Compositional Shape Analysis

Cristiano Calcagno
Imperial College, London
ccris@doc.ic.ac.uk

Dino Distefano
Queen Mary, University of London
ddino@dcs.qmul.ac.uk

Peter O'Hearn
Queen Mary, University of London
ohearn@dcs.qmul.ac.uk

Hongseok Yang
Queen Mary, University of London
hyang@dcs.qmul.ac.uk

Abstract

This paper describes a compositional shape analysis, where each procedure is analyzed independently of its callers. The analysis uses an abstract domain based on 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. Compositionality brings its usual benefits – increased potential to scale, ability to deal with unknown calling contexts, 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 interprocedural analysis algorithm. We have implemented our analysis algorithm and we report case studies on smaller programs to evaluate the quality of discovered specifications, and larger programs (e.g., an entire Linux distribution) to test scalability and graceful imprecision.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; D.3.4 [*Programming Languages*]: Processors; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

General Terms Verification, Reliability, Languages, Theory

Keywords Program Analysis, Proof Theory, Abduction

1. Introduction

The Case for Compositional Shape Analysis. A shape analysis attempts to discover invariants that describe the data structures in a program (e.g., [41, 2, 35, 19, 3]). We are interested in particular in the use of such analyses for (lightweight) program verification. A shape analysis can in principle prove that programs do not commit pointer-safety errors (dereferencing a null or dangling pointer, 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.

Recall that a semantic definition of a language is compositional if the meaning of a composite expression can be defined in terms of the meanings of its parts. Similarly, 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. Often, this definition is understood at the level of granularity of procedures or groups of recursive procedures. This paper is the, to our knowledge, first to define and evaluate a compositional shape analysis.

Compositionality has well-known potential benefits, which line up directly with some of the outstanding problems in making shape analysis more practical.

1. *The ability to analyze program parts, without having the context.* When an entire program is not available, or is very large, the user of a whole-program shape analysis must put in a significant amount of work before the analysis is run at all. This work consists in defining a "fake main program" which allocates data structures and calls the incomplete program's procedures, or it involves defining preconditions for the procedures. Both are time consuming and possible sources of error.

   The very definition of "compositional" presupposes that an analysis makes sense for an incomplete program, without having its context (or a "main" program) available.

   This is evidently relevant to the potential use of an analysis during program development, rather than only after a complete program has been written.

2. *Potential to scale.* Shape analyses are notoriously expensive. After years of research and many papers, we have only recently seen the accurate analysis of complete programs in the thousands of lines of code (up to 10K LOC) [19, 21, 28]. (Less accurate analyses, which typically cannot prove pointer safety on such programs, have been reported for larger code bases; e.g., [13, 20].) Papers in the field usually use test programs numbering in only the hundreds or tens of lines of code.

   With a compositional analysis it becomes relatively easy to get meaningful (if partial) results for large code bases.

   There is a further benefit that is worth mentioning: *Parallelization.* If we can analyze groups of procedures independently, it is easy to run the analysis in parallel and exploit the power of the multicore computers that are becoming mainstream.

   [*Aside*. 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.]

3. *Graceful imprecision.* There is no single existing shape domain that has 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) meaning-

ful results cease to be obtained even for portions of code which could be well treated, were a suitable precondition known.

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.

This discussion motivates the problem of defining and evaluating a compositional shape analysis.

Compositional Shape Analysis by Abductive Inference. Charles Peirce introduced *abductive inference* – inference of explanatory hypotheses – in the early 1900's in his writings on the scientific process [34]. Abduction was formulated distinguishing hypothesis formation from deductive and inductive 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 article [22] points to many formalizations and applications in the AI literature.)

The main contribution of this paper is conceptual in nature. We explain how abductive inference can be used to define a compositional, interprocedural shape analysis algorithm. We use abduction to generate preconditions, so that we can obtain a true Hoare triple for a procedure without knowing its calling context. This allows us to make a compositional, bottom-up program analysis, where callees are analyzed before callers.

The basic idea is this. During an analysis run we might find that we do not have enough information to perform an operation – a procedure call, or a dereferencing. We then perform abductive inference to infer what is missing. We percolate this information back to the preconditions of procedures and, in the end, this allows us to synthesize Hoare triples without examining the calling context of a procedure.

A benefit of this use of abduction is to generalize and enhance a method we described in [6] for generating preconditions in an intraprocedural (i.e., without procedures) analysis. The use of abduction makes possible to handle procedure calls. It is also more systematic: the method of finding preconditions is seen as part of a more general scheme – inference of missing hypotheses – rather than an *ad hoc* method based on pointer dereferencing faults (as was the case in [6]).

A major part of our contribution involves defining an abductive inference algorithm for use with a shape analysis. We define a new procedure for performing abductive inference for (certain) separation logic assertions which, following [4, 10], are used to define the abstract states in our analysis. In fact, to treat procedure calls we must deal with a more general problem, which we call *bi-abduction*, that infers "frames" describing extra, unneeded portions of state as well as the needed, missing portions (the "anti-frames").

We have implemented our analysis algorithm and done a number of case studies to evaluate it. These range from small programs operating over composite list structures, through to a medium-sized program (a firewire device driver), and on to larger code bases (including complete distributions of Apache, OpenSSL, Linux). The small and medium-sized examples are done to probe questions concerning the quality of specifications discovered by our analysis, and the larger ones to test scalability and graceful imprecision.

2. Bi-Abductive Inference for Specification Synthesis

In this section we explain informally how bi-abduction is used in our analysis. Subsequent sections develop the ideas formally.

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 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.

In this paper we will be solving a similar problem, but for separation logic rather than classical logic. So, we will have to solve problems of the form

$$A * ?? \vdash G$$

where we use the separating conjunction to partition the premises instead of, or in addition to, the usual additive conjunction of classical logic.

Generating Preconditions by Abduction. Suppose that during a 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: $A \stackrel{def}{=} x \mapsto 0$

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 0 (so a list of length 1).

The difficulty we face 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. Formally, it involves solving a question of the form

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

We can see the relevance of this to interprocedural analysis using an example. Suppose we are given a procedure summary (here, a spec using Hoare triples) of a procedure `merge(x,y)`. The summary might have been computed previously in an analysis, or it might have been supplied manually by a user. We have an enclosing procedure `p(y)` which calls `merge` as follows.

```
1  void p(lst_nd *y) { //Inferred Pre: list(y)
2    lst_nd *x;
3    x=malloc(sizeof(lst_nd)); x->tail =  0;
4    merge(x,y);
5    y=0;
6  } // Inferred Post: list(x)
7  void merge(lst_nd *x,lst_nd *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 heap. Just after line 3 we obtain the assertion $A = x \mapsto 0$. We call our abductive proof procedure, which tells us that $\mathsf{list}(y)$ is missing. 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 `merge(x,y)`. At the end of procedure `p()` we obtain $\mathsf{list}(x)$ as the computed post.

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 `merge()`.

Bi-Abduction. Abduction gives us a way to synthesize missing portions of state. We also have to synthesize additional, leftover portions of the heap by solving the more general problem

$$A * ?1 \;\vdash\; G * ?2.$$

Here, we refer to $?2$ as the frame, and $?1$ as the anti-frame.

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

```
1  void q(lst_nd *y) { //Inferred Pre: list(y)
2    lst_nd *x, *z;
3    x=malloc(sizeof(lst_nd)); x->tail=0;
4    z=malloc(sizeof(lst_nd)); z->tail=0;
5    // Abducted: list(y), Framed: z|->0
6    merge(x,y);
7    // Obtained Post: list(x)*z|->0
8    merge(x,z);
9  } // Inferred Post: list(x)
```

This time we infer anti-frame $\mathsf{list}(y)$ as before, but using bi-abduction we also infer $z{\mapsto}0$ as the frame axiom that won't be needed by procedure call at line 6. That is, we obtain a solution of the bi-abduction question

$$x{\mapsto}0 \,*\, z{\mapsto}0 \,*\, ?1 \;\vdash\; \mathsf{list}(x) \,*\, \mathsf{list}(y) \,*\, ?2$$

where the solution obtained is $?1 = \mathsf{list}(y)$, $?2 = z{\mapsto}0$. We tack the frame $?2$ 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 9.

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 a way to realize, in a program analysis, a key idea from [32], where specifications and proofs concentrate on the cells accessed by a program rather than the entire global state of a system. Synthesis of frames allows us to *use* small specifications of heap portions by slotting them into larger states found at call sites, where abduction of anti-frames helps us to *find* the small specs.[1]

3. Symbolic Heaps

Storage Model. 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 $a, b, c, \ldots$). Let Loc be a countably infinite set of locations, and let Val be a set of values that includes Loc. The storage model is as follows:

$$\mathsf{Heap} \overset{def}{=} \mathsf{Loc} \rightharpoonup_{\mathsf{fin}} \mathsf{Val} \qquad \mathsf{Stack} \overset{def}{=} (\mathsf{Var} \cup \mathsf{LVar}) \to \mathsf{Val}$$
$$\mathsf{State} \overset{def}{=} \mathsf{Stack} \times \mathsf{Heap}$$

Symbolic Heaps. *Symbolic heaps* are special separation logic formulae [4, 10], interpreted over State, and defined as follows:

E	$::=$	$x \mid a \mid t(E, \ldots, E)$	*Expressions*
Π	$::=$	$E{=}E \mid E{\neq}E \mid \mathsf{true} \mid \Pi \wedge \Pi$	*Pure formulae*
B	$::=$	$\ldots$	*Basic spatial predicates*
Σ	$::=$	$B \mid \mathsf{true} \mid \mathsf{emp} \mid \Sigma * \Sigma$	*Spatial formulae*
Δ	$::=$	$\Pi \wedge \Sigma$	*Quantifier-free symb. heaps*
H	$::=$	$\exists \vec{a}.\, \Delta$	*Symbolic heaps*

[1] Previous work has shown the benefit of inferring frames [39, 16, 28], but not anti-frames, and the resulting procedure summaries therefore describe larger portions of state than necessary.

Expressions are program or logical variables x, a. Or they are heap-independent terms $t(E_1, \ldots, E_n)$ (e.g., 0). Pure formulae are composed by the conjunction of equalities and disequalities between expressions, and describe properties of variables. The spatial formulae specify properties of the heap. 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 $\exists \vec{a}.\, \Pi \wedge \Sigma$, 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.

B is a collection of basic spatial predicates. One instantiation is

$$B \;\; ::= \;\; E{\mapsto}E \mid \mathsf{lseg}(E, E)$$

Here, the *points-to* predicate $x{\mapsto}y$ denotes a heap with a single allocated cell at address x with content y, and $\mathsf{lseg}(x, y)$ denotes a list segment from x to y (not included). For simplicity, we describe our algorithms and results mainly using this instantiation, although they work equally well for other more sophisticated instantiations [3, 8] after slight or no modifications; when some modifications are necessary, we will explain what they are.

In this paper, 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 $H_i = \exists \vec{a_i}.\, \Delta_i$ 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:

$$\Delta_1{*}\Delta_2 \overset{def}{=} (\Pi_1{\wedge}\Pi_2){\wedge}(\Sigma_1{*}\Sigma_2), \quad H_1{*}H_2 \overset{def}{=} \exists \vec{a_1}\vec{a_2}.\, \Delta_1{*}\Delta_2.$$

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

$$\begin{array}{rclrcl} \Delta_i * \Sigma & \overset{def}{=} & \Pi_i \wedge (\Sigma * \Sigma_i), & H_i * \Sigma & \overset{def}{=} & \exists \vec{a_i}.\, \Delta_i * \Sigma, \\ \Pi \wedge \Delta_i & \overset{def}{=} & (\Pi \wedge \Pi_i) \wedge \Sigma_i, & \Pi \wedge H_i & \overset{def}{=} & \exists \vec{a_i}.\, \Pi \wedge \Delta_i. \end{array}$$

4. Abduction for Separated Heap Abstractions

In this section we describe an algorithm for inferring answers to the abduction question:

Given Δ and H, find a symbolic heap M such that

$$\Delta * M \;\;\vdash\;\; H. \qquad (1)$$

The question, as put, can be answered trivially, by returning a false assertion. In this paper, we propose a criterion for judging the quality of solutions of (1).

$$\begin{array}{rl} M \precsim M' \overset{def}{\iff} & (M' \vdash M * \mathsf{true} \;\wedge\; M \nvdash M' * \mathsf{true}) \;\vee \\ & (M' \vdash M * \mathsf{true} \;\wedge\; M \vdash M' * \mathsf{true} \;\wedge\; M' \vdash M). \end{array}$$

Ideally, we would find solutions that are minimal w.r.t. $\precsim$ and consistent. As a semantic question about sets of concrete heaps (viewing an assertion as a set) the best solution always exists.[2] While this solution exists theoretically, it is not easy to compute and can lead to additional expense in the analysis (see Example 3 below). So, we present a more pragmatically-motivated procedure for abductive inference. The order $\precsim$ is used to inform design choices in the algorithm, rather than to identify the theoretically best solution. This is just how in abstract interpretation analyses often do not implement the "best abstract transformer" for pragmatic reasons.

[2] Semantically, the best solution is

$$\mathsf{min}(\{\; (s, h) \;\mid\; \Delta * \{(s, h)\} \;\vdash\; H \;\}),$$

where $\mathsf{min}(M)$ is the predicate defined below:

$$\{\; (s, h) \in M \;\mid\; \text{for all subheaps } h' \text{ of } h, \text{ if } \{(s, h')\} \vdash M, \text{ then } h' = h \;\}.$$

$$\frac{\Delta * [M] \;\triangleright\; H \qquad \Delta \;\vdash\; \exists\vec{a}.\,\Pi \qquad \mathsf{emp} \;\vdash\; \exists\vec{a}.\,\Sigma}{\Delta * [M] \;\triangleright\; H * (\exists\vec{a}.\,\Pi \wedge \Sigma)}\ \textsf{remove}$$

$$\frac{(E_0{=}E_1 \wedge \Delta) * [M] \;\triangleright\; \exists\vec{b}.\,\Delta'}{\Delta * E{\mapsto}E_0 * [\exists\vec{a}.\,E_0{=}E_1 \wedge M] \;\triangleright\; \exists\vec{a}\vec{b}.\,\Delta' * E{\mapsto}E_1}\ {\mapsto}\textsf{-match}$$
$$(\text{where } \vec{b} \cap \mathsf{FreeLVar}(E_1) = \emptyset)$$

$$\frac{\Delta * [M] \;\triangleright\; \exists\vec{a}.\,\Delta' * \mathsf{lseg}(E_0, E_1)}{\Delta * B(E, E_0) * [M] \;\triangleright\; \exists\vec{a}.\,\Delta' * \mathsf{lseg}(E, E_1)}\ \textsf{lseg-right}$$
$$(\text{where } B(E, E_0) \text{ is } E{\mapsto}E_0 \text{ or } \mathsf{lseg}(E, E_0))$$

$$\frac{(E{\neq}E_0 \wedge \Delta * \mathsf{lseg}(a, E_0)) * [M] \;\triangleright\; \exists\vec{b}.\,\Delta'}{\Delta * \mathsf{lseg}(E, E_0) * [E{\neq}E_0 \wedge M] \;\triangleright\; \exists a\vec{b}.\,\Delta' * E{\mapsto}a}\ \textsf{lseg-left}$$

$$\frac{}{(\Pi \wedge \mathsf{emp}) * [\exists\vec{a}.\,\Pi' \wedge \mathsf{emp}] \;\triangleright\; \exists\vec{a}.\,\Pi' \wedge \mathsf{emp}}\ \textsf{base-emp}$$

$$\frac{}{\Delta * [\exists\vec{a}.\,\Pi \wedge \mathsf{emp}] \;\triangleright\; \exists\vec{a}.\,\Pi \wedge \mathsf{true}}\ \textsf{base-true}$$

$$\frac{\Delta * [M] \;\triangleright\; \Delta' \qquad \Delta * \exists\vec{a}.\,B(E, E') \not\vdash \mathsf{false}}{\Delta * [M * \exists\vec{a}.\,B(E, E')] \;\triangleright\; \Delta' * \exists\vec{a}.\,B(E, E')}\ \textsf{missing}$$
$$(\text{where } B(E, E') \text{ is } E{\mapsto}E' \text{ or } \mathsf{lseg}(E, E'))$$

Figure 1. Proof Rules for Abductive Inference

Proof System for Abductive Inference. We introduce a proof system for deriving judgments of the form

$$\Delta * [M] \;\triangleright\; H,$$

where M, H are symbolic heaps and Δ is a quantifier-free symbolic heap. This judgment means that M is the missing anti-frame of the abduction question $\Delta * ?? \;\vdash\; H$, and each proof rule describes how to search for the solution of this question. The proof rules are presented in Figure 1. In the figure, we assume that all the bound logical variables are different from one another and also from free logical variables.

EXAMPLE 1. This example shows how our inference finds a solution of the question

$$x{\mapsto}y \,*\, [??] \;\triangleright\; x{\mapsto}a \,*\, \mathsf{lseg}(a, 0) \,*\, \mathsf{true}.$$

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

$$\frac{\dfrac{\dfrac{}{(y{=}a \wedge \mathsf{emp}) * [\mathsf{emp}] \;\triangleright\; \mathsf{true}}\ \textsf{base-true}}{(y{=}a \wedge \mathsf{emp}) * [\mathsf{lseg}(a, 0)] \;\triangleright\; \mathsf{lseg}(a, 0) * \mathsf{true}}\ \textsf{missing}}{x{\mapsto}y * [y{=}a \wedge \mathsf{lseg}(a, 0)] \;\triangleright\; x{\mapsto}a * \mathsf{lseg}(a, 0) * \mathsf{true}}\ {\mapsto}\textsf{-match}$$

The last step of the derivation shows that ${\mapsto}$-match is used to find the instantiation of a (i.e., $y = a$), and strengthens the assumption with this instantiation $y{=}a$. The other two steps move the remaining $\mathsf{lseg}(a, 0)$ predicate in the conclusion to the missing anti-frame part.

This example shows how our proof method goes beyond previous works on theorem proving with separation logic. Indeed, it is possible to obtain a sound theorem prover for abduction by a simple modification of a usual theorem prover for separation logic [4]. An abduction question $\Delta * ?? \;\vdash\; H$ is solved by attempting to show the entailment $\Delta \;\vdash\; H$ and, when this proof attempt fails with one undischarged assumption $\mathsf{emp} \;\vdash\; M$, we conclude that the missing heap is M. This prover is not powerful enough to find out the instantiation of logical variables, as we have done here. For instance, in our example, it would fail to find $y{=}a$, and infer $x{\mapsto}a * \mathsf{lseg}(a, 0)$ as a missing anti-frame. This means that the symbolic heap $x{\mapsto}y * x{\mapsto}a * \mathsf{lseg}(a, 0)$ is a new enhanced assumption by abduction, although it is an inconsistent formula. Certainly, this is not a desirable solution. □

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

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

The derivation below shows how our inference algorithm finds a solution of this abduction question:

$$\frac{\dfrac{\dfrac{\dfrac{}{\mathsf{emp} * [\mathsf{emp}] \;\triangleright\; \mathsf{true}}\ \textsf{base-true}}{\mathsf{emp} * [\mathsf{lseg}(y, 0)] \;\triangleright\; \mathsf{lseg}(y, 0) * \mathsf{true}}\ \textsf{missing}}{\mathsf{emp} * [\mathsf{lseg}(y, 0)] \;\triangleright\; \mathsf{lseg}(z, z) * \mathsf{lseg}(y, 0) * \mathsf{true}}\ \textsf{remove}}{x{\mapsto}z * [\mathsf{lseg}(y, 0)] \;\triangleright\; \mathsf{lseg}(x, z) * \mathsf{lseg}(y, 0) * \mathsf{true}}\ \textsf{lseg-right}$$

The last step subtracts $x{\mapsto}z$ from $\mathsf{lseg}(x, z)$, and the second last step removes the result $\mathsf{lseg}(z, z)$ of this subtraction, because $\mathsf{emp} \;\vdash\; \mathsf{lseg}(z, z)$. The remaining steps move the predicate $\mathsf{lseg}(y, 0)$ from the conclusion to the anti-frame. In this derivation, the application of the remove rule is crucial to obtain a better solution. Without using the rule, we could apply missing twice, once for $\mathsf{lseg}(z, z)$ and the next for $\mathsf{lseg}(y, 0)$, and this would give us

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

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

EXAMPLE 3. Consider the abduction question

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

Our algorithm finds as a solution $y \mapsto 3$. It is not the best solution: a better solution is the disjunction $y \mapsto 3 \vee (y{=}x \wedge \mathsf{emp})$.

Although we can see how to compute a best solution in this case, we do not do so for two pragmatic reasons. First, if we were to aim for the best then we would end up 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 [21]. Second, we have found in our experiments that the loss of precision caused by this choice does not hurt us too much. See, in particular, the results on the device drivers in Section 7. This is a trade-off, which could be revisited. □

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

$$\frac{\Delta' * [M'] \;\triangleright\; H' \qquad \text{Cond}}{\Delta * [M] \;\triangleright\; H}$$

Here Cond is a condition involving parts of Δ and H. The algorithmic reading of this rule is the following. In order to answer the entailment question $\Delta * ?? \vdash H$, the side condition Cond is first checked, and if it holds, we make a recursive call to answer the smaller question $\Delta' * ?? \vdash H'$. 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:

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

The inference engine then cancels out those facts, and continues the search for the solution with the reduced right-hand side $\exists\vec{b}.\,\Delta'$ 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 $\vec{a}$, which gives the result of the original search.

Our abduction algorithm tries to apply the rules in Figure 1 in the order in which they appear in the figure. It firsts attempts to use remove and eliminate 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, lseg-right and lseg-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 considered desirable according to the definition of $\precsim$. By trying the remove rule before missing, the inference prefers choosing the empty heap to moving a predicate from the conclusion to the missing anti-frame part. For instance, given $\mathsf{emp} * [??] \vdash \mathsf{lseg}(x,x)$, the remove rule infers emp as the missing anti-frame whereas the missing rule returns $\mathsf{lseg}(x,x)$. 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, lseg-right and lseg-left) before the moving-predicate rules ensures that the common parts between the assumption and the conclusion are cancelled out as much as possible, before trying to move predicates from the conclusion to the missing anti-frame part.

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 [38, 3, 8] 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, lseg-left, lseg-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 paper corresponds to a specific instantiation:

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

The missing rule is generalized in the following way:

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

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

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

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

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

Each of these axioms generates proof rules that replace $\mapsto$-match, lseg-left and lseg-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:

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

For each axiom of the second form

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

we include the following proof rule for abduction:

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

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

THEOREM 4. *If all the assumed axioms are sound, the proof system for abduction is sound. That is, if $\Delta * [M] \rhd H$ is derivable, then $\Delta * M$ semantically implies H.*

5. Bi-Abduction

We now consider the more general bi-abduction question

$$\Delta\ *\ ?1\ \vdash\ H\ *\ ?2.$$

for quantifier-free symbolic heaps Δ and normal symbolic heaps H.[3] 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 [4, 30, 11]. 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. In Algorithm 1 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 1 relies on an ordinary theorem prover for symbolic heaps.

We can combine these two procedures to obtain the algorithm BiAbd described in Algorithm 2. By convention, the algorithm raises exception fail if either of its internal procedure calls does.

THEOREM 5. $\mathsf{BiAbd}(\Delta,H) = (M,F) \implies \Delta * M \vdash H * F.$

Comparing Solutions. The soundness property in Theorem 5 can be satisfied trivially. We now define an order $\sqsubseteq$ on potential

[3] We consider the bi-abduction between Δ and H here, instead of normal symbolic heaps, because it is the question asked by our analysis. However, it is not technically difficult to extend our algorithm for the bi-abduction between normal symbolic heaps.

Algorithm 1 Finding a Missing Heap Portion

```
Abduce(Δ, H)  ≝
  1. Find a symbolic heap $M$ such that
         $\Delta * [M] \ \triangleright\ H$
     using the abduction algorithm from Section 4. If no such heap
     can be found, return fail.
  2. If $\Delta * M$ is (provably) inconsistent, return fail. Otherwise,
     return $M$.
```

Algorithm 2 Synthesizing Missing and Leftover Heaps, Jointly

```
BiAbd(Δ, H)  ≝
    $M := \mathsf{Abduce}(\Delta, H * \mathsf{true})$;    $L := \mathsf{Frame}(\Delta * M, H)$;
    return$(M, L)$
```

solutions which was used to design our algorithm.

$$M \precsim M' \overset{def}{\iff} (M' \vdash M * \mathsf{true} \ \wedge\ M \nvdash M' * \mathsf{true}) \ \vee$$
$$(M' \vdash M * \mathsf{true} \ \wedge\ M \vdash M' * \mathsf{true} \ \wedge\ M' \vdash M)$$
$$M \approx M' \overset{def}{\iff} M' \precsim M \ \wedge\ M' \succsim M$$

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

The definition of $\sqsubseteq$ is a lexicographic ordering of the order $M \precsim M'$ defined for abduction, and ordinary implication for leftover heaps. The bias on the anti-frame part is due to our application: the BiAbd algorithm is mainly used to infer preconditions of procedures. 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 the 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 missing 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 6. We illustrate the intuition behind $\sqsubseteq$ using

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

Consider the following three solutions of the question:

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

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., $M \not\precsim M'$) 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 $(M, L) \sqsubseteq (M'', L'')$. □

6. Compositional Interprocedural Shape Analysis

We illustrate our interprocedural shape analysis using a simple while language extended with procedure calls and parametrized by a collection of basic operations. Later we will instantiate the basic operations to be certain heap-manipulating commands.

$$\begin{array}{lll} e & ::= x \mid t(e_1, \ldots, e_n) & \textit{Prog. Expressions} \\ b & ::= \cdots & \textit{Booleans} \\ A & ::= \cdots & \textit{Atomic Commands} \\ c & ::= A \mid x{:=}f(\vec{e}) \mid c_1; c_2 \mid \textbf{if}\ b\ c_1\ c_2 & \textit{Commands} \\ & \mid\ \ \textbf{while}\ b\ c & \\ p & ::= \cdot \mid f(\vec{x})\{\textbf{local}\ \vec{y};\ c;\ \textbf{return}\ e\};\ p & \textit{Programs} \end{array}$$

A program p consists of a number of procedure definitions. For simplicity, we only consider procedures that return a single value, and that do not access any global variables.

The aim of our compositional shape analysis is to construct a spec table $\mathcal{T}(f)$ (or procedure summary [42]) for every procedure f in Proc. Spec tables are partial functions from symbolic heaps to sets of symbolic heaps:

$$\mathcal{T}(f) : \mathsf{Spec}, \quad \text{where } \mathsf{Spec} \overset{def}{=} \mathsf{SH} \rightharpoonup \mathcal{P}(\mathsf{SH}).$$

The domain of $\mathcal{T}(f)$ specifies the preconditions we consider for procedure f. For each P in the domain, the intended reading of $\mathcal{T}(f)(P) = \{Q_1, \ldots, Q_k\}$ is that

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

is a true Hoare triple. We will often write $\{P\}f(\vec{x})\{\mathcal{Q}\} \in \mathcal{T}$ to mean that $\mathcal{T}(f)(P) = \mathcal{Q}$, and we define the set of spec tables as

$$\mathsf{AllSpecs} \overset{def}{=} \mathsf{Proc} \rightarrow \mathsf{Spec}.$$

The analysis uses an abstract semantics

$$[\![c]\!]^{\mathsf{IP}} \ :\ \mathsf{AllSpecs} \rightarrow \mathcal{P}(\mathsf{SH}{\times}\mathsf{SH} \cup \{\top\}) \rightarrow \mathcal{P}(\mathsf{SH}{\times}\mathsf{SH} \cup \{\top\})$$

of commands that works on pairs (F, H) of the precondition part F and the current heap H. The transfer function for an individual statement can alter the H component, and add to the F component, giving $(F * M, H')$, meaning that the statement can reach H' if the missing part M is added to the start state.

We presume that we are given transfer functions $[\![A]\!]^{\mathsf{IP}}$ for the atomic commands. Later, we will spell out one such collection. The pivotal part of our development is the semantics $[\![x = f(\vec{e})]\!]^{\mathsf{IP}}$ of procedure call, which we describe first.

6.1 Abstract Semantics of Procedure Call

Besides frames and missing anti-frames, our semantics must take into account that, as in standard Hoare logic, all the logical variables (elements of LVar) in a Hoare triple are implicitly universally quantified. For example, in

$$\{x{\mapsto}a' * y{\mapsto}b'\}\mathtt{swap}(x, y)\{\mathsf{ret}{=}0 \wedge x{\mapsto}b' * y{\mapsto}a'\}.$$

a' and b' are logical variables, and the spec means that `swap` exchanges the contents of cells x and y.

We will be using abduction to instantiate logical variables, as well as to find missing heap portions. To see how this works, for the abduction question

$$x \mapsto e \ *\ [??] \ \vdash\ \ x \mapsto a$$

our algorithm finds $e{=}a \wedge \mathsf{emp}$ as the solution, where $e{=}a$ is the part telling us how to instantiate the logical variable.

Figure 2 shows the details of the abstract semantics of the procedure call $y{:=}f(\vec{e})$. The abstract execution

$$[\![y{:=}f(\vec{e})]\!]^{\mathsf{IP}}_{\mathcal{T}'}(F, H)$$

goes through every spec $\{P\}f(\vec{x})\{\mathcal{Q}\}$ of f in $\mathcal{T}'(f)$, and calls the BiAbd algorithm to check whether this spec can be used to update (F, H) appropriately. If BiAbd returns fail, the analysis ignores this spec, and moves on to the next. Otherwise, it massages the anti-frame and finds instantiations of parameters by a call to

```
$\mathcal{R} := \emptyset$;    Let $\exists\vec{a}.\Delta_H$ be $H$;
for all $\{P\}f(\vec{x})\{\mathcal{Q}\} \in \mathcal{T}'(f)$ do
  if $\mathsf{BiAbd}(\Delta_H, P) = (M, L) \neq \mathsf{fail}$,  and
      $\mathsf{Rename}(\Delta_H, M, P, \mathcal{Q}) = (\vec{e'}, \vec{b}, M_0) \neq \mathsf{fail}$
   then
     $\mathcal{R} := \mathcal{R} \cup \{(F * M_0,\ \exists\vec{a}.\,(Q * L)[y\vec{e'}/\mathsf{ret}\vec{b}]) \mid Q \in \mathcal{Q}\}$
  end if
end for;
return $\mathcal{R}$
```

Figure 2. Abstract Semantics of $[\![y{:=}f(\vec{e})]\!]^{\mathsf{IP}}_{\mathcal{T}'}(F, H)$

```
Let $\vec{b}$ be $\mathsf{FreeLVar}(P, \mathcal{Q})$;
Pick $\vec{e'}$ disjoint from $\vec{b}$ such that $\Delta_H * M \vdash \vec{e'}{=}\vec{b}$,
  but if cannot pick such $\vec{e'}$, return fail;
Pick $M_0$ disjoint from $\vec{b}$, $\vec{a}$ and program variables such that
        $\Delta_H * M_0 \vdash \Delta_H * M[\vec{e'}/\vec{b}]$,
  but if cannot pick such $M_0$, return fail;
return $(\vec{e'}, \vec{b}, M_0)$
```

Figure 3. Subroutine $\mathsf{Rename}(\Delta_H, M, P, \mathcal{Q})$

the Rename subroutine from Figure 3. 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 H. 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 $\vec{b}$ tell us how to instantiate logical variables in the specification, as discussed in the swap example at the beginning of this subsection. Although 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 7. We give a full description of the abstract semantics

$$[\![v := \mathtt{swap}(x, y)]\!]^{\mathsf{IP}}_{\mathcal{T}'}(F, H)$$

using the specification given earlier in this section. The semantics first invokes $\mathsf{BiAbd}(\ y{=}b \wedge x{\mapsto}y * z{\mapsto}0,\quad x{\mapsto}a' * y{\mapsto}b'\ )$ and infers M and L:

$$M \overset{def}{=} (a'{=}y \wedge b'{=}d \wedge y{\mapsto}d), \qquad L \overset{def}{=} z{\mapsto}0.$$

From this output, the analysis computes the three missing elements for analyzing the call $v{:=}\mathtt{swap}(x, y)$, which are a leftover frame $z{\mapsto}0$, a missing anti-frame $b{\mapsto}d$, and the instantiation $a'{=}y \wedge b'{=}d$ of logical variables a', b' in the spec of swap. Rename performs the computation of the anti-frame and the instantiation. These three elements form the result of analyzing the call. The current precondition F is updated by $*$-conjoining the missing anti-frame $b{\mapsto}d$:

$$F * b{\mapsto}d \quad\iff\quad x{=}a \wedge y{=}b \wedge a{\mapsto}c * b{\mapsto}d.$$

The current heap H is mutated according to the instantiated spec of swap with $a'{=}y \wedge b'{=}d$ and the leftover frame $z{\mapsto}0$, and becomes

$$v{=}0 \wedge x{\mapsto}d * y{\mapsto}y * z{\mapsto}0.$$

Thus, the result of $[\![v := \mathtt{swap}(x, y)]\!]^{\mathsf{IP}}_{\mathcal{T}'}(F, H)$ is the singleton set

$$\{\ (x{=}a \wedge y{=}b \wedge a{\mapsto}c * b{\mapsto}d,\quad v{=}0 \wedge x{\mapsto}d * y{\mapsto}y * z{\mapsto}0)\ \}.\square$$

Our algorithm is different in an important respect compared to typical forwards interprocedural analysis algorithms. Usually, if a call is found to match with one of the specifications in a procedure summary, 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 (through abduction): for this purpose, it makes sense to try as many of the specifications in a procedure summary as possible.

EXAMPLE 8. To illustrate the use of multiple specs consider the following example program:

```
void safe_reset_wrapper(int *y) {
  // Inferred Pre1: y=0 && emp
  // Inferred Pre2: y!=0 && y|->-
  safe_access(y);
}  // Inferred Post1: y=0 && emp
   // Inferred Post2: y!=0 && y|->0
void safe_reset(int *y) {// SUMMARY ONLY
 // Given Pre1: y=0 && emp
 // Given Pre2: y|->-
} // Given Post1: y=0 && emp
  // Given Post2: 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 $y{\neq}0 \wedge y{\mapsto}0$. If the analysis had searched for only one applicable spec, it would not have been able to find the second precondition.$\square$

6.2 Top-level Algorithm

We write Proc for the set of procedure names defined in a program, and assume that Proc is partitioned into $\mathsf{Proc}_0, \ldots, \mathsf{Proc}_n$ that satisfy the following calling relationship: if f is in Proc_i and f calls g, procedure g belongs to Proc_j for some $j \leq i$.

The analysis starts by building spec tables $\mathcal{T}_0$ for procedures in Proc_0. It infers preconditions for the procedures in Proc_0 using an interprocedural precondition discovery, and then computes postconditions for the inferred preconditions by the forward interprocedural shape analysis. Next, using the constructed $\mathcal{T}_0$, the analysis considers procedures in the next level Proc_1, and builds tables $\mathcal{T}_1$ for them, again by the interprocedural precondition discovery and the forward interprocedural shape analysis. This process is repeated for $\mathsf{Proc}_2, \ldots, \mathsf{Proc}_n$, and gives spec tables for all the procedures.

The top-level analysis algorithm is given in Algorithm 3, which follows the informal description in the previous paragraph. Note that the algorithm calls two subroutines:

$$\mathsf{AllSpecs} \overset{def}{=} \mathsf{Proc} \to \mathsf{Spec} \qquad \mathsf{AllPres} \overset{def}{=} \mathsf{Proc} \to \mathcal{P}(\mathsf{SH})$$
$$\begin{aligned} \mathsf{InferPre} &: \mathcal{P}(\mathsf{Proc}) \times \mathsf{AllSpecs} \to \mathsf{AllPres} \\ \mathsf{InferSpec} &: \mathcal{P}(\mathsf{Proc}) \times \mathsf{AllPres} \times \mathsf{AllSpecs} \to \mathsf{AllSpecs} \end{aligned}$$

The first subroutine $\mathsf{InferPre}(\mathsf{Proc}_k, \mathcal{T})$ is an interprocedural precondition discovery phase. Using the given spec tables $\mathcal{T}$ of called procedures, it discovers candidate preconditions for procedures in Proc_k. The second $\mathsf{InferSpec}(\mathsf{Proc}_k, \mathcal{P}, \mathcal{T})$ is our interprocedural forward shape analysis, and it constructs specs for procedures in Proc_k with respect to preconditions in $\mathcal{P}$ and adds them to $\mathcal{T}$.

The subroutine $\mathsf{InferPre}(\mathsf{Proc}_k, \mathcal{T}_{\mathsf{in}})$ discovers candidate preconditions for every f in Proc_k, while using spec tables $\mathcal{T}_{\mathsf{in}}$ to handle procedure calls in the body of f. Concretely, this discov-

Algorithm 3 Top-level Analysis Algorithm.

```
$\mathsf{AllSpecs} \stackrel{def}{=} \mathsf{Proc} \to \mathsf{Spec}$    $\mathsf{AllPres} \stackrel{def}{=} \mathsf{Proc} \to \mathcal{P}(\mathsf{SH})$

local $\mathcal{T}$:AllSpecs,  $\mathcal{P}$:AllPres,  $k$:Nats;
$\mathcal{T} := \lambda f.\lambda P.$undefined;  $\mathcal{P} := \lambda f.\emptyset$;  $k := 0$;
while $k \le n$ do
  $\mathcal{P} := \mathsf{InferPre}(\mathsf{Proc}_k, \mathcal{T})$;
  $\mathcal{T} := \mathsf{InferSpec}(\mathsf{Proc}_k, \mathcal{P}, \mathcal{T})$;
  $k := k{+}1$
end while;
return $\mathcal{T}$
```

Algorithm 4 $\mathsf{InferPre}(\mathsf{Proc}_k, \mathcal{T}_{\mathsf{in}})$.

```
local $\mathcal{T}$: AllSpecs,  $\mathcal{R} \subseteq \mathsf{SH}{\times}\mathsf{SH} \cup \{\top\}$;
$\mathcal{T} := \mathcal{T}_{\mathsf{in}}$;  $\mathcal{R} := \emptyset$;
repeat
  for all $f \in \mathsf{Proc}_k$ do
    Let $f(\vec{x})\{\mathbf{local}\ \vec{y}; c; \mathbf{return}\ e\}$ be the definition of $f$;
    Pick fresh $\vec{a}$ with $|\vec{a}|{=}|\vec{x}|$;
    $\mathcal{R} := [\![c;\ \mathsf{ret}{:=}e]\!]^{\mathsf{IP}}_{\mathcal{T}}(\{(\vec{x}{=}\vec{a} \wedge \mathsf{emp},\ \vec{x}{=}\vec{a} \wedge \mathsf{emp})\})$;
    if $\top \in \mathcal{R}$ then
      Reports the possibility of local memory errors at $f$
    end if;
    for all $(F, H) \in \mathcal{R}$ do
      if ($\mathcal{T}(f)(F)$ is undefined) then
        $\mathcal{T}(f)(F) := \{\exists \vec{b}.H[\vec{b}/\vec{x}\vec{y}] \mid$ fresh $\vec{b}$ s.t $|\vec{b}|{=}|\vec{x}\vec{y}|\}$;
      else
        $\mathcal{T}(f)(F) := \mathcal{T}(f)(F) \cup$
                    $\{\exists \vec{b}.H[\vec{b}/\vec{x}\vec{y}] \mid$ fresh $\vec{b}$ s.t $|\vec{b}|{=}|\vec{x}\vec{y}|\}$;
      end if
    end for
  end for
until $\mathcal{T}$ does not change;
return $\lambda f.$ if $(f \in \mathsf{Proc}_k)$ then $\mathsf{dom}(\mathcal{T}(f))$ else $\emptyset$
```

ery is done by a fixpoint computation, which constructs (possibly unsound) spec tables $\mathcal{T}$ for procedures in Proc_k. $\mathcal{T}$ includes all the spec tables in $\mathcal{T}_{\mathsf{in}}$, but those tables of $\mathcal{T}$ do not change during the fixpoint computation. For each procedure f in Proc_k, the subroutine abstractly runs the body of f with respect to the initial (F, H), and uses the result of this abstract run to update the spec table $\mathcal{T}(f)$ for f. This updating is repeated until $\mathcal{T}$ does not change, in which case the preconditions in $\mathcal{T}(f)$ are returned. The overall structure of the subroutine is described in Algorithm 4, where the abstract run of commands is specified using the abstract semantics $[\![-]\!]^{\mathsf{IP}}$. Note that the algorithm uses two sets of logical variables for each procedure f: $\vec{a}$ for caching the initial values of parameters $\vec{x}$, and $\vec{b}$ for existentially quantifying the parameters and local variables in the computed postconditions H for f.

We remark that our implementation also uses optimizations from the RHS interprocedural analysis algorithm [37], which avoid a certain amount of re-computation. We have, for simplicity, described a less efficient algorithm in the paper.

6.3 InferSpec Phase

After having run our interprocedural precondition-discovery algorithm we go through one more phase to find the final specifications for the procedures. This is a form of re-execution, which calculates postconditions from the given candidate preconditions, and also filters out unsafe preconditions. As in [6], filtering is needed because we are using abstraction to simplify preconditions, which is a potentially unsound step were we not to re-execute.

Algorithm 5 $\mathsf{InferSpec}(\mathsf{Proc}_k, \mathcal{P}_{\mathsf{in}}, \mathcal{T}_{\mathsf{in}})$.

```
local $\mathcal{T}$: AllSpecs,  $\mathcal{Q}$: $\mathcal{P}(\mathsf{SH} \cup \{\top\})$;
$\mathcal{T} := \lambda f.\lambda P.$ if $(f \in \mathsf{Proc}_k \wedge P \in \mathcal{P}_{\mathsf{in}}(f))$ then $\emptyset$ else $\mathcal{T}_{\mathsf{in}}(f)(P)$;
repeat
  for all $f \in \mathsf{Proc}_k$ and $F \in \mathsf{dom}(\mathcal{P}_{\mathsf{in}}(f))$ do
    Let $f(\vec{x})\{\mathbf{local}\ \vec{y}; c; \mathbf{return}\ e\}$ be the definition of $f$;
    Pick fresh $\vec{a}$ with $|\vec{a}|{=}|\vec{x}\vec{y}|$;
    $\mathcal{Q} := [\![c;\ \mathsf{ret}{:=}e]\!]^{\mathsf{I}}_{\mathcal{T}}(\{F\})$;
    if $((\top \in \mathcal{Q})\ \vee\ (\mathcal{T}(f)(F)$ is undefined$))$ then
      $\mathcal{T}(f)(F) :=$ undefined
    else
      $\mathcal{T}(f)(F) := \mathcal{T}(f)(F) \cup \{\exists \vec{a}.H[\vec{a}/\vec{x}\vec{y}] \mid H \in \mathcal{Q}\}$
    end if
  end for
until $\mathcal{T}$ does not change;
return $\mathcal{T}$
```

```
$\mathcal{R} := \emptyset$;   Let $\exists \vec{a}.\Delta_H$ be $H$;
for all $\{P\}f(\vec{x})\{\mathcal{Q}\} \in \mathcal{T}'(f)$ do
  $\vec{b} := \mathsf{FreeLVar}(P, \mathcal{Q})$;   Let $f(\vec{x})\{...\}$ be the definition of $f$;
  $R := \mathsf{BiAbd}(\Delta_H, P)$;
  if  $R = (\Pi \wedge \mathsf{emp},\ L) \neq \mathsf{fail}$ for some $\Pi, L$,  and
      $\Delta_H \vdash \Pi \Rightarrow \vec{e'}{=}\vec{b}$ for some $\vec{e'}$ disjoint from $\vec{b}$
   then
    $\mathcal{R} := \mathcal{R} \cup \{\ (\exists \vec{b}.\Pi,\ \{\exists \vec{a}.\,(Q * L)[y\vec{e'}/\mathsf{ret}\vec{b}] \ \mid\ Q \in \mathcal{Q}\})\ \}$
  end if
end for;
if (there is $\mathcal{R}_0 \subseteq \mathcal{R}$ s.t. $\vdash \bigvee\{\Pi' \mid (\Pi', \mathcal{Q}') \in \mathcal{R}_0\}$) then
  Pick a minimal such $\mathcal{R}_0$;
  return $\bigcup\{\mathcal{Q}' \mid (\Pi', \mathcal{Q}') \in \mathcal{R}_0\}$
else
  return $\{\top\}$
end if
```

Figure 4. Abstract Semantics of $[\![y{:=}f(\vec{e})]\!]^{\mathsf{I}}_{\mathcal{T}'}(H)$

The re-execution InferSpec takes three parameters. The first parameter is Proc_k, the names of the procedures whose summaries should be calculated, and the second is $\mathcal{P}_{\mathsf{in}}$ that records the preconditions for each procedure in Proc_k. The final parameter is $\mathcal{T}_{\mathsf{in}}$ that contains the already-computed procedure summaries (i.e., the summaries of those in Proc_j with $j < k$). Given these parameters, InferSpec constructs the procedure summaries $\mathcal{T}$ for Proc_k by a fixpoint computation. (Technically, $\mathcal{T}$ also contains summaries of procedures in Proc_j with $j < k$, but this part of $\mathcal{T}$ does not change during the fixpoint computation.) It goes through every procedure $f \in \mathsf{Proc}_k$ and every candidate precondition $P \in \mathcal{P}_{\mathsf{in}}(f)$, and calls a forward interprocedural analysis

$$[\![-]\!]^{\mathsf{I}} \quad : \quad \mathsf{AllSpecs} \to \mathcal{P}(\mathsf{SH} \cup \{\top\}) \to \mathcal{P}(\mathsf{SH} \cup \{\top\})$$

for the body of f. Then, the forward analysis gives the postcondition $\mathcal{Q}$ with

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

which is used to update $\mathcal{T}$. The fixpoint computation continues until $\mathcal{T}$ does not change. Algorithm 5 gives the details of InferSpec.

As in the previous section, the most interesting case of our abstract forward semantics $[\![-]\!]^{\mathsf{I}}$ is the procedure call; see Figure 4. (The other cases are identical to those of the standard intraprocedural analysis.) Suppose that the forward analysis is analyzing procedure call $y{:=}f(e)$ for a symbolic heap H under spec tables

$\mathcal{T}'$. For simplicity, we assume that neither H nor any preconditions of f in $\mathcal{T}'(f)$ have existential quantifications. The abstract run of $y{:=}f(e)$ searches for a collection of specs of f stored in $\mathcal{T}'(f)$ that can be used to transform the current symbolic heap H. Concretely, the analyzer first strips away the existential quantifications of H, and obtains $\exists\vec{a}.\,\Delta_H$. Then, it goes through every spec $\{P\}f(x)\{\mathcal{Q}\}$ in $\mathcal{T}'(f)$, and tries to match the precondition with the assertion at the call site. At this point we do something slightly unusual. Experimentally we have often found an assertion H that does not imply any of the preconditions in the summary, even after frame inference. However, if we do case analysis (excluded middle) $(b{\wedge}H){\vee}(\neg b{\wedge}H)$, then we can find that one or both of the disjuncts provably match preconditions. So, in this phase, we do case-split on demand, in an effort to boost the precision of our analysis. Technically, this is done, again, with the help of bi-abduction. In detail, the algorithm in Figure 4 checks whether the free logical variables $\vec{b}$ in the spec $\{P\}f(x)\{\mathcal{Q}\}$ can be instantiated by $\vec{e'}$ such that

$$\Pi' \wedge \Delta_H \;\vdash\; (P * L)[e\vec{e'}/x\vec{b}] \quad \text{for some } \Pi', L.$$

Here Π' is the condition used to split cases of H, L is a leftover frame, i.e., the part of H that is missing from the precondition P, and $\vec{e'}$ is the instantiation of implicitly universally quantified logical variables $\vec{b}$ in the spec. The checking is done by first invoking bi-abduction and finding Π and L such that

$$\mathsf{BiAbd}(\Delta_H, P) \;=\; (\Pi \wedge \mathsf{emp}, L)$$

and then determining whether, for some $\vec{e'}$ disjoint from $\vec{b}$,

$$\Delta_H \;\vdash\; \Pi \Rightarrow \vec{e'}{=}\vec{b}.$$

If those two steps succeed, the analyzer records the pair of a case-split condition and postcondition; otherwise it moves on to the next spec in $\mathcal{T}'(f)$. Finally, when all the specs have been examined, the analyzer selects a minimal collection of the recorded pairs

$$(\Pi'_1, \mathcal{Q}_1), \ldots, (\Pi'_n, \mathcal{Q}_n)$$

such that the Π'_j together cover all the cases of H, which is to say that $\Pi'_1 \vee \ldots \vee \Pi'_n$ is a tautology. The analysis result is the disjunction of $\mathcal{Q}_i$'s. (This case-splitting can be justified by the disjunction rule in Hoare logic.)

We point out two aspects of $[\![y{:=}f(e)]\!]^{\mathsf{I}}_{\mathcal{T}'}$. First, when the abstract semantics finds sufficiently many specs of f that are applicable (i.e., the disjunction of their preconditions is implied by the current symbolic heap), it ignores other specs of f. This contrasts with $[\![y{:=}f(e)]\!]^{\mathsf{IP}}_{\mathcal{T}'}$, which considers *all* specs of f. Second, if the analysis does not find any applicable spec, it does not attempt to add a new spec to f, unlike common global interprocedural analyses.

This phase also needs a semantics $[\![A]\!]^{\mathsf{I}}$ for the (so far unspecified) atomic commands. The basic fact is the following

THEOREM 9. *If the semantics $[\![A]\!]^{\mathsf{I}}$ of each atomic command is sound, then* InferSpec *returns true Hoare triples only.*

We could state the result more formally, by identifying a concrete semantics, a concretization function, and so on. But, there is no deep reason for why this result holds. It is just that, if we are given sound $[\![A]\!]^{\mathsf{I}}$ (wrt a given concrete semantics) then our treatment of procedures with $[\![-]\!]^{\mathsf{I}}$ is sound as well, and this semantics is then used by InferSpec to weed out any preconditions from the previous phase that do not guarantee safe execution.

6.4 Underlying Shape Analysis

We now describe the missing definitions of the abstract semantics, which have been postponed in previous sections. Our description assumes one particular instantiation of our programming language:

$$\begin{array}{rcll} b & ::= & e{=}e \mid e{\neq}e & \textit{Booleans} \\ A & ::= & a[e] \mid a & \textit{Atomic Commands} \\ a[e] & ::= & [e]{:=}e \mid \mathbf{free}(e) \mid x{:=}[e] & \\ a & ::= & x{:=}e \mid x{:=}\mathbf{new}(e) & \end{array}$$

In this instantiation, we have two classes of atomic commands. $a[e]$ attempts to dereference cell e, updating it ($[e_1]{:=}e_2$), disposing it ($\mathbf{free}(e)$), or reading its content ($x{:=}[e]$). The other atomic commands, denoted a, do not access existing cells.

The forward abstract semantics $[\![-]\!]^{\mathsf{I}}$ of commands is defined in three steps (as suggested in [41]), and implemented by functions:

$$\mathsf{rearr}(e) : \mathsf{SH} \to \mathcal{P}(\mathsf{SH} \cup \{\top\}),$$
$$\mathsf{exec}(A) : \mathsf{SH} \to \mathcal{P}(\mathsf{SH} \cup \{\top\}), \qquad \mathsf{abs} : \mathsf{SH} \to \mathsf{SH}.$$

The first step is *rearrangement*,[4] and it is implemented by the function $\mathsf{rearr}(e)$, which takes an abstract state and attempts to "concretise" the cell e that is accessed by the command in execution. If e is already explicit then this step is simply the identity function. The second step is *execution*, and it is defined by the function $\mathsf{exec}(A)$. This function symbolically executes the atomic command A in the rearranged heap. Finally, the third step is *abstraction* (or canonicalization), implemented by abs, which simplifies symbolic heaps and allows the convergence of the fixpoint computation.

For a function $t : D \to \mathcal{P}(D \cup \{\top\})$, let $t^\dagger$ be a function on $\mathcal{P}(D \cup \{\top\})$ defined by

$$t^\dagger(X) \;\overset{\textit{def}}{=}\; \{\top \mid \top \in X\} \,\cup\, (\bigcup\{t(x) \mid x \in (X \cap D)\}).$$

The forward abstract transfer functions of atomic commands are:[5]

$$\begin{array}{rcl} [\![a[e]]\!]^{\mathsf{I}}_{\mathcal{T}}(X) & \overset{\textit{def}}{=} & (\mathsf{abs}^\dagger \circ \mathsf{exec}(a[e])^\dagger \circ \mathsf{rearr}(e)^\dagger)(X) \\ [\![a]\!]^{\mathsf{I}}_{\mathcal{T}}(X) & \overset{\textit{def}}{=} & (\mathsf{abs}^\dagger \circ \mathsf{exec}(a)^\dagger)(X). \end{array}$$

The reader is referred to [10] for a detailed treatment of transfer functions defined in terms of rearr, exec and abs. The abstract semantics of the compound commands is standard:

$$\begin{array}{l} [\![c_1; c_2]\!]^{\mathsf{I}}_{\mathcal{T}}(X) \overset{\textit{def}}{=} ([\![c_2]\!]^{\mathsf{I}}_{\mathcal{T}} \circ [\![c_1]\!]^{\mathsf{I}}_{\mathcal{T}})(X) \\ [\![\mathbf{if}\ b\ c_1\ c_2]\!]^{\mathsf{I}}_{\mathcal{T}}(X) \overset{\textit{def}}{=} ([\![c_1]\!]^{\mathsf{I}}_{\mathcal{T}} \circ \mathsf{filt}(b))(X) \sqcup ([\![c_2]\!]^{\mathsf{I}}_{\mathcal{T}} \circ \mathsf{filt}(\neg b))(X) \\ [\![\mathbf{while}\ b\ c]\!]^{\mathsf{I}}_{\mathcal{T}}(X) \overset{\textit{def}}{=} \mathsf{filt}(\neg b)(\mathsf{lfix}\ \lambda X'.\, X \sqcup ([\![c]\!]^{\mathsf{I}}_{\mathcal{T}} \circ \mathsf{filt}(b))(X')) \end{array}$$

where $\mathsf{filt}(b)$ is used to filter out states that do not satisfy the boolean condition b. The semantics of function calls is already given in Section 6.3.

The abstract semantics $[\![-]\!]^{\mathsf{IP}}$ used by the precondition discovery follows the same pattern as the forward one $[\![-]\!]^{\mathsf{I}}$. The only difference is that it uses new versions of rearrangement, execution, abstraction functions that work on $\mathcal{P}((\mathsf{SH} \times \mathsf{SH}) \,\cup\, \{\top\})$:

$$\begin{array}{rcl} \mathsf{Rearr}(e) & : & \mathsf{SH} \times \mathsf{SH} \to \mathcal{P}((\mathsf{SH} \times \mathsf{SH}) \cup \{\top\}), \\ \mathsf{Exec}(A) & : & \mathsf{SH} \times \mathsf{SH} \to \mathcal{P}((\mathsf{SH} \times \mathsf{SH}) \cup \{\top\}), \\ \mathsf{Abs} & : & \mathsf{SH} \times \mathsf{SH} \to \mathsf{SH} \times \mathsf{SH}. \end{array}$$

These functions are defined in Figure 5. The major change is in the rearrangement function $\mathsf{Rearr}(e)$, which takes (F, H) and tries to expose a specified cell e from H. The rearrangement function of the forward analysis rearr is invoked to prove that a dereferenced cell e is allocated. The unusual step is that, in case this attempt fails, the subroutine adds the missing cell to the precondition and the current symbolic heap. A standard analysis would have stopped, reporting a possible fault. Note that before adding the points-to

[4] Rearrangement is the typical term used in separation logic based analyses. In [41], this step is called materialization of summary nodes.

[5] Here we view functions on SH as functions from SH to $\mathcal{P}(\mathsf{SH} \cup \{\top\})$ that always return singleton sets.

$$
\begin{array}{l}
\mathsf{Rearr}(e)(F,H) \overset{def}{=} \textbf{let } \mathcal{H} = \mathsf{rearr}(e)(H) \ \textbf{ and} \\
\qquad\qquad\qquad\qquad \mathcal{F} = \{(F,H') \mid H' \in \mathcal{H} \cap \mathsf{SH}\} \\
\qquad\qquad\qquad \textbf{in if } (\top \notin \mathcal{H}) \textbf{ then } \mathcal{F} \\
\qquad\qquad\qquad\quad \textbf{else if } (H \vdash e{=}a \text{ for some } a{\in}\mathsf{LVar}) \text{ and} \\
\qquad\qquad\qquad\qquad\qquad \neg(F{*}a{\mapsto}b \vdash \mathsf{false} \text{ for fresh } b{\in}\mathsf{LVar}) \\
\qquad\qquad\qquad\qquad \textbf{then } \mathcal{F} \cup \{(F * a{\mapsto}b,\ H * e{\mapsto}b)\} \\
\qquad\qquad\qquad\qquad \textbf{else } \mathcal{F} \cup \{\top\} \\
\mathsf{Exec}(A)(F,H) \overset{def}{=} \textbf{let } \mathcal{H} = \mathsf{exec}(A)(H) \\
\qquad\qquad\qquad \textbf{in } \{(F,H') \mid H' \in \mathcal{H}\} \cup \{\top \mid \top \in \mathcal{H}\} \\
\mathsf{Abs}(F,H) \overset{def}{=} (\mathsf{abs}(F), \mathsf{abs}(H))
\end{array}
$$

Figure 5. Rearrangement, Execution, Abstraction for $[\![A]\!]^{\mathsf{IP}}$.

relation to F, the expression e is rewritten in terms of a logical variable a to avoid conflicts with the program variables that change during the computation. We also point out that in order to stop the precondition assertion from growing forever (and therefore making the analysis diverge) Abs abstracts F as well as H.

7. Case Studies

The compositional analysis algorithm in this paper takes an abstract domain as an argument. We have implemented our algorithm in the SPACEINVADER tool, giving us a version "SPACEINVADER ABDUCTOR", or more briefly, ABDUCTOR. This instantiation of our compositional analysis algorithm uses the composite abstract domain from [3] as the base shape analysis. We used that domain because of its ability to deal with a variety of linear data structures. But since our analysis is parametric in the abstract domain, we could plug in other abstract domains, such as those from [19, 28], and we would immediately obtain a compositional analysis.

Small Examples. This first case study gives us some basic information on the quality of the specifications inferred by the compositional algorithm. It is entirely possible, after all, to obtain a useless compositional analysis simply by returning trivial specifications (the top of a lattice) for all procedures.

Our first case study illustrates the treatment of recursion (and, thus, the interprocedural aspect), both cyclic and acyclic lists, and nested structures. The particular data structure was a cyclic singly-linked list, where each node has a nested acyclic sub-list. We wrote recursive procedures for traversing, deleting, and inserting into such structures. In each case the ABDUCTOR found a precondition formula describing exactly this data structure: it described *only* the cells accessed by the algorithms. That the algorithm found these preconditions for algorithms performing nested traversals on nested data structures indicates that the analysis does not only find trivial specifications: accurate analysis within loops and recursions, and within lists, is needed to find the resulting specifications.[6]

Medium Example. The IEEE 1394 (firewire) Device Driver for Windows is around 10K LOC, and it contains 121 procedures. It was analyzed in a top-down fashion in [21]. We sought to use the same abstract domain, to compare our bottom-up analysis.

Our analysis was able to find consistent specifications (where 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 sub-procedures 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 [21]. 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.

We stress that, to obtain theses results for firewire, we had to slightly modify the code analyzed in [21]. The problem is that the driver works over lists where the nodes contain back-pointers to a common head node. When doing top-down analysis, with a given precondition, this "common" information is readily available. When going bottom-up, more general cases were considered by ABDUCTOR, that the abstract domain could not handle precisely. This is perhaps not surprising, as the domain from [21] was designed with top-down analysis in mind. But, it also illustrates that the bottom-up analysis places different stresses on an abstract domain than does top-down.

So, while we would claim that the better the abstract domain fits the data structures of a program the better are the triples discovered by our technique, we also emphasize that the problem of spec discovery might impact the design of abstract domains.

Large Programs and Complete Open Source Projects. In Table 1 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 two 2.66GHz Quad-Core Intel Xeon processors with 4GB 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 million (MLOC); 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 the number of processor cores indicated in brackets (e.g. 8 cores is indicated as Time (8)).

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 structure, e.g., in procedure `ap_find_linked_module` in Apache.) The point, for analysis, is that by combining abduction and frame inference we obtain specifications that (nearly) describe only the cells accessed

[6] The source code of the small examples can be found at:
`http://www.dcs.qmul.ac.uk/~ddino/small_examples_popl09`

Program	MLOC	Num. Procs	Proven Procs	Procs Coverage %	Time (1)	Time (8)
Linux kernel 2.6.25.4	2.473	101330	59215	58.4	6869.09	1739.28
Gimp 2.4.6	0.708	15114	6364	42.1	3601.16	1067.60
OpenSSL 0.9.8g	0.214	4818	2967	61.6	605.36	446.60
Sendmail 8.14.3	0.108	684	353	51.6	184.50	184.83
Apache 2.2.8	0.102	1870	881	47.1	294.67	104.48
OpenSSH 5.0	0.073	1135	519	45.7	142.56	30.24
Spin 5.1.6	0.019	357	197	55.2	772.82	253.96

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

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. To underline this independence we have reported parallel speedup in Table 1, for an implementation using eight cores.

Caveats. 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 [3], which does not deal well with arrays and pointer arithmetic. These are treated as non-deterministic operations that are imprecise but sound for the goal of proving pointer safety. Also, the analysis ignores concurrency. (Indeed, it may be that the compositional analysis gives new methods that might be used in shape analysis for concurrency [17].)

We have treated unknown library procedures as non-deterministic 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. Note that this "problem" does not impact the scalability aspect of our experiment, which is its primary purpose. (Of course, we would be interested in combining our work with techniques designed to deal with libraries in binary [15].)

Finally, 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 [33]. However, significant further work is needed in this direction.

At the beginning of the paper we made the case for examining a compositional shape analysis. We do not claim that compositional analyses are fundamentally superior to non-compositional ones. As we have argued in the introduction, they present several interesting properties. However, it is reasonable to envisage that, ultimately, effective and accurate analysis 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 non-compositional 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.

8. 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.[7] 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 [41], and there have been many subsequent works in search of ever better shape domains (e.g., [35, 5, 24, 26, 27, 3, 7]). 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 [19, 21, 28]. Other analyses sacrifice precision in order to gain scalability [13, 20]; they cannot, for example, give precise results on device drivers.

The techniques developed in this paper are in a sense complementary to the ideas in all these works. Although we have used the domain from [3, 21] in our experiments, the compositional analysis algorithm is parametrized by the abstract domain and we might swap in one of the other abstract domains, or others that are developed in the future to obtain ever better compositional analyses.

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, underapproximating 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 [6], 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 [25, 36, 1]. 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 have treated procedure calls in a local way, by passing only the reachable part of the abstract heap to a procedure [40, 16, 28]. 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 [9], and for non-shape domains a number of compositional analyses have appeared (e.g. [43, 31, 12, 18, 29]). They all use different techniques to those here.

[7] For example, [23] refers to itself as a compositional heap analysis, but it requires procedure summaries/specs to be provided by the user, so it performs a different task than here.

Giacobazzi has previously used abductive inference in the analysis of logic programs [14]. The problem he solves is dual to that here. Given a specification of a logic programming module and an implementation, the method of [14] 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-computer 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.

9. Conclusions

This paper has introduced the use of abductive inference to synthesize specifications in a shape analysis. We defined one particular abductive proof procedure for use with separated heap abstractions, and we explained how a more general notion of bi-abduction could be used to define a compositional interprocedural shape analysis.

Bi-abduction displays abduction as, in a sense, an inverse to the frame problem. Frame inference is about synthesizing leftover, extra portions of state. Abduction finds needed, missing portions of state, which we call anti-frames. The frames are not allowed to be updated, while the anti-frames can be. This informally-described inversion is curiously matched in our logical question

$$A * ?1 \vdash G * ?2$$

where $?2$ is the frame and $?1$ is the anti-frame. Furthermore, the two parts to the bi-abduction question play different but mutually supporting roles in our compositional analysis: the anti-frames help us *find* small specifications, that describe constrained portions of memory [32], while the frames allow us to *use* the small specs.

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.

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 non-trivial 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 non-trivial 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.

Finally, it seems that our scheme of using abduction to infer preconditions could conceivably be used to define compositional analyses for other abstract domains than shape domains.

Acknowledgements. We would like to thank Jan Tobias Mühlberg for providing the benchmarks which initially sparked this work, and Paul Kelly for giving us access to his group's 8-core machine for our experiments. Distefano was supported by a Royal Academy of Engineering research fellowship. Calcagno and Yang were supported by EPSRC Advanced Fellowships. O'Hearn was supported by a Royal Society Wolfson Research Merit Award.

References

[1] P.A. Abdulla, A. Bouajjani, J. Cederberg, F. Haziza, A. Rezine: Monotonic Abstraction for Programs with Dynamic Memory Heaps. In CAV'08, pp341-354.

[2] I. Balaban, A. Pnueli, and L. D. Zuck. Shape analysis by predicate abstraction. In *VMCAI'05*, *LNCS* 3385, pp. 164–180. Springer, 2005.

[3] J. Berdine, C. Calcagno, B. Cook, D. Distefano, P. O'Hearn, T. Wies, and H. Yang. Shape analysis of composite data structures. In *CAV'07*.

[4] J. Berdine, C. Calcagno, and P. O'Hearn. Symbolic execution with separation logic. In *APLAS'05*, 2005.

[5] A. Bouajjani, P. Habermehl, A. Rogalewicz, and T. Vojnar. Abstract tree regular model checking of complex dynamic data structures. In *SAS'06.*, *LNCS* 4134, pp. 52–70. Springer, 2006.

[6] C. Calcagno, D. Distefano, P. O'Hearn, and H. Yang. Footprint analysis: A shape analysis that discovers preconditions. In *SAS'07*, *LNCS* 4634, pp. 402–418. Springer, 2007.

[7] B. Chang and X. Rival. Relational inductive shape analysis. In *35th POPL*, pages 247–260. ACM, 2008.

[8] B. Chang, X. Rival, and G. Necula. Shape analysis with structural invariant checkers. In SAS'07, 2007.

[9] P. Cousot and R. Cousot. Compositional separate modular static analysis of programs by abstract interpretation. In *Proceedings of SSGRR*, Compact disk, L'Aquila, Italy, 2001.

[10] D. Distefano, P. O'Hearn, and H. Yang. A local shape analysis based on separation logic. In *TACAS'06*, 2006.

[11] D. Distefano and M. Parkinson. jStar: Towards Practical Verification for Java. In *OOPSLA*, 2008. To appear.

[12] N. Dor, M. Rodeh, and M. Sagiv. CSSV: Towards a realistic tool for statically detecting all buffer overflows in C. *PLDI*, 2003.

[13] R. Ghiya and . Hendren. Is it a tree, a DAG, or a cyclic graph? a shape analysis for heap-directed pointers in C. In *POPL'96*, pp. 1–15, 1996.

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

[15] D. Gopan and T. Reps. Low-level library analysis and summarization. In *CAV 2007*, pages 68–81.

[16] A. Gotsman, J. Berdine, and B. Cook. Interprocedural shape analysis with separated heap abstractions. In *SAS'06*,*LNCS* 4134, pp. 240–260.

[17] A. Gotsman, J. Berdine, B. Cook, and M. Sagiv. Thread-modular shape analysis In PLDI 2007.

[18] S. Gulwani and A. Tiwari. Computing procedure summaries for interprocedural analysis. In *ESOP'07*, LNCS 4421, pp. 253–267.

[19] B. Guo, N. Vachharajani, and D. August. Shape analysis with inductive recursion synthesis. In *PLDI'07*, 2007.

[20] B. Hackett and R. Rugina. Region-based shape analysis with tracked locations. In *POPL'05*, pages 310–323, 2005.

[21] H.Yang, O.Lee, J.Berdine, C.Calcagno, B.Cook, D.Distefano, and P.O'Hearn. Scalable shape analysis for systems code. In *CAV*, 2008.

[22] A. C. Kakas, R. A. Kowalski, and F. Toni. Abductive logic programming. *J. of Logic and Computation*, 2(6):719–770, 1992.

[23] V. Kuncak, P. Lam, and M. Rinard. Role analysis. In *29th POPL*, 2002.

[24] T. Lev-Ami, N. Immerman, and M. Sagiv. Abstraction for shape analysis with fast and precise transfomers. In *CAV*, 2006.

[25] T. Lev-Ami, M. Sagiv, T. Reps, and S. Gulwani:. Backward analysis for inferring quantified preconditions. Tel Aviv University Tech Report TR-2007-12-01, 2007.

[26] S. Magill, J. Berdine, E. Clarke, and B. Cook. Shape analysis with structural invariant checkers. In SAS'07, 2007.

[27] R. Manevich, J. Berdine, B. Cook, G. Ramalingam, and M. Sagiv. Shape analysis by graph decomposition. In *TACAS'07*, LNCS 4424, pp 3–18, Springer, 2007.

[28] M. Marron, M. Hermenegildo, D. Kapur, and D. Stefanovic. Efficient context-sensitive shape analysis with graph based heap models. In

CC'08, 2008.

[29] Y. Moy. Sufficient preconditions for modular assertion checking. In *9th VMCAI*, 2008.

[30] H. Nguyen and W.-N. Chin. Enhancing program verification with lemmas. In *CAV*, 2008.

[31] E. Nystrom, H. Kim, and W. Hwu. Bottom-up and top-down context-sensitive summary-based pointer analysis. *11th SAS*, 2004.

[32] P. O'Hearn, J. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *CSL'01*, 2001.

[33] P. O'Hearn, H. Yang and J. Reynolds. Separation and information hiding. POPL 2004, pp. 268-280.

[34] C. Peirce. *Collected papers of Charles Sanders Peirce*. Harvard University Press., 1958.

[35] A. Podelski and T. Wies. Boolean heaps. In *SAS'05*, LNCS 3672, pp. 268–283. Springer, 2005.

[36] A.Podelski, A.Rybalchenko, and T.Wies. Heap assumptions on demand. In *CAV*, 2008.

[37] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL'95*, 1995.

[38] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, 2002.

[39] N. Rinetzky, J. Bauer, T. Reps, M. Sagiv, and R. Wilhelm. A semantics for procedure local heaps and its abstractions. In *POPL'05*, 2005.

[40] N. Rinetzky, M. Sagiv, and E. Yahav. Interprocedural shape analysis for cutpoint-free programs. In *SAS'05*, LNCS 3672, pp. 284–302, Springer, 2005.

[41] M. Sagiv, T. Reps, R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM TOPLAS*, 20(1):1–50,1998.

[42] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S. Muchnick and J. Jones, editors, *Program Flow Analysis: Theory and Applications*. Prentice-Hall, 1981.

[43] J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java. In *OOPSLA*, pages 187–206, 2006.