailment, we set down the semantic notion of entailment, which will be used to judge the soundness of our inference techniques.

Definition 3.1 (*Semantic Entailment between Symbolic Heaps*). Given symbolic heaps H_1 and H_2, we define

$$H_1 \models H_2 \quad \text{iff} \quad \forall s, h.\, s,h \models H_1 \ \Rightarrow \ s,h \models H_2.$$

The reader might find it helpful to convince himself or herself of several example entailment questions.

SAMPLE TRUE AND FALSE ENTAILMENTS

$$
\begin{array}{rlrl}
x \mapsto x' * y \mapsto y' & \models x \neq y & \mathsf{ls}(x,x') * \mathsf{ls}(y,y') & \not\models x \neq y \\
x \mapsto x' * x \mapsto y' & \models \mathsf{false} & \mathsf{ls}(x,x') * \mathsf{ls}(y,y') & \not\models \mathsf{false} \\
\mathsf{emp} & \models \mathsf{ls}(x,x) & \mathsf{ls}(x,x) & \not\models \mathsf{emp} \\
x \mapsto x' * \mathsf{ls}(x',y) & \models \mathsf{ls}(x,y) & \mathsf{ls}(x,y) & \not\models \exists X.x \mapsto X * \mathsf{true}
\end{array}
$$

3.1.3. Abstraction. Generally, for whatever the primitive predicates in the symbolic heaps, we expect to have a function

$$\mathsf{abstract}^{\#} : \mathsf{SH} \to \mathsf{SH}.$$

Typically, such a function is like a syntactic analogue of the abstraction function in abstract interpretation. An essential property is

$$H \models \mathsf{abstract}^{\#} H.$$

This entailment is needed for the soundness of deductive analysis, and it also conveys a sense of generalization. In fact, for any concrete heap h, we can define a canonical symbolic heap H_h that identifies h up to isomorphism, and then the abstraction function $\mathsf{abstract}^{\#}(H_h)$ can be seen as generalizing from a particular concrete heap.

A useful, though not essential, property is that the range of $\mathsf{abstract}^{\#}$ be finite.

We will not set down an abstraction function here in detail, but we give two rewriting rules that have been used previously in the definition of particular such functions.

$$(\rho, \rho' \text{ range over } \mathsf{ls}, \mapsto)$$

$$\exists \vec{X}.\Delta * \rho(x, Y) * \rho'(Y, Z) \longrightarrow \exists \vec{X}.\Delta * \mathsf{ls}(x, Z) \quad \text{where } Y \text{ not free in } \Delta$$

$$\exists \vec{X}.\Delta * \rho(Y, Z) \longrightarrow \exists \vec{X}.\Delta * \mathsf{true} \quad \text{where } Y \text{ not provably reachable from program vars}$$

The first rule says to forget about the length of uninterrupted list segments, where there are no outside pointers (from Δ) into the internal point. Our abstraction "gobbles up" logical variables appearing in internal points of lists, by swallowing them into list segments, as long as these internal points are unshared. This is true of either free or bound logical variables. The requirement that they not be shared is an accuracy rather than soundness consideration; we stop the rules from firing too often, so as not to lose too much information [Distefano et al. 2006].

The second rule says to forget about the specifics of unreachable cells, and there it is significant that we use true rather than emp in right-hand side of the rule: true showing up in a symbolic heap is an indicator of a potential memory leak.

As an example of the first rule, if we have a concrete heap of size 4 corresponding to the following formula

$$x{\mapsto}X_1 * y{\mapsto}X_1 * X_1{\mapsto}X_2 * X_2{\mapsto}0,$$

then this is abstracted to

$$x{\mapsto}X_1 * y{\mapsto}X_1 * \mathsf{ls}(X_1, 0).$$

This step has generalized from a single (up to isomorphism) concrete heap to an infinite class of concrete heaps.

3.2. An Heuristic Algorithm for Abduction

We define a proof system for solving the abduction question for symbolic heaps, and explain how it gives rise to an algorithm. The basic idea is to distinguish a position in a sequent that holds abduced facts, and to fill in this position using decisions about matching predicates and instantiations of logical variables. This basic idea is illustrated in this section using a simple proof system for abduction that has been designed for use in a program proof tool that runs on large code bases. The procedure is purposely incomplete, avoiding consideration of expensive cases which seem not to come up so often in practice: this leads to a very fast algorithm, as well as a one that is relatively simple to describe.

3.2.1. Proof System for Abductive Inference. The proof system involves judgments of the form

$$H_1 * [M] \quad \triangleright \quad H_2, \tag{1}$$

$$\frac{H_1 * [M] \ \rhd \ H_2}{(\exists X.\, H_1) * [M] \ \rhd \ H_2}\ \textsf{exists} \quad (\text{where } X \notin \textsf{FreeLVar}(M, H_2))$$

$$\frac{(\Pi \wedge \Sigma) * [M] \rhd H \quad \Pi \wedge \Sigma \vdash \exists \vec{X}.\Pi' \quad \Pi \wedge \Pi' \wedge \textsf{emp} \vdash \Sigma'}{(\Pi \wedge \Sigma) * [M] \ \rhd \ H * (\exists \vec{X}.\, \Pi' \wedge \Sigma')}\ \textsf{remove}$$

$$\frac{(E_0{=}E_1 \wedge \Delta) * [M] \ \rhd \ \exists \vec{Y}.\, \Delta'}{\Delta * E{\mapsto}E_0 * [\exists \vec{X}.\, E_0{=}E_1 \wedge M] \ \rhd \ \exists \vec{X}\vec{Y}.\, \Delta' * E{\mapsto}E_1}\ {\mapsto}\textsf{-match} \quad (\text{where } \vec{Y} \cap \textsf{FreeLVar}(E_1) = \emptyset)$$

$$\frac{\Delta * [M] \ \rhd \ \exists \vec{X}.\, \Delta' * \textsf{ls}(E_0, E_1)}{\Delta * B(E, E_0) * [M] \ \rhd \ \exists \vec{X}.\, \Delta' * \textsf{ls}(E, E_1)}\ \textsf{ls-right} \quad (\text{where } B(E, E_0) \text{ is } E{\mapsto}E_0 \text{ or } \textsf{ls}(E, E_0))$$

$$\frac{(E{\neq}E_0 \wedge \Delta * \textsf{ls}(X, E_0)) * [M] \ \rhd \ \exists \vec{Y}.\, \Delta'}{\Delta * \textsf{ls}(E, E_0) * [E{\neq}E_0 \wedge M] \ \rhd \ \exists X\vec{Y}.\, \Delta' * E{\mapsto}X}\ \textsf{ls-left}$$

$$\frac{}{(\Pi \wedge \textsf{emp}) * [\exists \vec{X}.\, \Pi' \wedge \textsf{emp}] \ \rhd \ \exists \vec{X}.\, \Pi' \wedge \textsf{emp}}\ \textsf{base-emp}$$

$$\frac{}{\Delta * [\exists \vec{X}.\, \Pi \wedge \textsf{emp}] \ \rhd \ \exists \vec{X}.\, \Pi \wedge \textsf{true}}\ \textsf{base-true}$$

$$\frac{\Delta * [M] \ \rhd \ \Delta' \qquad \Delta * \exists \vec{X}.\, B(E, E') \not\vdash \textsf{false}}{\Delta * [M * \exists \vec{X}.\, B(E, E')] \ \rhd \ \Delta' * \exists \vec{X}.\, B(E, E')}\ \textsf{missing} \quad (\text{where } B(E, E') \text{ is } E{\mapsto}E' \text{ or } \textsf{ls}(E, E'))$$

Fig. 1. Proof rules for abductive inference.

where M and H_1 and H_2 are symbolic heaps. The M component in this judgment form is the abduced or "missing" fact, and the proof rules will be organized in such a way that proof search gives us a way to *find* M. Note that this is explicitly a three-place judgment $(\cdot) * [\cdot] \ \rhd \ (\cdot)$. The use of the $*[\cdot]$ notation is to evoke the intended soundness requirement, that $H_1 * M \models H_2$ should be a valid entailment.

The proof rules are presented in Figure 1. They are for the Simple List instantiation of symbolic heaps using $E{\mapsto}E$ and a single inductive predicate $\textsf{ls}(E, E)$. In Section 3.2.3, we will generalize the treatment to cover a range of inductive definitions. In Figure 1, and elsewhere in this section, we assume that all the bound logical variables are different from one another and also from free logical variables.

The rule remove in Figure 1 includes two plain entailment questions (not abduction questions) $H_0 \vdash H_1$ between symbolic heaps. What we require is a sound theorem prover for such entailments, where when the theorem prover says yes to $H_0 \vdash H_1$ it follows that $H_0 \models H_1$ holds. In this section, we do not describe such a theorem prover for plain entailment, viewing it as a parameter to our proof system. Theorem proving techniques for symbolic heaps have been described in Berdine et al. [2005b] and Nguyen and Chin [2008], and implemented in a number of tools including SMALLFOOT, SPACEINVADER, HIP/SLEEK, VERIFAST, SLAYER and JSTAR.

We illustrate the proof rules with a few examples. The first one involves instantiation of a logical variable which, during program analysis, is essential for using procedure summaries (specs) that themselves contain logical variables.

Example 3.2. Consider the inference question

$$x{\mapsto}y * [??] \ \rhd \ x{\mapsto}X * \textsf{ls}(X, 0) * \textsf{true}.$$

In this case, the abduction should infer that y is an instantiation of the logical variable X, in addition to the fact that the $\mathsf{ls}(X, 0)$ predicate is missing in the assumption. The inference algorithm finds such an instantiation using the $\mapsto$-match rule:

$$\dfrac{\dfrac{\dfrac{}{(y = X \wedge \mathsf{emp}) * [\mathsf{emp}] \;\rhd\; \mathsf{true}}\,\text{base-true}}{(y = X \wedge \mathsf{emp}) * [\mathsf{ls}(X, 0)] \;\rhd\; \mathsf{ls}(X, 0) * \mathsf{true}}\,\text{missing}}{x \mapsto y * [y = X \wedge \mathsf{ls}(X, 0)] \;\rhd\; x \mapsto X * \mathsf{ls}(X, 0) * \mathsf{true}}\,\mapsto\text{-match}$$

The last step of the derivation uses $\mapsto$-match to find an instantiation of X of the form $y = X$, and adds it to the assumption, before proceeding with the application of rule missing. The other two steps simply move the remaining $\mathsf{ls}(X, 0)$ predicate from the conclusion to the anti-frame part.

In relating this derivation to the literal proof rules, note that in the example we have used our convention, stated at the end of Section 3.1.1, to elide the pure part of a symbolic heap when it is true. Also, in the application of the missing we have elided the plain entailment statements (which are properly considered side conditions rather than premises): they are trivial in this case as both Π' and Σ' are instantiated to true in this application of the rule.

This example illustrates how our proof method goes beyond previous work on theorem proving with separation logic. It would be possible to obtain a sound theorem prover for abduction by a simple modification of an existing theorem prover [Berdine et al. 2005b]: An abduction question $H_1 * ?? \vdash H_2$ would be solved by attempting to prove the entailment $H_1 \vdash H_2$ and, when this proof attempt fails with one undischarged assumption $\mathsf{emp} \vdash M$, one could conclude that the missing heap is M. However, such a prover would not be powerful enough to find the instantiation of logical variables we have shown before. For instance, in Example 3.2, it would fail to find $y{=}X$, and instead infer $x{\mapsto}X * \mathsf{ls}(X, 0)$ as a missing anti-frame. As a consequence, it would generate the symbolic heap $x{\mapsto}y * x{\mapsto}X * \mathsf{ls}(X, 0)$ as a candidate antecedent, even though it is an inconsistent formula; certainly, this is not a desirable solution.

Example 3.3. Next, we consider a slightly modified version of the motivating example from Section 2:

$$x{\mapsto}z * [??] \;\rhd\; \mathsf{ls}(x, z) * \mathsf{ls}(y, 0) * \mathsf{true}$$

The following derivation finds a solution to this abduction question:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\mathsf{emp} * [\mathsf{emp}] \;\rhd\; \mathsf{true}}\,\text{base-true}}{\mathsf{emp} * [\mathsf{ls}(y, 0)] \;\rhd\; \mathsf{ls}(y, 0) * \mathsf{true}}\,\text{missing}}{\mathsf{emp} * [\mathsf{ls}(y, 0)] \;\rhd\; \mathsf{ls}(z, z) * \mathsf{ls}(y, 0) * \mathsf{true}}\,\text{remove}}{x \mapsto z * [\mathsf{ls}(y, 0)] \;\rhd\; \mathsf{ls}(x, z) * \mathsf{ls}(y, 0) * \mathsf{true}}\,\text{ls-right}$$

The last step subtracts $x{\mapsto}z$ from $\mathsf{ls}(x, z)$, and the second last step removes the result $\mathsf{ls}(z, z)$ of this subtraction, because $\mathsf{emp} \vdash \mathsf{ls}(z, z)$. The remaining steps move the predicate $\mathsf{ls}(y, 0)$ from the conclusion to the anti-frame.

In the derivation in Example 3.3, the application of the remove rule is crucial to obtain a better solution. Without using the rule, we could apply missing twice, once for $\mathsf{ls}(z, z)$ and then for $\mathsf{ls}(y, 0)$, and this would give us

$$x{\mapsto}z * [\mathsf{ls}(z, z) * \mathsf{ls}(y, 0)] \;\rhd\; \mathsf{ls}(x, z) * \mathsf{ls}(y, 0) * \mathsf{true}.$$

Note that the inferred anti-frame is not as good as $\mathsf{ls}(y, 0)$, because it has a bigger symbolic heap than $\mathsf{ls}(y, 0)$.

Example 3.4. Consider the abduction question:

$$x{\mapsto}3 \ * \ [??] \ \vdash \ y{\mapsto}3 \ * \ \mathsf{true}$$

Our proof system finds as a solution $y{\mapsto}3$. However, another solution is $(y = x \wedge \mathsf{emp})$, and (if we had disjunction) another would be $y{\mapsto}3 \vee (y = x \wedge \mathsf{emp})$.

Example 3.4 shows an important heuristic choice in the proof system: it does not explore all aliasing possibilities for solutions to the abduction problem. This choice is motivated by the fact that we use this proof system in an analyzer that discovers preconditions as well as the following three concerns. The first is that, to apply our algorithm to large code bases, we want to avoid creating large disjunctions that do case analysis on whether two pointers are equal or not (it would involve comparing the left sides of $\mapsto$ or linked list assertions, pairwise). Keeping control of disjunctions is essential for performance [Yang et al. 2008]. Second, as we discuss in the following sections, it is nontrivial to see how to compute the best solution at all (once we lay dow the notion of "best"), let alone efficiently. And third, it seems that in either by-hand or automatic program proofs this aliasing case is rarely needed. In particular, it will turn out that if a program checks for the aliasing (such as in traversal of cyclic list), two possibilities (aliased or not) are considered anyway by our program analysis algorithm: the conditional which checks for aliasing triggers a case analysis. However, if the program does not check aliasing, the prover doesn't either. We will return to this point in Section 5.1.

3.2.2. Reading the Proof System as an Algorithm. The proof system for abductive inference, when read in the usual premises-to-conclusion way, lets us easily see that the inferences we are making are sound. When read in the opposite direction, it can also be thought of as a specification of an algorithm for finding missing hypotheses M. The algorithm is obtained by reading the rules bottom-up, *and* by viewing the M parts as unknowns. There is also a pragmatically motivated order to the application of the rules, which we describe. This reading of the proof rules leads immediately to a recursive program, which forms the basis of our implementation.

The key point is that our proof rules have a special form

$$\frac{H_1' * [M'] \ \rhd \ H_2' \qquad \text{Cond}}{H_1 * [M] \ \rhd \ H_2}.$$

Here Cond is a condition involving parts of H_1 and H_2. The algorithmic reading of this rule is the following. In order to answer the entailment question $H_1 * ?? \vdash H_2$, the side condition Cond is first checked, and if it holds, we make a recursive call to answer the smaller question $H_1' * ?? \vdash H_2'$. The solution M' of this simpler question is then used to compute the solution M of the original question. For instance, the rule $\mapsto$-match fires when both the left-hand side and right-hand side have a points-to-fact involving E:

$$\Delta * E{\mapsto}E_0 * ?? \ \vdash \ \exists \vec{X}\vec{Y}.\, \Delta' * E{\mapsto}E_1.$$

The inference engine then cancels out those facts, and continues the search for the solution with the reduced right-hand side $\exists \vec{Y}.\, \Delta'$ and the reduced left-hand side Δ after adding the equality $E_0 = E_1$ concerning the contents of cell E. Later, when this new simplified search gives a result M, we conjoin the assumed equality to the computed missing anti-frame M and existentially quantify logical variables $\vec{X}$, which gives the result of the original search.

Algorithm 1 Finding Abduced Facts by Proof Search

INPUT: Symbolic heaps H_1 and H_2.
OUTPUT: M where $H_1 * [M] \rhd H_2$ is derivable, or exception fail.

$\mathsf{Abduce1}(H_1, H_2) \stackrel{def}{=}$

IF either axiom base-emp or base-true applies,
    return the M indicated by the axiom
ELSEIF one of the remaining rules from Figure 1 applies,
    select the lowest ranking rule CURRENTRULE
        (in order vertically from top of page),
    let $M' = \mathsf{Abduce1}(H_1', H_2')$, for H_1', H_2' as in CURRENTRULE,
    compute M from M' according to CURRENTRULE, return M.
ELSE return fail.

Our abduction algorithm tries to apply the rules in Figure 1 in the order in which they appear in the figure; see Algorithm 1. This algorithm has cubic worst-case running-time (and better typical case), if we count the calls to the underlying theorem prover $\vdash$ as having linear complexity. After checking whether an axiom applies, the algorithm attempts to use remove and eliminates a part of the symbolic heap on the right-hand side of $\rhd$ that holds for the empty heap. Once this phase is complete, the inference goes through each predicate on the right-hand side and tries to simplify the predicate using $\mapsto$-match, ls-right and ls-left. When this simplification process gets stuck, the algorithm applies missing, base-emp and base-true, and moves the remaining predicates from the right-hand side to the missing anti-frame part.

By arranging the order of rule applications in this way, our inference tries to minimize the size of the spatial part of the inferred missing anti-frame M. This is thought of as desirable according to the definition of the ordering on solutions $\precsim$ given in Section 3.3. By trying the remove rule before missing, the inference prefers choosing the empty heap to moving a predicate from the conclusion to the anti-frame part. For instance, given $\mathsf{emp} * [??] \vdash \mathsf{ls}(x, x)$, the remove rule infers emp as the missing anti-frame whereas the missing rule returns $\mathsf{ls}(x, x)$. The inference algorithm returns emp between the two, because it tries remove before missing. Also, the application of the simplification rules (i.e., $\mapsto$-match, ls-right and ls-left) before the missing rule ensures that the common parts between the assumption and the conclusion are canceled out as much as possible, before trying to move predicates from the conclusion to the anti-frame part.

Example 3.5. Algorithm 1 finds the anti-frames discussed in examples so far. An example of where it fails is $\mathsf{abduce}(x = 0, x \mapsto -)$. This returns fail, as no rule applies; in particular, the second premise of "missing" does not hold.

An example of where inconsistency arises is $\mathsf{abduce}(x \mapsto 2, x \mapsto 3)$. Here, the algorithm finds $2 = 3$ as the anti-frame. Of course, that is alright since for every A such that $x \mapsto 2 * A \models x \mapsto 3$, we must have that $x \mapsto 2 * A$ is inconsistent.

Example 3.6. The abduction algorithm in incomplete in the sense that it sometimes fails to find any solution at all when one exists. Consider the question:

$$x{\mapsto}3 \ * \ [??] \ \vdash \ y{\mapsto}3$$

There is a solution – $(y = x \wedge \mathsf{emp})$ – but our algorithm does not find it (and our proof system cannot prove it). The instance $\mathsf{abduce}(x \mapsto 3, y \mapsto 3)$ of our algorithm returns fail because no rule applies.

This is similar to Example 3.4, except that we do not have $*\mathsf{true}$ in the consequent. In that case, the algorithm does find a solution. As we will see later, it is abduction questions with $*\mathsf{true}$ in the consequent that we use in our applications to program analysis in this article. But, there are more complicated examples where the algorithm fails with $*\mathsf{true}$ in the consequent as well.[6]

3.2.3. A Framework of Abductive Inference for Inductive Predicates. For concreteness, in the description of the abductive inference system, we used a specific inductive predicate for list segments. We now describe a generalization that deals with different classes of inductive definitions, such as those for doubly liked lists, nested lists, trees and skip lists [Berdine et al. 2007; Chang et al. 2007; Reynolds 2002] which have been used for different abstract domains. For our generalization, we keep all the components of the abduction inference in the previous section, except for the four proof rules in Figure 1: $\mapsto$-match, ls-left, ls-right and missing.

Suppose that we have an abstract domain whose basic spatial predicates are ranged over by $B(E, \vec{E})$. Recall that the abstract domain used throughout the article corresponds to a specific instantiation:

$$B(E, E') \quad ::= \quad E{\mapsto}E' \;\mid\; \mathsf{ls}(E, E').$$

The missing rule is generalized in the following way:

$$\frac{\Delta * [M] \;\rhd\; \Delta' \qquad \Delta * \exists \vec{X}.\, B(E, \vec{E}) \;\not\vdash\; \mathsf{false}}{\Delta * [M * \exists \vec{X}.\, B(E, \vec{E})] \;\rhd\; \Delta' * \exists \vec{X}.\, B(E, \vec{E})} \text{ missing-gen.}$$

Note that this rule is almost the same as missing, except for the changes required to reflect the different sets of basic predicates.

To generalize the other rules, we need to make an assumption about the abstract domain: we assume that we are given a set of axioms involving basic special predicates, all of which have the form

$$\begin{array}{c}(\exists \vec{y}.\; \Pi(x, \vec{y}, \vec{z}) \wedge B(x, \vec{y}) * \Sigma(\vec{y}, \vec{z})) \;\vdash\; B'(x, \vec{z}), \quad \text{or} \\ \Pi(x, \vec{z}) \wedge B(x, \vec{z}) \;\vdash\; (\exists \vec{y}.\; B'(x, \vec{y}) * \Sigma(\vec{y}, \vec{z})).\end{array}$$

For example, the abstract domain of this article has the axioms below:

$$\begin{array}{l}(\exists y.\, y = z \wedge x{\mapsto}y * \mathsf{emp}) \;\vdash\; (x{\mapsto}z), \\ (\exists y.\, \mathsf{ls}(x, y) * \mathsf{ls}(y, z)) \;\vdash\; \mathsf{ls}(x, z), \\ x \neq z \wedge \mathsf{ls}(x, z) \;\vdash\; \exists y.(x{\mapsto}y) * \mathsf{ls}(y, z).\end{array} \tag{2}$$

Each of these axioms generates proof rules that replace $\mapsto$-match, ls-left and ls-right. For each axiom of the first form

$$(\exists \vec{y}.\; \Pi(x, \vec{y}, \vec{z}) \wedge B(x, \vec{y}) * \Sigma(\vec{y}, \vec{z})) \;\vdash\; B'(x, \vec{z}),$$

we define the following rule for abduction:

$$\frac{(\Pi(E, \vec{E}, \vec{E'}) \wedge \Delta) * [M] \;\rhd\; \exists \vec{Y}.\, \Delta' * \Sigma(\vec{E}, \vec{E'})}{\Delta * B(E, \vec{E}) * [\exists \vec{X}.\, \Pi(E, \vec{E}, \vec{E'}) \wedge M] \;\rhd\; \exists \vec{X}\vec{Y}.\, \Delta' * B'(E, \vec{E'})}.$$

For each axiom of the second form

$$\Pi(x, \vec{z}) \wedge B(x, \vec{z}) \;\vdash\; (\exists \vec{y}.\; B'(x, \vec{y}) * \Sigma(\vec{y}, \vec{z})),$$

[6]Nikos Gorogiannis, personal commmunication: Such examples will be described and analyzed in a forthcoming work of Gorogiannis, Kanovich and O'Hearn.

we include the following proof rule for abduction:

$$\frac{(\Pi(\vec{E},\vec{E}') \wedge \Delta * \Sigma(\vec{X},\vec{E}')) * [M] \;\; \rhd \;\; \exists \vec{Y}.\, \Delta'}{(\Delta * B(\vec{E},\vec{E}')) * [\Pi(\vec{E},\vec{E}') \wedge M] \;\; \rhd \;\; \exists \vec{X}\vec{Y}.\, \Delta' * B'(\vec{E},\vec{X})}.$$

The rules $\mapsto$-match, ls-left and ls-right presented earlier can be generated by following this recipe using the three axioms in (2).

THEOREM 3.7. *Suppose that the assumed entailment checking is sound, so that* $H_0 \vdash H_1$ *implies* $H_0 \models H_1$. *Then the proof system in Figure* 1 *is sound: if* $\Delta * [M] \rhd H$ *is derivable, then* $\Delta * M \models H$.

The theorem holds simply because all of the proof rules preserve validity, if validity of $\Delta * [M] \rhd H$ is interpreted as the truth of the entailment $\Delta * M \models H$.

3.3. On the Quality of Solutions

This subsection and the next contain some more detailed theoretical work on abduction for separation logic. They can be skipped, if desired, without loss of continuity.

As a general point, it is well known that solutions to abduction questions are not unique. Often, one looks for solutions that are minimal in some sense. In some cases there actually is a theoretically best answer, and it becomes a question of how difficult it is to compute the best answer. Failing that, having an ordering of "betterness" on solutions can prove helpful as a guide in the design of even incomplete algorithms. This section investigates these issues. We propose an ordering on solutions, show that a best solution always exists, and give an explicit description of it (though in infinitary terms that moves beyond symbolic heaps). In the next section, we will do some work on calculating the best-solution.

We should remark at once that the discussion surrounding Example 3.4 already establishes that the heuristic algorithm given earlier does not always compute the best solution according to the definition given in this section.

The work in this section will be conducted at a more abstract level than the previous section. We will work with predicates more general than those described by symbolic heaps, essentially taking a predicate to be synonymous with a set of stack-heap pairs. That is, predicates are elements of the powerset $\mathcal{P}(\mathsf{States})$, where $\mathsf{States} = \mathsf{Stack} \times \mathsf{Heap}$. We use F, G to range over such predicates, blurring syntax and semantics, and use the $*$ and $\wedge$ notation from earlier along with the Boolean algebra structure and residuated monoid structure of $\mathcal{P}(\mathsf{States})$. In particular, the negation and disjunction operators and the separating implication $-\!\!*$:

$$\begin{aligned} s,h &\models \neg F && \text{iff } s,h \not\models F \\ s,h &\models F \vee G && \text{iff } s,h \models F \text{ or } s,h \models G \\ s,h &\models F -\!\!* G && \text{iff } \forall h',\ h \uplus h'\downarrow \text{ and } s,h' \models F \text{ implies } s,h \uplus h' \models G. \end{aligned}$$

To follow this section, the reader should be comfortable with the semantic view of predicates as (essentially) elements of a certain lattice, in this case a Boolean lattice with an additional commutative residuated monoid structure (cf, a Boolean BI algebra [Pym et al. 2004]).

3.3.1. The Spatial Betterness Ordering. We are interested in solutions M to the question

$$F * M \models G. \tag{3}$$

There are many possible solutions, a trivial one of which is the assertion false.

We propose a criterion for judging the quality of solutions to (3). The order $M \precsim M'$, meaning that M is a better solution than M', is defined as follows:[7]

$$M \precsim M' \stackrel{def}{\iff} (M \leq M' \ \wedge \ M' \not\leq M) \ \vee \\ (M \leq M' \ \wedge \ M' \leq M \ \wedge \ M \models M'),$$

where $\leq$ is the following preorder

$$M \leq M' \stackrel{def}{\iff} M' \models M * \mathsf{true}.$$

This definition says that a solution M to an abduction question is better than another solution M' when M is spatially "smaller" ($M \leq M'$) or, if not spatially smaller it is logically stronger.

Example 3.8. Consider the symbolic heaps

$$M \triangleq \mathsf{ls}(y, 0)$$
$$M' \triangleq \mathsf{ls}(y, 0) * z{\mapsto}0$$
$$M'' \triangleq \mathsf{ls}(y, -).$$

We have that $M' \models M * \mathsf{true}$ therefore $M \leq M'$. Moreover $M \not\models M' * \mathsf{true}$, so $M' \not\leq M$: we have that M is strictly spatially smaller than M', as we expect intuitively since M' has an extra $*$-conjunct. Therefore, $M \precsim M'$. On the other hand, $M'' \leq M$ and $M \not\leq M''$, and so $M' \precsim M$.

For an example exercising the second disjunct in the definition of $\precsim$, consider $M = \mathsf{emp}$ and $M' = (\mathsf{emp} \vee x{\mapsto}0)$. We have that

$$\mathsf{emp} \models (\mathsf{emp} \vee x{\mapsto}0) * \mathsf{true} \qquad \text{and} \qquad (\mathsf{emp} \vee x{\mapsto}0) \models \mathsf{emp} * \mathsf{true}$$

moreover, since $\mathsf{emp} \models (\mathsf{emp} \vee x{\mapsto}0)$, we conclude that $\mathsf{emp} \precsim (\mathsf{emp} \vee x{\mapsto}0)$.

Ideally, we would like to find the best solution, that is, the minimal solution with respect to $\precsim$. As a semantic question, the best solution always exists theoretically, and can be defined using the following notion.

Definition 3.9 (*Function* min). The set of minimal states satisfying F is defined as follows

$$\mathsf{min}(F) = \{\, (s, h) \models F \mid \text{for all subheaps } h' \text{ of } h, \text{ if } (s, h') \models F, \text{ then } h' = h \,\}.$$

The set $\mathsf{min}(F)$ consists of the minimal elements of F only. In other words, a state L is filtered out if a strictly smaller state is also described by F. The function $\mathsf{min}(\cdot)$ is obviously idempotent, as the minimal elements of F remain minimal in $\mathsf{min}(F)$. $\mathsf{min}(\cdot)$ also interacts nicely with the $\leq$ relation.

[7]The definition in Calcagno et al. [2009a] had $M' \models M$ in the second disjunct by mistake.

LEMMA 3.10 (PROPERTIES OF $\leq$). *The following hold for any M,M':*

(1) $\leq$ *is a preorder.*
(2) $M \leq \mathsf{min}(M)$ *and* $\mathsf{min}(M) \leq M$.
(3) $(M \leq M'$ *and* $M' \leq M)$ *iff* $\mathsf{min}(M) = \mathsf{min}(M')$.

PROOF. Immediate from the definitions of $\leq$ and min. □

LEMMA 3.11 (PROPERTIES OF $\precsim$). *The following hold for any M,M':*

(1) $\precsim$ *is a partial order.*
(2) $M' \leq M$ *implies* $\mathsf{min}(M') \precsim M$*; in particular* $\mathsf{min}(M) \precsim M$.

PROOF. The fact that $\precsim$ is a preorder follows from the definition of $\precsim$ and the fact that $\leq$ (Lemma 3.10) and $\models$ are preorders. Antisymmetry follows from antisymmetry of $\models$.

We now prove that $M' \leq M$ implies $\mathsf{min}(M') \precsim M$. Assume $M' \leq M$. Then, by Lemma 3.10, we have $\mathsf{min}(M') \leq M$. If $M \not\leq \mathsf{min}(M')$, then $\mathsf{min}(M') \precsim M$ follows immediately by definition of $\precsim$. If instead $M \leq \mathsf{min}(M')$, then Lemma 3.10 gives $\mathsf{min}(M) = \mathsf{min}(\mathsf{min}(M'))$, so $\mathsf{min}(M) = \mathsf{min}(M')$ since min is idempotent. Since $\mathsf{min}(M) \models M$, we have in particular $\mathsf{min}(M') \models M$, therefore the second disjunct in the definition of $\precsim$ applies, and we can conclude $\mathsf{min}(M') \precsim M$. □

LEMMA 3.12 (MINIMAL SOLUTION WITH RESPECT TO $\leq$). *A minimal solution to the abduction question* (3) *with respect to* $\leq$ *is given by* $F \mathrel{-\!\!*} G$.

PROOF. Immediate from the definitions, and the fact that $\mathrel{-\!\!*}$ is the right adjunct of $*$: $F * M \models G$ iff $M \models F \mathrel{-\!\!*} G$. □

Since $\leq$ is a preorder but not a partial order, the minimal solution with respect to $\leq$ need not be unique. In fact, $\mathsf{min}(F \mathrel{-\!\!*} G)$ is another minimal solution with respect to $\leq$.

THEOREM 3.13 (UNIQUE MINIMAL SOLUTION WITH RESPECT TO $\precsim$). *The minimal solution to* (3) *with respect to* $\precsim$ *always exists, and is given by* $\mathsf{min}(F \mathrel{-\!\!*} G)$.

PROOF. First, we show that $\mathsf{min}(F \mathrel{-\!\!*} G)$ is a solution to (3). Since $F * (F \mathrel{-\!\!*} G) \models G$ and $\mathsf{min}(F \mathrel{-\!\!*} G) \models (F \mathrel{-\!\!*} G)$ both hold, it immediately follows that $F * \mathsf{min}(F \mathrel{-\!\!*} G) \models G$, as required.

Second, we show that $\mathsf{min}(F \mathrel{-\!\!*} G)$ is minimal with respect to $\precsim$. Consider an arbitrary set of heaps M such that $F * M \models G$. It follows that $M \models (F \mathrel{-\!\!*} G) * \mathsf{true}$, hence $(F \mathrel{-\!\!*} G) \leq M$ and Lemma 3.11 gives $\mathsf{min}(F \mathrel{-\!\!*} G) \precsim M$, as required. □

3.3.2. Characterizing the Minimal Solution. Note that the function min can be described using separation logic connectives, as

$$\mathsf{min}(F) \;=\; F \wedge \neg(F * \neg\mathsf{emp}).$$

Consequently, from Theorem 3.13 we obtain that the minimal solution to (3) with respect to $\precsim$ can be expressed in separation logic as

$$(F \mathrel{-\!\!*} G) \wedge \neg((F \mathrel{-\!\!*} G) * \neg\mathsf{emp}). \tag{4}$$

One might therefore ask why we do not take (4) as the answer to abduction, and leave it that. The answer is that (4), while perhaps theoretically interesting, begs the question posed, not giving us any information about the nature of the "missing memory".

The formula (4) also poses an, on the surface of it, difficult theorem proving question, which program analysis tools are not (yet) up to. One of the questions, for example,

that an analysis tool must be able to answer easily is whether a formula is inconsistent: throwing away inconsistent symbolic heaps is essential for pruning the search space. No existing automatic theorem provers for separation logic can deal with formulae in the form of (4).

A less formal statement of the limitations of (4) can be given in terms of program understanding. The special form of symbolic heaps given earlier is used by ABDUCTORto draw "pictures of memory." Such a picture is a "partial graph", or a graph with dangling links, in which $\mapsto$ or ls facts are the edges. These are used to show the user preconditions and postconditions of procedures that describe perhaps infinite classes of heaps, but that connect to the intuition of a heap as a graph; see Section 5 for examples. The formula (4) does not furnish enough information to draw these sorts of pictures, without further simplification or analysis of the formula.

Because of these points, we would like to have more direct solutions to abduction, when possible, without invoking advanced machinery such as $-\!\!*$ and $\neg$ nested around $*$. Section 3.2.1 gave one way to obtain such solutions, but the solutions obtained were not best according to $\precsim$, as Example 3.4 shows.

3.4. A Systematic Algorithm for a Restricted Fragment

Where earlier we described a fast but incomplete heuristic algorithm, now we investigate the possibility of an algorithm for abduction that computes the best solution. This is in fact a nontrivial problem, in that abduction is more challenging than pure entailment, and entailment in the presence of inductive definitions contains many unknown questions. After much work, decidability results were obtained for symbolic heaps with inductive definitions for just list segments [Berdine et al. 2004, 2005b], but no decidability results are known for entailment questions for wider classes of inductive definitions as considered in Section 3.2.3.

So, to make progress, in this section, we restrict our attention to symbolic heaps with $\mapsto$ but not any inductive predicates. This fragment itself could have practical interest: it could possibly form the foundation for less expressive analysis problems which do not require full-blown shape analysis (such as typestate-like analyses [Bierhoff and Aldrich 2005; DeLine and Fähndrich 2004]). Be that as it may, our aim here is more theoretical, to try to understand some of the issues surrounding the best solution.

We use a specialization of the generic grammar for symbolic heaps of Section 3.1, using only one basic spatial predicate

POINTS-TO INSTANTIATION

$$P ::= E = E \mid E \neq E$$
$$S ::= E \mapsto E$$

We will address the abduction question to find D such that

$$\Delta * D \models H. \tag{5}$$

where the quantifier-free symbolic heap Δ and the symbolic heap H are from the POINTS-TO INSTANTIATION. D here is a new syntactic category that allows disjunctions of symbolic heaps, which is what will allow us to describe the best solution.

$$D ::= H_1 \vee \cdots \vee H_n \qquad \textit{Disjunctions of symbolic heaps}$$

The notation $\exists X.\ D$ will be used as a shorthand for $\exists X.\ H_1 \vee \cdots \vee \exists X.\ H_n$ when D is of the form $H_1 \vee \cdots \vee H_n$.

Computing the best solution directly, even for the limited fragment studied here, proved to be a nontrivial task. We have broken down the problem into phases, the first of which uses a notion of compatible solution.

3.4.1. Compatible Solutions

Definition 3.14 (*Compatible/Incompatible Solutions*). We make use of the following separation logic idiom:

$$\mathsf{Elsewhere}(F) \stackrel{def}{=} \neg(F \mathrel{-\!\!*} \mathsf{false}), \text{ which says: "some separate heap satisfies } F\text{"}.$$

The reading of Elsewhere follows at once from the definition of $-\!\!*$:

$$\begin{aligned} s,h \models F \mathrel{-\!\!*} \mathsf{false} \text{ iff } & \forall h',\ h \uplus h'\downarrow \text{ and } s,h' \models F \text{ implies } s,h \uplus h' \models \mathsf{false} \\ = \ & \forall h',\ \neg(h \uplus h'\downarrow \text{ and } s,h' \models F) \\ = \ & \neg\exists h',\ h \uplus h'\downarrow \text{ and } s,h' \models F. \end{aligned}$$

We use Elsewhere to express the idea of "compatible" solutions to the abduction question (5). Let D be a solution to the abduction question (5). Then, $\neg\mathsf{Elsewhere}(\Delta)$ describes *completely incompatible solutions*, those for which there are no separate heaps satisfying Δ. Conversely, we can consider the compatible solutions.

— D is called a *compatible solution* if $D \models \mathsf{Elsewhere}(\Delta)$; that is, D implies that some separate heap satisfies Δ.
— The *compatible preorder* $\leq_c$ is defined as:

$$D \leq_c D' \stackrel{def}{\iff} (D \wedge \mathsf{Elsewhere}(\Delta)) \leq (D' \wedge \mathsf{Elsewhere}(\Delta)).$$

It is worth reminding that, from the definition of $\leq$, we then obtain the characterization

$$D \leq_c D' \iff (D' \wedge \mathsf{Elsewhere}(\Delta)) \models (D \wedge \mathsf{Elsewhere}(\Delta)) * \mathsf{true}.$$

Example 3.15. Consider the abduction question

$$y{\mapsto}3 * D \models y{\mapsto}3 * x{\mapsto}2.$$

Then $D_1 \stackrel{def}{=} y \neq x \wedge x{\mapsto}2$ is a compatible solution. We have that $y \neq x \wedge x{\mapsto}2 \models \neg(y{\mapsto}3 \mathrel{-\!\!*} \mathsf{false})$ because when $y \neq x \wedge x{\mapsto}2$ holds there is some separate heap satisfying $y{\mapsto}3$. On the other hand, $D_2 \stackrel{def}{=} (x = y \wedge y{\mapsto}4) \vee (y \neq x \wedge x{\mapsto}2)$ is not a compatible solution, because the disjunct $(x = y \wedge y{\mapsto}4)$ conflicts with $y{\mapsto}3$: it is not the case that any state satisfying D_2 guarantees that some separate heap satisfies $y{\mapsto}3$. Note as well that, although D_2 is not compatible, neither is it completely incompatible: we do not have $D_2 \models y{\mapsto}3 \mathrel{-\!\!*} \mathsf{false}$, which is to say that D_2 does not imply that no separate heap satisfies $y{\mapsto}3$. Finally, disjunctions are not the only source of failure of compatibility. $D_3 \stackrel{def}{=} x{\mapsto}2$ is a solution, but it is not compatible because there are models where $x = y$ holds, which stops us from concluding that (in all models) some separate heap satisfies $y{\mapsto}3$.

As far as the compatible preorder is concerned, we have that $D_1 \leq_c D_2$ and that D_1 and D_3 are equivalent modulo $\leq_c$ and both are minimal.

Notice that, if we go beyond the language of symbolic heaps, it is always possible to strengthen a solution D to a compatible solution $D \wedge \mathsf{Elsewhere}(\Delta)$. The compatible preorder $\leq_c$ is a modification of the $\leq$ order to ignore the incompatible portions of solutions.

$$\frac{}{(\Pi \wedge E{\mapsto}E' * \Sigma) * [\mathsf{false}] \ \rhd\ \Pi' \wedge \mathsf{emp}}\ \mathsf{false}$$

$$\frac{}{(\Pi \wedge \mathsf{emp}) * [\Pi' \wedge \mathsf{emp}] \ \rhd\ \Pi' \wedge \mathsf{emp}}\ \mathsf{emp}$$

$$\frac{}{(\Pi \wedge \Sigma) * [\Pi' \wedge \mathsf{emp}] \ \rhd\ \Pi' \wedge \mathsf{true}}\ \mathsf{true}$$

$$\frac{\left[(\Pi \wedge \Sigma^{-j}) * [D_j] \ \rhd\ \Pi' \wedge \Sigma'\right]_{j=1..n} \qquad (\Pi \wedge \Sigma) * [D] \ \rhd\ \Pi' \wedge \Sigma'}{(\Pi \wedge \Sigma) * [\bigvee_{j=1..n}(L_j{=}L' \wedge R_j{=}R' \wedge D_j) \vee (D * L'{\mapsto}R')] \ \rhd\ \Pi' \wedge L'{\mapsto}R' * \Sigma'}\ \mathsf{psto}$$
$$(\text{where } \Sigma \equiv *_{i=1..n} L_i{\mapsto}R_i \text{ and } \Sigma^{-j} \equiv *_{i=1..n, i\neq j} L_i{\mapsto}R_i)$$

$$\frac{\Delta * [D] \rhd \Delta'}{\Delta * [\exists \vec{X}.\ D] \ \rhd\ (\exists \vec{X}.\ \Delta')}\ \mathsf{exists}$$

Fig. 2. Proof rules for perfect abductive inference modulo $\leq_c$.

The algorithm for computing the best solution to (5) consists of three phases:

(i) derivation of a minimal solution D_1 with respect to the $\leq_c$ preorder using a proof system;
(ii) computation of the set of incompatible solutions $\neg\mathsf{Elsewhere}(\Delta)$ expressed as a disjunction D_2;
(iii) computation of $\mathsf{min}(D_1 \vee D_2)$ using a minimization algorithm.

Before describing these three steps in detail, we show that they indeed compute a minimal solution.

LEMMA 3.16 (COMPUTING A MINIMAL SOLUTION). *Let D be a minimal solution to* (5) *with respect to $\leq_c$. Then,* $\mathsf{min}(D \vee \neg\mathsf{Elsewhere}(\Delta))$ *is a minimal solution to* (5) *with respect to $\precsim$.*

PROOF. We will show that $(D \vee \neg\mathsf{Elsewhere}(\Delta))$ is a minimal solution with respect to $\leq$. Then, Lemma 3.11 implies that $\mathsf{min}(D \vee \neg\mathsf{Elsewhere}(\Delta))$ is the minimal solution with respect to $\precsim$.

To prove that $(D \vee \neg\mathsf{Elsewhere}(\Delta))$ is a minimal solution with respect to $\leq$, let D' be an arbitrary solution. We need to show that $(D \vee \neg\mathsf{Elsewhere}(\Delta)) \leq_c D'$. Since D' is a solution, it follows that $D \leq_c D'$ by minimality of D, that is, $(D' \wedge \mathsf{Elsewhere}(\Delta)) \models (D \wedge \mathsf{Elsewhere}(\Delta)) * \mathsf{true}$ holds. Therefore, $D' \models ((D \wedge \mathsf{Elsewhere}(\Delta)) * \mathsf{true}) \vee \neg\mathsf{Elsewhere}(\Delta)$ holds by Boolean logic (standard, rule for $\neg$), and so $D' \models (D * \mathsf{true}) \vee \neg\mathsf{Elsewhere}(\Delta)$ holds by monotonicity of $*$ wrt $\models$ and projection for $\wedge$. Hence, $D' \models (D \vee \neg\mathsf{Elsewhere}(\Delta)) * \mathsf{true}$ also holds by the fact that $\vee$ preserves $*$ and that $\neg\mathsf{Elsewhere}(\Delta) \models \neg\mathsf{Elsewhere}(\Delta)) * \mathsf{true}$. We have shown $(D \vee \neg\mathsf{Elsewhere}(\Delta)) \leq D'$, as required. □

3.4.2. Proof System for Perfect Solution with Respect to $\leq_c$. In this section, we introduce a proof system for deriving judgments of the form

$$\Delta * [D] \ \rhd\ H,$$

where Δ, D, H are just as in (5). The proof system is described in Figure 2. Its proof rules derive a solution D to the abduction question which is minimal with respect to the $\leq_c$ preorder. Recall that in the $\leq_c$ preorder the incompatible solutions are ignored. This simplifies the definition of the proof system. For example, consider the emp rule: if we had required minimality with respect to $\leq$, then the solution $(\Pi' \wedge \mathsf{emp})$ would not have been the minimal one, since the minimal solution with respect to $\leq$ includes all the incompatible solutions as well. The same observation applies to the other rules.

In the rule psto, we are using the letters L and R as a reading aide, to give the connotations "l-value" and "r-value". A crucial point is that this rule considers all aliasing possibilities, in contrast to the heuristic algorithm earlier, and as illustrated in the following example.

Example 3.17. An instance of the psto rule

$$\frac{\dfrac{}{\mathsf{emp} * [\mathsf{emp}] \rhd \mathsf{true}}\ \mathsf{true} \qquad \dfrac{}{x{\mapsto}3 * [\mathsf{emp}] \rhd \mathsf{true}}\ \mathsf{true}}{x{\mapsto}3 \ * \ [(x = y \wedge \mathsf{emp}) \vee (\mathsf{emp} * y{\mapsto}3)] \ \rhd \ y{\mapsto}3 \ * \ \mathsf{true}}\ \mathsf{psto}$$

computes the best solution to the abduction question from Example 3.4, where the heuristic algorithm got only one of the disjuncts. (In this rule note that emp on the top left of this derivation is Σ^{-j} in the inference rule, using the convention that emp is the 0-ary $*$-conjunction. Also, we have elided true$\wedge$ on several occasions, where true would bind to Π and Π' in a more literal depiction.)

A different instance finds a solution when the heuristic algorithm found none at all (Example 3.6):

$$\frac{\dfrac{}{\mathsf{emp} * [\mathsf{emp}] \rhd \mathsf{emp}}\ \mathsf{emp} \qquad \dfrac{}{x{\mapsto}3 * [\mathsf{false}] \rhd \mathsf{emp}}\ \mathsf{false}}{x{\mapsto}3 \ * \ [(x = y \wedge \mathsf{emp}) \vee (\mathsf{false} * y{\mapsto}3)] \ \rhd \ y{\mapsto}3}\ \mathsf{psto}$$

After obtaining this solution, using the identities $P \vee \mathsf{false} = P$ and $\mathsf{false} * Q = \mathsf{false}$, we end up with the expected solution $x = y \wedge \mathsf{emp}$.

The role of $\mapsto$ to the left in the false rule is worth remarking on here. Reading the rule downwards, this $E{\mapsto}F$ is unnecessary for soundness. But, reading the rules upwards, as a way of finding solutions, it is useful in saying that $E{\mapsto}F$ cannot imply emp, on the upper right of this last derivation.

LEMMA 3.18 (TERMINATION). *Given any Δ and H, there exists D such that $\Delta * [D] \rhd H$ is derivable, and the proof search is terminating.*

PROOF SKETCH. You read the rules upwards, as a way of *finding* solutions, similarly to the discussion in Section 3.2.2. First, the exists rule is applied to strip all existentials, remembering their names in a partially constructed solution. Then, the psto rule is applied repeatedly to remove $\mapsto$ facts from the right-hand side. At this point, we are left with a (partial) derivation in which there are no $\mapsto$ assertions on the right at the leaves. We can then match these assertions with the three axioms false, emp, and true. In each leaf, if the right-hand side has true in the spatial part, we match with the true rule (and generate $\Pi' \wedge \mathsf{emp}$ as this part of the answer to the overall abduction question). If the right-hand side has emp as the spatial part, then we use axiom false or emp, depending on whether or not $\mapsto$ is present in the left-hand side. □

LEMMA 3.19 (MINIMAL SOLUTION WITH RESPECT TO $\leq_c$). *If $\Delta * [D] \rhd H$ is derivable and $\Delta * F \models H$, then $D \leq_c F$.*

We remark that the definition of $\leq_c$ refers to a symbolic heap Δ, and the statement of this lemma also refers to a Δ: in the statement of the lemma it is intended that they are one and the same. Also, in the lemma the metavariable D ranges over disjunctions of symbolic heaps, while F ranges over arbitrary semantic predicates (sets of states), and we can use $-\!\!*$ and other constructs not available in symbolic heaps to describe the F's.

PROOF. The proof proceeds by induction on the derivation of $\Delta * [D] \rhd H$. Note that in applying the induction hypothesis, the relation $\leq_c$ varies: it can be different when the Δ part in the lemma varies; that is the case only in the psto rule.

For the axiom false, let $\Delta \stackrel{def}{=} \Pi \wedge E{\mapsto}F * \Sigma$, and suppose $\Delta * F \models \Pi' \wedge \mathsf{emp}$. To show $\mathsf{false} \leq_c F$, as required, boils down to showing that $F \wedge \mathsf{Elsewhere}(\Delta) \models (\mathsf{false} \wedge \neg I) * \mathsf{true}$, which is the same as saying that $F \wedge \mathsf{Elsewhere}(\Delta)$ is unsatisfiable. To see why this is the case, suppose, towards contradiction, that $F \wedge \mathsf{Elsewhere}(\Delta)$ is satisfiable. If we are given a model of $F \wedge \mathsf{Elsewhere}(\Delta)$, we can immediately obtain a model s, h of $\Delta * F$, and then h is the empty heap because we have presumed the entailment $\Delta * F \models \Pi' \wedge \mathsf{emp}$. But, h cannot be empty if is satisfies $\Delta * F$, as Δ contains a $\mapsto$: we have a contradiction, and so $F \wedge \mathsf{Elsewhere}(\Delta)$ must be unsatisfiable.

The arguments for the case of the other axioms true and emp are trivial.

For the case of the psto rule, let $\Delta \stackrel{def}{=} \Pi \wedge \Sigma$, $\Delta' \stackrel{def}{=} \Pi' \wedge \Sigma'$ and $\Delta^{-j} \stackrel{def}{=} \Pi \wedge \Sigma^{-j}$. We must prove

$$(req)\ \ F \wedge \mathsf{Elsewhere}(\Delta) \models \left(\left(\bigvee_{j=1..n}(L_j = L' \wedge R_j = R' \wedge D_j) \vee (D * L'{\mapsto}R')\right) \wedge \mathsf{Elsewhere}(\Delta)\right) * \mathsf{true}$$

And we get to use

(a) $\Delta * F \models \Pi' \wedge L'{\mapsto}R' * \Sigma'$
(b) $(\Delta \mathrel{-\!\!*} \Delta') \wedge \mathsf{Elsewhere}(\Delta) \models (D \wedge \mathsf{Elsewhere}(\Delta)) * \mathsf{true}$
(c) $(\Delta^{-j} \mathrel{-\!\!*} \Delta') \wedge \mathsf{Elsewhere}(\Delta^{-j}) \models (L_j{=}L' \wedge R_j{=}R' \wedge D_j \wedge \mathsf{Elsewhere}(\Delta^{-j})) * \mathsf{true}$
(d) Δ is *strictly exact* [Yang 2001]. For all s, there is at most one h_Δ with $s, h_\Delta \models \Delta$. (This holds of all contexts, incluidng Δ^{-j} in this lemma.)

Here, (a) says that F is a solution to the abduction question, (b) follows from the induction hypothesis for D and the fact that $(\Delta \mathrel{-\!\!*} \Delta')$ is a solution to $\Delta * ?? \rhd \Delta'$. (c) similarly uses the induction hypothesis for D_j, but note that it is stated using Δ^{-j}; this is the one place where $\leq_c$ changes (reflecting remark just prior to the proof) in an inductive case. (d) is true because Δ is quantifier free and contains only $\mapsto$ heap assertions.

To begin proving (req), assume $s, h \models F \wedge \mathsf{Elsewhere}(\Delta)$. We at once obtain that there is h_Δ where $h \uplus h_\Delta$ is defined and $s, h_\Delta \models \Delta$. By (a) we obtain that $s, h \uplus h_\Delta \models \Pi' \wedge L'{\mapsto}R' * \Sigma'$. The rest of the proof goes by case analysis, according to whether or not L' is allocated in h.

If L' is allocated in h, let h^- be h with the denotation of L' removed. If we can show $s, h^- \models (D \wedge \mathsf{Elsewhere}(\Delta)) * \mathsf{true}$, then we will obtain $s, h \models ((D * L'{\mapsto}R') \wedge \mathsf{Elsewhere}(\Delta)) * \mathsf{true}$, which will give us the rhs of (req). Since $s, h \models \mathsf{Elsewhere}(\Delta)$, we obtain immediately that $s, h^- \models \mathsf{Elsewhere}(\Delta)$. Next, we claim that $s, h^- \models \Delta \mathrel{-\!\!*} \Delta'$. To see why, consider an arbitrary separate heap satisfying Δ with s. It can only be h_Δ because of (d), and we already know that $s, h \uplus h_\Delta \models \Pi' \wedge L'{\mapsto}R' * \Sigma'$: it follows that $s, h^- \uplus h_\Delta \models \Pi' \wedge \Sigma'$. By the semantics of $\mathrel{-\!\!*}$, this establishes $s, h^- \models \Delta \mathrel{-\!\!*} \Delta'$. Now, using (b), we get $s, h^- \models (D \wedge \mathsf{Elsewhere}(\Delta)) * \mathsf{true}$, which is what we wanted to show.

In the other case, if L' is not allocated in h, then it must correspond to one $\mapsto$ fact within Δ, and we consider the corresponding l-value amongst the $L_1, .., L_n$, call it L_j (now j is fixed). It should be clear that that j is chosen here in such a way that, if we define $h_{\Delta^{-j}}$ to be h_Δ with the denotation of L' removed from its domain, then $s, h_{\Delta^{-j}} \models \Delta^{-j}$.

If we can show $s, h \models (\Delta^{-j} \mathrel{-\!\!*} \Delta') \wedge \mathsf{Elsewhere}(\Delta^{-j})$, satisfaction of the left-hand side of (c), then we will get satisfaction of the right-hand side of (c) and, hence, the right-hand side of (req). For the $\mathsf{Elsewhere}(\Delta^{-j})$ conjunct, because we have assumed $s, h \models \mathsf{Elsewhere}(\Delta)$, we obviously have $s, h \models \mathsf{Elsewhere}(\Delta^{-j})$. For the $\Delta^{-j} \mathrel{-\!\!*} \Delta'$ conjunct,

we know that there is one and only one separate h' where $s, h' \models \Delta^{-j}$, because Δ^{-j} is strictly exact and because $s, h \models \mathsf{Elsewhere}(\Delta^{-j})$; and, as we remarked previously, this h' is $h_{\Delta^{-j}}$. So if $s, h \uplus h_{\Delta^{-j}} \models \Delta'$, then we will have shown $s, h \models (\Delta^{-j} \mathrel{-\!\!*} \Delta')$. Earlier in the proof (before case analysis on L' being allocated) we obtained $s, h \uplus h_\Delta \models \Pi' \wedge L' \mapsto R' * \Sigma'$ from (a), and this obviously implies $s, h \uplus h_{\Delta^{-j}} \models \Pi' \wedge \Sigma' (\equiv \Delta')$ by the definition of $h_{\Delta^{-j}}$ as h_Δ with the denotation of L' removed, and this establishes satisfaction of the $\mathrel{-\!\!*}$ conjunct. This completes the proof of the psto case.

The case exists can be proven using the following general principle, which holds for arbitrary separation logic formulae F, G:

$$F \mathrel{-\!\!*} \exists x.\, G \models \exists x.\, (F \mathrel{-\!\!*} G) \quad [\text{provided } F \text{ is } \textit{strictly exact}]$$

with the observation that any quantifier-free symbolic heap Δ is strictly exact (as defined in (d) in the psto case).

Consider a derivation of $\Delta * [\exists \vec{X}.\ D_{\vec{X}}] \rhd (\exists \vec{X}.\ \Delta'_{\vec{X}})$ from $\Delta * [D_{\vec{X}}] \rhd \Delta'_{\vec{X}}$ using the exists rule, where for clarity we use subscripts $\vec{X}$ for formulae where $\vec{X}$ can occur free. Consider an arbitrary F such that $\Delta * F \models \exists \vec{X}.\ \Delta'_{\vec{X}}$ holds. We need to prove that $\exists \vec{X}.\ D_{\vec{X}} \leq_c F$ holds.

Define $F_{\vec{X}} \stackrel{def}{=} F \wedge (\Delta \mathrel{-\!\!*} \Delta'_{\vec{X}})$. By semantic considerations, $\Delta * F_{\vec{X}} \models \Delta'_{\vec{X}}$ holds, therefore $D_{\vec{X}} \leq_c F_{\vec{X}}$ holds by induction hypothesis, hence $\exists \vec{X}.\ D_{\vec{X}} \leq_c \exists \vec{X}.\ F_{\vec{X}}$ also holds. By this principle we have $\exists \vec{X}.\ F_{\vec{X}} = F \wedge (\Delta \mathrel{-\!\!*} \exists \vec{X}.\ \Delta'_{\vec{X}})$, and since $F \models \Delta \mathrel{-\!\!*} \exists \vec{X}.\ \Delta'_{\vec{X}}$ holds by hypothesis on F, we have $\exists \vec{X}.\ F_{\vec{X}} = F$. So we can conclude $\exists \vec{X}.\ D_{\vec{X}} \leq_c F$, as required. □

A delicacy. We note a delicate point in the proof system of Figure 2. The exists rule is complete, as well as sound, for the fragment we consider, but it is incomplete when one adds predicates for linked lists. To see why, consider that

$$x \neq 0 \wedge \mathsf{ls}(x, 0) * ??? \vdash \exists x'. x \mapsto x' * \mathsf{true}$$

has solution emp, but

$$x \neq 0 \wedge \mathsf{ls}(x, 0) * ??? \vdash x \mapsto x' * \mathsf{true}$$

has no solution (except false). Generally, the rule depends on Δ being "strictly exact", which is to say that, for all s, there is at most one h making $s, h \models \Delta$ true. Lists break exactness, where $\mapsto$ does not. This is only one of the obstacles that would need to be overcome to extend the approach of this section to inductive predicates.

3.4.3. Incompatible Solutions. The second phase of our algorithm is the computation of the incompatible solutions to (5) using a disjunction of symbolic heaps. (Strictly speaking, these do not take the right-hand side of the abduction into account, so are the candidate solutions for *any* abduction question with *Delta* on the left.) Recall that the incompatible solutions are described by $\neg\mathsf{Elsewhere}(\Delta)$ (or, equivalently, $\Delta \mathrel{-\!\!*} \mathsf{false}$). Let

$$\Delta \equiv \left(\wedge_{i=1..n} A_i\right) \wedge \left(*_{j=1..m} E_j \mapsto E'_j\right),$$

where each A_i is either an equality or a disequality. Then, the following disjunction, which we call $\mathsf{Incompat}(\Delta)$, describes the incompatible solutions:

$$\left(\vee_{i=1..n} \neg A_i\right) \vee \left(\vee_{j=1..m} E_j{=}0\right) \vee \left(\vee_{i,j=1..m, i\neq j} E_i{=}E_j\right) \vee \left(\vee_{j=1..m} \exists X. E_j \mapsto X * \mathsf{true}\right)$$

This definition captures all the states that cannot be combined with Δ, based on four reasons: the pure parts are inconsistent; the left-hand side of $\mapsto$ cannot be 0; all the

left-hand sides of $\mapsto$ in Δ must be distinct; a $\mapsto$ in Δ cannot be consistently $*$-conjoined with another $\mapsto$ with the same left-hand side. It is a straightforward exercise to verify that this definition correctly captures incompatibility.

LEMMA 3.20. $\mathsf{Incompat}(\Delta)$ *and* $\neg\mathsf{Elsewhere}(\Delta)$ *are semantically equivalent.*

Example 3.21. Consider the symbolic heap

$$H \triangleq x = y \wedge x{\mapsto}0 * w{\mapsto}0$$

According to our definition, all incompatible solutions are given by the formula

$$x \neq y \vee x = 0 \vee w = 0 \vee x = w \vee \exists X.x{\mapsto}X * \mathsf{true} \vee \exists Y.w{\mapsto}Y * \mathsf{true}$$

The disjuncts each capture a straightforward intuition: if $x \neq y$ in s state, then H cannot hold elsewhere because H requires $x = y$; if x or y is zero, then this precludes H holding elsewhere, because H requires x and y to be allocated (which implies non-0); if either x or y is allocated in a given state ("here"), then H cannot hold in a separate heap ("elsewhere") because that precludes x and y being allocated in the separate heap.

3.4.4. Computing min. The third phase of our algorithm consists in applying min to the result of the first two phases. In this section, we show how the result of applying min can be described as a disjunction of symbolic heaps.

The min is constructed using a formula $F - G$ for subtraction.

Definition 3.22 (*Subtraction*). Let F and G be arbitrary predicates (sets of states). The operator $-$ is defined as $F - G \stackrel{def}{=} F \wedge \neg((\neg\mathsf{emp}) * G)$. In words, $F - G$ consists of those states satisfying F for which no strictly smaller substate satisfies G.

When talking about semantic predicates, we say that "X is not free in F" to mean that $s, h \in F$ iff $s', h \in F$ for all s' like s except taking a possibly different value at X. The following lemma records properties of subtraction.

LEMMA 3.23 (PROPERTIES OF SUBTRACTION) .

(1) $\mathsf{min}(F) = F - F$.
(2) $(F \vee G) - H = (F - H) \vee (G - H)$.
(3) $(\exists X.F) - G = \exists X.(F - G)$, *provided* X *not free in* G.
(4) $F - (G \vee H) = (F - G) \wedge (F - H)$.
(5) $F - (\exists X.G) = \forall X.(F - G)$, *provided* X *not free in* F.
(6) $\mathsf{min}(F_1 \vee \cdots \vee F_n) = G_1 \vee \cdots \vee G_n$
where $G_i = ((F_i - F_1) \cdots - F_n)$

The proof of each of these properties is straightforward. The crucial point, (6), is a direct statement of the following idea: a state is in $\mathsf{min}(F_1 \vee \cdots \vee F_n)$ just if it is in one of the F_i's, and not in a smaller state for any other disjunct (else it would not be minimal). Our algorithm for computing min consists of applying directly point (6), which only depends on an algorithm for computing binary subtraction $H - H'$.

Quantifier-Free Subtraction. Suppose we are given quantifier-free symbolic heaps Δ and Δ', and we want to compute $\Delta - \Delta'$. A saturation Π is a collection of equalities and disequalities such that for any two expressions E_1 and E_2 (amongst those in Δ and Δ'), we have $E_1{=}E_2 \in \Pi$ or $E_1{\neq}E_2 \in \Pi$. As a result, both $(\Pi \wedge \Delta)$ and $(\Pi \wedge \Delta')$ will denote essentially unique heaps (modulo renaming), and they are amenable to a syntactic form of subtraction.

We define a syntactic check $\Delta \subseteq_\Pi \Delta'$ to capture the intuition that, under a saturated Π, the heap described by Δ is a subheap of the one described by Δ'. Formally, let

$\Delta \equiv (\Pi_1 \wedge \Sigma_1)$ and $\Delta' \equiv (\Pi_2 \wedge \Sigma_2)$, and define $\Delta \subseteq_\Pi \Delta'$ if for each $E_1 \mapsto E_1'$ in Σ_1 there exists $E_2 \mapsto E_2'$ in Σ_2 such that $E_1 = E_2 \in \Pi$ and $E_1' = E_2' \in \Pi$. The strict inclusion $\Delta \subset_\Pi \Delta'$ is defined as $\Delta \subseteq_\Pi \Delta'$ and $\Delta' \not\subseteq_\Pi \Delta$.

To compute $\Delta - \Delta'$, consider all the saturations $\Pi_1, \ldots, \Pi_n$ such that $\Delta' \subset_{\Pi_i} \Delta$. Those are all the saturations under which Δ is removed by the subtraction of Δ', so we obtain the result by negating them:

$$\Delta - \Delta' = ((\neg\Pi_1) \vee \cdots \vee (\neg\Pi_n)) \wedge \Delta.$$

By pushing negations inside and using de-Morgan duality, we actually get a disjunction of Π's. So the solution has the following shape:

$$(\Pi_1' \vee \cdots \vee \Pi_k') \wedge \Delta.$$

and we can then use distribution of $\vee$ over $\wedge$ to obtain

$$(\Pi_1' \wedge \Delta) \vee \cdots \vee (\Pi_k' \wedge \Delta)$$

and this is a disjunction of symbolic heaps with the same meaning as $\Delta - \Delta'$.

Quantifier-Removal for Pure Formulae. To extend the previous subtraction case to handle quantifiers, we first establish a quantifier-removal result for the special case of a pure formula $\exists X.\Pi$. Note that this makes essential use of the restricted nature of Π as only equalities and disequalities of variables or constants. To convert such a formula to an equivalent quantifier-free formula, first repeatedly replace $\exists X.(X = E \wedge \Pi)$ by $\exists X.\Pi[E/X]$ until no equalities remain involving X. Second, repeatedly replace $\exists X.(X \neq E \wedge \Pi)$ by $0 \neq 0$ if $E \equiv X$, or by $\exists X.\Pi$ if $E \not\equiv X$. This second step is an equivalence because, at this stage, there are no equalities involving X in Π: X can be chosen to be not equal to E iff it can be chosen at all to satisfy Π. We are left with a formula $\exists X.\Pi'$ equivalent to the $\exists X.\Pi$ we started with, where X is not free in Π', so it is equivalent to the quantifier-free formula Π'.

Subtraction with Quantifiers. We handle symbolic heaps with quantifiers as follows, using cases of Lemma 3.23 and observing suitable α-renaming.

$$\begin{aligned}(\exists\vec{X}.\Delta) - (\exists\vec{Y}.\Delta') &= [by\,(3)]\ \exists\vec{X}.(\Delta - (\exists\vec{Y}.\Delta')) \\ &= [by\,(5)]\ \exists\vec{X}.\forall\vec{Y}.(\Delta - \Delta') \\ &= \exists\vec{X}.\forall\vec{Y}.((\Pi_1' \vee \cdots \vee \Pi_k') \wedge \Delta)\end{aligned}$$

where the final step uses the result of quantifier-free subtraction $\Delta - \Delta'$ described above. To represent the result as a symbolic heap we just need to remove quantifiers from the pure part: $\forall\vec{Y}.(\Pi_1' \vee \cdots \vee \Pi_k') = \Pi_1'' \vee \cdots \vee \Pi_j''$ for some j.

This final step can be done as follows. First, replace $\forall\vec{Y}.(\Pi_1' \vee \cdots \vee \Pi_k')$ by $\neg\exists\vec{Y}.\neg(\Pi_1' \vee \cdots \vee \Pi_k')$. Second, use de-Morgan and distribution laws to convert $\neg(\Pi_1' \vee \cdots \vee \Pi_k')$ into disjunctive normal form in the usual way, giving $A_1 \vee \cdots \vee A_m$ where each A_i is a conjunction of equalities and disequalities. Fourth, drive $\exists$ inwards to convert $\neg\exists\vec{Y}.(A_1 \vee \cdots \vee A_m)$ to $\neg((\exists\vec{Y}.A_1) \vee \cdots \vee (\exists\vec{Y}A_m))$. Fifth, apply the $\exists$-removal for pure formulae as previously described, giving us an equivalent quantifier-free formula $\neg(B_1 \vee \cdots \vee B_m)$. Finally, convert to disjunctive normal form giving us our $\Pi_1'' \vee \cdots \vee \Pi_j''$.

Then, as in the previous case, we can use distribution to obtain a disjunction of symbolic heaps

$$(\Pi_1'' \wedge \Delta) \vee \cdots \vee (\Pi_j'' \wedge \Delta)$$

and we have shown how to calculate subtraction of symbolic heaps with quantifiers.

Example 3.24. For an example involving quantifier elimination, consider

$$(\exists Y, Z.Y \mapsto Z * Z \mapsto 3) - (\exists X.X \mapsto X).$$

First, as stated previously, we compute

$$\exists Y, Z.\forall X.(Y \mapsto Z * Z \mapsto 3 - X \mapsto X).$$

The inner subtraction gives $\neg(Z = X \wedge X = 3)$ in the pure part, that is, $Z \neq X \vee X \neq 3$. Then, quantifier elimination: $\forall X.(Z \neq X \vee X \neq 3) = (Z \neq 3)$. So the final solution is

$$\exists Y, Z.Z \neq 3 \wedge Y \mapsto Z * Z \mapsto 3$$

and one can see that this symbolic heap cannot be validated in a heap where an allocated pointer points to itself.

Computing Multi-Nary Subtraction. We have just described a binary algorithm that gives $H - H' = H_1 \vee \cdots \vee H_n$ for some n, for individual symbolic heaps H and H'. To obtain an algorithm that takes a disjunction of symbolic heaps as input and returns their min, we directly apply Lemma 3.23(6) repeatedly, and use 3.23(2) to redistribute the resulting disjunction at each nesting step in equation 3.23(6).

Putting Everything Together. Putting together the results in this section, we can first compute a minimal solution D with respect to $\leq_c$, then calculate the incompatible solutions I, and finally minimize $D \vee I$ using the algorithm just given. By Lemma 3.16, this gives us the minimal solution with respect to $\precsim$. We conclude by stating our main result on the systematic algorithm for the POINTS-TO INSTANTIATION of symbolic heaps.

THEOREM 3.25 (MINIMAL SOLUTION ALGORITHM). *The minimal solution to the abduction question* (5) *is expressible as a disjunction D of symbolic heaps, which can be effectively computed with an algorithm.*

3.5. Bi-Abduction and Framing: Execution-Oriented Program-Proof Rules

Now we take a step from the abduction question itself towards program analysis. Bi-abduction is a combination of the frame inference problem [Berdine et al. 2005b] and abduction as defined earlier in this article. We will see that this notion gives us a way to reason compositionally about the heap, using a form of symbolic execution, and helps in porting the principle of local reasoning to automatic program analysis.

Definition 3.26 (*Bi-Abduction*). Given symbolic heaps Δ and H find *symbolic heaps* ?*anti-frame and* ?*frame* such that

$$\Delta * ?\textit{anti-frame} \vdash H * ?\textit{frame}$$

It would be possible to consider a mixed proof system for this problem, but it turns out that there is a way to answer the question by appealing to separate frame inference and abduction procedures.

Several frame inference procedures have been described in previous papers [Berdine et al. 2005b; Distefano and Parkinson 2008; Nguyen and Chin 2008]. Here we assume a given procedure Frame, which returns either a symbolic heap or an exception fail. Frame must satisfy

$$\mathsf{Frame}(H_0, H_1) = L(\neq \mathsf{fail}) \implies H_0 \vdash H_1 * L$$

indicating that if frame inference succeeds in finding a "leftover" heap L, then the indicated entailment holds.

Algorithm 2 Finding a Missing Heap Portion

$\mathsf{Abduce}(\Delta, H) \stackrel{def}{=}$

(1) Find a symbolic heap M such that

$$\Delta * [M] \ \rhd \ H$$

using the abduction algorithm from Section 3.2. If no such heap can be found, return fail.
(2) If $\Delta * M$ is (provably) inconsistent, return fail. Otherwise, return M.

Algorithm 3 Synthesizing Missing and Leftover Heaps, Jointly

$\mathsf{BiAbd}(\Delta, H) \stackrel{def}{=}$

$M := \mathsf{Abduce}(\Delta, H * \mathsf{true});$

$L := \mathsf{Frame}(\Delta * M, H);$

return(M, L)

A frame inferencing procedure gives us a way to automatically apply the frame rule of separation logic [Ishtiaq and O'Hearn 2001; O'Hearn et al. 2001]. Normally, the rule is stated

$$\frac{\{A\}C\{B\}}{\{A * F\}C\{B * F\}} \text{Frame Rule (Usual Version).}$$

By combining the frame rule and the rule of consequence, we obtain a rule which tells us how to adapt a specification by tacking on the additional resource L found by a frame inferencing algorithm.

$$\frac{\{A\}C\{B\} \quad \mathsf{Frame}(P, A) = L}{\{P\}C\{B * L\}} \text{Frame Rule (Forwards Analysis Version).}$$

In the analysis version of the frame rule, we think of $\{A\}C\{B\}$ as the given specification of C, which might be obtained from a procedure summary, and P as the current state at the program point where C resides within a larger program. The rule tells us how to symbolically execute the program on the formulae, by propagating $B * L$ to the post-state.

In Algorithm 2, we define a further procedure Abduce, satisfying the specification

$$\mathsf{Abduce}(\Delta, H) = M (\neq \mathsf{fail}) \implies \Delta * M \vdash H,$$

meaning that it soundly finds missing heap portions. The second step of Algorithm 2 relies on an ordinary theorem prover for symbolic heaps.

We can combine these two procedures to obtain the algorithm BiAbd described in Algorithm 3. The use of $* \mathsf{true}$ in the abduction question of the algorithm lets us ignore the leftover frame when computing the missing heap. By convention, the algorithm raises exception fail if either of its internal procedure calls does.

THEOREM 3.27. $\mathsf{BiAbd}(\Delta, H) = (M, F) \implies \Delta * M \vdash H * F.$

We can begin to make the link between bi-abduction and program analysis with a new inference rule.

$$\frac{\{A\}C\{B\} \quad \mathsf{BiAbd}(P, A) = (M, L)}{\{P * M\}C\{B * L\}} \text{ Frame Rule (Bi-Abductive Analysis Version).}$$

Logically, the bi-abductive frame rule is just a consequence of the usual frame rule and the Hoare rule of consequence. The relevance to program analysis comes from a bottom-up reading in terms of finding missing and leftover heaps: we are given P, A and B, and we compute M and L which are sufficient to obtain a pre/post spec of C, which includes P as part of the pre.

3.5.1. Comparing Bi-Abductive Solutions. As before, this theoretical material on the quality of solutions can be skipped without loss of continuity.

The soundness property in Theorem 3.27 can be satisfied trivially. We define an order $\sqsubseteq$ on potential solutions

$$M \approx M' \stackrel{def}{\iff} M \precsim M' \ \wedge \ M \succsim M'$$

$$(M, L) \sqsubseteq (M', L') \stackrel{def}{\iff} (M \precsim M') \ \vee \ (M' \approx M \ \wedge \ L \vdash L').$$

The definition of $\sqsubseteq$ is a lexicographic ordering of the order $M \precsim M'$ defined for abduction (see Section 3.3.1), and ordinary implication for leftover heaps. The bias on the anti-frame part is useful in some application of BiAbd algorithm such as in program analysis for inferring preconditions of procedures as we will do in Section 4.2. The missing anti-frame part is used to update the precondition being discovered by the analysis. The second disjunct means that if two solutions have equally good missing anti-frames, the better one should have a logically stronger leftover frame.

Our BiAbd algorithm first attempts to find a good missing anti-frame M, and then tries to find a good frame L. This order of searching for a solution reflects our emphasis on the quality of the missing anti-frame part, as in the definition of $\sqsubseteq$.

Example 3.28. We illustrate the intuition behind $\sqsubseteq$ using

$$x{\mapsto}0 \ * \ M \ \vdash \ \mathsf{ls}(x, 0) * \mathsf{ls}(y, 0) * L$$

Consider the following three solutions to the question:

$$\begin{array}{ll} M \stackrel{def}{=} \mathsf{ls}(y, 0) & L \stackrel{def}{=} \mathsf{emp} \\ M' \stackrel{def}{=} \mathsf{ls}(y, 0) * z{\mapsto}0 & L' \stackrel{def}{=} z{\mapsto}0 \\ M' \stackrel{def}{=} \mathsf{ls}(y, 0) & L' \stackrel{def}{=} \mathsf{true} \end{array}$$

According to the order we have just defined, the best solution among the above three is (M, L), and it is what the algorithm BiAbd returns. It is better than (M', L'), because its missing anti-frame M is strictly better than M' (i.e., $M \precsim M'$ and $M \not\approx M'$) since it describes smaller heaps than M'. The solution (M, L) is also better than (M', L') but for a different reason. In this case, the missing anti-frames M, M' have the same quality according to our definition (i.e., $M \approx M'$). However, this time the deciding factor is the comparison of the leftover frames of the solutions; L is stronger than L', so $(M, L) \sqsubseteq (M', L')$.

It is possible to investigate the question of best answer to the bi-abduction question, like we did for abduction. One can show, in particular, that a strongest frame exists when the right-hand side of the frame inference question is precise. But, we leave a full study of this as a problem for future work.

3.6. Discussion

In this section, we have given the first proof procedures for abduction in separation logic, one which is heuristic in nature and the other which is more systematic. A tremendous amount more could be attempted, theoretically, in this direction.

Our work on the systematic procedure is along the lines of what is called "computing all explanations" in the AI literature [Eiter and Makino 2002], while our heuristic algorithm is more reminiscent of the problem of finding one solution when there might be many possibilities [Paul 1993]. One can also ask whether a solution must contain a particular literal, and given a candidate solution one can ask whether it is minimal. For all of these questions one can ask about complexity and decidability as well. The same goes for frame inference. Note that these questions are unlikely to be straightforward to answer, as abduction is at least as difficult as entailment, and relatively little is known about decidability and complexity of entailment for symbolic heaps beyond very specific fragments based on simple lists. There has been significant work on the complexity of abduction in propositional logic (e.g., [Creignou and Zanuttini 2006; Eiter and Gottlob 1995]), but that work does not immediately carry over to our setting. The substructural nature of separation logic as well as the use of existential quantifiers and (especially) inductive definitions in symbolic heaps raise special problems.

There is probably much worthwhile work to be done to increase understanding in the direction of completeness and complexity of algorithms for abduction and bi-abduction, and that is a valuable direction for future work. However, rather than pause to pursue these questions now, we move on to the question of the *use* of abduction and bi-abduction: if we assume an algorithm for bi-abduction as given, to what use might it be put in automatic program analysis?

4. ALGORITHMS FOR AUTOMATIC PROGRAM ANALYSIS

In this section we show how the previous ideas can be put to work in algorithms for automatic program analysis. We begin by describing an algorithm PREGEN for generating preconditions, and then we use it in an algorithm for generating pre/post specs.

4.1. Procedures, Summaries, and Control-Flow Graphs

We represent a program by a control-flow graph in which the edges are procedure calls $x{:=}f(\vec{e})$. Each procedure name f is associated with a *summary*, a set of Hoare triples

$$\{P\}\, f(\vec{x})\{Q_1 \vee \cdots \vee Q_k\},$$

which determines the meanings of the edges.

This format for the edges and their specifications, while seemingly spare, gives us considerable expressive power. Typical built-in atomic commands or libraries can be given initial specification which get an analysis started at the leaves of a call tree in a system: some examples, based on the "small axioms" from O'Hearn et al. [2001], are in Figure 3. In the figure, $[x] := y$ and $\mathbf{free}(x)$ are viewed as procedures that return no values (or that nondeterministically set the return value ret to anything). The statement `return`$[y]$ can be used with our procedure call syntax in the form $y :=$ `return`$[x]$ to obtain the same effect (and same small axiom) as what is written $y := [x]$ in O'Hearn et al. [2001].

In C-language terms, $[x]$ in Figure 3 can be thought of as corresponding to a dereferencing `x->tl` where x is a pointer to a struct of type

```
struct node { struct node* tl; };
```

and we will use examples based on this struct throughout this section. We stress, though, that the commands and their axioms in Figure 3 are given just as an illustration: our technical development does not depend at all on this struct or this unary $x{\mapsto}y$ predicate. For records with multiple fields we would need a more liberal $\mapsto$ where a pointer points to a tuple, as in Berdine et al. [2005b, 2007] (see Section 3.3.1). A more liberal $\mapsto$ is used in the ABDUCTOR tool.

$\{\mathsf{emp}\}\mathbf{malloc}()\{(\mathsf{ret}\mapsto -) \vee (\mathsf{ret} = \mathsf{nil} \wedge \mathsf{emp})\}$ $\quad\quad$ $\{x\mapsto -\}\mathbf{free}(x)\{\mathsf{emp}\}$

$\{x\mapsto - \wedge y = Y\}[x] := y\{x\mapsto Y \wedge y = Y\}$ $\quad\quad$ $\{x\mapsto X\}\mathtt{return}[x]\{x\mapsto X \wedge \mathsf{ret} = X\}$

Fig. 3. Example pre-defined summaries.

Using **assume** statements which check for boolean conditions on edges allows us to model the conditional branching of source-level `if` and `while` statements using the procedure-summary format of control-flow graphs (see Example 4.4). Also, source statements for objects with multiple fields would map to more complex forms than the simple heap lookup $[x]$. We ignore these issues in the technical part of our work, expecting that it is relatively clear how `while` programs with a collection of embedded procedure calls and heap-mutating operations can be modelled within this format of control-flow graphs with procedure summaries.

Notice the disjunction in the postcondition for **malloc** in Figure 3: it says that the return value is either an allocated cell or an error flag indicating that the allocation was unsuccessful (incidentally illustrating disjunctive posts, which are essential for nontrivial precision). This allows us to deal with the issue mentioned at the end of Section 2.2 concerning error values returned from **malloc**.

In representing programs using graphs plus summaries, we assume that the procedures f do not alter global variables (the x's appearing to the left of := on control-flow edges $x{:=}f(\vec{e})$), though they might alter global heap cells ($[x]$); this assumption will be reflected in the transfer function for processing procedure calls in our analysis. In practice, when translating a procedure to control-flow-graph form the variables x in $x{:=}f(\vec{e})$ can correspond to local variables of the procedure, as well as additional temporary variables inserted by a compilation step.[8]

With these preliminary considerations, we give our formal definition.

Definition 4.1 (**Program**). A *program* is a tuple $(L, \mathtt{start}, \mathtt{finish}, T, \mathsf{summary})$ of

— finite set of program locations L
— designated program locations $\mathtt{start}, \mathtt{finish} \in L$
— a set T of transitions $(\ell, x := f(\vec{v}), \ell')$, where ℓ, ℓ' are program locations
— a set $\mathsf{summary}(f)$ of procedure specs

$$\{P\}\, f(\vec{x})\{\mathcal{Q}\}$$

for each procedure f appearing amongst the transitions in T. Here, P is a symbolic heap and $\mathcal{Q}$ is a set of symbolic heaps. The intended meaning of a *spec* is that

$$\{P\}\, f(\vec{x})\{Q_1 \vee \cdots \vee Q_k\}$$

is a true Hoare triple, where $\mathcal{Q} = \{Q_1, \ldots, Q_k\}$.

4.2. Precondition Generation

4.2.1. The PREGEN *Algorithm.* The PREGEN algorithm, Algorithm 4, takes a set of abstract states as an argument (the "seeded" precondition) and returns a set of abstract states as a result (the "generated" precondition). It is a modification of a standard worklist or abstract reachability algorithm [Jhala and Majumdar 2009].

The algorithm works as if it is trying to find some postconditions to give a proof of the program starting from the seeded preconditions. The proof attempt is done according to the "tight interpretation of triples" in separation logic [O'Hearn et al. 2001],

[8]In case the address $\&x$ of a variable is taken, we would explicitly allocate a cell corresponding to x and replace any assignments to it by assignments to the allocated cell; and for this we usually use an allocation operation without a disjunctive post (that caters for error).

where a memory fault or failure to satisfy the given precondition of a procedure call invalidates any proof from the seeded preconditions. Where, usually, encountering a potential fault invalidates a proof attempt when using tight Hoare triples, instead of giving up PREGEN uses this failure information to guide the generation of preconditions.

In more detail, when a sufficient precondition cannot be found that enables a program operation, PREGEN uses abductive inference to synthesize a candidate precondition M describing the missing memory (line 13 in the algorithm). An assertion L describing the leftover memory not needed by the procedure is also synthesized at this point, and it is combined with one of the procedure's specified post-states to obtain a new state for the worklist algorithm to use at program point ℓ'. The way this is done follows very much the ideas in the examples given in Section 2, using bi-abduction. But the formalities involve intricate manipulations concerning logical program variables—the $\vec{e'}$ and $\vec{Y}$ information in line 13—which gets us into the territory of "adaptation" rules in Hoare logic [Hoare 1971; Naumann 2001], with some new twists brought on by the abduction question. We have encapsulated these details inside a procedure AbduceAndAdapt, which will be described in Section 4.2.3.

In order to help convergence of the algorithm, abstraction is applied to candidate preconditions (line 14 in the algorithm) as well as to current abstract states (line 18). Abstraction performs generalization in shape analysis, as we indicated in Section 2.4. When applied to the current state this generalization constitutes a sound deductive inference. But, when applied to a precondition, it is inductive rather than deductive. As is usually the case in scientific reasoning, there is no guarantee that inductive reasoning steps are deductively valid, and we check our work in a separate phase (Section 4.3).

A typical way to call the PREGEN algorithm is with a single initial seed state of the form

$$x_1 = X_1 \wedge \cdots \wedge x_n = X_n \wedge \mathsf{emp},$$

where the x_i are procedure parameters, which appear freely in a control-flow-graph, and the X_i are variables that do not appear at all in the program (e.g., the example in Section 2.4). PREGEN can then use the X_i to refer to quantities that existed at procedure-call time, and it tries to express the precondition in terms of these. After the computation ends and we have expressed a precondition in terms of the X_i, we can then turn around and use x_i in its place. This is the pattern used in all the examples in Section 2, and it is the way our implementation works. (Note that, in case one of the x_i is not assigned to in the program, we don't really need the X_i, and can use x_i at later points in the program to refer back to the pre-state).

4.2.2. Toy Examples Illustrating the Algorithm. We give several toy examples to illustrate aspects of the algorithm. We give the examples using C program syntax rather than the graph form, expecting that no confusion will arise.

Our first example shows a savings in the algorithm based on the idea behind separation logic's frame rule.

Example 4.2. Here we have a program that frees two nodes given as input

```
void continue_along_path(struct node *x,struct node *y)  {
 free(x); free(y); }
```

If we apply the PREGEN algorithm with start state $x = X \wedge y = Y \wedge \mathsf{emp}$; it finds at the `free(x)` statement that there is not enough memory in the precondition, so it abduces the precondition $x \mapsto -$. It then continues on to abduce $y \mapsto -$ at the `free(y)` statement,

Algorithm 4 Precondition Generation

Input: A set of symbolic heaps, *Seed*
Output: A set of symbolic heaps, the candidate preconditions

PREGEN(*Seed*) =

```
1: ws := {(start, p, p) | p ∈ Seed}; reach := ∅;
2: while ws ≠ ∅ do
3:    choose (ℓ, pre, curr) ∈ ws and remove it from ws
4:    if (ℓ, pre, curr) ∉ reach then reach := reach ∪ {(ℓ, pre, curr)};
5:
6:    // Loop over all out-edges from ℓ
7:    for all out-edges (ℓ, x := f(ē), ℓ') from ℓ in T do
8:       Let ∃X⃗.Δ be curr; // strip existentials, prepare to process specs
9:
10:      // Loop in order to process all specs of f
11:      for all {P} f(x⃗) {Q} ∈ summary(f) do
12:         P' := P[ē/x⃗]; Q' := Q[ē/x⃗];    // Instantiate Params
13:         if AbduceAndAdapt(Δ, P', Q', X⃗) = (M, L, ē', Y⃗) ≠ fail then
14:            newpre := abstract#(pre ∗ M)
15:            //Loop over post-states
16:            for all Q ∈ Q' do
17:               Pick a fresh Z
18:               newpost := abstract#(∃X⃗ Z . (Q ∗ L)[Z/x][x/ret, ē'/Y⃗])
19:               ws := ws ∪ {(ℓ', newpre, newpost)}
20:            end for
21:         end if
22:
23:      end for
24:   end for
25: end while
26:
27: return {p | ∃c.(finish, p, c) ∈ reach}
```

which it separately conjoins to the previously abduced fact, giving us $x\mapsto - * y\mapsto -$ overall.

The pertinent point in this example is how, when symbolic execution encounters its first error, it continues executing after the statement, and conjoins subsequently abduced facts to previously conjoined ones. It would have been possible to get the same overall effect by restarting symbolic execution from the beginning, with a new precondition, as soon as an error is found, but this would be wasteful: it would repeat work up until this error. In this example, such work is trivial, but suppose that before the `free(x)` statement we had a long path of statements, π: then we would need to re-analyze π were we to immediately start again.

It is separation logic's frame rule that suggests to us that this re-execution could be redundant. If you analyze a path π, obtaining a pre/post pair $\{H_0\}\,\pi\,\{H_1\}$, and then for a subsequent statement a, we obtain $\{H_1 * A\}\,a\,\{H_2 * L\}$ using bi-abduction where A is nontrivial, then the frame rule tells us that $\{H_0 * A\}\,\pi\,\{H_1 * A\}$ holds, and hence we get $\{H_0 * A\}\,\pi;a\,\{H_2 * L\}$ using the Hoare rule for sequential composition.

This discussion provides a partial explanation for why we have preferred to continue symbolic execution after a potential error has been diagnosed and its treatment abduced. However, we are making these remarks mostly to illustrate points in the PREGEN algorithm, rather than to give an ultimate justification (which is not possible). Indeed, it is possible to find examples where a stop-first-and-re-execute strategy would work better than our continue-along-path strategy (we invite the reader to think of such examples, which involve π, having branches rather than being a straightline path).

Our second example concerns a difference in the way that procedure summaries are used, compared to usual interprocedural analysis algorithms. When a procedure call is considered, PREGEN considers *all* of the procedure preconditions (line 11 in the algorithm). In a normal interprocedural analysis (like the right-hand side algorithm [Reps et al. 1995]), usually, if a call is found to match with one of the specifications in a procedure summary then that specification is used and others are ignored. The reason is that, unless the abstract domain supports conjunction (meet), using more than one spec can only lead to decreased precision and efficiency. In our case, although our analysis is forwards-running, its primary purpose is to help generate preconditions: for this purpose, it makes sense to try as many of the specifications in a procedure summary as possible.

Example 4.3.

```
1  void safe_reset_wrapper(int *y) {
2    // Inferred Pre1: y=0 && emp
3    // Inferred Pre2: y!=0 && y|->-
4    safe_reset(y);
5  }  // Inferred Post1: y=0 && emp
6     // Inferred Post2: y!=0 && y|->0
7  void safe_reset(int *y) {// SUMMARY ONLY
8   // Given Pre1: y=0 && emp
9   // Given Pre2: y!=0 && y|->-
10 } // Given Post1: y=0 && emp
11   // Given Post2: y!=0 && y|->0
```

The analysis of `safe_reset_wrapper` starts with emp, meaning that the heap is empty. When it hits the call to `safe_reset`, the analysis calls the abduction algorithm, and checks whether emp is abducible to the preconditions of the two specs. It finds that emp is abducible with the precondition $y = 0 \wedge \mathsf{emp}$ of the first spec. Instead of ignoring the remaining specs (like a standard forward analysis would), our analysis considers the abducibility of emp with the precondition $y\mapsto -$ of the other spec. Since the analysis considers both specs, it eventually infers two preconditions: $y = 0 \wedge \mathsf{emp}$, and $y \neq 0 \wedge y\mapsto -$. If the analysis had searched for only one applicable spec, it would not have been able to find the second precondition.

Our third example concerns path sensitivity, and what we call the "assume as assert" heuristic. This heuristic is something that is important to take into account when building a practical program analyzer for generating preconditions. It allows one, in a not inconsiderable number of cases, to generate preconditions that cover more rather than fewer paths through the code, as we now explain.

As preparation for this example, recall that the usual specifications of the "assume" and "assert" statements are

$$\{B \wedge \mathsf{emp}\}\mathbf{assert}\,B\{B \wedge \mathsf{emp}\} \qquad \{\mathsf{emp}\}\mathbf{assume}\,B\{B \wedge \mathsf{emp}\},$$

where B is a pure Boolean. Using the frame rule, from these specs, we obtain the familiar

$$\{B \wedge P\}\mathbf{assert}\, B\{B \wedge P\} \qquad \{P\}\mathbf{assume}\, B\{B \wedge P\}$$

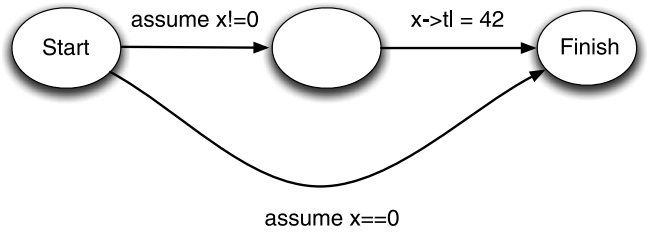
for any P, indicating that for **assert** B must be true beforehand, where for **assume** it need not be.

The **assume** statement is important for analyzing paths, because the conditional statement `if` B `then` C_1 `else` C_2 can be represented as a nondeterministic choice $(\mathbf{assume}\, B); C_1 \; + \; (\mathbf{assume} \neg B); C_2$. In control-flow graphs, this nondeterminism is represented by distinct edges out of a node, as in the following example.

Example 4.4. Consider the program

```
void why_assume_as_assert(struct node *x)      {
    if (x!=0)   { x->tl = 42;  }     }
```

As a control-flow graph, it is



If we interpret the assume statement in the usual way, as $\{\mathsf{emp}\}\mathbf{assume}_B\{B \wedge \mathsf{emp}\}$, then running PREGEN would only give us a single, overly-specific precondition $x \mapsto Z$ which (on its own) rules out the `x==0` path.

However, if we interpret the assume statement as if it were an assert, using $\{B \wedge \mathsf{emp}\}\mathbf{assume}_B\{B \wedge \mathsf{emp}\}$ then we are forced to record the path-specific information in each branch in the precondition, and we will obtain a pair of preconditions

$$x \mapsto Z \qquad \text{and} \qquad x = 0 \wedge \mathsf{emp}$$

where the second one does not rule out the `x==0` path. (Here, in the formula, we use the mathematical $x = 0$ for equality, rather than the C notation `x==0`; we will assume that the reader can disambiguate between the C notation = (for assignment) and the mathematical one (for equality): the latter will be used in assertions, and the former only in program examples.)

It is not unusual to find procedures that dereference memory only after checking some conditions, as in this example, in which case we would certainly like to (for precision reasons) be able to distinguish the cases when the memory is needed from when it is not. Thus, we see that, for the purpose of precondition rather than postcondition generation, it can be helpful to interpret the **assume** statement as if it were **assert** by using

$$\{B \wedge \mathsf{emp}\}\mathbf{assume}_B\{B \wedge \mathsf{emp}\}$$

because this allows us to generate more general preconditions.

This "assume as assert" trick is reminiscent of the situation in symbolic testing, where one wants accurate path information when generating constraints (preconditions) that, when solved, give test cases that drive a program towards error. Our

implementation first tries this trick, but falls back on "assume as assume" when that does not work. We stress that "assume as assert" is an heuristic choice, and there are cases where it does not work well (see, e.g., Example 4.6) in those cases, falling back on "assume as assume" can still give useful preconditions.

Our next two examples concern termination and divergence of the input program. Notice that at line 26 the PREGEN algorithm reports those preconditions that have been carried along (and perhaps altered) all the way up to the final state. The effect of this is that the analysis result ignores generated preconditions that exist *only* at intermediate states, and this makes the analysis tend towards preconditions that guarantee convergence.

Example 4.5. For the program

```
void avoid_provable_divergence(int x) {// Inferred Pre: x!=0 && emp
         if (x==0) { while (1) {};}        }
```

the analysis finds $x \neq 0 \wedge \mathsf{emp}$ as the precondition. This pre is sufficient to avoid the `if` branch. However, there is another precondition, $x = 0$, which is still safe: it leads into the `if` branch, and then the infinite loop.

What is happening is that the working set and reachable set both get a triple $(\ell, x = 0, x = 0)$, where ℓ is the program point immediately before the `while` loop, and inside the `if` branch, but at the program point ℓ' after the loop (but still in the `if`) no states are reachable. This triple is entered into the working set at line 19 of PREGEN (Algorithm 4), after bi-abduction has found $x = 0$ as both the pre and post of an assume statement **assume** $x = 0$ corresponding to one branch of the `if` statement in avoid_provable_divergence. (Note that the pre is obtained using the "assume as assert" trick, but even without this trick; "assume as assume" would give emp as the pre, but we would face the same divergence issue.) Then, after the PREGEN algorithm loops around the same triple can be entered into the reachable set at line 4. The exit of the loop `while (1) {}` corresponds to an edge **assume** false in a control-flow-graph, this results in no states being propagated to program point ℓ' after the loop. That is why the analysis fails to report the precondition $x = 0$ that leads to divergence.

The phenomenon illustrated in this example shows up not infrequently in natural programs. Indeed, for the classic linked-list programs considered in Section 5.1—traversing, deleting from, inserting into, etc.—our analysis finds preconditions that guarantee convergence.

On the other hand, it is not the case that the algorithm always finds preconditions that are sufficient to ensure termination; in particular, that is the case when one cannot find preconditions that unambiguously prove divergence or convergence.

Example 4.6. For this program

```
void not_total_correctness()     } // Inferred Pre:  emp
  struct node *y;
  y = malloc(sizeof(struct node));
  if (y) {while (1) {};}         }
```

the PREGEN algorithm finds emp as the precondition. PREGEN gets to the triple (`finish`, emp, $\neg y \wedge$ emp) at the procedure's end as a result of the condition corresponding to failure of the `if` test in the program. The `while` loop filters out the state $(\ell, \mathsf{emp}, y \mapsto -)$ at the program point ℓ just inside the `if`, and so that does not lead to

any reachable triple at `finish`. The net effect, then, is that emp is found as a candidate precondition.

Thus, we have found a precondition for a procedure that is not guaranteed to terminate. `not_total_correctness()` diverges when `malloc()` succeeds, and converges if `malloc()` fails in its attempt to allocate. The analysis treats `malloc()` as a nondeterministic choice between success and failure, as set down by the axiom in Figure 3. And independently of algorithmics, in terms of this semantics there exists no precondition which guarantees termination of `not_total_correctness()`. (Of course, such a pre could exist with a more detailed semantics taking into account size of memory.)

In this example, there is no way to express the loop condition (y) in terms of variables existing in the pre-state. This is one of the cases where we fall back on default to "assume as assume" rather than treat "assume as assert".

The choice to use the final state to record our candidate preconditions is in no way canonical and we could, if we wanted, alter the algorithm to record some of these safe preconditions that provably lead to divergence. But, there is no canonical such choice as far as we know, and we gave these examples more to help explain how the algorithm is working than to defend our use of the final state and the resultant leaning to preconditions that imply termination.

Remarks on Algorithmic Refinements. The algorithm as given in Algorithm 4 ignores several issues and possible refinements that one would naturally consider in making a practical algorithm that works on real-world code. The intention has been to present an idealized algorithm showing the basic idea of how you can use abduction to do interprocedural analysis, without being distracted by other (important) issues that are independent of the main new technical contributions of this work. We briefly mention two of these other issues.

(1) *Partial Concretization.* In shape analysis, it is often important, for precision reasons, to convert an abstract state to one that has more concrete information. In Sagiv et al. [1998], this phenomenon is called "materialization of a summary node", and in other shape analyses it involves unrolling inductive definitions or using other consequences of induction [Berdine et al. 2005b; Distefano et al. 2006]. To see the issue here, consider the abductive entailment question

$$x \neq \mathsf{nil} \wedge \mathsf{ls}(x, \mathsf{nil}) * ?? \vdash x \mapsto X * \mathsf{true}.$$

There is no consistent answer to this question using symbolic heaps, because we cannot match X on the left-hand side. However, if we unroll the inductive definition of $\mathsf{ls}(x, \mathsf{nil})$ on the left and strip the disjunct made inconsistent by $x \neq \mathsf{nil}$ we obtain $\exists Z\, .x \mapsto Z * \mathsf{ls}(Z, \mathsf{nil})$. If we then strip the existential (as we can do when on left of $\vdash$), we can consider a different abduction question

$$x \mapsto Z * \mathsf{ls}(Z, \mathsf{nil}) * ?? \vdash x \mapsto X * \mathsf{true},$$

which can now be solved easily as $Z = X \wedge \mathsf{emp}$.
Because the literal Algorithm 4 does not have partial concretization, it cannot find a consistent precondition for a statement $x := [y]$ that follows a procedure that generates a linked list of indeterminate length, or of a run of the algorithm that starts with $x \neq \mathsf{nil} \wedge \mathsf{ls}(x, \mathsf{nil})$ as the seed state. However, if we insert some concretization after line 8 of the algorithm, this example can easily be dealt with. Such concretization is implemented in ABDUCTOR and, for the primitive statements, follows the detailed description in Calcagno et al. [2007a], where we refer for further information.

(2) *Disjunctions.* We have assumed in describing the algorithm that Abduction and Frame Inference each return at most one formula, in which case L and M would be disjunctions. The systematic abductive procedure from Section 3.4 in fact would return such a disjunction, and in frame inference it can be useful to perform some case analysis, such as in the question

$$\mathsf{ls}(x, y) * y{\mapsto}0 \vdash x{\mapsto}- * ??,$$

where we could obtain

$$(x{\neq}y \wedge y{\mapsto}0 * \mathsf{true}) \vee (x{=}y \wedge \mathsf{emp})$$

as a solution. The issue of whether we let L or M contain multiple answers is a pragmatic question, but is not theoretically difficult or problematic. Essentially, this is because of the validity of a disjunctive frame rule

$$\frac{\{P\}\ a\ \{\vee_{k\in K} Q_k\} \qquad H * \bigvee_{i\in I} A_i \vdash P * \bigvee_{j\in J} F_j}{\{\bigvee_{i\in I} H * A_i\}\ a\ \{\bigvee_{j\in J, k\in K} Q_k * F_j\}},$$

which is logically equivalent to the ordinary frame rule, using the fact that the separating conjunction $*$ distributes over disjunction:

$$P * \bigvee_{i\in I} Q_i \ = \ \bigvee_{i\in I} P * Q_i.$$

The ABDUCTOR tool allows some extra disjunctions of this form, especially when it comes to using partial concretization in conjunction with framing.

4.2.3. Bi-Abductive Analysis of Procedure Calls. We describe in detail how procedure calls are interpreted in PREGEN. This involves filling in the step

$$\mathsf{AbduceAndAdapt}(\Delta, P', \mathcal{Q}', \vec{X}) = (M, L, \vec{e'}, \vec{Y}) \neq \mathsf{fail}$$

at line 13 in Algorithm 4. This finds missing and leftover heaps M and L, and instantiations of variables.

Conceptually, our work amounts to applying the bi-abductive frame rule

$$\frac{\{A\}C\{B\} \quad \mathsf{BiAbd}(P, A) = (M, L)}{\{P * M\}C\{B * L\}}\ \mathsf{Frame\ Rule\ (Bi\text{-}Abductive\ Analysis\ Version)}$$

together with several considerations about disjunctions and variable renaming.

The variable issues arise from the need to adapt a spec at a procedure-call site, by instantiating procedure parameters and logical variables in a specification. For example, given a spec

$$\{x{\mapsto}X * y{\mapsto}Y\}\mathtt{swap}(x, y)\{\mathsf{ret{=}nil} \wedge x{\mapsto}Y * y{\mapsto}X\}$$

we could consider a call with pre-state

$$\{z{\mapsto}7 * w{\mapsto}11\}\mathtt{call\ swap}(z, w)\{???\}$$

and the task of adaptation is to find the instantiation $x := z, y := w, X := 7, Y := 11$ which allows us to infer $z{\mapsto}11 * w{\mapsto}7$ as the post.

The issue of adaptation has been well studied in Hoare logic [Cook 1978; Hoare 1971; Kleymann 1999]. Our treatment here has two additions. First, and most importantly, we will perform adaptation of the missing anti-frame M. Adaptation will attempt to express the anti-frame in terms of logical variables that could exist at procedure precondition time, instead of in terms of program variables at a call site. Second,

```
AbduceAndAdapt(Δ, P, Q, X⃗) =
  if BiAbd(Δ, P) = (M, L) ≠ fail and Rename(Δ, M, P, Q, X⃗) = (e⃗, Y⃗, M0) ≠ fail then
    return (M0, L, e⃗, Y⃗)
  else
    return fail
  end if

Rename(Δ, M, P, Q, X⃗) =
  Let Y⃗ be FreeLVar(P, Q);
  Pick e⃗ disjoint from Y⃗ such that Δ ∗ M ⊢ e⃗=Y⃗,
    but if cannot pick such e⃗, return fail;
  Pick M0 disjoint from Y⃗, X⃗ and program variables such that
              Δ ∗ M0 ⊢ Δ ∗ M[e⃗/Y⃗],
    but if cannot pick such M0, return fail;
  return (e⃗, Y⃗, M0)
```

Fig. 4. Adaptation in the presence of Abduction.

we will be using abduction to instantiate logical variables, as well as to find missing heap portions.

The need to express the anti-frame in terms of quantities existing in the pre-state was already illustrated in the `free_list` example from Section 2.4.

The definition of the AbduceAndAdapt procedure is in Figure 4. It attempts to do bi-abduction using the procedure from Algorithm 3. In case bi-abduction succeeds, the next step massages the anti-frame and finds instantiations of parameters by a call to the Rename subroutine. Rename performs essential but intricate trickery with variables, as is usual in Hoare logic treatments of procedures. Generally, the anti-frame M_0 that it finds will be expressed in terms of logical variables that are fresh or free in Δ. This is to ensure that it is independent of program variables that might be modified between the start of a procedure and the point of discovery of M_0, allowing it to be used later as a precondition. The vectors $\vec{e}$ and $\vec{Y}$ tell us how to instantiate logical variables in the specification, as discussed in the `swap` example at the beginning of this section.

While technical in nature, properly dealing with the issues tackled by Rename is essential for the precision of specification discovery. Many procedures will have logical variables in their specifications, and imprecise treatment of them would lead to an unacceptably imprecise analysis.

Example 4.7. We give a description of the abstract semantics of v := swap(x, y) with

$$pre \stackrel{def}{=} (x = X \wedge y = Y \wedge X{\mapsto}Z), \quad curr \stackrel{def}{=} (y = Y \wedge x{\mapsto}y * z{\mapsto}0)$$

using the specification of `swap` given earlier in this section. The semantics first invokes BiAbd($y = Y \wedge x{\mapsto}y * z{\mapsto}0$, $x{\mapsto}X' * y{\mapsto}Y'$) and infers M and L:

$$M \stackrel{def}{=} (X' = y \wedge Y' = W \wedge y{\mapsto}W), \qquad L \stackrel{def}{=} z{\mapsto}0.$$

From this output, the analysis computes the three necessary elements for analyzing the call v:=swap(x, y), which are a leftover frame $z{\mapsto}0$, a missing anti-frame $y{\mapsto}W$, and the instantiation $X' = y \wedge Y' = W$ of logical variables X', Y' in the spec of `swap`. Rename then computes $(\vec{e'}, \vec{Y}, M_0)$, where $M_0 \stackrel{def}{=} (Y{\mapsto}W)$ is expressed in terms of existing logical variable Y and fresh W, $\vec{e'} \stackrel{def}{=} (y, W)$ and $\vec{Y'} \stackrel{def}{=} (X', Y')$. These elements form the result

of analyzing the call. The precondition *pre* is updated by $*$-conjoining the missing anti-frame $Y \mapsto W$:

$$pre * Y \mapsto W \iff x = X \wedge y = Y \wedge X \mapsto Z * Y \mapsto W.$$

The current heap *curr* is mutated according to the instantiated spec of `swap` with $X' = y \wedge Y' = W$ and the leftover frame $z \mapsto 0$, and becomes

$$v = 0 \wedge x \mapsto W * y \mapsto Z * z \mapsto 0.$$

Thus, the result of $v :=$ `swap`(x, y) on $(\ell, pre, post)$ is the singleton set

$$\{ (\ell', x = X \wedge y = Y \wedge X \mapsto Z * Y \mapsto W, \quad v = 0 \wedge x \mapsto W * y \mapsto Z * z \mapsto 0) \}.$$

Example 4.8. This is an example of when the attempt to rename an abduced assertion in terms of quantities available at the precondition is not possible.

```
1  void cannot_verify(int x) {
2    int y = f(x);
3    g(y);  // Abduced: y!=0, but cannot express at pre-state
4  }
5
6  int f(int x){//SUMMARY ONLY
7     // Given Pre: emp
8 }  // Given Post: emp
9
10  int g(int x){//SUMMARY ONLY
11     // Given Pre: x!=0
12 }  // Given Post: emp
```

The procedure `cannot_verify` here calls other procedures `f` and `g`. We are given specs for `f` and `g`, but not their code. Procedure `f` has a spec that is noncommittal about the value of `f(x)`; logically, it can be any integer, so that the spec views `f` as essentially a nondeterministic selection of integers. Procedure `g` expects an integer argument that is nonzero.

At line 3 when we call `g`, we will not know the value of `y`. Abduction will then tell us that we *should* have `y!=0`. However, it is not possible to express the assumption `y!=0` at line 3 in terms of the value `X` of `x` at the initial state: there is no assertion at the pre-state that implies that `y!=0` at line 3.

Note that it might or might not be that the case that the code of `f` (which we are not given) in this example is non-deterministic. If `f` is, say, a memory-safe function that is difficult to analyze for full correctness (it might involve hashing or cryptography), then a program analysis will treat it as if it was nondeterministic.

There is a similarity between this example and Example 4.6, where we also encountered an assertion that could not be pushed back to the pre-state. But there we had a get-out clause, falling back on "assume as assert", where here the failure to re-express the assertion leads to a failed verification.

4.2.4. Bi-Abduction and Footprints. We said earlier that our program analysis would "shoot for" footprints, descriptions of the cells used by a program. We give a theoretical account of some issues surrounding this idea. This section may be omitted without loss of continuity.

There is little we can say about entire programs because, as mentioned earlier, an abduced precondition got from a proof attempt that follows one path in a program is not necessarily relevant to other paths (though we heuristically hope that it will be). So

we concentrate on sequential compositions of two actions in this subsection, ignoring parameters, to focus on the essence of the analysis of paths.

Intuitively, the safe states of a command C are those states on which C will not cause a memory error (such as dereferencing a dangling pointer), and the footprint consists of the minimal such states. To describe footprints logically,we will call upon the concept of a precise predicate, which is one that, for any state, there is at most one substate (think of it as the "accessed cells") beneath it that satisfies the predicate.

Definition 4.9 (*Precise Predicate*). A predicate P is precise if, for every s, h there is at most one $h_f \subseteq h$ where $s, h_f \models P$.

Definition 4.10 (*Safe States and Footprints*). Let C be a sequence of actions, where $\mathsf{spec}(\alpha) = \{P_\alpha\}\alpha\{Q_\alpha\}$ and each P_α is precise.

(1) The safe states $\mathsf{safe}(C)$ are defined by induction as follows:
   (a) $\mathsf{safe}(\alpha) \stackrel{def}{=} P_\alpha * \mathsf{true}$.
   (b) $\mathsf{safe}(\alpha; C) \stackrel{def}{=} P_\alpha * (Q_\alpha \mathrel{-\!\!*} (\mathsf{safe}(C)))$.
(2) The *footprint* of C, written $\mathsf{foot}(C)$, is defined as

$$\mathsf{foot}(C) \stackrel{def}{=} \mathsf{min}(\mathsf{safe}(C)).$$

We have defined the safe states of sequential composition using a particular formula, rather than referring to an independent semantics of programs as state transformers. However, this definition can be seen to agree with "the set of states on which $C_1; C_2$ is not $\top$", using the semantic framework of abstract separation logic [Calcagno et al. 2007b].[9]

Recall from Lemma 3.13 that the best solution to the abduction question $P * ??? \models Q$ is given by $\mathsf{min}(P \mathrel{-\!\!*} Q)$. The next theorem shows that the canonical spec of a sequential composition of two actions describes the footprint, if we assume that the bi-abductive theorem prover computes the best antiframe.

THEOREM 4.11 (DERIVING FOOTPRINT). *Suppose* $\mathsf{spec}(\alpha) = \{P_\alpha\}\alpha\{Q_\alpha\}$ *and each* P_α *is precise, and* C *is a sequence of actions. Suppose further that each call to the bi-abductive prover in the definition of the canonical spec* $\mathsf{spec}(C)$ *delivers the best antiframe, and let* P_C *be the precondition of the canonical spec. Then,* $\mathsf{foot}(C) = P_C$.

PROOF. The case of a single action is trivial.

For $a; C$ we claim that

$$\mathsf{foot}(\alpha; C) = P_\alpha * M,$$

where M is the best solution to the abduction question $Q_\alpha * ??? \models P_C * \mathsf{true}$. If we can establish this, we are done.

By induction, we know that $\mathsf{foot}(C) = P_C$, and therefore that $\mathsf{safe}(C) = P_C * \mathsf{true}$. By Theorem 3.13, we know that $M \equiv \mathsf{min}(Q_\alpha \mathrel{-\!\!*} (P_C * \mathsf{true}))$. We can then derive

$$\begin{aligned} \mathsf{foot}(\alpha; C) &= \mathsf{min}(P_\alpha * (Q_\alpha \mathrel{-\!\!*} (P_C * \mathsf{true}))) \\ &= \mathsf{min}(P_\alpha) * \mathsf{min}(Q_\alpha \mathrel{-\!\!*} (P_C * \mathsf{true})) \\ &= P_\alpha * M, \end{aligned}$$

where the last two steps follow from precision of P_α. □

[9]We remark that this correspondence breaks down for some imprecise assertions. This is not a problem because here footprints should be precise, and so we are justified in the precision requirement.

Algorithm 5 INFERSPECS Algorithm

Input: A set of symbolic heaps, *Seed*
Output: A set of pre/post pairs

```
INFERSPECS(Seed) =
  Specs := ∅
  Candidate Preconditions := PREGEN(Seed)
  for all P ∈ Candidate Preconditions do
    if PostGen(P) = Q ≠ fail then
      Specs := Specs ∪ (P, Q)
    end if
  end for
  return Specs
```

This result gives some indication that using bi-abduction to analyze paths can lead to accurate results. Further theoretical work in this direction could prove useful in future work.

4.3. Pre/Post Synthesis: Revising Our Work after Induction and Abduction

Now that we have a way to generate candidate preconditions, we can generate pre/post specs for a code fragment (and specs of procedures that it calls) by employing a standard, nonabductive forwards analysis. The INFERSPECS algorithm, Algorithm 5, does this. It first uses PREGEN to generate a number of candidate preconditions. Then, for each candidate precondition, it uses a forwards analysis POSTGEN to attempt to generate postconditions. In cases where the forwards analysis fails to find a proof, the candidate precondition is filtered from the output. (the $PostGen(P) = \mathcal{Q} \neq \mathsf{fail}$ check). Here is an example showing the need for filtering.

Example 4.12. When PREGEN is run on the following program

```
void nondet-example(struct node *x,*y) {
    if (nondet()) {x->tl = 0;} else {y->tl = 0;}  },
```

where `nondet()` is a Boolean whose value we do not know at precondition time (e.g., it can be generated by `malloc()`). The abduction question asked when PREGEN encounters statement `x → tl = 0` is $x = X \wedge y = Y * ?? \models x\mapsto - * \mathsf{true}$, and this results in $x\mapsto -$ being generated as a precondition. Similarly, `y → tl = 0` leads to precondition $y\mapsto -$. In the end, PREGEN discovers two preconditions, $x\mapsto -$ and $y\mapsto -$, neither of which is safe.

When POSTGEN is run on these preconditions it returns fail for each, and this causes INFERSPECS to filter both of them out. The failure in POSTGEN occurs at line 12 of the algorithm. When $x\mapsto -$ is the beginning preconditon, the failure happens when analyzing the statement `y → tl = 0` in nondet-example, the reason being that $x\mapsto -$ does not entail $y\mapsto - * \mathsf{true}$. The reason for the failure for the $y\mapsto -$ pecondition is the mirror image.

What is happening here is that an abduced precondition can rule out an error for one path, but it might not be enough to rule out errors on other paths.[10]

[10]This is similar to what happens in abstraction refinement, where synthesizing a new predicate from a path might say nothing about other paths.

Another example concerns abstraction rather than path sensitivity.

Example 4.13. When PREGEN is run on the following program

```
void needs_even_list(struct node *x)  {
       while (x!=0) { x=x->tl; x=x->tl; } }
```

using the abstract domain from Section 3.1, it generates two preconditions,

$$x{=}0 \wedge \mathsf{emp} \qquad \text{and} \qquad x{\mapsto}X * \mathsf{ls}(X, 0).$$

The left-hand precondition is safe, where the right-hand precondition is unsafe and filtered-out by INFERSPECS.

The problem in this example is that the program fools the abstraction function $\mathsf{abstract}^{\#}$ from Section 3.1.3. When PREGEN uses $\mathsf{abstract}^{\#}$ to generalize the precondition, it forgets about whether the list under consideration was of odd or even length, where this program is safe for only even-length lists given as input: that is, $\mathsf{abstract}^{\#}$ introduces a list segment predicate into the precondition, and the predicate we are using is oblivious to the distinction between oddness and evenness.

We could address the problem just shown for `needs_even_list()` by using a different abstract domain than in Section 3.1.3, a domain that distinguishes lists of odd and even lengths. But, that fix would be scuppered by other examples: no matter what abstract domain we consider, there will always be a program that fools the analysis, by general computability considerations.

Given that PREGEN discovers some unsafe preconditions, there are several ways to react. One would be to record the postconditions as well as preconditions of PREGEN and place it in an iterative loop, that calls the algorithm again on generated preconditions, until abduction always returns emp (so that we have a proof, modulo several other considerations). Iteration would be a good way to deal with `nondet-example`, were we could discover the safe precondition $x{\mapsto}- * y{\mapsto}-$ by running PREGEN twice. While iteration is a fundamentally interesting possibility, we would eventually want to run a standard forwards analysis because PREGEN makes several choices (assume-as-assert, trying all procedure specs) which are oriented to precondition generation rather than accurate postcondition generation.

Thus, to infer pre/post specs, we also employ the POSTGEN algorithm (Algorithm 6), which is itself a standard forwards analysis algorithm. One pertinent point is the exit from the for loop at line 19; as soon as we find one spec that matches the current abstract state, we do not try other specs. Another point is the return statement in line 23: in case no match is found for a procedure call, we know that our proof attempt has failed and we give up. Finally, we have a procedure $\mathsf{FrameAndAdapt}$ (line 12) which must apply the frame rule of separation logic, the rule of consequence, and variable adaptation. It does less work that the corresponding $\mathsf{AbduceAndAdapt}$ used in PREGEN, and we omit its formal definition.

Again, POSTGEN and INFERSPECS are idealized algoritms, and there are pragmatic variants of them. One variant would be to have INFERSPECS call PREGEN several times before POSTGEN. Another would be to use a partial join operator in various places, in order to speed up the analysis [Yang et al. 2008]. You could use join on the current heap component of POSTGEN to keep the number of abstract states from growing too rapidly; the same can in fact be done for PREGEN. Another is to use join on the preconditions in INFERSPECS, after PREGEN is called and before POSTGEN: thinking of the output of INFERSPECS as a procedure summary, for use subsequently in an analysis of a larger system, this is an attempt to shrink the summary. Mixed strategies

Algorithm 6 Standard Forwards Analysis Algorithm

Input: A symbolic heap, *seed*
Output: A set of symbolic heaps, the postconditions, or fail

POSTGEN(*seed*) =

```
1:  ws := {(start, seed)};  reach := ∅;
2:  while ws ≠ ∅ do
3:    choose (ℓ, curr) ∈ ws and remove it from ws
4:    if (ℓ, curr) ∉ reach then reach := reach ∪ {(ℓ, curr)};
5:    for all out-edges (ℓ, x := f(e⃗), ℓ′) from ℓ in T do
6:
7:      Let ∃X⃗.Δ be curr; // strip existentials, prepare to process specs
8:      found good spec := false
9:      // Loop in order to process specs of f
10:     for all {P}f(x⃗){Q} ∈ summary(f) do
11:       P′ := P[e⃗/x⃗]; Q′ := Q[e⃗/x⃗];    // Instantiate Params
12:       if FrameAndAdapt(Δ, P′, Q′) = (L, e⃗′, Y⃗) ≠ fail then
13:         //Loop over post-states
14:         for all Q ∈ Q′ do
15:           Pick a fresh Z
16:           newpost := abstract#(∃X⃗ Z . (Q ∗ L)[Z/x][x/ret, e⃗′/Y⃗])
17:           ws := ws ∪ {(ℓ′, newpost)}
18:         end for
19:         found good spec := true; goto line 22
20:       end if
21:     end for
22:     if not found good spec then
23:       return fail
24:     end if
25:   end for
26: end while
27: return {post | (finish, post) ∈ reach}
```

are also possible: attempt to use join in creating a summary, and if that does not work, fall back on the unjoined candidate preconditions. The ABDUCTOR tool implements a number of these possibilities (as usual, there is no single best configuration amongst these options).

4.3.1. Recipe for Compositional Analysis. A compositional method of program analysis is an immediate consequence of the ability to generate pre/post specs for a program fragment without relying on the program's calling context.

Suppose, first, that the program under consideration has no recursion, and can be represented accurately in the model of programs of this section. We are talking about sequential, possibly nondeterministic programs, without concurrency and without code pointers. Then, the compositional analysis algorithm proceeds as follows, given a codebase (even an incomplete codebase without a "main" program).

(1) Start at the leaves of the call-tree, and compute pre/post specs for the procedures there.

(2) Go up in the tree computing pre/post specs for procedures using specs previously obtained for procedures lower in the call tree.

For programs with recursion an alteration to this scheme is necessary, where one groups collections of mutually recursive procedures together. This is possible using a bottom-up version of the classic right-hand side interprocedural analysis algorithm.

There is nothing novel about the general points just made, and we avoid a formal description of them here (the POPL version of this article [Calcagno et al. 2009a] contains a partial formalization). So we take it as given in the next section that a compositional bottom-up program analysis follows at once from the ideas in this section.

4.4. The Soundness Property

When applied to an entire codebase, our analysis can generate a collection of pre/post specs for each procedure. In this section, we describe the soundness property that the analysis satisfies. The purpose is to make clear what is computed by our analysis, rather than to take an extended detour into program semantics, and this section is therefore kept brief.

We think of a program according to Definition 4.1 as a representation of an individual procedure, where the free variables are the parameters. Accordingly, we expect a program to determine a state transformation which, when given values for the parameters, produces a nondeterministic function that takes an initial heap to a final heap, or an indication $\top$ of potential memory fault.

$$\mathsf{ConcreteProcs} \stackrel{def}{=} \mathsf{Stack} \to \mathsf{Heap} \to \mathcal{P}(\mathsf{Heap})^{\top}$$

Note that, by this definition, ConcreteProcs ordered pointwise is a complete lattice, and we will use its meet operation $\bigcap$.

For example, the meaning of a program that deletes all the elements in a linked list pointed to by x will take a heap h and return $\top$ if x does not point to a nil-terminated list in h, and otherwise it will return $\{h'\}$ where h' is the heap obtained from h by deleting this linked list. The use of a set of output heaps rather than a single one allows from nondeterminism (e.g., `malloc` nondeterministically returns an unused cell). The reason for putting fault on the top is related to partial correctness (safety but not liveness), and is explained further in Calcagno et al. [2007b].

In whole-program analysis, one often mimics the function in a concrete semantics with a function between abstract states, and one gets a typical inequational diagram saying that if the abstract transfer function maps one abstract state to another, then a certain simulation relation holds when we concretize abstract values. But, our analysis over-approximates the entire functional meaning of a procedure, without explicitly using a function on the entire abstract domain (such a function is just too big). This is what is sometimes referred to as a relational analysis [Cousot and Cousot 2002]. The abstract domain AbstractProcs avoids the function space, and instead follows the procedure summary idea, where an element of the abstract domain is a collection of pre/post specs. Although, in the worst case, this collection could be large, the intention is that it can often be very small in practice, and in any case for the domain to allow for small specifications rather than only entire tables involving the entire abstract domain.

$$\mathsf{AbstractProcs} \stackrel{def}{=} \mathcal{P}(\mathsf{Specs})$$

$$\mathsf{Specs} \stackrel{def}{=} \mathsf{AbstractStates} \times \mathcal{P}(\mathsf{AbstractStates}).$$

To show the sense in which an element of the abstract domain over-approximates, an entire procedure meaning we connect the abstract and concrete domains with a function

$$\gamma : \mathsf{AbstractProcs} \longrightarrow \mathsf{ConcreteProcs}.$$

We first define a subsidiary function

$$\mu : \mathsf{Specs} \longrightarrow \mathsf{ConcreteProcs}$$

and then use it to define γ.

So, suppose we have a single spec $(p, (q_1, ..., q_n)) \in \mathsf{Specs}$. As preparation for this definition of μ, if s is a stack and $\vec{v}$ is an assignment of values to (all) logical variables, then $s + \vec{v}$ is s altered so that the logical variables are updated according to $\vec{v}$. We will use a meet indexed by assignments $\vec{v}$ to model the idea that logical variables are universally quantified outside of Hoare triples. The definition is

$$\begin{aligned} &\mu(p, (q_1, ..., q_n))s\,h \\ &\stackrel{def}{=} \textstyle\bigsqcap_{\vec{v}} \mathsf{proj}\Big(\bigsqcap\big\{(\cup_i [\![q_i]\!]) * \{(s + \vec{v}, h_1)\} \mid h = h_0 \bullet h_1 \text{ and } (s + \vec{v}, h_0) \in [\![p]\!]\big\}\Big), \end{aligned}$$

where $\mathsf{proj} : \mathcal{P}(\mathsf{Stack} \times \mathsf{Heap})^\top \to \mathcal{P}(\mathsf{Heap})^\top$ is the evident projection function that picks out the heap, and $[\![p]\!], [\![q_i]\!] \in \mathcal{P}(\mathsf{Stack} \times \mathsf{Heap})$ are denotations of assertions. In this formula, the expression $(\cup_i [\![q_i]\!]) * \{(s, h_1)\}$ is like a postcondition using a disjunction and a singleton assertion $\{(s, h_1)\}$ as the frame using the frame rule.

We then define γ again using the meet in ConcreteProcs:

$$\gamma(D) \stackrel{def}{=} \bigsqcap\{\mu d \mid d \in D\}.$$

In case D is empty, this is the greatest element $\top$ $(= \lambda s.\top)$ of the lattice ConcreteProcs. Thus, our analysis always finds an over-approximation. When it to find a pre/post spec for a procedure the over-approximation is, for trivial reasons, $\top$.

With this connection between the abstract and concrete domains we are in a position to state the soundness property. To do this, we assume that a program `Prog` determines a function

$$[\![\texttt{Prog}]\!] \in \mathsf{ConcreteProcs}$$

which works by assuming a sequential runtime model and the greatest state transformation which satisfies the Hoare triples associated with each edge [Calcagno et al. 2007b].[11]

THEOREM 4.14 (SOUNDNESS OF PROGRAM ANALYSIS). *Suppose* INFERSPECS *(Seed) terminates and returns D, where Seed* $= x_1 = X_1 \wedge \cdots \wedge x_n = X_n \wedge \mathsf{emp}$ *and* $x_1, ..., x_n$ *contains all the free variables in the program* `prog` *under consideration.*

(1) (Soundness stated in Abstract Interpretation style)

$$\gamma D \sqsupseteq [\![\texttt{Prog}]\!].$$

(2) (Soundness stated in Hoare Logic style)
If $(P, \{Q_1, ..., Q_k\}) \in D$*, then*

$$\{P\}\texttt{prog}\{Q_1 \vee \cdots \vee Q_k\}$$

is a true Hoare triple according to the tight (fault-avoiding) interpretation of triples in separation logic (cf., Ishtiaq and O'Hearn [2001], Reynolds [2002], Calcagno et al. [2007b]).

[11]It makes sense to use the greatest such state transformer, as it over-approximates any functions satisfying the specs and can be seen as what the specs denote.

We have stated this theorem in a somewhat informal manner. Filling in the details mainly involves an exercise in the semantics of programs used in [Calcagno et al. 2007b]. Once this is done, the argument for soundness used that POSTGEN works by using sound proof rules. For instance, the treatment of each procedure call by the analysis can be justified by a valid Hoare triple for the call, and the worklist algorithm used in the analysis can be explained by the disjunction rule, the rule for loops and other rules for compound commands in Hoare logic.

5. CASE STUDIES

ABDUCTOR is a prototype program analysis tool we have developed implementing the ideas in the previous sections. In this section, we report on case studies applying ABDUCTOR.

5.1. Small Examples: Simple Linked-List Programs

We first consider examples that probe the question of the quality of the specifications inferred by the compositional algorithm. Our intention here is to test the algorithm itself, and not the underlying abstract domain. For this we concentrate on simple linked-list programs for which accurate abstract domains are known for computing postconditions, if the human supplies a precondition. The purpose of our work here is to see whether good preconditions can be inferred by our PREGEN algorithm, in some cases, where an abstract domain is known to work well (for forwards analysis).

Table I shows the results of applying the ABDUCTOR analysis to a set of typical list programs. The Candidate Pre column reports the number of the (initially) computed preconditions. Amongst all these candidates, some can be unsafe and there can be redundancy in that one can imply another. The Unsafe Pre column indicates the preconditions filtered out when we re-execute the analysis. In the Discovered Precondition column we have dropped the redundant cases and used implication to obtain a compact representation that could be displayed in the table. For the same program, the table shows different preconditions on different lines. For all tests except one (`merge.c`), our analysis produced a precondition from which the program can run safely, without generating a memory fault, obtaining a true Hoare triple. We comment on a few representative examples.

`del-doublestar` uses the usual C trick of double indirection to avoid unnecessary checking for the first element, when deleting an element from a list.

```
void del-doublestar(nodeT **listP, elementT value) {  nodeT *currP, *prevP;
  prevP=0;
  for (currP=*listP; currP!=0; prevP=currP, currP=currP->next) {
    if (currP->elmt==value) {  /* Found it. */
      if (prevP==0) *listP=currP->next;
      else prevP->next=currP->next;
      free(currP);
} } }
```

The first discovered precondition is

```
listP|->x_ * ls(x_,x1_) * x1_|->elmt:value.
```

This shows the cells that are accessed when the element being searched for happens to be in the list. Note that it does not record list items that might follow the value:

Table I. Running ABDUCTOR on Simple List Programs

Program	Candidate Pre	Uns Pre	Discovered Preconditions
append.c	4	0	ls(x,0)
append-dispose.c	17	0	ls(x,0)*ls(y,0)
copy.c	4	0	ls(c,0)
create.c	1	0	emp
del-doublestar.c	10	0	listP↦x_*ls(x_,x1_) *x1_↦elmt:value, listP↦x_*ls(x_,0)
del-all.c	4	0	ls(c,0)
del-all-circular.c	3	0	c↦c_*ls(c_,c)
del-lseg.c	48	0	z≠0∧ls(c,z)*ls(z,0), z≠w∧ls(c,z)*ls(z,w)*w↦0, z≠w∧w_≠0∧ls(c,z)*ls(z,w)*w↦w_, z≠c∧c↦0, z≠c∧z≠c_∧c↦c_*ls(c_,0), c=0∧emp
find.c	12	0	ls(c,b)*b↦0, b_≠0∧ls(c,b)*b↦b_, b≠c∧b≠c_∧c↦c_*lseg(c_,0), b≠c∧c↦0, c=0∧emp
insert.c	10	0	e1≠0∧e2≠0∧c_≠d_∧ c↦c_*ls(c_,d_)*d↦dta:e3, e1≠0∧e2≠0∧c_≠0∧c↦c_*ls(c_,0), e1≠0∧c↦0, e1≠0∧e2=0∧c↦c_*c_↦-, e1=0∧c↦-, c=0∧emp
merge.c	30	30	—
reverse.c	4	0	ls(c,0)
traverse-circ.c	3	0	c↦c_*ls(c_,c)

they are not accessed.[12] A postcondition for this precondition has just a list segment running to x1_:

```
listP|->x_ * ls(x_,x1_).
```

The other precondition

```
listP|->x_ * ls(x_,0)
```

corresponds to when the element being searched for is not in the list.

The algorithm fails to discover a circular list in the precondition

```
listP|->x_ * ls(x_,x_).
```

[12]This point could be relevant to interprocedural analysis, where Rinetzky et al. [2005a] and Gotsman et al. [2006] pass a useful but coarse over-approximation of the footprint to a procedure, consisting of all abstract nodes reachable from certain roots.

The program infinitely loops on this circular input, but does not commit a memory safety violation, so that would be a safe precondition. This is a further instance of the phenomenon highlighted in Example 4.5, where the analysis tends to exclude preconditions that lead to guaranteed divergence.

In contrast, for the procedure that explicitly traverses a circular list

```
void traverse-circ(struct node *c)  {
  struct node *h;
  h=c; c=c->tl;
  while (c!=h) { c=c->tl;}
},
```

the analysis does find a circular list as the pre. An important point is that the boolean guard of the loop `(c!=h)` here essentially covers the alias check that is ignored by our abduction algorithm. Before exiting the loop, the algorithm will have discovered as precondition a list segment from `h` to (the current) `c`. At this point, the assume-as-assert interpretation of conditionals illustrated in Example 4.4, when applied to the condition `c=h` at loop exit, turns the list segment into a circular list.

Further issues can be seen by contrasting `append.c` and `append-dispose.c`. The former is the typical algorithm for appending two lists `x` and `y`. The computed precondition is

```
ls(x_,0)
```

Again, notice that nothing reachable from `y` is included, as the appended list is not traversed by the algorithm: it just swings a pointer from the end of the first list. However, when we post-compose appending with code to delete all elements in the acyclic list rooted at `x`, which is what `append-dispose.c` does, then the footprint requires an acyclic list from `y` as well

```
ls(x,0) * ls(y,0)
```

The only program for which we failed to find a safe precondition was `merge.c`, the usual program to merge two sorted lists: instead, ABDUCTOR returned all unsafe candidates (which were pruned at re-execution time). To see the reason of this failure, consider the case that `merge.c` traverses only one of the two input sorted lists until the end. This case happens when the other partially traversed list contains values larger than those stored in the full-traversed list. ABDUCTOR correctly finds that only one list is traversed completely, but it does not know that this partial traversal is due to the values stored in the list. As a result, it freely generalises its finding using the abstraction, and suggests candidate preconditions which say that there are two disjoint lists, one terminating with null but the other ending with any values. These candidate do not guarantee the memory safety of `merge.c`, hence, being discarded at re-execution time.

The few number of unsafe preconditions encountered indicates that the inferences performed to generate preconditions, while unsound in general, can nonetheless be reasonably effective, at least in certain circumstances. Of course, this is a highly controlled case study, and the programs are extremely simple. But, it is significant that only a few years ago performing a forwards analysis with a user-supplied precondition for such programs was considered enough of a research contribution to merit publication [Balaban et al. 2005; Bouajjani et al. 2005; Distefano et al. 2006; Manevich et al. 2005]; the work in this subsection represents an advance in the state of the art, in that it shows that such preconditions can often be inferred.

5.2. Medium Example: IEEE (1394) Firewire Device Driver

We do not mean to imply that the results of ABDUCTOR are always as rosy as the picture painted in the application to simple list programs. We ran ABDUCTOR as well on the IEEE 1394 (firewire) Device Driver for Windows, code of around 10K LOC containing 121 procedures, and there our results were more mixed. The driver was analyzed in a top-down fashion in our previous work on SPACEINVADER [Yang et al. 2008]. SPACEINVADER implements a whole-program analysis which, after supplying environment code which nondeterministically called the dispatch routines with appropriately initialized data structures, proved the absence of pointer-safety faults in a version of the driver where the faults our analysis had found were fixed. We applied ABDUCTOR to the same driver, but without the environment code.

ABDUCTOR was able to find consistent specifications (where the precondition is not inconsistent) for all 121 of the procedures. Note that many of these procedures have to "fit together" with one another. If we found a completely imprecise spec of one procedure, then this might be inconsistent with another procedure's calling expectations. Put another way, our analysis has found proofs of the Hoare triples for higher-level procedures, which would have been impossible if the specs discovered for subprocedures were too imprecise for the relevant call sites.

More anecdotally, looking at top-level procedures we find that the specifications describe complex, nested structures (not necessarily weakest liberal preconditions), which could only be found if the analysis was tracking traversals of these structures with some degree of precision. To take one example, the analysis of top-level procedure `t1394Diag_Pnp` discovers preconditions with several circular linked lists, some of which have nested acyclic sub-lists. This is as expected from Yang et al. [2008]. But, some of the preconditions are unexpected. Usually, the firewire driver collaborates with other lower-level drivers, and this collaboration involves the dereference of certain fields of these lower-level drivers. So, if a human (as in our previous work) writes preconditions for the driver, he or she normally specifies that the collaborating lower-level drivers are allocated, because otherwise, the preconditions cannot ensure pointer safety. What the bottom-up analysis finds is that these lower-level drivers are not necessarily dereferenced; dereferencing depends on the values of parameters. The preconditions discovered by ABDUCTOR thus clarify this dependency relation between the values of parameters and the dereference of lower-level drivers.

However, for one of the routines, `t1394Diag_PnpRemoveDevice`, ABDUCTOR found an "overly specific" precondition which rules out many of the paths in the code. The specification would not fit together with the environment code if we were to have included it, and we cannot thereby claim that ABDUCTOR has verified the driver as thoroughly as the top-down analysis. This kind of incompleteness is not completely unexpected in compositional analyses, which do not have the benefit of the context of a program to help the analysis along. As said previously, the abstract domain used in the experiments has been designed with a whole-program analysis in mind. It might be that a more expressive abstract domain could assist the compositional analysis to overcome these difficulties in synthesizing a more general precondition for `t1394Diag_PnpRemoveDevice`.

One might speculate that a compositional analysis technique could help the user to formulate environment code when one seeks to "close the system" in a verification effort, without expecting that the automatic compositional method will be sufficient to close the system in a satisfactory way, without human intervention, in all circumstances.

Table II. Case Studies with Large Programs (timeout = 1s)

Program	KLOC	Num. Procs	Proven Procs	Proven %	Time
Linux kernel 2.6.30	3032	143768	86268	60.0	9617.44
Gimp 2.4.6	705	16087	8624	53.6	8422.03
Gtk 2.18.9	511	18084	9657	53.4	5242.23
Emacs 23.2	252	3800	1630	42.9	1802.24
Glib 2.24.0	236	6293	3020	48.0	3240.81
Cyrus imapd 2.3.13	225	1654	1150	68.2	1131.72
OpenSSL 0.9.8g	224	4982	3353	67.3	1449.61
Bind 9.5.0	167	4384	1740	39.7	1196.47
Sendmail 8.14.3	108	820	430	52.4	405.39
Apache 2.2.8	102	2032	1066	52.5	557.48
Mailutils 1.2	94	2273	1533	67.4	753.91
OpenSSH 5.0	73	1329	594	44.7	217.81
Squid 3.1.4	26	419	281	67.1	107.85

5.3. Large Programs and Complete Open Source Projects

In Table II, we report case studies running ABDUCTOR on larger open source projects (e.g., a complete Linux Kernel distribution). The purpose of these examples is not to test precision: rather, they probe scalability and graceful imprecision. The case studies were run on a machine with a 3.33 GHz Intel Core i5 processor with 4 GB memory. The number of lines of C code (LOC) was measured by instrumenting gcc so that only code actually compiled was counted. The table reports for each test: the number of lines of code with unit one thousand (KLOC); the number of procedures analyzed (Num. Procs); the number of procedures with at least one consistent spec (Proven Procs); the percentage of procedures with at least one consistent spec (Procs Coverage %); and the execution time in seconds (Time) with one core.

Running with a timeout of one second for each procedure, we observed that only a very low percentage of procedures timed out. More often our analysis failed to find a nontrivial spec (a spec with a consistent precondition). The percentage of procedures analyzed is somewhat encouraging, and might be improved by using better base abstract domains or human intervention.

For the great majority of the procedures, relatively simple specifications were found, which did not involve linked list predicates. This is because a minority of procedures actually traverse data structures. (The analysis did many times find linked list structure, for example, in procedure `ap_find_linked_module` in Apache or in the Cyrus imapd's procedures `freeentryatts` and `freeattvalues` discussed in this section.) The point, for analysis, is that by combining abduction and frame inference we obtain specifications that (nearly) describe only the cells accessed by a procedure. This modularity means that linked lists do not have to be threaded throughout the specifications of all procedures.

There is no deep reason for our scalability, except perhaps that we attempt to discover small specs (hence reducing the number as well). We can easily employ a timeout strategy because of compositionality: if one procedure times out, we can still get results for others. Even more importantly, we can analyze each file independently of others: we do not have to load the entire source program into memory to analyze it, which would quickly overspill the RAM and cause the analysis to thrash. Beside the experiments reported in Table II, we ran the analysis of Linux in a eight core machine, trying

with both one core and eight cores and we observed a 4x speed-up. Compositionality gives a easy and natural way to exploit parallelism in multi-core machines.

Our compositional algorithm is parametrized by the base abstract domain, and so an instantiation of it will inherit any limitations of the base domain. For our experiments we used the domain introduced in Berdine et al. [2007]. Our symbolic heaps have pure and spatial part; non pointer variables involve the pure part, and for the pure part we have a simple domain with constants, equalities and inequalities; better domains could be plugged in. We remark also that this domain does not deal well with arrays and pointer arithmetic. These are treated as nondeterministic operations that are imprecise but sound for the goal of proving pointer safety.

5.3.1. Focussing on the Cyrus imapd Example. Currently, ABDUCTOR provides little support for interpreting the data resulting from the analysis. Given this current user support and the huge quantity of results we decided to look closely only at the data related to Cyrus imapd.[13] Here we briefly summarize the outcomes. As indicated above consistent specifications were found for 68.2% of the procedures. Among the discovered specifications, we observed that 18 procedures (i.e., 1% of the total and 1.5% of the successfully analyzed procedures) reported preconditions involving complex data structures (e.g., different kinds of nested and non-nested lists). This indicates that a minority of procedures actually traverse data structures.

Figure 5 reports (in a pictorial form) one of the three (heap) specifications discovered for the procedure `freeentryatts`. The precondition is given by the box on the top labeled by "PRE1". On the bottom there are two post-conditions labelled by "POST1" and "POST2", respectively. The intuitive meaning is that when running the procedure `freeentryatts` starting from a state satisfying PRE1, the procedure does not commit any pointer errors, and if it terminates it will reach a state satisfying either POST1 or POST2. A pre (or a post) displays a heap structure. A small rectangle with a label denotes an allocated cell, a dashed small rectangle stands for a possibly dangling pointer or nil. A long filled rectangle is a graphical representation of a higher-order list predicate $\mathsf{hls}\, k\, \phi\, e\, e$ introduced in Section 3.1.1. The parameter k denoting whether the list is surely not empty or not is written at the top of the box (lsPE/lsNE). The parameter ϕ describing the internal structure of the elements of the higher-order list is depicted by a large dashed box. Hence we can observe that the footprint of `freeentryatts` consists of a noncircular singly linked-list whose elements are `struct` with fields `attvalues`, `next`, and `entry`. The field `attvalues` points to a list of `struct` with fields `value`, `next`, and `attrib`. In other words, the footprint of `freeentryatts` consists of a nested noncircular singly linked-list.

Figure 6 shows one specification of the function `freeattvalues`. It deallocates the fields in the list pointed to by its formal parameter `l`. The procedure `freeentryatts` calls `freeattvalues(l->attvalues)` asking to free the elements of the inner list. Notice how the bottom-up analysis composes these specifications. In `freeentryatts`, the elements of the inner list pointed to by `attvalues` are deallocated by using (composing) the specification found for `freeattvalues` which acts on a smaller footprint.[14] The field `entry` is instead deallocated directly inside `freeentryatts`.

[13]We used `cyrus-imapd-2.3.13` downloaded from `http://cyrusimap.web.cmu.edu`.

[14]The reader may wonder why in the postcondition of `freeattvalues` we have a nonempty lists case whereas in the `freeentryatts` we have a case for possibly-empty lists. The reason can be explained as follows. Using the two postconditions of `freeattvalues` (one describing the case for nonempty lists and the other for the empty list) will result in two postconditions for `freeentryatts` in the case of lists. However, these two cases are then joined together [Yang et al. 2008] producing a single post-condition for possibly empty lists in the `freeentryatts`'s postconditions.

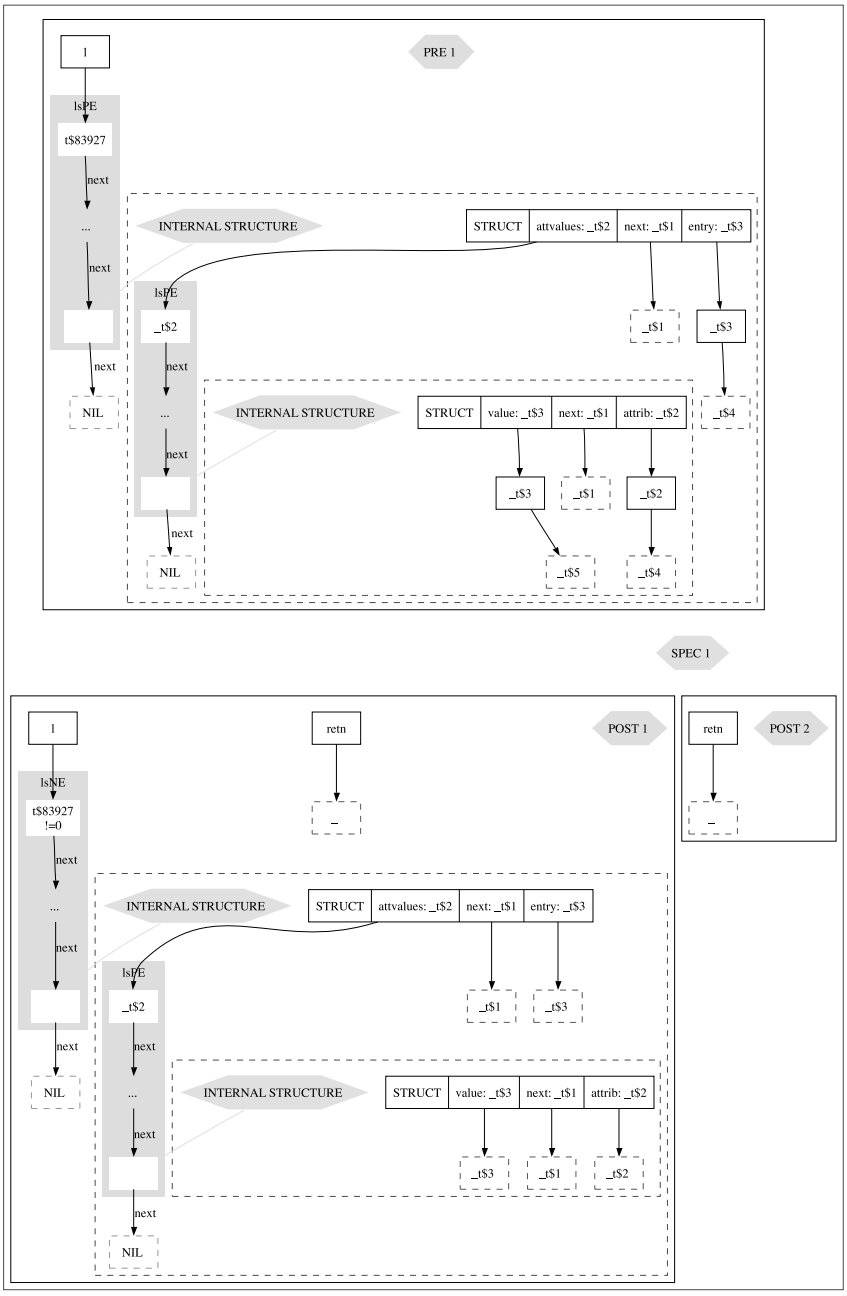


Fig. 5. A specification automatically synthesized by ABDUCTOR for the procedure `freeentryatts` of the Cyrus imapd example.

This relation between `freeentryatts` and `freeattvalues` illustrates, in microcosm, the modularizing effect of bi-abductive inference. The specification of `freeattvalues` does not need to mention the enclosing list from `freeentryatts`, because of the principle of local reasoning. In a similar way, if a procedure touches only two or three cells,

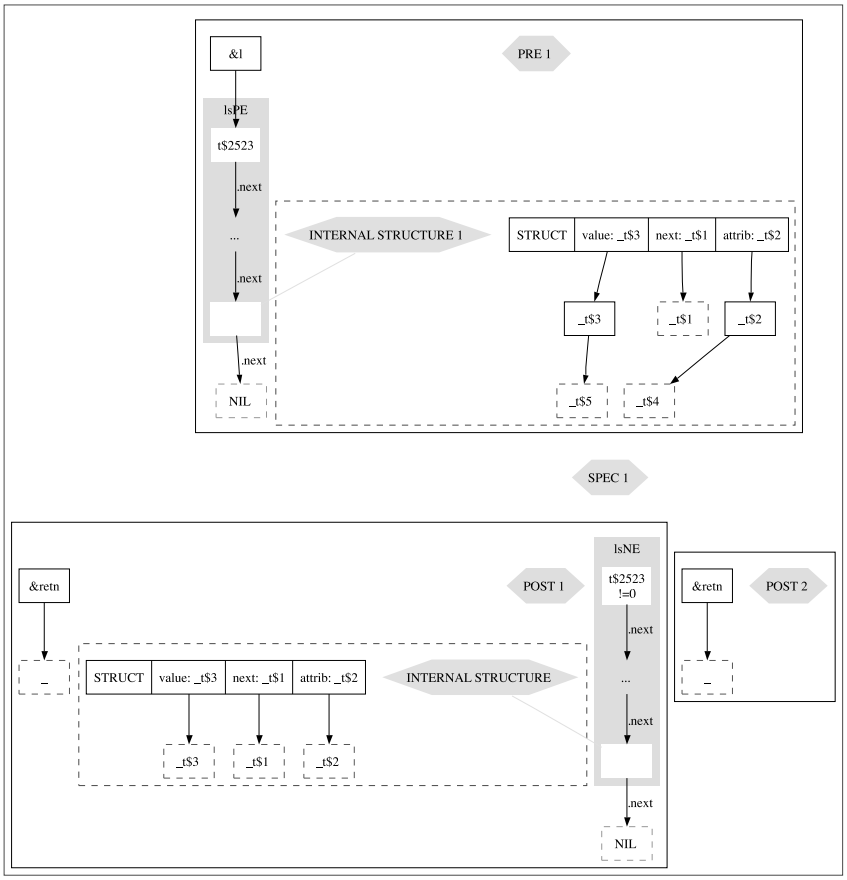


Fig. 6. A specification for the procedure `freeattvalues` called by `freeentryatts`.

there will be no need to add any predicates describing entire linked structures through its verification. In general, analysis of a procedure does not need to be concerned with tracking an explicit description of the entire global state of a system, which would be prohibitively expensive.

Only 4 procedures timed out (i.e., 0.4% of the total). Among the procedures for which the analysis was unable to synthesize specifications, 84 potential memory leaks were reported. A potential leak is reported when a spatial predicate in a symbolic heap is not provably reachable from program variables. A quick inspection of these possible errors revealed that 19 cases (22.6%) were clearly real leaks, whereas 26 cases (30.9%) were false bugs. For the remaining 39 cases (46.4%), it was not easy to establish whether or not they were genuine bugs. This would require a good knowledge of the source code and/or better user support in reporting possible errors, a feature that is currently lacking in ABDUCTOR. Nevertheless, given that ABDUCTOR was not designed as a bug catcher, but rather as a proof tool, we found the unveiling of several real bugs a pleasant surprising feature of our technology.

In this context, we add a final consideration. We emphasize that ABDUCTOR computes a genuine over-approximation (with respect to an idealized model) in the sense of abstract interpretation [Cousot and Cousot 1977]. Thus, in contrast to several unsound bug-catching tools that can detect some heap errors, when ABDUCTOR finds a

specification it has constructed a proof which shows that no pointer errors can occur (when the precondition is satisfied). For instance, from Figure 5, we can infer that `freeentryatts` does not leak memory, does not dereference a null/dangling pointer, and does not double-free memory.

5.3.2. Caveats. The work in this section applies a sound algorithm (INFERSPECS) to large code, but it does so in a way that sometimes goes beyond the assumptions of its model, or by using a model that does not completely match C's runtime in all its detail. A first limitation in this respect is that the analysis ignores concurrency.[15] Based on concurrent separation logic [O'Hearn 2007], one might expect that the results of a sequential analysis could in some circumstances be sound in a concurrent model [Berdine et al. 2005a], but we do not wish to make strong claims in this regard. Second we have treated unknown library procedures as nondeterministic assignments without side effects. In many cases, this is a sound interpretation, for the purposes of pointer safety. However, we simply did not have the time to manually inspect all of the C libraries to provide specs for them. (Of course, we would be interested in combining our work with techniques designed to deal with libraries in binary [Gopan and Reps 2007].) Third, our method does not deal well with code pointers, a longstanding open problem. We have treated code pointers as if they are unknown procedures. This, in effect, assumes that a code pointer is part of an "object" with a different footprint than the current procedure. One often sees this idiom, for example in the relation between the Linux kernel and device drivers. Our position also calls to mind the hypothetical frame rule from O'Hearn et al. [2009]. However, significant further work is needed in this direction. Fourth, the abstract domain used in our case studies does not treat arrays and other non-pointer values precisely. Our symbolic heaps have pure and spatial part: non pointer variables involve the pure part, and for the pure part we have a simple domain with constants, equalities and inequalities. Better domains for analyzing non-pointer values could be plugged in in place of the particular domain used in our case studies. A fifth caveat is that we apply INFERSPECS to large projects on a by-procedure basis by intercepting calls to gcc scheduled by a makefile for the projects. This has the result of treating some procedures as unknown when they needn't be, and between-file cycles in the call graph are not detected.

We make these remarks to be clear about limitations of our implementation. Some of these "problems" are fundamental (e.g., concurrency), and others are straightforward (e.g., makefile). But they are all independent of the scientific contributions of this paper, which concern bi-abduction and automation of program analyses: the case studies in this section were done to illustrate the automation, scalability and graceful imprecision, but the above applications on the large code bases have admitted limitations as regards accuracy. Of course, the caveats listed and others are ones that should be addressed in mature industrial-strength tools, going beyond a scientific prototype like reported here.

5.4. Discussion

At the beginning of the article we made the case for examining a compositional shape analysis. We do not claim that compositional analyses are fundamentally superior to noncompositional ones. They present several interesting properties, particularly as regards scalability, automation and the ability to deal with incomplete programs. However, they can have limitations as regards precision, as some of our case studies confirmed. It is reasonable to envisage that, ultimately, effective and accurate analysis

[15]However, initial results have shown that the compositional analysis may be used to obtain new methods in shape analysis for concurrency [Calcagno et al. 2009b].

of large software will be done by considering a mix of techniques. An analysis might work compositionally most of the time, choosing where to employ more precise and perhaps noncompositional methods. For instance, we might run an analysis bottom-up, interpreting callees before callers, but then employ a top-down narrowing phase afterwards to improve precision on certain needed parts of the code. One could also use a compositional analysis in concert with other techniques (e.g., interactive proof), to reduce the overall human effort.

6. RELATED WORK

As we mentioned in the Introduction, we are interested in accurate heap analyses that can be used to prove pointer safety. We use the term "analysis" to refer to methods that discover loop invariants and pre- and postconditions, and confine our attention to such verification methods in this section. Of course, our method might be used in concert with static verifiers that use user-supplied annotations.

The kind of shape analysis done here is one that attempts to be accurate in the presence of deep heap update, where a heap mutation is made some undetermined length down a linked structure. The first such analysis was presented in the fundamental [Sagiv et al. 1998], and there have been many subsequent works in search of ever better shape domains (e.g., Podelski and Wies [2005], Bouajjani et al. [2006], Lev-Ami et al. [2006], Magill et al. [2007], Manevich et al. [2007], Berdine et al. [2007], and Chang and Rival [2008]). Scalability is a significant problem for these precise analyses. Several papers on deep update have reported whole-program analyses with experimental results on programs in the thousands of lines of code [Guo et al. 2007; Marron et al. 2008; Yang et al. 2008]. Other analyses sacrifice precision in order to gain scalability [Ghiya and Hendren 1996; Hackett and Rugina 2005]; they cannot, for example, prove absence of pointer-safety faults in many circumstances.

We emphasize that the work in this article is not aimed at supplanting existing shape analyses, but in finding ways to boost them by making them compositional. In our formulation and our experiments, we naturally built on our own prior work on SPACEINVADER. But it is likely that the ideas could be used similarly with other forms of shape analysis, such as based on automata [Bouajjani et al. 2006], graphs [Marron et al. 2008], or 3-valued logic [Sagiv et al. 2002].

Thus, this article might be viewed as providing techniques whereby we take an abstract domain A for shape analysis, and turn it into a compositional analysis technique $C[A]$. The contribution is in the passage $A \mapsto C[A]$. But we are still left with the question of which domains A to choose from. We have chosen one domain in our experiments, but the question of what the good (precise and efficient and expressive) domains A are remains a significant problem, on which further research is needed.

While we showed one way to effect this passage $A \mapsto C[A]$, we do not claim that bi-abductive inference is the only possible way one might obtain a compositional shape analysis. Indeed, other approaches might occur more immediately to the reader: particularly, under-approximating backwards program analysis.

Defining a backwards shape analysis with acceptable precision and efficiency is, as far as we are aware, an open problem. Prior to our precursor paper [Calcagno et al. 2007a], we formulated and implemented a backwards shape analysis of our own. It created an enormous number of abstract states, and when it took several seconds to analyze a trivial list traversal program we abandoned the approach.

Subsequently, several groups described ways to obtain preconditions in shape analysis by going backwards [Abdulla et al. 2008; Lev-Ami et al. 2007; Podelski et al. 2008]. None is formulated in an interprocedural manner, and they report experimental results only for programs in the tens of lines of code. Further research is needed to develop or evaluate the ideas in these works.

Previous works in shape analysis have treated procedure calls in a local way, by passing only the reachable part of the abstract heap to a procedure [Gotsman et al. 2006; Marron et al. 2008; Rinetzky et al. 2005b]. The method here is more strongly local; by using the idea of finding/using "small specifications" that only mention the footprints of procedures, we are able to be more aggressive and sometimes pass fewer than the reachable cells.

The general issues regarding compositional program analysis are well known [Cousot and Cousot 2001], and for nonshape domains or bug-catching applications, a number of compositional analyses have appeared (e.g., Whaley and Rinard [2006], Nystrom et al. [2004], Dor et al. [2003], Gulwani and Tiwari [2007], Jung and Yi [2008], and Moy [2008]). They all use different techniques to those here.

Giacobazzi [1994] has previously used abductive inference in the analysis of logic programs. The problem he solves is dual to that here. Given a specification of a logic programming module and an implementation, the method of Giacobazzi [1994] infers constraints on undefined literals in the module: it is top-down synthesis of constraints on literals referred to in open code. In a procedural language the corresponding problem is to start with a Hoare triple for an "outer" procedure, whose body refers to another unknown procedure, and to infer a constraint on the unknown procedure. Here, we infer the spec for the "outer" procedure, relying on previously computed specs for procedures called in its body. It would be interesting to attempt to apply abduction in Giacobazzi's way to procedural code, to infer constraints from open code.

Gulwani et al. [2008] have used abduction to under-approximate logical operators such as conjunction and disjunction in their work on program analysis for abstract domains with quantification. Their application of abduction is completely different to that here.

The PREGEN algorithm can be considered as part of a wider trend, whereby the verification method is guided by using the information in failed proofs. In this, it is somewhat reminiscent of counterexample-guided abstraction refinement (or CEGAR, [Jhala and Majumdar 2009]). The difference with CEGAR is that we use abduction to find new information to put into a precondition, without changing the abstract domain, where CEGAR attempts to refine the abstraction used in a developing proof (or the abstract domain), without changing the precondition. Like in CEGAR, once a proof has failed there is no theoretical reason to think that an abduced precondition will be sound for the entire program: it works only for a path up to a given point. It is necessary to do more work, in an interactive loop or in re-execution as we have done here, to find a proof.

Other analogies with and comparisons to CEGAR can be made. For example, the discussion after Example 4.2 of the stop-first-and-re-execute strategy with -continue-along-path partially calls to mind some aspects of the algorithm in SLAM [Ball et al. 2006] versus other algorithms such as BLAST [Henzinger et al. 2002], which attempt to avoid repeating work; the analogy is not exact, however. Perhaps the larger issue, though, is not about detailed similarities and differences, but the possibility of finding a generalized method that might extend the reach of compositional analysis techniques to a broader range of verification problems.

Since the first version of this work was published in POPL'09, there have been further developments that build on the work there. The paper [Gulavani et al. 2009] introduces a bottom-up shape analysis which does not need to use a re-execution phase as we have. They achieve this using an abstract domain that does not need the canonicalization or abstraction phase that is usually present in shape analysis. This method has produced detailed specs of some programs approaching functional correctness, and provides interesting possibilities for future shape analyses (the main question being the extent to which the avoidance of abstraction step can be maintained). Another

paper [Luo et al. 2010] tackles the problem of inferring specs of unknown procedures, analogous to the problem in the previously mentioned work of Giacobazzi, and also defines an alternate algorithm for abduction, and an alternate order for comparing quality of solutions. Other papers study memory leaks [Distefano and Filipovic 2010] and concurrency analysis [Calcagno et al. 2009b].

7. CONCLUSIONS

The major new technical contributions of this article are the following.

(1) The definition of proof techniques and algorithms for abduction and bi-abduction for separated heap abstractions.
(2) A novel method for generating preconditions of heap manipulating programs using bi-abduction.
(3) A compositional algorithm for generating summaries of programs (procedure specs) using bi-abduction.
(4) The first analysis of its kind (a shape analysis) that can scale to large programs.

We did case studies ranging from small and medium-sized examples to test precision, to larger code bases, including Linux, OpenSSL and Apache. No previous shape analysis for deep update has approached code bases of comparable size. However, the precision level of our analysis is no greater than the abstract domain it is based on. So, our method should be viewed as a way of boosting existing shape analyses, rather than as competitors to them.

The analysis results we obtain on the large programs are partial, but this is another benefit of our method. The analysis is able to obtain nontrivial Hoare triples for collections of procedures, even when for other procedures it obtains imprecise results or takes too long. For the procedures we did not successfully analyze, we could in principle consider using other methods such as manually-supplied assertions to help the analysis along, interactive proofs, or even other abstract domains. In fact, for the eventual aim of proving nontrivial properties (e.g., memory safety) of entire large code bases, it seems likely that a mixture of techniques, with different degrees of automation, will be needed, and it is in easing their blending that compositional methods have much to offer.

We acknowledge that this article has left a number of theoretical stones unturned. For example, in introducing abduction for resource logics, we have given two particular algorithms, but these are by no means definitive. It would be interesting to understand complexity and completeness questions for general classes of assertion logics. Similarly, we have shown one way to obtain a compositional program analysis, but one can entertain many variants of the algorithm. Also, new questions (such as concerning abduction and adaptation of Hoare logic specs) are arising. In making these remarks we emphasize that soundness has not been compromised: when the analysis says "yes" it means that the inferred Hoare triple is true, and that a class of bugs is excluded, according to a well-defined model: unanswered technical questions concern completeness or precision or complexity, rather than soundness.

In a general sense, the most important contribution of this work is the explanation and demonstration of how an increase in automation and scalability is possible using abductive inference. In doing the work, we aimed to get to the demonstration point without pausing (for perhaps years) on some of the natural and nontrivial technical problems that arose, some of which we have just mentioned. But we hope that this article will trigger work on new questions and applications: there appear to be rich directions for further research and deeper understanding surrounding the use of

abduction in program analysis, as well as, we hope, increased potential for applications of automatic program verification technology.

ACKNOWLEDGMENTS

We would like to thank Jan Tobias Mühlberg for providing the benchmarks which initially sparked this work, Paul Kelly for giving us access to his group's 8-core machine, and Wei-Ngan Chin for useful comments.

REFERENCES

ABDULLA, P. A., BOUAJJANI, A., CEDERBERG, J., HAZIZA, F., AND REZINE, A. 2008. Monotonic abstraction for programs with dynamic memory heaps. In *Proceedings of the 20th International Conference on Computer Aided Verification, 20th International Conference (CAV)*. 341–354.

BALABAN, I., PNUELI, A., AND ZUCK, L. D. 2005. Shape analysis by predicate abstraction. In *Proceedings of the 6th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*. Lecture Notes in Computer Science, , vol. 3385. Springer, 164–180.

BALL, T., BOUNIMOVA, E., COOK, B., LEVIN, V., LICHTENBERG, J., MCGARVEY, C., ONDRUSEK, B., RAJAMANI, S. K., AND USTUNER, A. 2006. Thorough static analysis of device drivers. In *Proceedings of the EuroSys Conference*. 73–85.

BERDINE, J., CALCAGNO, C., AND O'HEARN, P. W. 2004. A decidable fragment of separation logic. In *Proceedings of the 24th International Conference on Foundations of Software Technology and Theoretical Computer Science*. Vol. 3328. 97–109.

BERDINE, J., CALCAGNO, C., COOK, B., DISTEFANO, D., O'HEARN, P., WIES, T., AND YANG, H. 2007. Shape analysis of composite data structures. In *Proceedings of the 19th International Conference on Computer Aided Verification (CAV)*. 178–192.

BERDINE, J., CALCAGNO, C., AND O'HEARN, P. W. 2005a. Smallfoot: Automatic modular assertion checking with separation logic. In *Proceedings of the 4th International Symposium on Formal Methods for Components and Objects*. Lecture Notes in Computer Science, vol. 4111. 115–137.

BERDINE, J., CALCAGNO, C., AND O'HEARN, P. W. 2005b. Symbolic execution with separation logic. In *Proceedings of the 3rd Asian Symposium on Programming Languages and Systems (APLAS)*. 52–68.

BIERHOFF, K. AND ALDRICH, J. 2005. Lightweight object specification with typestates. In *Proceedings of the 13th ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE)*. 217–226.

BOUAJJANI, A., HABERMEHL, P., MORO, P., AND VOJNAR, T. 2005. Verifying programs with dynamic 1-selector-linked structures in regular model checking. In *Proceedings of the 11th Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. 13–29.

BOUAJJANI, A., HABERMEHL, P., ROGALEWICZ, A., AND VOJNAR, T. 2006. Abstract tree regular model checking of complex dynamic data structures. In *Proceedings of the 13th International Symposium on Static Analysis (SAS)*. Lecture Notes in Computer Science, vol. 4134. 52–70.

CALCAGNO, C., DISTEFANO, D., O'HEARN, P. W., AND YANG, H. 2006. Beyond reachability: Shape abstraction in the presence of pointer arithmetic. In *Proceedings of the 13th International Symposium on Static Analysis (SAS)*. Lecture Notes in Computer Science, vol. 4134. 182–203.

CALCAGNO, C., DISTEFANO, D., O'HEARN, P., AND YANG, H. 2007a. Footprint analysis: A shape analysis that discovers preconditions. In *Proceedings of the 14th International Symposium on Static Analysis (SAS)*. Lecture Notes in Computer Science, vol. 4634. 402–418.

CALCAGNO, C., O'HEARN, P., AND YANG, H. 2007b. Local action and abstract separation logic. In *Proceedings of the 22nd IEEE Symposium on Logic in Computer Science (LICS)*. 366–378.

CALCAGNO, C., DISTEFANO, D., O'HEARN, P., AND YANG, H. 2009a. Compositional shape analysis by means of bi-abduction. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 289–300.

CALCAGNO, C., DISTEFANO, D., AND VAFEIADIS, V. 2009b. Bi-abductive resource invariant synthesis. In *Proceedings of the 7th Asian Symposium on Programming Languages and Systems (APLAS)*. 259–274.

CHANG, B. AND RIVAL, X. 2008. Relational inductive shape analysis. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 247–260.

CHANG, B., RIVAL, X., AND NECULA, G. 2007. Shape analysis with structural invariant checkers. In *Proceedings of the 14th International Symposium on Static Analysis (SAS)*. Lecture Notes in Computer Science, Springer. 384–401.

COOK, S. A. 1978. Soundness and completeness of an axiomatic system for program verification. *SIAM J. Comput. 7*, 70–90.

COUSOT, P. AND COUSOT, R. 1977. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM Symposium on Principles of Programming Languages (POPL)*. 238–252.

COUSOT, P. AND COUSOT, R. 2001. Compositional separate modular static analysis of programs by abstract interpretation. In *Proceedings of SSGRR*.

COUSOT, P. AND COUSOT, R. 2002. Modular static program analysis. In *Proceedings of the 11th International Conference on Compiler Construction (CC)*. Lecture Notes in Computer Science, vol. 2304. 159–178.

COUSOT, P., COUSOT, R., FERET, J., MAUBORGNE, L., MINÉ, A., MONNIAUX, D., AND RIVAL, X. 2005. The ASTRÉE analyzer. In *Proceedings of the 14th European Symposium on Programming (ESOP)*. Lecture Notes in Computer Science, vol. 3444. 21–30.

CREIGNOU, N. AND ZANUTTINI, B. 2006. A complete classification of the complexity of propositional abduction. *SIAM J. Comput. 36*, 1, 207–229.

DELINE, R. AND FÄHNDRICH, M. 2004. Typestates for objects. In *Proceedings of the 18th European Conference on Object-Oriented Programming (ECOOP)*. 465–490.

DISTEFANO, D. AND FILIPOVIC, I. 2010. Memory leaks detection in java by bi-abductive inference. In *Proceedings of the 13th International Conference on Fundamental Approaches to Software Engineering (FASE)*. Lecture Notes in Computer Science, vol. 6013. 278–292.

DISTEFANO, D. AND PARKINSON, M. 2008. jStar: Towards Practical Verification for Java. In *Proceedings of the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. 213–226.

DISTEFANO, D., O'HEARN, P., AND YANG, H. 2006. A local shape analysis based on separation logic. In *Proceedings of the 12th Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. 287–302.

DOR, N., RODEH, M., AND SAGIV, M. 2003. CSSV: Towards a realistic tool for statically detecting all buffer overflows in C. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI)*. 155–167.

EITER, T. AND GOTTLOB, G. 1995. The complexity of logic-based abduction. *J. ACM 42*, 1, 3–42.

EITER, T. AND MAKINO, K. 2002. On computing all abductive explanations. In *Proceedings of the 18th National Conference on Artificial Intelligence and 14th Conference on Innovative Applications of Artificial Intelligence (AAAI/IAAI)*. 62–67.

GHIYA, R. AND HENDREN, L. 1996. Is it a tree, a DAG, or a cyclic graph? A shape analysis for heap-directed pointers in C. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 1–15.

GIACOBAZZI, R. 1994. Abductive analysis of modular logic programs. In *Proceedings of the International Logic Programming Symposium*. The MIT Press, 377–392.

GOPAN, D. AND REPS, T. 2007. Low-level library analysis and summarization. In *Proceedings of the 9th International Conference on Computer Aided Verification (CAV)*. 68–81.

GOTSMAN, A., BERDINE, J., AND COOK, B. 2006. Interprocedural shape analysis with separated heap abstractions. In *Proceedings of the 13th International Symposium on Static Analysis (SAS)*. Lecture Notes in Computer Science, vol. 4134. 240–260.

GULAVANI, B. S., CHAKRABORTY, S., RAMALINGAM, G., AND NORI, A. V. 2009. Bottom-up shape analysis. In *Proceedings of the 16th International Symposium on Static Analysis (SAS)*. 188–204.

GULWANI, S. AND TIWARI, A. 2007. Computing procedure summaries for interprocedural analysis. In *Proceedings of the 16th European Symposium on Programming (ESOP)*. Lecture Notes in Computer Science, vol. 4421. Springer, 253–267.

GULWANI, S., MCCLOSKEY, B., AND TIWARI, A. 2008. Lifting abstract interpreters to quantified logical domains. In *Proceedings of the 35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 235–246.

GUO, B., VACHHARAJANI, N., AND AUGUST, D. 2007. Shape analysis with inductive recursion synthesis. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM.

HACKETT, B. AND RUGINA, R. 2005. Region-based shape analysis with tracked locations. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languagesc (POPL)*. ACM, 310–323.

HENZINGER, T., JHALA, R., MAJUMDAR, R., AND SUTRE, G. 2002. Lazy abstraction. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, 58–70.

HOARE, C. A. R. 1971. Procedures and parameters: An axiomatic approach. In *Proceedings of the Symposium on the Semantics of Algebraic Languages*. Lecture Notes in Math, vol. 188. 102–116.

ISHTIAQ, S. AND O'HEARN, P. W. 2001. BI as an assertion language for mutable data structures. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, 14–26.

JHALA, R. AND MAJUMDAR, R. 2009. Software model checking. *ACM Comput. Surv. 41*, 4.

JUNG, Y. AND YI, K. 2008. Practical memory leak detector based on parameterized procedural summaries. In *Proceedings of the 7th International Symposium on Memory Management*. 131–140.

KAKAS, A. C., KOWALSKI, R. A., AND TONI, F. 1992. Abductive logic programming. *J. Logic Comput. 2*, 6, 719–770.

KLEYMANN, T. 1999. Hoare logic and auxiliary variables. *Form. Asp. Comput. 11*, 5, 541–566.

LEV-AMI, T., IMMERMAN, N., AND SAGIV, M. 2006. Abstraction for shape analysis with fast and precise transfomers. In *Proceedings of the 18th International Conference on Computer Aided Verification (CAV)*.

LEV-AMI, T., SAGIV, M., REPS, T., AND GULWANI, S. 2007. Backward analysis for inferring quantified preconditions. Tech rep. TR-2007-12-01, Tel-Aviv University.

LUO, C., CRACIUN, F., QIN, S., HE, G., AND CHIN, W.-N. 2010. Verifying pointer safety for programs with unknown calls. *J. Symb. Comput. 45*, 11, 1163–1183.

MAGILL, S., BERDINE, J., CLARKE, E., AND COOK, B. 2007. Arithmetic strengthening for shape analysis. In *Proceedings of the 14th International Symposium on Static Analysis (SAS)*. 419–436.

MANEVICH, R., YAHAV, E., RAMALINGAM, G., AND SAGIV, M. 2005. Predicate abstraction and canonical abstraction for singly-linked lists. In *Proceedings of the 6th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCA)*. Lecture Notes in Computer Science, vol. 3385. 181–198.

MANEVICH, R., BERDINE, J., COOK, B., RAMALINGAM, G., AND SAGIV, M. 2007. Shape analysis by graph decomposition. In *Proceedings of the 13th Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Lecture Notes in Computer Science, vol. 4424. Springer, 3–18.

MARRON, M., HERMENEGILDO, M., KAPUR, D., AND STEFANOVIC, D. 2008. Efficient context-sensitive shape analysis with graph based heap models. In *Proceedings of the 17th International Conference on Compiler Construction (CC)*. Lecture Notes in Computer Science, vol 4959. 245–259.

MOY, Y. 2008. Sufficient preconditions for modular assertion checking. In *Proceedings of the 9th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*. 188–202.

NAUMANN, D. A. 2001. Calculating sharp adaptation rules. *Inf. Process. Lett. 77*, 2-4, 201–208.

NGUYEN, H. H. AND CHIN, W.-N. 2008. Enhancing program verification with lemmas. In *Proceedings of the 20th International Conference on Computer Aided Verification (CAV)*.

NYSTROM, E., KIM, H., AND HWU, W. 2004. Bottom-up and top-down context-sensitive summary-based pointer analysis. In *Proceedings of the 11th International Symposium on Static Analysis (SAS)*. 165–180.

O'HEARN, P. W. 2007. Resources, concurrency and local reasoning. *Theoret. Comput. Sci. 75*, 1-3, 271–307.

O'HEARN, P. W., REYNOLDS, J. C., AND YANG, H. 2001. Local reasoning about programs that alter data structures. In *Proceedings of the 15th International Workshop on Computer Science Logic (CSL)*. 1–19.

O'HEARN, P. W., YANG, H., AND REYNOLDS, J. C. 2009. Separation and information hiding. *ACM Trans. Program. Lang. Syst. 31*, 3.

PAUL, G. 1993. Approaches to abductive reasoning: an overview. *Artif. Intell. Rev. 7*, 2, 109–152.

PEIRCE, C. 1958. *Collected Papers of Charles Sanders Peirce*. Harvard University Press.

PODELSKI, A. AND WIES, T. 2005. Boolean heaps. In *Proceedings of the 12th International Symposium on Static Analysis (SAS)*. Lecture Notes in Computer Science, vol. 3672. Springer, 268–283.

PODELSKI, A., RYBALCHENKO, A., AND WIES, T. 2008. Heap assumptions on demand. In *Proceedings of the 18th International Conference on Computer Aided Verification (CAV)*.

PYM, D., O'HEARN, P., AND YANG, H. 2004. Possible worlds and resources: the semantics of BI. *Theoret. Comput. Sci. 315*, 1, 257–305.

REPS, T., HORWITZ, S., AND SAGIV, M. 1995. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*.

REYNOLDS, J. C. 2002. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science (LICS)*. 55–74.

RINETZKY, N., BAUER, J., REPS, T., SAGIV, M., AND WILHELM, R. 2005a. A semantics for procedure local heaps and its abstractions. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*.

RINETZKY, N., SAGIV, M., AND YAHAV, E. 2005b. Interprocedural shape analysis for cutpoint-free programs. In *Proceedings of the 12th International Symposium on Static Analysis (SAS)*. Lecture Notes in Computer Science, vol. 3672. 284–302.

SAGIV, M., REPS, T., AND WILHELM, R. 1998. Solving shape-analysis problems in languages with destructive updating. *ACM Trans. Prog. Lang. Syst. 20*, 1, 1–50.

SAGIV, M., REPS, T., AND WILHELM, R. 2002. Parametric shape analysis via 3-valued logic. *ACM Trans. Prog. Lang. Syst. 24*, 3, 217–298.

WHALEY, J. AND RINARD, M. 2006. Compositional pointer and escape analysis for Java. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*. 187–206.

YANG, H. 2001. Local reasoning for stateful programs. Ph.D. thesis, University of Illinois, Urbana-Champaign.

YANG, H., LEE, O., BERDINE, J., CALCAGNO, C., COOK, B., DISTEFANO, D., AND O'HEARN, P. W. 2008. Scalable shape analysis for systems code. In *Proceedings of the 20th International Conference on Computer Aided Verification (CAV)*.

Received August 2010; revised May 2011; accepted May 2011