Interprocedural Shape Analysis with Separated Heap Abstractions

Alexey Gotsman[1], Josh Berdine[2], and Byron Cook[2]

[1] University of Cambridge
Alexey.Gotsman@cl.cam.ac.uk
[2] Microsoft Research Cambridge
{jjb,bycook}@microsoft.com

Abstract. We describe an interprocedural shape analysis that makes use of spatial locality (i.e. the fact that most procedures modify only a small subset of the heap) in its representation of abstract states. Instead of tracking reachability information directly and aliasing information indirectly, our representation tracks reachability indirectly and aliasing directly. Computing the effect of procedure calls and returns on an abstract state is easy because the representation exhibits spatial locality mirroring the locality that is present in the concrete semantics. The benefits of this approach include improved speed, support for programs that deallocate memory, the handling of bounded numbers of heap cutpoints, and support for cyclic and shared data structures.

1 Introduction

Interprocedural shape analysis engines infer and prove properties about the shapes of dynamically-allocated linked data structures constructed by imperative programs with (possibly recursive) procedures. We present a local interprocedural shape analysis tool, called SUMMATE, that is efficient and more accurate than previously reported results. The tool's advantage comes from the representation used for abstract program states, which consists of circumscribed portions of a program's heap. The shape of an abstracted portion of heap is determined solely by the representation of *only that* portion of heap. Representing heap portions independently is accomplished by building the shape of the heap into the notion of abstraction, using formulæ in separation logic. That is, abstracted heaps have known shape and are specified using inductive predicates that make positive statements (saying what the shape *is*, rather than what it *is not*) of each circumscribed portion of the abstracted heap.

The benefit of our representation for interprocedural analysis is that, when a procedure is called, the portion of the heap that it will not access can easily be separated from the rest, and easily recombined with the modified heap upon procedure return. Furthermore, spatial locality of code (i.e. the fact that each program statement accesses only a very limited portion of the concrete state) matches the spatial locality *in the representation*, dramatically reducing the amount of reasoning that must be performed when summarizing how procedure calls and returns change the symbolic representation of a program's state. This is because each instruction can only affect one of the separated heap portions—the transfer functions are parametric with respect to the untouched heap portions.

Our approach provides support for cyclic and shared data structures, procedures that deallocate memory, and a bounded number of *heap cutpoints*[3] [13]—all of which appear commonly in programs. For this reason our analysis is more accurate and applicable: it can be directly applied to, and give precise results for, a larger set of programs than previously reported tools.

Note that our approach also has a limitation: it is specialized to a limited set of data-structures such as linked lists, doubly-linked lists and trees. Our analysis is fortified with inductive axioms from [3]. These axioms accelerate the analysis. However: in order to support new data structures we would need to do additional manual work up-front before fortifying the analysis further. We return to this point in Sect. 6.

2 Fundamentals

SUMMATE implements a fixed-point computation over an abstract domain built from assertions expressed in separation logic. In essence, the analysis performed by SUMMATE can be viewed as a method of constructing proofs in standard separation logic (in fact, our proof of the analysis' soundness is based on this observation). In this section we describe SUMMATE's fundamental operations as proof rules in separation logic. Later, in Sect. 3, we go into more specific detail.

2.1 Abstract Representation of States

SUMMATE's states denote sets of store-heap pairs, and are represented as formulæ of separation logic's assertion language, which include:

$$\begin{array}{ll} F, E ::= \mathsf{nil} \mid x \mid x' & \text{expressions} \\ Q, P ::= \mathsf{emp} \mid E{\mapsto}F \mid P * Q \mid \mathsf{true} \mid E{=}F \mid P \wedge Q \mid P \vee Q \mid \cdots & \text{assertions} \end{array}$$

Expressions are independent of the heap, while assertions are not. Primed variables are implicitly existentially quantified. The formal semantics of assertions is standard, e.g. as in [12], but informally:

- emp describes states where the heap is empty, with no allocated locations;
- $E{\mapsto}F$ describes states where the heap contains a single allocated location E, with contents F;
- $P{*}Q$ describes states where the heap is the union of two disjoint heaps (with no locations in common), one satisfying P and the other satisfying Q;

[3] Roughly speaking: a cutpoint is a location in the portion of the heap that the procedure may access distinct from all the actual parameters of the procedure, that the rest of the state knows about in some way, either as the contents of a heap location or value of a variable.

– true describes all states;
– $E{=}F$ describes states where the store gives E and F equal values;
– $P \land Q$ describes states which satisfy both P and Q; and
– $P \lor Q$ describes states which satisfy either P or Q.

Possibly infinite sets of concrete states are finitely represented using inductive predicate assertions. For example, using the predicate $\mathsf{ls}(x, y)$ defined as $x{\neq}y \land (x{\mapsto}y \lor \exists x'.\, x{\mapsto}x' * \mathsf{ls}(x', y))$ the symbolic heap $\mathsf{ls}(x, \mathsf{nil})$ represents all of the states in which $x{\neq}\mathsf{nil}$ and the heap has the shape of a linked list starting from location x and ending with nil. There are unboundedly many such states, as the length of the list is unconstrained. Similarly, representations built from formulæ such as $\mathsf{tree}(x)$ or $\mathsf{dlist}(p, f, n, b)$ [12] constitute abstractions of unboundedly many concrete states, shaped like trees or doubly-linked lists. In each case, the abstracted heaps have known shape, specified declaratively by an inductive predicate. The abstraction comes from not tracking the precise number of inductive unfoldings from the base case.

2.2 Local Reasoning for Procedures

Interprocedural analyses commonly compute so-called procedure summaries that approximate the semantics of a procedure by associating representations of the program state at procedure entry to corresponding result states at procedure exit. We represent such computed summaries as a sequence Γ of triples $\{P\}\ f(\vec{x})\ \{Q\}$ in separation logic.[4] In separation logic, a triple $\{P\}\ C\ \{Q\}$ is valid if executing command C from any state satisfying assertion P does not violate memory safety and, if execution terminates, results in a state satisfying assertion Q. Lying behind this is a semantics of commands which results in a memory fault when accessing dangling pointers or other memory locations not guaranteed to be allocated. So validity of $\{P\}\ C\ \{Q\}$ ensures that P describes all the memory (except that which gets freshly allocated) that may be accessed during the execution of C, that is, the *footprint* of C.

The technical foundation of our approach to local interprocedural analysis is the Frame rule [10]:

$$\textsc{Frame}\quad \frac{\{P\}\ C\ \{Q\}}{\{P * R\}\ C\ \{Q * R\}}\ C \text{ does not modify variables in } R$$

If P ensures C's footprint is allocated, then according to Frame, executing C in the presence of additional memory R results in the same behavior, and C does not touch the extra memory. Since we represent concrete states with formulæ, Frame expresses how commands exhibit spatial locality in the abstract representation.

Our aim is to define an analysis which exploits this locality by using Frame in the case where C is a recursive procedure call $f(\vec{x})$ in order to send only part

[4] In this way, each triple in Γ corresponds to an entry in the table computed by tabulation algorithms such as [11], where the set of exit states there is expressed using logical disjunction in the Q's.

of the heap P at the call site to the procedure, while holding the rest of the heap R aside, to be added to the heap Q that results from executing f. This is formalized in the following proof rule for local recursive procedure calls:

$$\textsc{LocalProcCall}\quad \frac{S \vDash P\sigma * R \qquad Q\sigma * R \vDash T}{\mathit{\Gamma}\,,\{P\}\ f(\vec{x})\ \{Q\} \vdash \{S\}\ f(\vec{x}\sigma)\ \{T\}}$$

This rule is not primitive, but derivable (see Appendix A) from Frame and Hoare logic rules. Here $\vDash$ means semantic consequence and σ is an injective substitution map from variables, including the formals, to variables, including the actuals. σ adapts the hypothesis (i.e. the procedure summary), which is expressed not in terms of the actual, but formal, parameters (and possibly other variables, as discussed below), to a specification expressed in terms of the actual parameters. The rule LocalProcCall says that to compute the post-heap of a call to procedure f starting from pre-heap S:

1. split S into two disjoint ($*$-conjoined) heaps $P\sigma$ (a *local heap*) and R (a *frame*);
2. express the pre-heap $P\sigma$ in terms of the formal parameters, yielding P, which is applicable to the summary of the procedure f;
3. compute the post-heap Q of the procedure call on P;
4. express Q in terms of the actual parameters, yielding $Q\sigma$;
5. $*$-conjoin $Q\sigma$ with the frame axiom R, yielding the post-heap T.

Note that the choice of splitting $P\sigma * R$ of S is not important for soundness: any splitting is sound, but if too small a heap $P\sigma$ is chosen, a false memory fault will be discovered, and no post-heap Q will exist. Here (as in [15]) we choose to split the heap so that we send the procedure all of the heap reachable from the actual parameters.

2.3 Cutpoints

For the procedure call rules above, note that the free variables of the pre- and post-conditions of the procedure summaries in $\mathit{\Gamma}$ need not contain *only* the formal parameters. For instance, $\{x{\mapsto}y * y{\mapsto}\mathsf{nil}\}\ f(x)\ \{x{\mapsto}y\}$ is a perfectly reasonable procedure summary, whose pre- and post-conditions happen to contain a variable y which does not occur in the command $f(x)$. Such variables, commonly referred to as *ghost* variables, can be instantiated to whatever value is appropriate at a particular call site using Hoare logic's substitution rule (Subst, see Appendix A), which carries over to our LocalProcCall-based analysis.

For an instance of where this arises in programs, consider the code in Fig. 1, which represents a sequence of operations on a stack `s` implemented as a linked-list. Imagine that we are trying to summarize the effect that `foo` can have on the stack passed to it. Notice that, while the stack `s` is reachable from `foo`, both `i1` and `i2` will contain pointers into `s` but are not reachable from `foo`. Pointers such

```
Node *s, *i1, *i2;
int x, y, z;
/* ... */
s = push(s, x);
i1 = iterator(s);
s = push(s, y);
i2 = iterator(s);
s = push(s, z);
foo(s);
i2 = next(i2);
assert(i1 == i2);
```

Fig. 1. Simple example code fragment with cutpoints

as `i1` and `i2` are known as *cutpoints*. It is difficult to make a scalable analysis that will be accurate enough to prove that the `assert` cannot fail.

Without special consideration, cutpoints can be treated just as any other ghost variable. Since we use the standard store-based semantics, and a separated abstract representation, there are no problems splitting heaps in ways that create pointers that dangle across the split. Hence, if we were not worried about computability and finiteness considerations, the presence of cutpoints would be irrelevant: simply ignoring them and treating them just like any other ghost variable is sound and maximally precise. As a result, our representation enables our analysis to accurately and efficiently handle cutpoints: they cost no more than any other variable which appears in procedure summary pre- or post-conditions.

However, there is a problem in that just ignoring cutpoints potentially leads to unboundedly many of them, which breaks finiteness of the abstract domain. A solution is to abstract cutpoints beyond some bounded number by breaking the connection between them and the pointers to them, that is, by forgetting the destinations of pointers to cutpoints. In this manner, our analysis treats bounded numbers of cutpoints, and is parametric in the bound.

Hoare logic's rule of semantic consequence provides a mechanism for performing this abstraction. A cutpoint c in heap splitting $P' * R$ can simply be existentially quantified, since $P' \vDash \exists c.\, P'$ is a particular semantic consequence, thereby breaking the connection between pointers in R to c. Quantifying cutpoints is productive since quantified variables do not contribute to the size of the abstract domain, in contrast with unquantified variables. So, if a splitting $S \vDash P' * R$ contains cutpoints $\vec{c}$, then we can abstract them using the derived rule:

$$\textsc{LocalProcCallCut}\quad \frac{S \vDash P' * R \qquad \exists \vec{c}.\, P' \vDash P\sigma \qquad Q\sigma * R \vDash T}{\Gamma\,, \{P\}\ f(\vec{x})\ \{Q\} \vdash \{S\}\ f(\vec{x}\sigma)\ \{T\}}$$

The operational reading of this rule is like that of LocalProcCall except that after splitting the pre-state into $P' * R$, the variables denoting cutpoints in P' should be existentially quantified.

3 Implementation of the Analysis

We consider a simple programming language of while loops and recursive procedures extended with the usual four heap operations for loading from, storing to, allocating, and deallocating heap locations. Note, that although the analysis presented in this section operates on the abstract domain of separation logic formulæ including inductive predicates for lists only, the interprocedural analysis technique can be extended to include inductive predicates for other data structures such as trees and doubly-linked lists.

The syntax of the language is defined as follows:

$G ::= E{=}F \mid \neg(E{=}F)$	branch guards
$S ::= \mathtt{skip} \mid x{:=}\,E \mid x{:=}\,\mathtt{new}() \mid \mathtt{assume}(G)$	safe commands
$A(E) ::= \mathtt{dispose}(E) \mid x{:=}\,[E] \mid [E]{:=}\,F$	dangerous commands
$T ::= S \mid A(E) \mid f(\vec{x})$	atomic commands
$C ::= T \mid \mathtt{if}\ (G)\ \{C\}\ \mathtt{else}\ \{C\} \mid \mathtt{while}\ (G)\ \{C\}$	commands
$D ::= f(\vec{x})\ \{\mathtt{local}\ \vec{y}\ \mathtt{in}\ C\}$	procedure declarations
$M ::= \mathtt{letrec}\ D, \ldots, D\ \mathtt{in}\ \mathsf{main}(\vec{x})$	programs

Here variables $x, y, \ldots$ range over some infinite set Var; existentially quantified variables $x', y', \ldots$ range over some disjoint infinite set Var$'$; and, for each program, procedure names f range over some fixed finite set. We also assume given a set of variables Ghost $\subset$ Var used for ghost variables in the analysis to replace cutpoints during the processing of procedure calls. Quantified variables cannot appear in programs, but are included since expressions also appear in formulæ (Sect. 2.1). For convenience of later definitions, commands S are syntactically distinguished from commands $A(E)$. The difference between the two is that for a command S, execution is always safe, while execution of a command $A(E)$ may be unsafe, due to accessing heap location E.

Complications due to reference parameters are orthogonal to our concerns of interproceduralty, so we only consider procedures with value parameters. Additionally, for simplicity of presentation, we treat only programs without global variables or functions returning values (standard treatments such as [8, 5] can be adopted). As a result, for $f(\vec{x})\ \{\mathtt{local}\ \vec{y}\ \mathtt{in}\ C\}$ we require that the list of local variables $\vec{y}$ contain all the free variables $fv(C)$ of C except the formals $\vec{x}$. Finally, we assume programs have been syntactically preprocessed to ensure $\vec{x} \cap \vec{y} = \emptyset$ for each procedure declaration, and actual parameters are distinct variables.

The informal meaning of commands is as follows:

- `skip` accesses no heap, and has no effect;
- $x{:=}\,E$ does not access the heap, and results in a state where x has the value of E (using the overwritten value of x);
- allocation $x{:=}\,\mathtt{new}()$ requires no heap and returns an uninitialized location that is distinct from all other allocated locations (though may be pointed to by a previously dangling pointer);

- `dispose` takes a single location and deallocates it, possibly creating dangling pointers in the process;
- $x{:=}[E]$ accesses heap location E and results in a state where the heap is unmodified and x has value equal to the contents of E;
- $[E]{:=}F$ accesses location E and changes its contents to F;
- $\texttt{assume}(G)$ acts as a filter on the state space of programs—G must be true after `assume` is executed;
- and the meaning of the control-flow commands is standard.

We will argue correctness of the analysis by generating proofs in separation logic out of its results, rather than directly in terms of the concrete semantics of the programming language. Therefore we do not present the concrete semantics in any detail here, it is entirely standard and appears elsewhere (such as [12]). Instead, the separation logic axioms for commands [12] together with the rules from the previous section specify the meaning of the programming language in enough detail for our present purpose. We use the following rule (which is derived from standard rules for recursive procedure declarations [8] and variable declarations, see Appendix A) to define the semantics of procedure declarations:

$$\textsc{RecProcDeclLocals}\quad \frac{\Gamma\,,\{P\}\ f(\vec{x})\ \{Q\} \vdash \{P\}\ C\ \{T\} \qquad \exists\vec{y}.\,T \vDash Q \qquad \Gamma\,,\{P\}\ f(\vec{x})\ \{Q\} \vdash \{R\}\ C'\ \{S\}}{\Gamma \vdash \{R\}\ \texttt{letrec}\ f(\vec{x})\ \{\texttt{local}\ \vec{y}\ \texttt{in}\ C\}\ \texttt{in}\ C'\ \{S\}}$$

where $\vec{x} \cap \vec{y} = \emptyset$, $\vec{y} \cap fv(P) = \emptyset$, $fv(P)$ is the set of all free variables of P. The side condition $\vec{y} \cap fv(P) = \emptyset$ is needed so that variables in P do not clash with local variables of f. Existential quantification of the local variables ensures that they are not visible to a caller after the call returns.

3.1 Symbolic Heaps

Summate's analysis represents sets of concrete program states with sets of symbolic heaps Q of form $\Pi \wedge \Sigma$, where Π and Σ are given by:

$$\Pi ::= \mathsf{true} \mid \Pi \wedge \Pi \mid E{=}E \qquad \Sigma ::= \mathsf{emp} \mid \Sigma * \Sigma \mid E{\mapsto}E \mid \mathsf{ls}(E,E) \mid \mathsf{junk}$$

Symbolic heap formulæ consist of two parts: a Boolean formula Π built from $=$ and $\wedge$ which is insensitive to the heap; and a heap formula Σ which expresses heap shape. The meaning of these formulæ is as in Sect. 2.1, with the addition that junk describes at least one allocated location. Recall also that $\mathsf{ls}(E,F)$ describes non-empty acyclic singly-linked lists. Cyclic lists can be expressed using multiple predicates: e.g. $\mathsf{ls}(x,y') * \mathsf{ls}(y',x)$. Note that $x{\mapsto}x$ is a cycle of length one, while $\mathsf{ls}(x,x)$ is inconsistent.

Formulæ are considered up to symmetry of $=$, permutations across $\wedge$ and $*$ (e.g. $\Pi \wedge B_0 \wedge B_1$ and $\Pi \wedge B_1 \wedge B_0$ are equated), unit laws for true and emp, idempotency of $- * \mathsf{junk}$ (e.g. $\mathsf{junk} * \mathsf{junk}$ and junk are equated), adding or removing consequences of equalities present in the pure part, and interchanging equal (due to the equalities in the pure part) variables in the spatial part. So, $x = y \wedge y = z \wedge \mathsf{ls}(v,x)$ and $x = y \wedge y = z \wedge x = z \wedge \mathsf{ls}(v,y)$ are considered equal. We denote the set of symbolic heaps with $\mathcal{SH}$.

3.2 Intraprocedural Analysis

As a part of its *inter*procedural analysis, SUMMATE must also implement an *intra*procedural analysis. For this SUMMATE implements the analysis from [6, 4]. This analysis is defined in Appendix B. A complete exposition is found in [6].

The intraprocedural analysis defines a set of canonical symbolic heaps $\mathcal{CSH} \subset \mathcal{SH}$ on which the analysis operates, a canonicalization function $\mathsf{can}\colon \mathcal{SH} \to \mathcal{CSH}$, which returns a canonical symbolic heap abstracting a given symbolic heap, and a decision procedure for consistency of canonical symbolic heaps. Note that canonical symbolic heaps are written in the same language as symbolic heaps, i.e. they can have existential quantifiers. A key property of the abstract domain proved in [6] is that although the number of symbolic heaps over a finite number of unquantified variables is infinite, the domain of consistent and canonical symbolic heaps over a finite number of unquantified variables is finite. Hence, due to the presence of canonicalization in the analysis, fixed-point computations over the abstract domain converge in a finite number of steps.

For each atomic command C the intraprocedural analysis defines a transfer function $\mathcal{A}_C\colon \mathcal{SH} \to (2^{\mathcal{CSH}} \cup \{\top\})$ that, given an initial symbolic heap, returns either $\top$ (meaning that a possible memory error has been encountered) or a set of consistent canonical symbolic heaps representing the effect of the command on the initial symbolic heap. If the former case is encountered, our analysis terminates and reports a possible bug.

While performing fixed-point computations both our intraprocedural and interprocedural analyses use subset inclusion as a domain ordering between sets of symbolic heaps. Other, less coarse, approximations of entailment between symbolic heaps (e.g. [3, 2]) would be possible (but note that convergence of fixed-point computation is a question if the entailment prover is not transitive).

3.3 Analyzing Procedure Calls and Returns

In this section we give a detailed explanation of how SUMMATE treats procedure calls and returns. This treatment follows the operational reading of the rules LOCALPROCCALL and RECPROCDECLLOCALS.

According to the proof rule LOCALPROCCALL, to process procedure call $f(\vec{x}\theta)$ we have to determine which part of the symbolic heap at the call-site to send to the procedure. As noted in Sect. 2.2, we send to the procedure the part of the heap reachable from the actual parameters in the formula representing the symbolic heap. Formally, let Σ be the spatial part of a consistent symbolic heap and U be a set of expressions. Let V be the minimal set of expressions such that:

$$U \cup \{F \mid \exists E, \Sigma_1.\, E \in V \text{ and } \Sigma = H(E,F) * \Sigma_1\} \subseteq V$$

Here $H(E,F)$ stands for either $E \mapsto F$ or $\mathsf{ls}(E,F)$. We denote the part of Σ reachable from U with $\mathsf{Reach}(\Sigma, U)$ and define it as the $*$-conjunction of the following set of formulæ:

$$\{\Sigma_1 \mid \exists E, F, \Sigma_2.\, E \in V \text{ and } \Sigma = \Sigma_1 * \Sigma_2 \text{ and } \Sigma_1 = H(E,F)\}$$

Let $\mathsf{Unreach}(\Sigma, U)$ be the formula consisting of all $*$-conjuncts from Σ that are not in $\mathsf{Reach}(\Sigma, U)$.

Consider a procedure call statement $f(\vec{x}\theta)$ with formal parameters $\vec{x}$ and the map from formal parameters to actual parameters θ and let $Q_{\mathrm{call}} = \Pi \wedge \Sigma$ be the heap at the call site. To take the equalities in Π into account while computing the part of Σ reachable from actual parameters we require that the variables in Σ be chosen so that for each equivalence class generated by the equalities in Π at most one variable from this equivalence class is present in Σ (with preference given to unquantified variables over quantified ones, and to actual parameters over other variables). We denote the part of the heap to be sent to the procedure with $\mathsf{local}(\Pi \wedge \Sigma, \vec{x}\theta)$ and define it as:

$$\mathsf{local}(\Pi \wedge \Sigma, \vec{x}\theta) = \mathsf{can}\Big(\exists\big(\mathit{fv}(\Pi \wedge \Sigma)\setminus\mathit{fv}(\mathsf{Reach}(\Sigma, \vec{x}\theta))\big).\ \Pi \wedge \mathsf{Reach}(\Sigma, \vec{x}\theta)\Big)$$

The local heap is obtained by taking the part of the heap reachable in the formula from the actual parameters, projecting it onto those variables appearing in the representation of the reachable heap, and canonicalizing the result. The set of cutpoints in the result is $\mathsf{Cut} = \mathit{fv}(\mathsf{local}(Q_{\mathrm{call}}, \vec{x}\theta))\setminus\vec{x}\theta$. The frame in this case is given by $\mathsf{frame}(\Pi \wedge \Sigma, \vec{x}\theta) = \Pi \wedge \mathsf{Unreach}(\Sigma, \vec{x}\theta)$.

Having obtained a local heap we have to express it in terms of the formal parameters. As follows from the proof rule RecProcDeclLocals, we also have to rename cutpoints so as they do not clash with the local variables of the procedure f. Hence, we rename them to variables in Ghost. Let $\mathsf{ghost}(V)$ be a function that given the set of variables V returns a bijective partial function from variables in Ghost to V and let $\eta = \mathsf{ghost}(\mathsf{Cut})$. Then the heap Q_{entry} at the entry point of the procedure f (expressed in terms of formal parameters and ghost variables) is given by $\mathsf{local}(Q_{\mathrm{call}}, \vec{x}\theta)(\theta \cup \eta)^{-1}$.

Example 1. Suppose that before executing the procedure call `foo`(a) (with the formal parameter x) we have a symbolic heap $a{=}d \wedge \mathsf{ls}(a, b) * \mathsf{ls}(b, \mathsf{nil}) * c{\mapsto}b$ so that the tail of the list pointed to by a is shared. Then the part of the heap reachable from actual parameters (in this case just a) is $\mathsf{ls}(a, b) * \mathsf{ls}(b, \mathsf{nil})$ and the local heap is $\mathsf{can}(a{=}d' \wedge \mathsf{ls}(a, b) * \mathsf{ls}(b, \mathsf{nil})) = \mathsf{ls}(a, b) * \mathsf{ls}(b, \mathsf{nil})$. We remind the reader that primed variables are implicitly existentially quantified. Here b is a cutpoint, so we choose a variable $X \in$ Ghost and rename b to X and a to x obtaining $\mathsf{ls}(x, X) * \mathsf{ls}(X, \mathsf{nil})$ as a heap at the entry point of the procedure. $\square$

Note that for simplicity of presentation the analysis described above precisely handles only the cutpoints that arise from stack sharing, i.e. in the situation when a location in the local heap is equal to the value of a variable of the caller and distinct from all the actual parameters. Such cutpoints are defined by unquantified variables in the symbolic heap at the call-site. The other kind of cutpoints result from heap sharing, i.e. in the situation when a location in the local heap is equal to the contents of a location in the frame and distinct from all the local variables of the caller. Such cutpoints are defined by quantified variables in the symbolic heap at the call-site (e.g. x' in the local heap $\mathsf{ls}(x, x')$

with frame $\mathsf{ls}(y, x')$ where x is an actual parameter and y is a local variable of the caller). This kind of cutpoints can be handled precisely in a similar fashion (i.e. by replacing them with ghost variables in the local heap).

The way of processing procedure calls described above gives the most precise treatment to cutpoints—no information is lost when the heap at the call site has a cutpoint. However, in the case of recursive procedures the renaming of cutpoints to ghost variables can result in the number of unquantified variables in canonical symbolic heaps growing unboundedly. At the same time, as noted in Sect. 3.1, the abstract domain of canonical symbolic heaps is finite only if the number of unquantified variables is bounded. Hence, as it stands now, the analysis may not terminate. To solve this problem we put a bound m on the maximal number of cutpoints that can appear in a symbolic heap, i.e. on the cardinality of the set Ghost. Our analysis is parametric in this bound. Whenever during a call the number of cutpoints exceeds m, we existentially quantify the *new* cutpoints introduced by this call (i.e. variables in $\mathsf{Cut}\setminus\mathrm{Ghost}$) using the proof rule LocalProcCallCut. This guarantees finiteness of the abstract domain, and hence, termination of the analysis. The local heap in the case when we abstract cutpoints is defined as:

$$\begin{gathered}\mathsf{local}(\Pi \wedge \Sigma, \vec{x}\theta) = \\ \mathsf{can}(\exists(\mathsf{Cut}\setminus\mathrm{Ghost} \cup fv(\Pi \wedge \Sigma)\setminus fv(\mathsf{Reach}(\Sigma, \vec{x}\theta))).\ \Pi \wedge \mathsf{Reach}(\Sigma, \vec{x}\theta))\end{gathered}$$

Example 2. Consider the previous example and suppose that $m{=}0$. Then the local heap will be $\mathsf{can}(a{=}d' \wedge \mathsf{ls}(a, b') * \mathsf{ls}(b', \mathsf{nil})) = \mathsf{ls}(a, \mathsf{nil})$. We existentially quantified the cutpoint and this resulted in it being eliminated by the subsequent canonicalization. The heap at the entry point of the procedure is $\mathsf{ls}(x, \mathsf{nil})$. In this case we lost the information about c pointing to a node in the list. $\square$

We observe that the pathological cases where the number of cutpoints grows unboundedly while analyzing recursive procedures is rarely encountered in practice (especially if some sort of dead variable analysis is used to eliminate unnecessary unquantified variables). Even when the number of cutpoints in a symbolic heap at the call-site exceeds m our analysis is still able to obtain some information (though not the most precise).

According to RecProcDeclLocals, while processing procedure returns we have to existentially quantify the local variables of the procedure in order to obtain a summary (and canonicalize the result so that it is in our abstract domain). To obtain a symbolic heap at the return-site of the caller we just have to rename formal parameters to actual parameters and ghost variables to cutpoints in the resulting heap, and $*$-conjoin it with the frame. The result is guaranteed to be canonicalized.

3.4 Control-Flow Graphs

Before performing the interprocedural analysis we apply a standard translation from the program to its control-flow graph (cfg). A cfg is defined by the set of nodes N and the control-flow relation $F \subseteq N \times L \times N$, where L is the set of edge labels, $L = T \cup \{\texttt{return}, \texttt{quantify\_locals}\}$, T is the set of atomic commands.

We translate each procedure f independently, distinguishing its entry node $\mathsf{entry}(f)$ (the node from which the execution of the procedure starts) and exit node $\mathsf{exit}(f)$ (the node from which the procedure returns).

As noted in Sect. 3.3 we have to existentially quantify all the local variables before returning from a procedure. Therefore, as the last statement of each procedure we add a statement (labeled with `quantify_locals`) with the transfer function that existentially quantifies the local variables in the given heap and canonicalizes the result. Let $\mathsf{end}(f)$ be the node of the CFG preceding this statement, so that $(\mathsf{end}(f), \texttt{quantify\_locals}, \mathsf{exit}(f)) \in F$.

For each procedure call statement $f(\vec{x}\theta)$ (where $\vec{x}$ are the formal parameters, θ is the map from the formal parameters to the actual parameters) we introduce two nodes—a call node and a return node—and add two edges to the CFG, one connecting the call node to the entry node of the procedure f (labeled with $f(\vec{x}\theta)$), and the other connecting the exit node of the procedure f to the return node (labeled with `return`). We connect the statement preceding $f(\vec{x}\theta)$ to the call node and the return node to the statement succeeding $f(\vec{x}\theta)$.

While translating the program to the CFG we translate `while` and `if` statements in the standard way using `assume` statements. Note that although we define our analysis using such a representation, the proof of its soundness relies upon the fact that the resulting CFG is obtained from a well-structured program since it uses Hoare logic's proof rules for `while` and `if`.

3.5 Interprocedural Analysis

To perform the interprocedural analysis using the treatment of procedure calls and returns in our analysis proposed in Sect. 3.3, we adapt the Reps-Horwitz-Sagiv algorithm [11, 14] for using symbolic heaps as the abstract domain and efficiently handling procedure summaries with multiple cutpoints.

The analysis tabulates a function $\varphi \colon N \to 2^{\mathcal{CSH} \times \mathcal{CSH}}$. Intuitively, $\varphi(n)$ represents the set of pairs (Q_1, Q_2) of symbolic heaps at the entry point of a function containing the node n (Q_1) and at the node n (Q_2) such that there exists an execution of a sequence of program statements between these two points transforming Q_1 to Q_2.

The function computed by the analysis is the least function φ satisfying the equations in Fig. 2 under the following order: $\varphi_1 \sqsubseteq \varphi_2 \Leftrightarrow \forall n.\ \varphi_1(n) \subseteq \varphi_2(n)$.

We assume that we are given a symbolic heap I representing the initial state at the start of main expressed purely in terms of the formal parameters of main (the equation for $\varphi(\mathsf{entry}(\mathsf{main}))$). In the equations for $\varphi(n_{\text{entry}})$ and $\varphi(n_{\text{return}})$ the symbolic heap Q_{entry} at the entry point of the procedure is obtained from a heap Q_{call} at the call-site as it is described in Sect. 3.3. Note that in order to effectively treat procedure summaries containing cutpoints (i.e. ghost variables), the procedure is analyzed on this heap only if it has not been analyzed for another heap equal to the current one up to a bijective renaming of ghost variables (this equality can be decided in time polynomial in the length of the symbolic heaps).

$$\varphi(\mathsf{entry}(\mathsf{main})) = \varphi(\mathsf{entry}(\mathsf{main})) \cup (I \times I);$$

$$\begin{gathered}\varphi(n_{\text{entry}}) = \varphi(n_{\text{entry}}) \cup \{(Q_{\text{entry}}, Q_{\text{entry}}) \mid \exists n_{\text{call}}, Q_0, Q_{\text{call}}, \eta. \\ (n_{\text{call}}, f(\vec{x}\theta), n_{\text{entry}}) \in F \wedge (Q_0, Q_{\text{call}}) \in \varphi(n_{\text{call}}) \wedge \\ \eta = \mathsf{ghost}(fv(\mathsf{local}(Q_{\text{call}}, \vec{x}\theta)) \setminus \vec{x}\theta) \wedge Q_{\text{entry}} = \mathsf{local}(Q_{\text{call}}, \vec{x}\theta)(\theta \cup \eta)^{-1} \wedge \\ (\neg \exists Q'_{\text{entry}}, \eta'. \ (Q'_{\text{entry}}, Q'_{\text{entry}}) \in \varphi(n_{\text{entry}}) \wedge Q'_{\text{entry}}(\theta \cup \eta') = Q_{\text{entry}}(\theta \cup \eta))\}\end{gathered}$$

for each $n_{\text{entry}} = \mathsf{entry}(f)$ for some procedure f;

$$\begin{gathered}\varphi(n_{\text{return}}) = \{(Q_0, (Q_{\text{exit}}\sigma) * \mathsf{frame}(Q_{\text{call}}, \vec{x}\theta)) \mid \exists Q_{\text{entry}}, \eta. \ (Q_0, Q_{\text{call}}) \in \varphi(n_{\text{call}}) \wedge \\ \sigma = \theta \cup \eta \wedge \mathsf{local}(Q_{\text{call}}, \vec{x}\theta) = Q_{\text{entry}}\sigma \wedge (Q_{\text{entry}}, Q_{\text{exit}}) \in \varphi(n_{\text{exit}})\}\end{gathered}$$

for each pair of a call node n_{call} and a return node n_{return} for a statement $f(\vec{x}\theta)$; here $n_{\text{entry}} = \mathsf{entry}(f)$, $n_{\text{exit}} = \mathsf{exit}(f)$;

$$\varphi(n_2) = \{(Q_0, Q_2) \mid \exists n_1, C, Q_1. \ (n_1, C, n_2) \in F \wedge (Q_0, Q_1) \in \varphi(n_1) \wedge Q_2 \in \mathcal{A}_C(Q_1)\}$$

for all other nodes n_2.

Fig. 2. The equations defining the analysis. For simplicity we show only the case when cutpoints are not abstracted. All substitutions are injective.

Similarly, in the equation for $\varphi(n_{\text{return}})$ we search for summaries with the initial state equal to Q_{entry} up to a bijective renaming of ghost variables.

Example 3. Consider the following program fragment:

```
append(x, y);
append(u, v);
append(x, z)
```

Here `append(a, b)` receives as parameters head nodes of two lists and destructively appends the second list to the end of the first one. Suppose the initial state of the program consists of five disjoint lists $\mathsf{ls}(x, \mathsf{nil}) * \mathsf{ls}(y, \mathsf{nil}) * \mathsf{ls}(z, \mathsf{nil}) * \mathsf{ls}(u, \mathsf{nil}) * \mathsf{ls}(v, \mathsf{nil})$. The analysis will process each call to append in turn. The local heap of the first call to `append` expressed in terms of formal parameters is $\mathsf{ls}(a, \mathsf{nil}) * \mathsf{ls}(b, \mathsf{nil})$. As the analysis has no summaries for `append`, it will go on analyzing `append` on the local heap and will discover a post-heap $\mathsf{ls}(a, b) * \mathsf{ls}(b, \mathsf{nil})$. Hence, the heap at the return-site of the call will be $\mathsf{ls}(x, y) * \mathsf{ls}(y, \mathsf{nil}) * \mathsf{ls}(z, \mathsf{nil}) * \mathsf{ls}(u, \mathsf{nil}) * \mathsf{ls}(v, \mathsf{nil})$.

The local heap of the second call is again $\mathsf{ls}(a, \mathsf{nil}) * \mathsf{ls}(b, \mathsf{nil})$. The analysis will reuse the the summary discovered before and the heap at the return-site will be $\mathsf{ls}(u, v) * \mathsf{ls}(v, \mathsf{nil}) * \mathsf{ls}(x, y) * \mathsf{ls}(y, \mathsf{nil}) * \mathsf{ls}(z, \mathsf{nil})$.

The local heap of the third call expressed in terms of actual parameters is $\mathsf{ls}(x, y) * \mathsf{ls}(y, \mathsf{nil}) * \mathsf{ls}(z, \mathsf{nil})$. Here we have a cutpoint y. We replace it with a ghost variable Y, rename actuals to formals and obtain $\mathsf{ls}(a, Y) * \mathsf{ls}(Y, \mathsf{nil}) * \mathsf{ls}(b, \mathsf{nil})$ as a local heap. As there are no summaries for this local heap, the analysis will have to analyze `append` once again discovering a post-heap $\mathsf{ls}(a, Y) * \mathsf{ls}(Y, b) * \mathsf{ls}(b, \mathsf{nil})$.

Hence, the heap at the return-site of the call will be $\mathsf{ls}(u,v) * \mathsf{ls}(v,\mathsf{nil}) * \mathsf{ls}(x,y) * \mathsf{ls}(y,z) * \mathsf{ls}(z,\mathsf{nil})$. $\square$

4 Soundness

Following Lee, Yang, and Yi [9] we show the soundness of our analysis via translation to program proofs in separation logic; each run of the analysis determines a collection of proofs.

Suppose the analysis has not encountered a possible memory error and φ is the least function satisfying the equations in Fig. 2. Let $\psi_n(Q) = \{R \mid (Q,R) \in \varphi(n)\}$. Intuitively, for a node $n \in N$ and a symbolic heap $Q \in \mathcal{CSH}$, $\psi_n(Q)$ gives the set of symbolic heaps corresponding to the possible states at the node n reachable from the state $\{Q\}$ at the entry point to the procedure containing n.

Let s be a set of symbolic heaps. We define the separation logic formula representing this set as a disjunction of the formulæ representing the heaps in s: $\mathsf{means}(s) = \bigvee \{Q \mid Q \in s\}$. Note that $\mathsf{means}(\emptyset) = \mathsf{false}$. Throughout this section $\varGamma$ denotes the set of specifications of all the procedures obtained as a result of the analysis: $\varGamma = \{\{Q\}\ f(\vec{x})\ \{\mathsf{means}(\psi_{\mathsf{exit}(f)}(Q))\} \mid \psi_{\mathsf{entry}(f)}(Q) \neq \emptyset\}$.

Theorem 1. *Suppose the analysis succeeded, i.e. a possible memory error has not been encountered. Let C be a command, n_1 respectively n_2 be the nodes of the control-flow graph immediately preceding respectively following the command, and n_0 be the entry node of the procedure containing n_1 and n_2. Then for each symbolic heap Q such that $\psi_{n_0}(Q) \neq \emptyset$, the following judgment holds in separation logic: $\varGamma \vdash \{\mathsf{means}(\psi_{n_1}(Q))\}\ C\ \{\mathsf{means}(\psi_{n_2}(Q))\}$.*

The proof proceeds by induction on the structure of the command C. The cases for all the commands except for procedure call are similar to the ones in [9] and use the usual axioms and inference rules of separation logic [12]. The proof in the case of procedure call relies upon the proof rule LocalProcCallCut. Taking $n_1 = \mathsf{entry}(f)$ and $n_2 = \mathsf{end}(f)$ for each procedure f in the program in Theorem 1 and using RecProcDeclLocals, we obtain:

Corollary 1. *Let $\vec{D}$ be the list of all procedure declarations in the program, $\vec{v}$ the list of the formal parameters of* main*. Then if the analysis succeeds,*

$$\vdash \{\mathsf{means}(I)\}\ \texttt{letrec}\ \vec{D}\ \texttt{in}\ \mathsf{main}(\vec{v})\ \{\mathsf{means}(\psi_{\mathsf{end(main)}}(I))\}.$$

Corollary 1 justifies that the success of our analysis implies that the program is memory-safe and the computed post-condition is a valid one.

5 Experimental Results

In order to evaluate the performance of our analysis we have applied Summate to the list processing programs proposed in the literature, including those in [15] and [14]. The results are displayed in Table 1. The tests were performed on

Program	**Iterative**			**Recursive**		
	1	**2**	**3**	**1**	**2**	**3**
`create`	0.004	—	—	0.004	—	—
`deallocate`	0.004	0.005	0.007	0.003	0.005	0.005
`traverse`	0.004	0.005	0.011	0.004	0.005	0.008
`find`	0.004	0.008	0.022	0.005	0.010	0.034
`insert`	0.007	0.019	0.082	0.006	0.014	0.057
`remove` (element is not in the list)	0.006	0.025	0.115	0.006	0.010	0.022
`remove` (element is in the list)	0.004	0.006	0.007	0.004	0.005	0.006
`reverse` (acyclic list)	0.006	0.015	0.066	0.004	0.007	0.015
`reverse` (panhandle list)	0.030	0.032	0.175	—	—	—
`reverse_via_append`	0.006	0.020	0.142	0.006	0.013	0.047
`append`	0.005	0.015	0.066	0.004	0.008	0.021
`merge`	0.150	0.036	0.884	0.009	0.051	1.138
`splice`	0.010	0.024	0.041	0.006	0.010	0.019
`reverse8`	0.011	0.072	0.540	0.008	0.024	0.090

Table 1. Experimental results for iterative and recursive versions of simple list-processing functions with three different clients. The meaning of the programs is straightforward from their names (`reverse_via_append` reverses a list by appending its head to its reversed tail, `reverse8` reverses a list 8 times). Recursive `reverse` we used is not suitable for reversing a panhandle list. Times are given in seconds. SUMMATE did not require more than 600KB in any of these cases. `reverse_via_append` is a recursive function; as in [15] its "iterative" version uses an iterative version of `append`.

a 2GHz Pentium 4 Linux PC with 512MB of memory. Each program consists of a list-processing function and a client calling the function. We consider both iterative and recursive versions of functions. For each function except `create` we use three different clients. The first one corresponds to a cutpoint-free call. For reversing a panhandle list the second client calls the function so that the heap at the call-site is $\mathsf{ls}(x, y) * \mathsf{ls}(y, z') * \mathsf{ls}(z', y)$ (one cutpoint), the third client—$\mathsf{ls}(x, y) * \mathsf{ls}(y, z) * \mathsf{ls}(z, y)$ (two cutpoints). Here (and throughout this section) x, y, and z are local variables of the client. For `append`, `merge`, and `splice` the second client calls the function two times, hence, creating a cutpoint; e.g. for `append`:

```
xy = append(x, y);
xyz = append(xy, z)
```

The third client performs the call three times (thereby creating two cutpoints). For all the other programs the second client calls functions on a list in the case when a part of the list is shared (i.e. the heap at the call-site is $\mathsf{ls}(x, y) * \mathsf{ls}(y, \mathsf{nil}) * \mathsf{ls}(z, y)$). In this case one cutpoint is created in each call. The third client calls functions on a list in the case when it has two pointers to the middle of the list thereby modeling the situation shown in the example in Sect. 2.3 (i.e. the heap at the call-site is $\mathsf{ls}(x, y) * \mathsf{ls}(y, z) * \mathsf{ls}(z, \mathsf{nil})$). In this case two cutpoints are created in each call (except for `reverse8`, which creates an additional cutpoint

Program	**Iterative**		**Recursive**	
	Time (sec)	**Memory (KB)**	**Time (sec)**	**Memory (KB)**
`mergesort`	6.159	2288	0.211	368
`quicksort`	—	—	0.300	608
`insertionsort`	0.058	368	0.042	368
`tailsort`	0.008	368	0.007	368

Table 2. Experimental results for iterative and recursive versions of list sorting programs. `mergesort` and `tailsort` are recursive, `insertionsort`—iterative. As in [15] their "iterative" respectively "recursive" versions are obtained by using these functions with iterative respectively recursive versions of `insert` or `merge`.

because the program keeps track of the former head of the list, i.e. the tail of the reversed list).

For each program we were able to prove memory safety, absence of memory leaks, and the fact that the acyclicity of lists is preserved. In the cases when the caller had variables pointing to the middle of a list (i.e. we had calls with cutpoints), we have proved that the elements pointed to by the variables are still present in the resulting list in the order determined by the semantics of the list processing function. Besides, for each particular program the post-condition obtained as the result of the analysis could give some more information. For instance, we were able to prove that after `reverse` or `reverse_via_append` the head of the list moves to its tail, that the result of `append`, `merge`, `splice` still contains the heads of both source lists, and that `insert` actually inserts the element it is given into the list. `remove` was tested two times: in the case when the element being removed is present in the list, and in the case when it is not. In the former case the accurate treatment of cutpoints by our analysis allowed for proving that this element is deleted from the list.

In all these experiments the bound on the number of cutpoints was set to 3. A larger bound would not affect either precision or complexity of the analysis, since 3 is the maximal number of cutpoints created at a time in the programs considered. Setting the bound to a lower number makes the analysis less precise.

We also tested our implementation on list sorting programs. The results for them are shown in Table 2. The client in the programs calls a sorting function on a list once. For each of the programs we proved memory safety and preservation of the list acyclicity. `insertionsort` and `mergesort` have calls with a cutpoint. Accurate processing of this cutpoint by our analysis allowed us to prove that the head of the source list is present in the sorted list.

We have not done a systematic benchmarking of Summate against the other executable shape analysis tools in the same conditions. However, it is fair to say that Summate is at least competitive with the previously reported tools with respect to speed and memory, and clearly better with respect to accuracy:

– We observe a speed-up of up to 3 orders of magnitude in comparison with the numbers reported in [15] and [14]. However: this difference could likely be attributed to differences in machine configuration.
– Summate consumes less memory than previously reported local interprocedural shape analyses.
– Other than Summate, no local shape analysis tool accurately treats calls with multiple cutpoints.
– We have reported experimental results for programs operating on shared and cyclic data structures and programs that deallocate memory.

6 Conclusions

Summate implements an interprocedural shape analysis that makes use of spatial locality. Summate's abstraction simply tracks declarative representations of independent heap portions. Consequently, computing the effect of procedure calls and returns on an abstract state is easy.

Summate is the most accurate interprocedural shape analysis, due to its support of memory disposal, cyclic and shared data structures, and its handling of bounded numbers of cutpoints. To the best of our knowledge, no other tool precisely and efficiently supports these features combined. Furthermore: our interprocedural analysis can be formulated in terms of a handful of proof rules[5] and (unlike in previous efforts [13]) the proof of its soundness follows from them straightforwardly.

Related work. Hackett & Rugina [7], describe an analysis where transfer function computations benefit from using a form of local reasoning similar to ours. However, procedure summaries are represented in terms of global states and so analysis of procedure calls does not benefit from locality.

Several papers have described TVLA-based interprocedural shape analyses (i.e. [14, 13, 15]) where the procedure summaries operate on local heaps. However: in this work the analysis must dynamically find a way to divide the heap such that the overall shape is preserved. This is delicate with the TVLA reachability-based representation, since the separation significantly alters the represented reachability information. A consequence of this is that accurate treatment of cutpoints is expensive (e.g. in [14] all cutpoints must be abstracted away into a single cutpoint). Furthermore, the transfer function computation in this context is non-local and still expensive, because the analysis must propagate updates throughout the state. Summate's separated representation ensures that the difference between the states in the transfer function computation is limited. This is, in part, because instead of storing reachability information in a quickly queryable form, we only update information from which reachability could be computed. This is possible since the precise structure of abstracted heaps is known.

[5] This is similar in spirit to [1].

It is important to note the limitations of Summate's abstraction. Summate is much faster than tools such as TVLA [16], but in some ways can also be less general. Computing a transfer function requires case analysis. Since our representation specifies the precise structure of the abstracted heap, the case analysis phase of the transfer function can rely on it, and so the number of resulting possible cases is significantly lower. However, we use carefully hand-crafted inductive predicates and axioms. These inductive predicates and axioms only needed to be designed once [3], but a similar exercise must be done in order to support additional data types.

Acknowledgments The authors benefited greatly from discussions of this and related work with Dino Distefano, Peter O'Hearn, Tal Lev-Ami, Roman Manevich, Hongseok Yang, and Greta Yorsh. The authors would also like to thank Noam Rinetzky for providing the test programs used in [15] and the anonymous reviewers for useful suggestions.

References

[1] T. Amtoft and A. Banerjee. Information flow analysis in logical form. In *SAS*, volume 3148 of *LNCS*, pages 100–115, 2004.

[2] J. Berdine, C. Calcagno, and P. O'Hearn. Symbolic execution with separation logic. In *APLAS*, volume 3780 of *LNCS*, pages 52–68, 2005.

[3] J. Berdine, C. Calcagno, and P. W. O'Hearn. A decidable fragment of separation logic. In *FSTTCS*, volume 3328 of *LNCS*, 2004.

[4] J. Berdine, B. Cook, D. Distefano, and P. W. O'Hearn. Automatic termination proofs for programs with shape-shifting heaps. In *CAV*, 2006.

[5] S. A. Cook. Soundness and completeness of an axiomatic system for program verification. *SIAM J. on Computing*, 7:70–90, 1978.

[6] D. Distefano, P. W. O'Hearn, and H. Yang. A local shape analysis based on separation logic. In *TACAS*, volume 3920 of *LNCS*, pages 287–302, 2006.

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

[8] C. A. R. Hoare. Procedures and parameters: An axiomatic approach. In *Symposium on the Semantics of Algorithmic Languages*, pages 102–116, 1971.

[9] O. Lee, H. Yang, and K. Yi. Automatic verification of pointer programs using grammar-based shape analysis. In *ESOP*, volume 3444 of *LNCS*, pages 124–140, 2005.

[10] P. O'Hearn, J. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *CSL*, volume 2142 of *LNCS*, pages 1–19, 2001.

[11] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL*, pages 49–61, 1995.

[12] J. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74, 2002.

[13] N. Rinetzky, J. Bauer, T. Reps, M. Sagiv, and R. Wilhelm. A semantics for procedure local heaps and its abstractions. In *POPL*, pages 296–309, 2005.

[14] N. Rinetzky, M. Sagiv, and E. Yahav. Interprocedural functional shape analysis using local heaps. Tech. Rep. 26, Tel Aviv Univ., Nov. 2004.

$$\textsc{ProcCall}\quad \frac{}{\Gamma\,, \{P\}\ f(\vec{x})\ \{Q\} \vdash \{P\}\ f(\vec{x})\ \{Q\}} \qquad \textsc{Subst}\quad \frac{\{P\}\ C\ \{Q\}}{\{P\sigma\}\ C\sigma\ \{Q\sigma\}}\ \begin{array}{l} C \text{ modifies } x \text{ implies} \\ x\sigma \in \mathrm{Var}\setminus \bigcup_{y\neq x} \mathit{fv}(y\sigma)\end{array}$$

$$\textsc{Conseq}\quad \frac{P \vDash R \qquad \{R\}\ C\ \{S\} \qquad S \vDash Q}{\{P\}\ C\ \{Q\}} \qquad \textsc{VarDecl}\quad \frac{\{P\}\ C\ \{Q\}}{\{P\}\ \texttt{local}\ \vec{x}\ \texttt{in}\ C\ \{Q\}}\ \vec{x} \cap \mathit{fv}(P,Q) = \emptyset$$

$$\textsc{RecProcDecl}\quad \frac{\Gamma\,, \{P\}\ f(\vec{x})\ \{Q\} \vdash \{P\}\ C\ \{Q\} \qquad \Gamma\,, \{P\}\ f(\vec{x})\ \{Q\} \vdash \{R\}\ C'\ \{S\}}{\Gamma \vdash \{R\}\ \texttt{letrec}\ f(\vec{x})\ \{C\}\ \texttt{in}\ C'\ \{S\}}$$

Fig. 3. Standard rules of Hoare logic

[15] N. Rinetzky, M. Sagiv, and E. Yahav. Interprocedural shape analysis for cutpoint-free programs. In *SAS*, volume 3672 of *LNCS*, pages 284–302, 2005.

[16] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM TOPLAS*, 24(3):217–298, 2002.

A Proof Rules

The proof rules used by our interprocedural analysis can be derived from the Frame rule and Hoare logic rules, which are listed in Fig. 3. The corresponding derivations, which also involve some predicate calculus, are given in Fig. 4.

B Intraprocedural Analysis

In this section we describe Summate's *intra*procedural analysis. This analysis is essentially the same as that of [6, 4].

Transfer functions are defined by this analysis in terms of the symbolic execution relation $\leadsto$, the rearrangement relation $\rightarrow_E$, and the abstraction relation $\rightarrow^*$. Each individual concrete state can be expressed exactly by a symbolic heap, i.e. there is a subset of symbolic heaps which are simply different syntax for concrete states. In the usual concrete semantics, each command only accesses a small portion of the state: its *footprint*. From this perspective, symbolic execution ($\leadsto$) expresses the usual concrete semantics of commands[6] in terms of symbolic heaps, where the footprint of the command is expressed as one of the formulæ that is alternate syntax for a concrete state. The task of rearrangement ($\rightarrow_E$) is then to transform an arbitrary symbolic heap, via case analysis, into a set of symbolic heaps where the footprint of the next command is concrete. Abstraction ($\rightarrow$) then takes the symbolic heaps resulting from symbolic execution and maps them into a finite subdomain of symbolic heaps, ensuring that fixed-point computations converge.

[6] Except that the rule for `dispose` does not quite yield the strongest postcondition.

RecProcDeclLocals

$$\dfrac{\dfrac{\dfrac{\dfrac{\Gamma' \vdash \{P\}\ C\ \{T\} \quad \dfrac{}{T \vDash \exists\vec{y}.\,T}}{\Gamma' \vdash \{P\}\ C\ \{\exists\vec{y}.\,T\}}\,\textsc{Conseq}}{\Gamma' \vdash \{P\}\ \texttt{local}\ \vec{y}\ \texttt{in}\ C\ \{\exists\vec{y}.\,T\}}\,\textsc{VarDecl} \qquad \exists\vec{y}.\,T \vDash Q}{\Gamma' \vdash \{P\}\ \texttt{local}\ \vec{y}\ \texttt{in}\ C\ \{Q\}}\,\textsc{Conseq} \qquad \Gamma' \vdash \{R\}\ C'\ \{S\}}{\Gamma \vdash \{R\}\ \texttt{letrec}\ f(\vec{x})\ \{\texttt{local}\ \vec{y}\ \texttt{in}\ C\}\ \texttt{in}\ C'\ \{S\}}$$

where $\Gamma' = \Gamma\,,\{P\}\ f(\vec{x})\ \{Q\}$ and the first rule applied is RecProcDecl;

LocalProcCall

$$\dfrac{S \vDash P\sigma * R \qquad \dfrac{\dfrac{\dfrac{}{\Gamma' \vdash \{P\}\ f(\vec{x})\ \{Q\}}\,\textsc{ProcCall}}{\Gamma' \vdash \{P\sigma\}\ f(\vec{x}\sigma)\ \{Q\sigma\}}\,\textsc{Subst}}{\Gamma' \vdash \{P\sigma * R\}\ f(\vec{x}\sigma)\ \{Q\sigma * R\}}\,\textsc{Frame} \qquad Q\sigma * R \vDash T}{\Gamma' \vdash \{S\}\ f(\vec{x}\sigma)\ \{T\}}\,\textsc{Conseq}$$

where $\Gamma' = \Gamma\,,\{P\}\ f(\vec{x})\ \{Q\}$;

LocalProcCallCut

$$\dfrac{\dfrac{S \vDash P' * R \qquad \dfrac{\dfrac{\dfrac{}{P' \vDash \exists\vec{c}.\,P'} \qquad \exists\vec{c}.\,P' \vDash P\sigma}{P' \vDash P\sigma}}{P' * R \vDash P\sigma * R}}{S \vDash P\sigma * R} \qquad Q\sigma * R \vDash T}{\Gamma\,,\{P\}\ f(\vec{x})\ \{Q\} \vdash \{S\}\ f(\vec{x}\sigma)\ \{T\}}\,\textsc{LocalProcCall}$$

Fig. 4. Derivations of procedure call and declaration rules

The definition of these relations asks several types of questions about symbolic heaps: entailment of an equality ($Q \vdash E{=}F$), or of a disequality ($Q \vdash E{\neq}F$), inconsistency ($Q \vdash \mathsf{false}$), or testing if a location is guaranteed to be allocated ($Q \vdash \mathit{allocated}(E)$). We also sometimes ask the negations of these questions. Decision procedures for these queries are defined in [6].

Symbolic Execution ($\rightsquigarrow$). The symbolic execution relation captures the effect of executing an atomic command from a symbolic heap. That is, $Q_0 \overset{C}{\rightsquigarrow} Q_1$ means that Q_1 over-approximates the concrete states that can result from executing C on states satisfying Q_0. The symbolic execution rules are reported in Fig. 5.

$$\begin{array}{rcll}
Q & \overset{\texttt{skip}}{\rightsquigarrow} & Q & \textsc{Skip} \\
Q & \overset{x:=E}{\rightsquigarrow} & x{=}E[x'/x] \wedge Q[x'/x] & \textsc{Assign} \\
Q & \overset{x:=\texttt{new}()}{\rightsquigarrow} & Q[x'/x] * x{\mapsto}y' & \textsc{New} \\
Q & \overset{\texttt{assume}(E=F)}{\rightsquigarrow} & Q \wedge E = F \text{ if } Q \nvdash E{\neq}F & \textsc{AssumeT} \\
Q & \overset{\texttt{assume}(E\neq F)}{\rightsquigarrow} & Q \text{ if } Q \nvdash E = F \text{ and } Q \nvdash \mathsf{false} & \textsc{AssumeF} \\
Q * E{\mapsto}F & \overset{\texttt{dispose}(E)}{\rightsquigarrow} & Q & \textsc{Dispose} \\
Q * E{\mapsto}F & \overset{x:=[E]}{\rightsquigarrow} & x{=}F[x'/x] \wedge (Q * E{\mapsto}F)[x'/x] & \textsc{Load} \\
Q * E{\mapsto}F & \overset{[E]:=G}{\rightsquigarrow} & Q * E{\mapsto}G & \textsc{Store}
\end{array}$$

Fig. 5. Symbolic Execution ($\rightsquigarrow$). Here x', y' are globally fresh ([2] allows more local freshness constraints).

$$\begin{array}{rcll}
Q * F{\mapsto}G & \rightarrow_E & Q * E{\mapsto}G & \text{if } Q \vdash E{=}F \quad \textsc{Switch} \\
Q * \mathsf{ls}(F,G) & \rightarrow_E & Q * E{\mapsto}G & \text{if } Q \vdash E{=}F \quad \textsc{Unroll1} \\
Q * \mathsf{ls}(F,G) & \rightarrow_E & Q * E{\mapsto}x' * \mathsf{ls}(x',G) & \text{if } Q \vdash E{=}F \text{ and } x' \text{ fresh} \quad \textsc{Unroll}{>}1 \\
Q & \rightarrow_E & \top & \text{if } Q \nvdash \mathit{allocated}(E) \quad \textsc{Crash}
\end{array}$$

Fig. 6. Rearrangement ($\rightarrow_E$)

Rearrangement ($\rightarrow_E$). Symbolic execution does not operate on arbitrary prestates. For instance, Load requires that the source heap cell be explicitly known. In order to put symbolic heaps into the form required for symbolic execution of a command, we use the rearrangement relation $\rightarrow_E$, defined by the axioms shown in Fig. 6. When rearrangement fails to reveal the required location E, it indicates a potential memory safety violation and returns $\top$.

Abstraction ($\rightarrow$). Abstraction is accomplished by certain separation logic implications that rewrite a symbolic heap to a logically weaker one. The abstraction relation on symbolic heaps $Q_0 \rightarrow Q_1$ is defined by the axioms shown in Fig .7.

Note that the heap abstraction is defined solely in terms of the representation of heaps, i.e., the dynamic information the analysis knows. The precision of this abstraction immaterializes static information about the program text used by shallow analyses, such as allocation-sites.

We call a symbolic heap Q canonical if it is maximally abstracted, i.e. $Q \nrightarrow$ and denote the set of all canonical symbolic heaps with $\mathcal{CSH}$. A canonicalization function is defined in [6]. This function, can, is based on a fixed sequence of abstraction axiom applications, and transforms a symbolic heap to a canonical symbolic heap abstracting it, i.e. $Q \rightarrow^* \mathsf{can}(Q)$ and $\mathsf{can}(Q) \nrightarrow$.

$$\begin{array}{rcll}
z'{=}E \wedge Q & \to & Q[E/z'] & \textsc{Subst} \\
Q * H_0(E,x') * H_1(x',F) & \to & Q * \mathsf{ls}(E,\mathsf{nil}) & \text{if } Q \vdash F{=}\mathsf{nil} \\
 & & & \textsc{AppendLsNil} \\
Q * H_0(E,x') * H_1(x',F_0) * H_2(F_1,G) & \to & Q * \mathsf{ls}(E,F_0) * H_2(F_1,G) & \text{if } Q \vdash F_0{=}F_1 \\
 & & & \textsc{AppendLsGuard} \\
Q * H(x',E) & \to & Q * \mathsf{junk} & \textsc{Junk} \\
Q * H_0(x',y') * H_1(y',x') & \to & Q * \mathsf{junk} & \textsc{JunkCycle}
\end{array}$$

Fig. 7. Abstraction ($\to$). Here $H(E,F)$ stands for either $E{\mapsto}F$ or $\mathsf{ls}(E,F)$; and x', y' do not occur other than where explicitly indicated.

Transfer Functions. The transfer function $\mathcal{A}_C$ for an atomic command C transforms a given symbolic heap to either $\top$ (indicating a possible crash) or a set of consistent canonical symbolic heaps. Transfer functions are defined separately for safe commands S and unsafe ones $A(E)$ in the following way:

$$\mathcal{A}_S(Q_0) = \{\mathsf{can}(Q_1) \mid Q_0 \overset{S}{\rightsquigarrow} Q_1\}$$

$$\mathcal{A}_{A(E)}(Q_0) = \begin{cases} \top, & \text{if } Q \to_E \top \\ \{\mathsf{can}(Q_2) \mid \exists Q_1.\, Q_0 \to_E Q_1 \wedge Q_1 \overset{A(E)}{\rightsquigarrow} Q_2\}, & \text{otherwise} \end{cases}$$

Example 4. Suppose we want to compute the value of the transfer function for the command `x = [x]` on the symbolic heap $\mathsf{ls}(x,\mathsf{nil})$. The rearrangement phase will transform the heap into two symbolic heaps $x{\mapsto}\mathsf{nil}$ and $x{\mapsto}x' * \mathsf{ls}(x',\mathsf{nil})$ thereby making the information that x is allocated in the heap explicit. The symbolic execution phase will then symbolically simulate the effect of the command on the heaps producing $x = \mathsf{nil}$ and $x = x' \wedge x''{\mapsto}x' * \mathsf{ls}(x',\mathsf{nil})$. Finally, the abstraction phase will leave the first heap unchanged and will canonicalize the second heap to $\mathsf{junk} * \mathsf{ls}(x,\mathsf{nil})$. Hence, the value of the transfer function is $\{x = \mathsf{nil},\ \mathsf{junk} * \mathsf{ls}(x,\mathsf{nil})\}$. $\square$

Finiteness. A key property of the abstract domain $\mathcal{CSH}$ proved in [6] is that the domain of consistent and canonical symbolic heaps $\{Q \mid Q \nvdash \mathsf{false} \wedge Q \not\to\}$ over a finite number of unquantified variables is finite.