Verifying Programs with Unknown Calls in Separation Logic: Technical Report

Chenguang Luo[1], Florin Craciun[1], Shengchao Qin[1], Guanhua He[1], and Wei-Ngan Chin[2]

[1] Durham University, Durham DH1 3LE, UK
[2] National University of Singapore
{chenguang.luo,florin.craciun,shengchao.qin,guanhua.he}@durham.ac.uk
chinwn@comp.nus.edu.sg

Abstract. We study the automated verification of pointer safety for heap-manipulating imperative programs with unknown procedure calls. Given a Hoare-style partial correctness specification $S = \{\mathsf{Pre}\}\ C\ \{\mathsf{Post}\}$ in separation logic, where the program C contains calls to some unknown procedure U, we infer a specification S_U for the unknown procedure U from the calling contexts. We show that the problem of verifying the program C against the specification S can be safely reduced to the problem of proving that the procedure U (once its code is available) meets the derived specification S_U. The expected specification S_U for the unknown procedure U is automatically calculated using an abduction-based shape analysis adapted from the bottom-up shape analysis by Calcagno et al. [5]. We have also implemented a prototype system to validate the viability of our approach.

Key words: Inference, verification, abduction, separation logic

1 Introduction

While automated verification of memory safety remains a big challenge [12, 19], especially for substantial system software such as the Linux distribution and device drivers, significant advances have been seen recently on the automated verification of pointer safety for such software [5, 20]. The abduction-based compositional shape analysis [5] is able to calculate the pre/post specifications for procedures in a bottom-up approach, based on the call-dependency graph. It can verify the pointer safety of a large portion of the Linux kernel and many device drivers manipulating shared mutable data structures. One issue that has not been dealt within their work is unknown procedure calls. In their SpaceInvader tool, unknown calls are currently ignored and replaced by the empty statement skip during the verification. This may lead to imprecise or unsound results in general. Our work here is to investigate this issue carefully and provide a better solution to the verification of pointer safety of programs with unknown calls.

Automated program verifiers usually require to have access to the entire given program which, in practice, may not be completely available for various reasons. For instance, some programs (e.g. in C) may contain unknown calls that correspond to function pointers, some programs (e.g. in OO) may contain calls to interface methods whose actual implementations may not be available statically, or some programs may have calls to library procedures whose codes are not available during verification. Other possible scenarios lie in mobile code as well as software upgrading, where program fragments may be unavailable at verification time. To deal with the verification of programs with unknown procedure calls, current automated program verifiers either

– ignore the unknown procedure calls, e.g. replacing them by skip [5], which can be unsound in general; or
– assume that the program and the unknown procedure have disjoint memory footprints so that the unknown call can be safely ignored due to the hypothetical frame rule [17]; however, this assumption does not hold in most cases; or
– use specification mining [1] to discover possible specifications for the (unknown part of the) program, which is performed dynamically by observing the execution traces and is not likely to be exhaustive for all possible program behaviors; or
– take into account all possible implementations for the unknown procedure [9, 11]. In general, there can be too many such candidates, making the verification almost impossible at compile time; or
– simply stop at the first unknown procedure call and provide an incomplete verification, which is obviously undesirable.

Approach and contributions. We propose a different approach in this paper to the verification of programs with unknown procedure calls. Given a specification $S = \{\mathsf{Pre}\}\ C\ \{\mathsf{Post}\}$ for the program C containing calls to an unknown procedure U, our solution is to proceed with the verification for the known fragments of C, and at the same time infer a specification S_U that is expected for the unknown procedure U based on the calling context(s). The problem of verifying the program C against the specification S can now be safely reduced to the problem of verifying the procedure U against the inferred specification S_U, provided that the verification of the known fragments does not cause any problems. The inferred specification is subject to a later verification when an implementation or a specification for the unknown procedure becomes available (e.g. at loading time in Java).

The intuition of our method to infer the unknown procedure's specification can be divided into two steps. The first step is to analyze the code before the unknown procedure call to discover its precondition. The second is to analyze the code after the unknown call in order to discover its postcondition. For the second step, one might suggest to use a deductive backwards analysis, starting from the postcondition of the program being verified, to derive an expected postcondition for the unknown procedure. We cannot follow this suggestion due to a technical challenge: backwards reasoning over the separation domain is simply too costly to implement ([5]). Therefore, we exploit an abductive forwards reasoning [5, 10] to derive the unknown procedure's postcondition, and in this way the verification can be accomplished.

Our paper makes the following technical contributions:

- We propose a novel framework in separation logic for the verification of pointer safety for programs with unknown calls. It differs from the aforementioned approaches, and to some extent solves the problems some of them may cause.
- The *top-down* feature of our approach can potentially benefit the general software development process. Given the specification for the caller procedure, it can be used to infer the specification for the callee procedures.
- We have enhanced the abduction mechanism, resulting in an improved algorithm for the verification use.
- We have successfully incorporated the call-by-reference mechanism into the forward analyses in the separation domain.
- We have built a prototype system to test the viability of our approach and we have conducted some initial experimental studies to evaluate the precision of the results.

Outline. Section 2 employs a motivating example to informally illustrate our main approach. Section 3 presents the programming language and the abstract domain for our analysis. Section 4 introduces our improved abductive reasoning compared with previous work [5]. Section 5 defines the two abstract semantics used in our verification. Section 6 depicts our verification algorithms. Experimental results are shown in Section 7, followed by some concluding remarks.

2 A Motivating Example

In this section, we illustrate informally, via an example, how our analysis infers the specification for an unknown procedure. Our analysis makes use of a separation domain similar to that in the SpaceInvader tool [5, 8]. To keep the presentation simple, we use a small imperative language with both call-by-value and call-by-reference parameters for procedures. Formal details about the abstract domain and the language will be given in Section 3.

Example 1 (Motivating example). Our goal is to verify the procedure `findLast` against the given pre/post specifications shown in Figure 1. The data structure `node` $\{$ `int data; node next` $\}$ defines a node in a linked list. The predicate $\mathsf{ls}(x, y)$ used in the pre/post specifications as well as other places denotes a (possibly empty) list segment referred to by x and ended with the pointer y (i.e., y denotes the `next` field of the last node).

According to the given specification, the procedure `findLast` takes in a non-empty linked list x and stores a reference to the last node of the list in the call-by-reference parameter z. Here we group call-by-value and call-by-reference parameters together and use semicolon `;` to separate them. Note that `findLast` calls an unknown procedure `unkProc` at line `4`.

```
// Given Specification:
     // $\mathsf{Pre}_{\texttt{findLast}} := \mathsf{ls}(x, \texttt{null}) \wedge x{\neq}\texttt{null}$
     // $\mathsf{Post}_{\texttt{findLast}} := \mathsf{ls}(x, z) * z{\mapsto}\texttt{null}$
void findLast(node x; ref node z) {
0   node w, y;
0a   // $\Delta := \mathsf{Pre}_{\texttt{findLast}}$                    $\Delta \vdash x{\neq}\texttt{null}$
1   w := x.next;
1a   // $\Delta := x{\mapsto}w * \mathsf{ls}(w, \texttt{null}) \wedge x{\neq}\texttt{null}$
2   if (w == null) z := x;
2a   // $\Delta := x{\mapsto}w \wedge x{\neq}\texttt{null} \wedge w{=}\texttt{null} \wedge z{=}x$
2b   // $\exists w, y \cdot \Delta \vdash \mathsf{Post}_{\texttt{findLast}}$
3   else {
3a   // $\Delta := x{\mapsto}w * \mathsf{ls}(w, \texttt{null}) \wedge x{\neq}\texttt{null} \wedge w{\neq}\texttt{null}$
3b   // $H := \mathsf{Local}(\Delta, \{x, y\}) := x{\mapsto}w * \mathsf{ls}(w, \texttt{null}) \wedge x{\neq}\texttt{null} \wedge w{\neq}\texttt{null}$
3c   // $\mathsf{R}_0 := \mathsf{Frame}(\Delta, \{x, y\}) := \mathsf{emp} \wedge x{\neq}\texttt{null} \wedge w{\neq}\texttt{null}$
4   unkProc(x; y);
4a   // $\Delta := \mathsf{R}_0 * \mathsf{M}_0$      $\mathsf{M}_0 := (\mathsf{emp} \wedge x{=}a \wedge y{=}b)$      $\mathsf{M} := \mathsf{M}_0$
4b   // $\Delta \nvdash [y/x]\ \mathsf{Pre}_{\texttt{findLast}}$
4c   // $\Delta * [\mathsf{M}_1] \rhd [y/x]\ \mathsf{Pre}_{\texttt{findLast}}$
4d   // $\mathsf{M}_1 := \mathsf{ls}(y, \texttt{null}) \wedge y{\neq}\texttt{null}$         $\mathsf{M} := \mathsf{M} * \mathsf{M}_1$
4e   // $\Delta * \mathsf{M}_1 \vdash [y/x]\ \mathsf{Pre}_{\texttt{findLast}} * \mathsf{R}_1$      $\mathsf{R}_1 := \mathsf{emp} * x{=}a \wedge y{=}b$
5   findLast(y; z);
5a   // $\Delta := ([y/x]\ \mathsf{Post}_{\texttt{findLast}}) * \mathsf{R}_1$
5b   // $\Delta \nvdash \mathsf{Post}_{\texttt{findLast}}$
5c   // $\Delta * [\mathsf{M}_2] \rhd \mathsf{Post}_{\texttt{findLast}}$       $\mathsf{M}_2 := \mathsf{ls}(x, y)$       $\mathsf{M} := \mathsf{M} * \mathsf{M}_2$
6   } }
6a // $\mathsf{Pre}_{\texttt{unkProc}} := \exists w \cdot [a/x, b/y]\ H$
6b // $\mathsf{Post}_{\texttt{unkProc}} := [a/x, b/y]\ \mathsf{M}$
```

Fig. 1. Verification of `findLast` calling an unknown procedure `unkProc`.

We conduct a symbolic execution([3, 16]) on the procedure body starting with the precondition $\mathsf{Pre}_{\texttt{findLast}}$ (line `0a`). The results of our analysis (e.g. the abstract states) are marked as comments in the code. The analysis carries on as a standard forward shape analysis until it reaches the unknown procedure call at line `3`.

At line `3`, the current symbolic heap Δ is split into two disjoint parts: the local part H (line `3b`) that is depended on, and possibly mutated by, the unknown procedure; and the frame part R_0 (line `3c`) that is not accessed by the unknown procedure. Intuitively, the local part of a symbolic heap w.r.t. a set of variables X is the part of the heap reachable from variables in X (together with the aliasing information); while the frame part denotes the unreachable heap part (together with the aliasing information). For example, for a symbolic heap $\mathsf{ls}(x, w) * \mathsf{ls}(y, z) * \mathsf{ls}(z, \texttt{null}) \wedge w{=}z$, its local part w.r.t. $\{x\}$ is $\mathsf{ls}(x, w) * \mathsf{ls}(z, \texttt{null}) \wedge w{=}z$, and its frame part w.r.t. $\{x\}$ is $\mathsf{ls}(y, z) \wedge w{=}z$. We will have their formal definitions in Section 6.

We take H (line `3b`) as a crude precondition for the unknown procedure, since it denotes the symbolic heap that is accessible, and hence potentially usable, by the unknown call. The frame part R_0 is not touched by the unknown call and will remain in the post-state, as shown in line `4a`.

At line `4a`, the abstract state after the unknown call consists of two parts: one is the aforesaid frame R_0 not accessed by the call, and the other is due to the procedure's postcondition which is unfortunately not available. Our next step is to discover the postcondition by examining the code fragment after the unknown call with an abductive reasoning (line `4a` to `5c`).

Initially, we assume the unknown procedure having an empty heap M_0 as its postcondition[1], and gradually discover the missing parts of the postcondition during the symbolic execution of the code fragment after the unknown call. To do that, our analysis keeps track of a pair (Δ, M) at each program point, where Δ refers to the current heap state, and M denotes the expected postcondition discovered so far for the unknown procedure. The notations M_i are used to represent parts of the discovered postcondition.

At line `5`, the procedure `findLast` is called recursively. Since the current heap state does not satisfy the precondition of `findLast` (as shown in line `4b`), the verification fails. However, this is not necessarily due to a program error; it may be due to the fact that the unknown call's postcondition is still unknown. Therefore, our analysis performs an abductive reasoning (line `4c`) to infer the missing part M_1 for Δ such that $\Delta * \mathsf{M}_1$ entails the precondition of `findLast` w.r.t. some substitution $[y/x]$. As shown in line `4d`, M_1 is inferred to be $\mathsf{ls}(y, \texttt{null}) \wedge y{\neq}\texttt{null}$, which is accumulated into M as part of the expected postcondition of the unknown procedure. (We will explain the details for abductive reasoning in Sec 3.) Now the heap state combined with the inferred M_1 meets the precondition of the procedure `findLast`, and also generates a residual frame heap R_1 (line `4e`).

The heap state Δ immediately after the recursive call (line `5a`) is formed by `findLast`'s postcondition and the frame R_1, and it is expected to establish the postcondition of `findLast` for the overall verification to succeed. However, it does not (as shown in line `5b`). Again this might be due to the fact that part of the unknown call's postcondition is still missing. Therefore, we perform another abductive reasoning (line `5c`) to infer the missing M_2 as follows:

$$\mathsf{ls}(y, z) * z{\mapsto}\texttt{null} \wedge x{=}a \wedge y{=}b * [\mathsf{M}_2] \triangleright \mathsf{ls}(x, z) * z{\mapsto}\texttt{null}$$

In this case, our abductor returns $\mathsf{M}_2 := \mathsf{ls}(x, y)$ as the result which is then added into M by separation conjunction, as shown in line `5c`.

Finally, we generate the expected pre/post-specification for the unknown procedure (lines `6a` and `6b`). The precondition is obtained from the local pre-state of the unknown call, H, by replacing all variables that are aliases of a (or b) with the formal parameter a (or b). The postcondition is obtained from the accumulated abduction result, M, after performing the same substitution. Our

[1] Note that we introduce fresh logical variables a and b to record the value of x and y when `unkProc` returns.

discovered specification for the unknown procedure `unkProc(a;b)` is:

$$\begin{aligned} \mathsf{Pre}_{\texttt{unkProc}} &:= \exists w \cdot a \mapsto w * \mathsf{ls}(w, \texttt{null}) \wedge a \neq \texttt{null} \wedge w \neq \texttt{null} \\ \mathsf{Post}_{\texttt{unkProc}} &:= \mathsf{ls}(a, b) * \mathsf{ls}(b, \texttt{null}) \wedge b \neq \texttt{null} \end{aligned}$$

The entire program is verified, if `unkProc` meets the derived specification.

3 Abstract Domain and Programming Language

In this section, we first depict the syntax of a language in which programs may invoke unknown procedures, and then present the abstract domain for our analysis.

To focus only on key issues, we use a simple imperative language:

$$\begin{aligned} E \quad &=_{df}\ x \mid \texttt{null} \\ \mathsf{b} \quad &=_{df}\ E_1{=}E_2 \mid E_1{\neq}E_2 \\ A[E] &=_{df}\ [E] := E_1 \mid \mathsf{dispose}(E) \mid x := [E] \\ A \quad &=_{df}\ x := E \mid x := \mathsf{new}(E) \mid \mathsf{skip} \\ C \quad &=_{df}\ A[E] \mid A \mid f(\boldsymbol{x};\boldsymbol{y}) \mid C_1; C_2 \mid \\ &\qquad \mathsf{if\ b\ then}\ C_1\ \mathsf{else}\ C_2\ \mathsf{fi} \mid \mathsf{while\ b\ do}\ C\ \mathsf{od} \\ U \quad &=_{df}\ \mathsf{unkFn}(\boldsymbol{x};\boldsymbol{y}) \mid \{\ \mathsf{unkFn}(\boldsymbol{x}_0;\boldsymbol{y}_0); C_1; \\ &\qquad \mathsf{unkFn}(\boldsymbol{x}_1;\boldsymbol{y}_1); C_2; \ldots; C_{n-1}; \mathsf{unkFn}(\boldsymbol{x}_n;\boldsymbol{y}_n)\} \mid \\ &\qquad \mathsf{if\ b\ then}\ V\ \mathsf{else}\ C\ \mathsf{fi} \mid \mathsf{if\ b\ then}\ C\ \mathsf{else}\ V\ \mathsf{fi} \mid \\ &\qquad \mathsf{if\ b\ then}\ V_1\ \mathsf{else}\ V_2\ \mathsf{fi} \mid \mathsf{while\ b\ do}\ V\ \mathsf{od} \\ V \quad &=_{df}\ \{\ C_1; U_{(\boldsymbol{x}_1;\boldsymbol{y}_1)}; C_2\ \}_{(\boldsymbol{x}_0;\boldsymbol{y}_0)} \\ P \quad &=_{df}\ \cdot \mid P; f(\boldsymbol{x}; \mathsf{ref}\ \boldsymbol{y})\ \{\ \mathsf{local}\ \boldsymbol{z}; C\ \} \mid \\ &\qquad P; f(\boldsymbol{x}; \mathsf{ref}\ \boldsymbol{y})\ \{\ \mathsf{local}\ \boldsymbol{z}; V\ \} \end{aligned}$$

Note that expressions (E) are program variables to record memory locations in the heap, and all program variables are assumed of the same type, reference. The language has both heap sensitive ($A[E]$) and heap insensitive (A) atomic commands. The former requires access to the heap location referred to by E, while the latter does not. The command C contains calls to known procedures only, while the commands U and V comprise unknown procedure calls. Note also that our language allows both call-by-value and call-by-reference parameters for procedures, and for convenience, we group call-by-value parameters on the left and call-by-reference ones on the right, separated by a semicolon.

The unknown commands U and V specify the possible scenarios in which an unknown call ($\mathsf{unkFn}(\boldsymbol{x};\boldsymbol{y})$) may occur. Note that U and V may be annotated with two sets of variables, e.g. $(\boldsymbol{x}_0, \boldsymbol{y}_0)$ in the definition of V, where $\boldsymbol{x}_0$ denotes variables that can be accessed, but cannot be modified by V, and where $\boldsymbol{y}_0$ denotes variables that may be mutated by V. The same annotation applies to U. These annotations can be obtained automatically via a pre-processing phase of the analysis.

Example 2 (Unknown procedure and unknown block). The parse tree of `findLast`'s body (omitting local variable definition) in our motivating example is as in Figure 2.

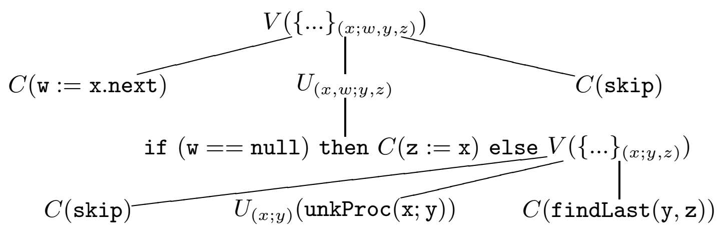
$V(\{...\}_{(x;w,y,z)})$
$C(\texttt{w := x.next})$
$U_{(x,w;y,z)}$
$C(\texttt{skip})$
$\texttt{if (w == null) then } C(\texttt{z := x}) \texttt{ else } V(\{...\}_{(x;y,z)})$
$C(\texttt{skip})$
$U_{(x;y)}(\texttt{unkProc(x; y)})$
$C(\texttt{findLast(y, z)})$

Fig. 2. The parse tree of `findLast`'s body.

A program P is composed of several procedures (with one of them as the entry method). A procedure can be totally known to the verifier (so that its body is composed by C), or it may contain an unknown block V.

We use a *memory model* similar to the standard one for separation logic [18]:

$$\begin{array}{lll}\mathsf{Stack} & =_{df} & (\mathsf{Var} \cup \mathsf{LVar} \cup \mathsf{SVar}) \rightarrow \mathsf{Val} \\ \mathsf{Val} & =_{df} & \mathsf{Loc} \cup \{\texttt{null}\} \\ \mathsf{Heap} & =_{df} & \mathsf{Loc} \rightharpoonup \mathsf{Val} \\ \mathsf{State} & =_{df} & \mathsf{Stack} \times \mathsf{Heap}\end{array}$$

The slight difference is that we have three (disjoint) set of variables: a finite set of program variables $\mathsf{Var} = \{x, y, ..\}$, logical variables $\mathsf{LVar} = \{x', y', ..\}$, and the special (logical) variables $\mathsf{SVar} = \{a, b, ..\}$, with the last set reserved for unknown procedures for specification purpose. As usual, Loc is a countably infinite set of locations, subsumed by the set of values Val. The function Heap denotes a partial mapping from locations to values and a program state is a pair of stack and heap.

An abstract (program) state in our analysis is a *symbolic heap* representing a set of concrete heaps. It is defined as follows:

$$\begin{array}{lll}E & =_{df} x \mid x' \mid a & \text{Expressions} \\ \Pi & =_{df} E_1{=}E_2 \mid E_1{\neq}E_2 \mid \texttt{true} \mid \Pi_1{\wedge}\Pi_2 & \text{Pure formulae} \\ \mathsf{B}(E_1, E_2) & =_{df} E_1{\mapsto}E_2 \mid \mathsf{ls}(E_1, E_2) & \text{Basic separation predicates} \\ \Sigma & =_{df} \mathsf{B}(E_1, E_2) \mid \texttt{true} \mid \mathsf{emp} \mid \Sigma_1 * \Sigma_2 & \text{Separation formulae} \\ \Delta & =_{df} \Pi \wedge \Sigma & \text{Quantifier-free symbolic heaps} \\ H & =_{df} \exists \boldsymbol{x}.\Delta & \text{Symbolic heaps}\end{array}$$

The expressions (x, x' and a) correspond to the three kinds of variables (program, logical and special ones). Pure formulae Π express the aliasing information among expressions. The basic separation predicate $\mathsf{B}(E_1, E_2)$ denotes either a singleton heap or a list segment [8,16]. A (possibly empty) list segment is inductively defined as follows:

$$\mathsf{ls}(E_1, E_2) =_{df} (\mathsf{emp} \wedge E_1{=}E_2) \vee (\exists E_3 \cdot E_1{\mapsto}E_3 * \mathsf{ls}(E_3, E_2))$$

A symbolic heap H is made up by a pure formula Π and a separation formula Σ, possibly with existential quantifications over them. The separation formula Σ is formed by the predicates $B(E_1, E_2)$, `true` (arbitrary heaps) and `emp` (an empty heap) via separation conjunction [18]. We use SH to denote the set of all symbolic heaps and we will use the two terms "symbolic heap" and "abstract state" interchangeably.

4 Abduction

As shown in Example 1 (Section 2), when analyzing the code after an unknown call, due to the lack of information about the unknown procedure, it is possible that the current state is too weak to meet the required precondition for the next instruction. As a consequence, the symbolic execution fails. A technique called abduction (or abductive reasoning) [5,10] can be used to discover a symbolic heap M to make the entailment $\Delta * \mathsf{M} \vdash H * \mathsf{R}$ succeed, when the entailment $\Delta \vdash H * \mathsf{R}$ fails. Here R denotes the frame part of the entailment. For instance, the entailments at line `4b` and line `5b` failed in Example 1, and in both cases, the abduction algorithm was called to find the missing M.

One problem in abduction is that there can be many solutions of M for the entailment $\Delta * \mathsf{M} \vdash H * \mathsf{R}$ to succeed. For instance, `false` can be a solution but should be avoided where possible. As another example, for the entailment $\mathsf{ls}(y,z) * \mathsf{M} \vdash \mathsf{ls}(x,z) * \mathsf{R}$ (a similar one was at line `5c` in Example 1 in Sec 2), an abductive reasoning may return, for example, two different solutions: $M = \mathsf{ls}(x,y)$ with $R = \mathsf{emp}$ (our solution) or $M = \mathsf{ls}(x,z)$ with $R = \mathsf{ls}(y,z)$ (the solution given by Calcagno et al. [5]). A partial order $\preceq$ over symbolic heaps was given by Calcagno et al. [5] to make the selection from different solutions:

$$\begin{aligned}\mathsf{M} \preceq \mathsf{M}' =_{df}\ & (\mathsf{M}' \vdash \mathsf{M} * \mathtt{true} \wedge \mathsf{M} \nvdash \mathsf{M}' * \mathtt{true}) \vee \\ & (\mathsf{M}' \vdash \mathsf{M} * \mathtt{true} \wedge \mathsf{M} \vdash \mathsf{M}' * \mathtt{true} \wedge \mathsf{M}' \vdash \mathsf{M})\end{aligned}$$

Intuitively speaking, consider two symbolic heaps M and M', if M' satisfies M (possibly with some frame), then M is less than M' under $\preceq$. For example, we have $\mathsf{ls}(x,y) \preceq \mathsf{ls}(x,y) * \mathsf{ls}(y,z)$, and $\mathsf{ls}(x,y) \preceq x{=}y$.

With such partial order $\preceq$, we can judge the quality of different solutions for abduction by always choosing the least one according to $\preceq$.

Two solutions can not be distinguished when they are not comparable under this partial order. However, we may still prefer one solution over another in this case, as will be discussed in Section 6. We would expect that the solution to incur as few free variables as possible in the frame part (R). For this reason, we define a new order as follows:

$$\begin{aligned}\mathsf{M} \preceq^{\Delta}_{H} \mathsf{M}' =_{df}\ & \mathsf{M}{\preceq}\mathsf{M}' \vee (\mathsf{M}{\not\preceq}\mathsf{M}' \wedge \mathsf{M}'{\not\preceq}\mathsf{M}\ \wedge \\ & \Delta * \mathsf{M} \vdash H * \mathsf{R} \wedge \Delta * \mathsf{M}' \vdash H * \mathsf{R}' \wedge \\ & \mathsf{fv}(\mathsf{R}) \subseteq \mathsf{fv}(\mathsf{R}'))\end{aligned}$$

Note that our partial order is defined over the set of solutions for one abduction. Intuitively, given two solutions which are not comparable w.r.t. $\preceq$, our partial order may still be able to compare them, if one incurs fewer free variables in the frame R than the other. To ensure that the abductor will choose the former, but not the latter, we enrich the abductor [5] with the following new rules so that it attempts to introduce less free variables in the frame, where possible:

$$\frac{(E_0{=}E_1 \wedge \Delta) * [\mathsf{M}] \triangleright \exists \boldsymbol{y}.\Delta'}{\Delta * E_0{\mapsto}E * [\exists \boldsymbol{x}'.E_0{=}E_1 \wedge \mathsf{M}] \triangleright \exists \boldsymbol{x}'\boldsymbol{y}'.\Delta' * E_1{\mapsto}E}\ \textsf{t-match}$$

$$\frac{\Delta * [\mathsf{M}] \triangleright \exists \boldsymbol{x}'.\Delta' * \mathsf{ls}(E_0, E_1)}{\Delta * \mathsf{B}(E_0, E) * [\mathsf{M}] \triangleright \exists \boldsymbol{x}'.\Delta' * \mathsf{B}(E_1, E)}\ \textsf{t-right}$$

$$\frac{E{\neq}E_0 \wedge \Delta * \mathsf{ls}(E_0, x')[\mathsf{M}] \triangleright \exists \boldsymbol{y}'.\Delta'}{\Delta * \mathsf{ls}(E_0, E) * [E_0{\neq}E \wedge \mathsf{M}] \triangleright \exists x'\boldsymbol{y}'.\Delta' * x'{\mapsto}E}\ \textsf{t-left}$$

Compared with the abductor in Calcagno et al. [5], when the matching from the head of a list segment fails, our abductor also tries to match from the tail, instead of directly applying missing rule to introduce a new separation predicate, which could bring in more free variables in the frame. Here the first rule introduces aliasing information between two heads of pointing-to relationships in the premise and conclusion of the abduction, respectively. The last two rules try to match from the end of a basic separation predicate; if the matching succeeds then the matched part will be dropped from both sides to reduce the complexity of the predicates. For instance, in Example 1 in Section 2, for the last abduction

$$\exists y\ .\ \mathsf{ls}(y, z) * z{\mapsto}\texttt{null} * [\mathsf{M}] \triangleright \mathsf{ls}(x, z) * z{\mapsto}\texttt{null}$$

their abductor will return $\mathsf{ls}(x, z)$ as M, by trying to match the head of list segment on both sides of $\triangleright$, and adding the missing heap part to the left hand side once the matching fails. This solution is not optimal in the light that it introduces a free variable z in the frame. Comparatively, our abduction tries to match also from the tail of a list segment, finding $\mathsf{ls}(x, y)$ as the result for the aforesaid abduction, which is a minimal solution under our partial order $\preceq^{\Delta}_{H}$.

5 Abstract Semantics

This section introduces two kinds of abstract semantics we use to analyze the program: an underlying semantics from the local shape analysis [8] and another semantics based on both the first one and abduction from the composite shape analysis [5].

We denote the specifications of a procedure $f(\boldsymbol{x}; \boldsymbol{y})$ as a subset of $\mathsf{SH} \times (\mathsf{SH} \cup \{\top\})$ (where $\top$ stands for a fault abstract state). Note that for a call-by-reference parameter x, both $\mathsf{old}(x)$ and x may occur in a postcondition with the former referring to the value of x in the pre-state (as in JML [13–15] or Spec# [2]). To illustrate, a postcondition $\mathsf{ls}(\mathsf{old}(x), x)$ means that there is a list segment beginning with the initial value of x (present in the precondition) to the final

x when the procedure returns. The set of all specifications for all procedures is defined as

$$\mathsf{AllSpec} =_{df} \mathcal{P}(\mathsf{Name} \times \mathcal{P}(\mathsf{SH} \times (\mathsf{SH} \cup \{\top\})))$$

where Name refers to the set of all function names. Then our underlying abstract semantics is defined as

$$[\![C]\!] =_{df} \mathsf{AllSpec} \to \mathcal{P}^{\top}(\mathsf{SH}) \to \mathcal{P}^{\top}(\mathsf{SH})$$

where $\mathcal{P}^{\top}(\mathsf{SH})$ for $\mathcal{P}(\mathsf{SH}) \cup \{\top\}$. Given a program C, a specification table $\mathcal{T} \in \mathsf{AllSpec}$, a set of abstract states S, $[\![C]\!]_{\mathcal{T}} S$ returns another set of abstract states.

Example 3 (Underlying semantics). For the `findLast` in our motivating example, suppose the specification table $\mathcal{T}$ is

$$\{(\mathtt{findLast}(\mathsf{a},\mathsf{b}),\ \{(\mathsf{ls}(a,\mathtt{null}) \wedge a{\neq}\mathtt{null}, \mathsf{ls}(a,b) * b{\mapsto}\mathtt{null})\})\}$$

Then we know the symbolic execution $[\![\mathtt{findLast}(\mathsf{x},\mathsf{y})]\!]_{\mathcal{T}}\{\mathsf{ls}(x,\mathtt{null}) \wedge x{\neq}\mathtt{null}\}$ will give $\{\mathsf{ls}(x,y) * y{\mapsto}\mathtt{null}\}$ as result.

The foundation of the first underlying semantics is the basic transition functions from a symbolic heap to a set of symbolic heaps below:

$\mathsf{rearr}(E)$	$=_{df}$	$\mathsf{SH} \to \mathcal{P}^{\top}(\mathsf{SH}[E])$	Rearrangement
$\mathsf{exec}(A[E])$	$=_{df}$	$\mathsf{SH}[E] \to \mathsf{SH} \cup \{\top\}$	Heap-sensitive execution
$\mathsf{exec}(A)$	$=_{df}$	$\mathsf{SH} \to \mathsf{SH}$	Heap-insensitive execution
abs	$=_{df}$	$\mathsf{SH} \to \mathsf{SH}$	Abstraction

where $\mathsf{SH}[E]$ denotes a set of symbolic heaps in which each element has E exposed as the head in one of its pointing-to conjuncts ($E{\mapsto}F$). Here $\mathsf{rearr}(E)$ attempts to unroll the shape beginning with E to expose it as the head of a pointing-to predicate, say, E points to another expression. (If such unrolling fails then it will return $\top$.) This is a preparation for the second transition function $\mathsf{exec}(A[E])$, as it tries to perform some dereference of E. The third function, like the semantics for stack-based variable assignment and heap allocation, does not require such exposure of E, and thus is called heap-insensitive compared with the second one. The last transition function conducts a rolling over the shapes, to eliminate unimportant cutpoints to ensure termination.

The rules for the basic transition functions are adopted from Distefano et al. [8], where the logical variable x'' is always fresh.

Rules for rearrangement:

$$\begin{array}{ll}
\mathsf{rearr}(E)\ (\exists \boldsymbol{x}'.\Pi{\wedge}\Delta * \mathsf{ls}(E,F)) =_{df} \{\exists \boldsymbol{x}'.\Pi{\wedge}E{=}F{\wedge}\Delta, \\
\qquad\qquad \exists \boldsymbol{x}'.\Pi{\wedge}\Delta * E{\mapsto}F, \\
\qquad\qquad \exists \boldsymbol{x}', x''.\Pi{\wedge}\Delta * E{\mapsto}x'' * \mathsf{ls}(x'',F)\} \\
\quad \mid\ (\exists \boldsymbol{x}'.\Pi{\wedge}\Delta * E{\mapsto}F) =_{df} \{\exists \boldsymbol{x}'.\Pi{\wedge}\Delta * E{\mapsto}F\} \\
\quad \mid\ (\exists \boldsymbol{x}'.\Pi{\wedge}\Delta * \mathsf{ls}(G,F)) =_{df} \textbf{if}\ \Pi \vdash G{=}E\ \textbf{then} \\
\qquad\qquad \{\mathsf{rearr}(E)(\exists \boldsymbol{x}'.\Pi{\wedge}\Delta * \mathsf{ls}(E,F))\}\ \textbf{else}\ \{\top\} \\
\quad \mid\ (\exists \boldsymbol{x}'.\Pi{\wedge}\Delta * G{\mapsto}F) =_{df} \textbf{if}\ \Pi \vdash G{=}E\ \textbf{then} \\
\qquad\qquad \{\mathsf{rearr}(E)(\exists \boldsymbol{x}'.\Pi{\wedge}\Delta * E{\mapsto}F)\}\ \textbf{else}\ \{\top\} \\
\quad \mid\ (\exists \boldsymbol{x}'.\Pi{\wedge}\Delta) =_{df} \{\top\}
\end{array}$$

Rules for symbolic execution:

$$\begin{array}{l}
\mathsf{exec}\ (x := E)(\exists \boldsymbol{x}'.\mathit{\Pi}{\wedge}\mathit{\Delta}) =_{\mathit{df}} \exists \boldsymbol{x}'.x{=}[x''/x]E \wedge [x''/x](\mathit{\Pi}{\wedge}\mathit{\Delta})\\
\mathsf{exec}\ (x := [E])(\exists \boldsymbol{x}'.\mathit{\Pi}{\wedge}\mathit{\Delta} * E{\mapsto}F) =_{\mathit{df}} \exists \boldsymbol{x}'.x{=}[x''/x]F \wedge [x''/x](\mathit{\Pi}{\wedge}\mathit{\Delta} * E{\mapsto}F)\\
\qquad\qquad \mid (\exists \boldsymbol{x}'.\mathit{\Pi}{\wedge}\mathit{\Delta}) =_{\mathit{df}} \top\\
\mathsf{exec}\ ([E] := G)(\exists \boldsymbol{x}'.\mathit{\Pi}{\wedge}\mathit{\Delta} * E{\mapsto}F) =_{\mathit{df}} \exists \boldsymbol{x}'.\mathit{\Pi}{\wedge}\mathit{\Delta} * E{\mapsto}G\\
\qquad\qquad \mid (\exists \boldsymbol{x}'.\mathit{\Pi}{\wedge}\mathit{\Delta}) =_{\mathit{df}} \top\\
\mathsf{exec}\ (x := \mathsf{new}(E))(\exists \boldsymbol{x}'.\mathit{\Pi}{\wedge}\mathit{\Delta}) =_{\mathit{df}} \exists \boldsymbol{x}'.[x''/x](\mathit{\Pi}{\wedge}\mathit{\Delta}) * x{\mapsto}[x''/x]E\\
\mathsf{exec}\ (\mathsf{dispose}(E))(\exists \boldsymbol{x}'.\mathit{\Pi}{\wedge}\mathit{\Delta} * E{\mapsto}F) =_{\mathit{df}} \exists \boldsymbol{x}'.\mathit{\Pi}{\wedge}\mathit{\Delta}\\
\qquad\qquad\quad \mid (\exists \boldsymbol{x}'.\mathit{\Pi}{\wedge}\mathit{\Delta}) =_{\mathit{df}} \top
\end{array}$$

Rules for abstraction:

$$\frac{}{\exists \boldsymbol{z}'.\ E{=}x' \wedge \mathit{\Pi} \wedge \mathit{\Sigma} \rightsquigarrow \boldsymbol{z}'.\ [E/x']\mathit{\Pi} \wedge \mathit{\Sigma}}\ \mathsf{Equality}$$

$$\frac{x'{\notin}\mathsf{LVar}(\mathit{\Sigma})}{\exists \boldsymbol{z}'.\ E{\neq}x' \wedge \mathit{\Pi} \wedge \mathit{\Sigma} \rightsquigarrow \exists \boldsymbol{z}'.\ \mathit{\Pi} \wedge \mathit{\Sigma}}\ \mathsf{Disequality}$$

$$\frac{x'{\notin}\mathsf{LVar}(\mathit{\Pi}, \mathit{\Sigma})}{\exists \boldsymbol{z}'.\ \mathit{\Pi} \wedge \mathit{\Sigma}\ *\ \mathsf{B}(x', E) \rightsquigarrow \exists \boldsymbol{z}'.\ \mathit{\Pi} \wedge \mathit{\Sigma}\ *\ \mathtt{true}}\ \mathsf{Junk1}$$

$$\frac{x', y'{\notin}\mathsf{LVar}(\mathit{\Pi}, \mathit{\Sigma})}{\exists \boldsymbol{z}'.\ \mathit{\Pi} \wedge \mathit{\Sigma}\ *\ \mathsf{B}(x', y')\ *\ \mathsf{B}(y', x') \rightsquigarrow \exists \boldsymbol{z}'.\ \mathit{\Pi} \wedge \mathit{\Sigma}\ *\ \mathtt{true}}\ \mathsf{Junk2}$$

$$\frac{x'{\notin}\mathsf{LVar}(\mathit{\Pi}, \mathit{\Sigma}, E, F) \qquad \mathit{\Pi} \vdash F{=}\mathtt{null}}{\exists \boldsymbol{z}'.\ \mathit{\Pi} \wedge \mathit{\Sigma} * \mathsf{B}_1(E, x') * \mathsf{B}(x', F) \rightsquigarrow \exists \boldsymbol{z}'.\ \mathit{\Pi} \wedge \mathit{\Sigma} * \mathsf{ls}(E, \mathtt{null})}\ \mathsf{Abs1}$$

$$\frac{x'{\notin}\mathsf{LVar}(\mathit{\Pi}, \mathit{\Sigma}, E, F, E_1, F_1) \qquad \mathit{\Pi} \vdash F{=}E_1}{\begin{array}{l}\exists \boldsymbol{z}'.\ \mathit{\Pi} \wedge \mathit{\Sigma} * \mathsf{B}_1(E, x') * \mathsf{B}(x', F) * \mathsf{B}(E_1, F_1) \rightsquigarrow\\ \quad \exists \boldsymbol{z}'.\ \mathit{\Pi} \wedge \mathit{\Sigma} * \mathsf{ls}(E, F) * \mathsf{B}(E_1, F_1)\end{array}}\ \mathsf{Abs2}$$

A lifting function $p^\dagger$ is defined over the transition functions above to lift their domains and ranges to a powerset of SH, plus $\top$:

$$p^\dagger S =_{\mathit{df}} \{H' \mid \exists H{\in}S\ .\ (H{\neq}\top \wedge p\ H{=}H') \vee H{=}H'{=}\top\}$$

and this function is overloaded for rearr only to lift its domain to $\mathcal{P}^\top(\mathsf{SH})$:

$$\mathsf{rearr}(E)^\dagger S =_{\mathit{df}} \bigcup_{H \in S \setminus \{\top\}} \mathsf{rearr}(E)\ H \cup \{\top | \top{\in}S\}$$

Different from Distefano et al. [8], we need to adapt the underlying abstract semantics to procedure invocation as well:

$$\begin{array}{rl}
[\![f(\boldsymbol{x}; \boldsymbol{y})]\!]_{\mathcal{T}} S =_{\mathit{df}} & \{[\boldsymbol{x}/\boldsymbol{a}, \boldsymbol{y}/\boldsymbol{b}, \boldsymbol{y}'/\mathsf{old}(\boldsymbol{b})]\mathsf{Post} * [\boldsymbol{y}'/\boldsymbol{y}]R \mid \mathit{\Delta}{\in}S \wedge\\
& \quad (\mathsf{Pre}, \mathsf{Post}){\in}\mathsf{Spec}_f \wedge \mathit{\Delta} \vdash [\boldsymbol{x}/\boldsymbol{a}, \boldsymbol{y}/\boldsymbol{b}]\mathsf{Pre} * R\}\\
& \qquad \text{where } (f, \mathsf{Spec}_f) \in \mathcal{T} \text{ and } \boldsymbol{y}' \text{ are fresh}
\end{array}$$

where $\mathcal{T}$ is an element of $\mathsf{AllSpec}$.

The following function, filt, is to filter out any symbolic heap that does not satisfy certain aliasing constraint:

$$\begin{aligned}
&\mathsf{filt}\ (E{=}F)(H) =_{df} \textbf{if } H \nvdash E{\neq}F \textbf{ then } H \wedge E{=}F \textbf{ else } \text{undefined}\\
&\mathsf{filt}\ (E{\neq}F)(H) =_{df} \textbf{if } H \nvdash E{=}F \textbf{ then } H \wedge E{\neq}F \textbf{ else } \text{undefined}
\end{aligned}$$

The program constructors' semantics are based on the atomic ones defined above:

$$\begin{array}{ll}
[\![\mathsf{b}]\!]_{\mathcal{T}} S & =_{df}\ \mathsf{filt}(\mathsf{b})^{\dagger} S\\
[\![A[E]]\!]_{\mathcal{T}} S & =_{df}\ \mathsf{abs}^{\dagger} \circ \mathsf{exec}(A[E])^{\dagger} \circ \mathsf{rearr}(E)^{\dagger} S\\
[\![A]\!]_{\mathcal{T}} S & =_{df}\ \mathsf{abs}^{\dagger} \circ \mathsf{exec}(A)^{\dagger} S\\
[\![C_1; C_2]\!]_{\mathcal{T}} S & =_{df}\ [\![C_2]\!]_{\mathcal{T}} \circ [\![C_1]\!]_{\mathcal{T}} S\\
[\![\mathsf{if\ b\ then}\ C_1\ \mathsf{else}\ C_2\ \mathsf{fi}]\!]_{\mathcal{T}} S & =_{df}\ [\![\mathsf{b}; C_1]\!]_{\mathcal{T}} S \cup [\![\neg \mathsf{b}; C_2]\!]_{\mathcal{T}} S\\
[\![\mathsf{while\ b\ do}\ C\ \mathsf{od}]\!]_{\mathcal{T}} S & =_{df}\ [\![\neg \mathsf{b}]\!]_{\mathcal{T}} (\mathsf{lfix}\ \lambda S'. S \cup [\![\mathsf{b}; C]\!]_{\mathcal{T}} S')
\end{array}$$

Next we define the abstract semantics with abduction used in our analysis to discover the effect of unknown procedure calls. This semantics is adopted from Calcagno et al. [5]:

$$[\![C]\!]^{\mathsf{A}} =_{df} \mathsf{AllSpec} \rightarrow \mathcal{P}^{\top}(\mathsf{SH} \times \mathsf{SH}) \rightarrow \mathcal{P}^{\top}(\mathsf{SH} \times \mathsf{SH})$$

Here we mean $\mathcal{P}^{\top}(\mathsf{SH}\times\mathsf{SH})$ by $\mathcal{P}((\mathsf{SH}\cup\{\top\})\times(\mathsf{SH}\cup\{\top\}))$. Given a specification table $\mathcal{T} \in \mathsf{AllSpec}$, each element of the input (or output) for $[\![C]\!]^{\mathsf{A}}_{\mathcal{T}}$ is a pair of two symbolic heaps, of which the first denotes the current program state, and the second stands for the abduction result. In our framework, this semantics is used to accumulate the discovered effect of unknown procedure calls into the second symbolic heap in the pair.

Example 4 (Abstract semantics with abduction). Consider the same setting for `findLast` in Example 3. For the semantics with abduction $[\![\mathtt{findLast(x,y)}]\!]^{\mathsf{A}}_{\mathcal{T}}$ $\{\mathsf{ls}(x, \mathtt{null})\}$, we will have $\{(\mathsf{ls}(x,y) * y{\mapsto}\mathtt{null}, x{\neq}\mathtt{null})\}$ as result. Here in order to achieve the first element in the pair as the final state $(\mathsf{ls}(x,y) * y{\mapsto}\mathtt{null})$, the second element $(x{\neq}\mathtt{null})$ must be added to its corresponding input state $\mathsf{ls}(x,\mathtt{null})$.

This semantics also consists of the basic transition functions which composes the atomic instructions' semantics and then the program constructors' semantics.

Here the basic transition functions are lifted as

```
$\mathsf{Rearr}(E) =_{df}$
    let $\mathcal{H}$=$\mathsf{rearr}(E)(H)$ and$S$=$\{(H', \mathsf{M}) | H' \in \mathcal{H} \cap \mathsf{SH}\}$
    in if $(\top \notin \mathcal{H})$ then $S$
        elseif $(H \vdash E{=}a$ for some $a \in \mathsf{SVar})$ and
               $(\mathsf{M} \nvdash a{\mapsto}x'$ for fresh $x' \in \mathsf{LVar})$
        then $S \cup \{(H * E{\mapsto}x', \mathsf{M} * a{\mapsto}x')\}$
        else $S \cup \{\top\}$
$\mathsf{Exec}(A)(H, \mathsf{M}) =_{df}$ let $\mathcal{H}$=$\mathsf{exec}(A)(H)$
    in $\{(H', \mathsf{M}) | H' \in \mathcal{H}\} \cup \{\top | \top \in \mathcal{H}\}$
               where $A$ is $[E] := G$ or $\mathsf{dispose}(E)$
$\mathsf{Abs}(H, \mathsf{M}) =_{df} (\mathsf{abs}(H), \mathsf{abs}(\mathsf{M}))$
```

In the definition of Exec we need to divert from Calcagno et al. [5], specifically for variable assignments. As can be seen below, when x is assigned to a new value, the original value is still preserved with a substitution $\sigma = [x''/x]$ where x'' is fresh. Doing this allows us to keep the connection among the history values of a variable and its latest value, which may be essential as a link from the unknown procedure's postcondition to its caller's postcondition.

```
$\mathsf{Exec}(x := E)(\exists \boldsymbol{x}'.\Pi \wedge \Delta, \mathsf{M}) =_{df} (\exists \boldsymbol{x}'.x{=}\sigma E \wedge \sigma(\Pi \wedge \Delta), \sigma \mathsf{M})$
$\mathsf{Exec}(x := [E])(\exists \boldsymbol{x}'.\Pi \wedge \Delta * E{\mapsto}F) =_{df} (\exists \boldsymbol{x}'.x{=}\sigma F \wedge \sigma(\Pi \wedge \Delta * E{\mapsto}F), \sigma \mathsf{M})$
        $| \ (\exists \boldsymbol{x}'.\Pi \wedge \Delta) =_{df}$ if $\exists \boldsymbol{x}'.\Pi \wedge \Delta * E{\mapsto}F \nvdash$ false
                           then $(\exists \boldsymbol{x}'.x{=}\sigma F \wedge \sigma(\Pi \wedge \Delta * E{\mapsto}F), \sigma(\mathsf{M} * E{\mapsto}F))$
                           else $(\top, \sigma \mathsf{M})$
$\mathsf{Exec}(x := \mathsf{new}(E))(\exists \boldsymbol{x}'.\Pi \wedge \Delta) =_{df} (\exists \boldsymbol{x}'.\sigma(\Pi \wedge \Delta) * x{\mapsto}\sigma E, \sigma \mathsf{M})$
```

The filter function Filt now only works on the first symbolic heap of a pair:

$$\mathsf{Filt}(\mathsf{b})(H, \mathsf{M}) =_{df} (\mathsf{filt}(\mathsf{b})(H), \mathsf{M})$$

And the abstract semantics for the program constructors is as follows.

$$
\begin{array}{ll}
[\![\mathsf{b}]\!]^{\mathsf{A}}_{\mathcal{T}} S & =_{df} \mathsf{Filt}(\mathsf{b})^{\dagger} S \\
[\![A[E]]\!]^{\mathsf{A}}_{\mathcal{T}} S & =_{df} \mathsf{Abs}^{\dagger} \circ \mathsf{Exec}(A[E])^{\dagger} \circ \mathsf{Rearr}(E)^{\dagger} S \\
[\![A]\!]^{\mathsf{A}}_{\mathcal{T}} S & =_{df} \mathsf{Abs}^{\dagger} \circ \mathsf{Exec}(A)^{\dagger} S \\
[\![C_1; C_2]\!]^{\mathsf{A}}_{\mathcal{T}} S & =_{df} [\![C_2]\!]^{\mathsf{A}}_{\mathcal{T}} \circ [\![C_1]\!]^{\mathsf{A}}_{\mathcal{T}} S \\
[\![\mathsf{if\ b\ then\ } C_1 \mathsf{\ else\ } C_2 \mathsf{\ fi}]\!]^{\mathsf{A}}_{\mathcal{T}} S & =_{df} [\![\mathsf{b}; C_1]\!]^{\mathsf{A}}_{\mathcal{T}} S \sqcup [\![\neg \mathsf{b}; C_2]\!]^{\mathsf{A}}_{\mathcal{T}} S \\
[\![\mathsf{while\ b\ do\ } C \mathsf{\ od}]\!]^{\mathsf{A}}_{\mathcal{T}} S & =_{df} [\![\neg \mathsf{b}]\!]^{\mathsf{A}}_{\mathcal{T}} (\mathsf{lfix}\ \lambda S'. S \sqcup [\![\mathsf{b}; C]\!]^{\mathsf{A}}_{\mathcal{T}} S')
\end{array}
$$

At last is the semantics for procedure invocation, with abduction:

$$\begin{aligned}[\![f(\boldsymbol{x};\boldsymbol{y})]\!]^{\mathsf{A}}_{\mathcal{T}}S =_{df} & \{([\boldsymbol{x}/\boldsymbol{a},\boldsymbol{y}/\boldsymbol{b},\boldsymbol{y}'/\mathsf{old}(\boldsymbol{b})]\mathsf{Post} * [\boldsymbol{y}'/\boldsymbol{y}]R, [\boldsymbol{y}'/\boldsymbol{y}]\mathsf{M}) \mid (\Delta,\mathsf{M}){\in}S\wedge \\ & \quad (\mathsf{Pre},\mathsf{Post}){\in}\mathsf{Spec}_f \wedge \Delta \vdash [\boldsymbol{x}/\boldsymbol{a},\boldsymbol{y}/\boldsymbol{b}]\mathsf{Pre} * R\} \cup \\ & \{([\boldsymbol{x}/\boldsymbol{a},\boldsymbol{y}/\boldsymbol{b},\boldsymbol{y}'/\mathsf{old}(\boldsymbol{b})]\mathsf{Post} * [\boldsymbol{y}'/\boldsymbol{y}]R, [\boldsymbol{y}'/\boldsymbol{y}](\mathsf{M} * \mathsf{M}')) \mid \\ & \quad (\Delta,\mathsf{M}){\in}S \wedge (\mathsf{Pre},\mathsf{Post}){\in}\mathsf{Spec}_f \wedge \Delta{\nvdash}[\boldsymbol{x}/\boldsymbol{a},\boldsymbol{y}/\boldsymbol{b}]\mathsf{Pre} * R \wedge \\ & \quad \Delta * [\mathsf{M}'] \triangleright [\boldsymbol{x}/\boldsymbol{a},\boldsymbol{y}/\boldsymbol{b}]\mathsf{Pre}\} \\ & \text{where } (f,\mathsf{Spec}_f) \in \mathcal{T} \text{ and } \boldsymbol{y}' \text{ are fresh}\end{aligned}$$

where $\mathcal{T}$ is an element of $\mathsf{AllSpec}$.

6 Verification

Based on the abstract semantics defined in the last section, we present in this section our algorithms for the verification of programs with unknown calls.

6.1 Main Verification Algorithm

Our verification algorithm given in Figure 3 attempts to verify the body of the current procedure (the third input, comprising an unknown command U) against the given specifications (the second input). The first input gives a set of known procedure specifications, which are necessary as the current procedure may invoke known procedures apart from unknown ones. If the verification succeeds, it returns specifications that are expected for all unknown procedures invoked within U for the whole verification to succeed. If it fails, we know that the current procedure cannot meet one or more given specifications, no matter what specifications are given to the invoked unknown procedures. Returned specifications will be expressed using special variables $\boldsymbol{a}, \boldsymbol{b}$, etc. as in the earlier example.

For each pair $(\mathsf{Pre}, \mathsf{Post})$ to verify against (line 2), the algorithm works in three steps. Based on the underlying semantics mentioned earlier, it first computes the post-states of C_1 (i.e. S_0) from Pre (line 3), from which it extracts the preconditions for $U_{(\boldsymbol{x};\boldsymbol{y})}$ using the function Local. Intuitively, it extracts the part of each Δ_1 reachable from the variables that may be accessed by U, namely, $\boldsymbol{x}$ and $\boldsymbol{y}$ (line 6). Here $\mathsf{fv}(\Delta)$ stands for all free (program and logical) variables occurring in Δ. The function $\mathsf{Local}(\Pi \wedge \Sigma, \{\boldsymbol{x}\})$ is defined as follows:

$$\begin{aligned}\mathsf{Local}(\Pi{\wedge}\Sigma, \{\boldsymbol{x}\}) =_{df} & \exists \mathtt{fv}(\Pi{\wedge}\Sigma) \setminus \mathtt{ReachVar}(\Pi{\wedge}\Sigma, \{\boldsymbol{x}\}) \cdot \\ & \Pi * \mathtt{ReachHeap}(\Pi{\wedge}\Sigma, \{\boldsymbol{x}\})\end{aligned}$$

where $\mathtt{ReachVar}(\Pi{\wedge}\Sigma, \{\boldsymbol{x}\})$ is the minimal set of variables reachable from $\{\boldsymbol{x}\}$:

$$\begin{aligned}&\{\boldsymbol{x}\} \cup \{z_2 \mid \exists z_1, \Pi_1 \cdot z_1{\in}\mathtt{ReachVar}(\Pi{\wedge}\Sigma, \{\boldsymbol{x}\}) \wedge \Pi{=}(z_1{=}z_2) \wedge \Pi_1\} \cup \{z_2 \mid \\ &\quad \exists z_1, \Sigma_1 \cdot z_1{\in}\mathtt{ReachVar}(\Pi{\wedge}\Sigma, \{\boldsymbol{x}\}) \wedge \Sigma{=}\mathsf{B}(z_1, z_2) * \Sigma_1\} \subseteq \mathtt{ReachVar}(\Pi{\wedge}\Sigma, \{\boldsymbol{x}\})\end{aligned}$$

where $\mathsf{B}(z_1, z_2)$ stands for either $z_1{\mapsto}z_2$ or $\mathsf{ls}(z_1, z_2)$. And the formula $\mathtt{ReachHeap}$ $(\Pi{\wedge}\Sigma, \{\boldsymbol{x}\})$ denotes the part of Σ reachable from $\{\boldsymbol{x}\}$ and is formally defined as the $*$-conjunction of the following set of formulae:

$$\{\Sigma_1 \mid \exists z_1, z_2, \Sigma_2 \cdot z_1{\in}\mathtt{ReachVar}(\Pi{\wedge}\Sigma, \{\boldsymbol{x}\}) \wedge \Sigma{=}\Sigma_1{*}\Sigma_2 \wedge \Sigma_1{=}\mathsf{B}(z_1, z_2)\}$$

```
Verify : AllSpec × $\mathcal{P}$(SH × SH) × $V$ ⇀ AllSpec
**Algorithm** Verify($\mathcal{T}$, Spec$_V$, $\{C_1; U_{(\boldsymbol{x};\boldsymbol{y})}; C_2\}_{(\boldsymbol{x}_0;\boldsymbol{y}_0)}$)
1  Spec$_U$ := $\emptyset$
2  **foreach** (Pre, Post) ∈ Spec$_V$ **do**
3    $S_0 := [\![C_1]\!]_{\mathcal{T}}\{\mathsf{Pre} \wedge \boldsymbol{y}_0{=}\mathsf{old}(\boldsymbol{y}_0)\}$
4    **foreach** $\Delta_1 \in S_0$ **do**
5      **if** $\Delta_1 = \top$ **then return** fail **endif**
6      Pre$_U$ := Local($\Delta_1$, $\{\boldsymbol{x},\boldsymbol{y}\}$)
7      Denote $\boldsymbol{z} = \mathsf{fv}(\mathsf{Pre}_U) \setminus \{\boldsymbol{x},\boldsymbol{y}\}$
8      $S := [\![C_2]\!]^{\mathsf{A}}_{\mathcal{T}}\{([\mathsf{old}(\boldsymbol{b})/\boldsymbol{y}]\,\mathsf{Frame}(\Delta_1,\{\boldsymbol{x},\boldsymbol{y}\}) \wedge$
               $\boldsymbol{x}{=}\boldsymbol{a} \wedge \boldsymbol{y}{=}\boldsymbol{b} \wedge \boldsymbol{z}{=}\boldsymbol{c}, \mathsf{emp} \wedge \boldsymbol{x}{=}\boldsymbol{a} \wedge \boldsymbol{y}{=}\boldsymbol{b} \wedge \boldsymbol{z}{=}\boldsymbol{c})\}$
9      $S' := \{\ (\Delta,\mathsf{M}) \mid (\Delta,\mathsf{M}) {\in} S \wedge \Delta \vdash \mathsf{Post} * \mathtt{true}\ \} \ \cup$
            $\{\ (\Delta * \mathsf{M}', \mathsf{M} * \mathsf{M}') \mid (\Delta,\mathsf{M}) {\in} S \wedge$
              $\Delta \nvdash \mathsf{Post}{*}\mathtt{true} \wedge \Delta{*}[\mathsf{M}'] \triangleright \mathsf{Post}\ \}$
10     **if** $\exists(\Delta,\mathsf{M}){\in}S'\ .\ \mathsf{fv}(\mathsf{M}) \nsubseteq \mathsf{ReachVar}(\Delta, \{\boldsymbol{a},\boldsymbol{b}\})$
         **then return** (fail, M) **endif**
11     **foreach** $(\Delta, \mathsf{M}) \in S'$ **do**
12       Pre$_U$ := $[\boldsymbol{a}/\boldsymbol{x}, \boldsymbol{b}/\boldsymbol{y}, \boldsymbol{c}/\boldsymbol{z}]$Pre$_U$
13       Post$_U$ := sub_alias(M, $\{\boldsymbol{a},\boldsymbol{b},\boldsymbol{c}\}$)
14       $g := (\mathsf{fv}(\mathsf{Pre}_U) \cap \mathsf{fv}(\mathsf{Post}_U)) \cup \{\boldsymbol{a},\boldsymbol{b}\}$
15       Spec$_U$ := Spec$_U \cup \{(\exists(\mathsf{fv}(\mathsf{Pre}_U) \backslash g) {\cdot} \mathsf{Pre}_U, \mathsf{Post}_U)\}$
16     **end foreach**
17   **end foreach**
18 **end foreach**
19 $\mathcal{T}_U$ := CaseAnalysis($\mathcal{T}$, Spec$_U$, $U$)
20 Post_Check($\mathcal{T} \uplus \mathcal{T}_U$, Spec$_V$, $\{C_1; U; C_2\}$)
**end Algorithm**
```

Fig. 3. The main verification algorithm.

The second step is to symbolically execute C_2, using the abstract semantics with abduction, to discover the postconditions for $U_{(\boldsymbol{x};\boldsymbol{y})}$ (lines 8–10). At line 8, since we know nothing about U, we take emp as the post-state of U. Therefore, the initial state for the symbolic execution of C_2 is simply the frame part of state not touched by U. Here Frame is formally defined as

$$\mathsf{Frame}(\Pi \wedge \Sigma, \{\boldsymbol{x}\}) =_{df} \Pi \wedge \mathtt{UnreachHeap}(\Pi \wedge \Sigma, \{\boldsymbol{x}\})$$

where $\mathtt{UnreachHeap}(\Pi \wedge \Sigma, \{\boldsymbol{x}\})$ is the formula consisting of Π and all $*$-conjuncts from Σ which are not in $\mathtt{ReachHeap}(\Pi \wedge \Sigma, \{\boldsymbol{x}\})$.

Note that $\boldsymbol{x}{=}\boldsymbol{a} \wedge \boldsymbol{y}{=}\boldsymbol{b} \wedge \boldsymbol{z}{=}\boldsymbol{c}$ are used to record the snapshot of variables associated with U using the special variables $\boldsymbol{a}, \boldsymbol{b}$ and $\boldsymbol{c}$. The symbolic execution of C_2 at line 8 returns a set S of pairs (Δ, M) where Δ is a possible post state of C_2 and M records the discovered effect of U. At line 9, we check whether or not each Δ can establish the postcondition Post for the whole procedure. If not, another abduction $\Delta{*}[\mathsf{M}'] \triangleright \mathsf{Post}$ is invoked to discover further effect M' which is then added into M.

There can be some complication here. Note that there can be a potential bug in the program, or the given specification is not sufficient. As a consequence of that, the result M returned by our abductor may contain more information than what can be expected from U, in which case we cannot simply regard the whole M as the postcondition of U. For example, consider the code fragment `unknown(x); z:=y.next` with the precondition $x{\mapsto}\texttt{null}$. Before the second instruction (dereference of `y.next`) we use abduction to get $y{\neq}\texttt{null}$. However, noting the fact $y{\notin}\texttt{ReachVar}(\Delta, \{x\})$ where $\Delta = \mathsf{emp}{\wedge}y{\neq}\texttt{null}$ is the state immediately after the unknown call plus the abduction result, we know that from the unknown call's parameters (x), y is not reachable, and hence the unknown call will never establish a state where $y{\neq}\texttt{null}$. In that case we are assured that the procedure being verified cannot meet the specification.

To detect such situation, we introduce the checking in line 10. It tests whether the whole abduction result is reachable from variables accessed by U. If not, then the unreachable part cannot be expected from U, which indicates a possible bug in the program or some inconsistency between the program and its specification. In such cases, the algorithm returns an additional formula that can be used by a further analysis to either identify the bug or strengthen the specification. Recall the example presented in last paragraph: since $y{\neq}\texttt{null}$ cannot be established by the unknown call, if we add it to the precondition of the code fragment (to form a new precondition $x{\mapsto}\texttt{null} \wedge y{\neq}\texttt{null}$), then the verification with the new specification can move on and will potentially succeed. We will exemplify this later with experimental results.

The third step (lines 11–16) is to form the derived specifications for U in terms of variables $\boldsymbol{a}, \boldsymbol{b}$ and g, where g denotes logical variables not directly accessed by U, but occurring in both pre- and postconditions. The formula $\mathsf{sub\_alias}(\mathsf{M}, \{\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}\})$ is obtained from M by replacing all variables with their aliases in $\{\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}\}$. It is defined as

$$\begin{aligned}\mathsf{sub\_alias}(\mathsf{M}, \{\boldsymbol{x}\}) =_{df} & \ (\{[x/x'] \mid x{\in}\{\boldsymbol{x}\} \wedge x'{\in}\mathsf{aliases}(\mathsf{M}, x)\} \ \mathsf{M}) \wedge \\ & \bigwedge\{x = x' \mid x, x'{\in}\{\boldsymbol{x}\} \wedge x'{\in}\mathsf{aliases}(\mathsf{M}, x)\}\end{aligned}$$

where we put a set of substitutions before a formula M to mean to apply all those substitutions to M, and $\mathsf{aliases}(\mathsf{M}, x)$ returns all the aliases of x in M.

Finally, at line 19, the obtained specifications Spec_U for U are passed to the case analysis algorithm (given in Figure 4) to derive the specifications of unknown procedures invoked in U. At line 20, we conduct a post analysis for soundness purpose which will be explained in the next section.

In order to discover specifications for unknown procedures invoked in U, the algorithm in Figure 4 conducts a case analysis according to the structure of U. In the first case (line 2), U is simply an unknown call. In this situation, the algorithm simply returns all the pre-/postcondition pairs from Spec_U as the unknown procedure's specifications.

In the second case (line 4), U is an if construct and each branch contains an unknown block. The algorithm uses the main algorithm to verify the two branches separately with preconditions $\mathsf{Pre}{\wedge}\mathsf{b}$ and $\mathsf{Pre}{\wedge}\neg\mathsf{b}$ respectively, where

```
$\mathsf{CaseAnalysis} : \mathsf{AllSpec} \times \mathcal{P}(\mathsf{SH} \times \mathsf{SH}) \times U \rightharpoonup \mathsf{AllSpec}$
Algorithm $\mathsf{CaseAnalysis}(\mathcal{T}, \mathsf{Spec}_U, U)$
 1 switch $U$
 2    case $\mathsf{unkFn}(\boldsymbol{x}; \boldsymbol{y})$
 3       return $\{(\mathsf{unkFn}, \mathsf{Spec}_U)\}$
 4    case $\mathsf{if\ b\ then}\ V_1\ \mathsf{else}\ V_2\ \mathsf{fi}$
 5       $\mathsf{Spec}_T := \{(\mathsf{Pre}{\wedge}\mathsf{b}, \mathsf{Post}) \mid (\mathsf{Pre}, \mathsf{Post}) \in \mathsf{Spec}_U\}$
 6       $\mathsf{Spec}_F := \{(\mathsf{Pre}{\wedge}\neg\mathsf{b}, \mathsf{Post}) \mid (\mathsf{Pre}, \mathsf{Post}) \in \mathsf{Spec}_U\}$
 7       $R_1 := \mathsf{Verify}(\mathcal{T}, \mathsf{Spec}_T, V_1)$
 8       $R_2 := \mathsf{Verify}(\mathcal{T}, \mathsf{Spec}_F, V_2)$
 9       return $R_1 \uplus R_2$
10    case $\mathsf{if\ b\ then}\ V\ \mathsf{else}\ C\ \mathsf{fi}$
11       $\mathsf{Spec}_T := \{(\mathsf{Pre}{\wedge}\mathsf{b}, \mathsf{Post}) \mid (\mathsf{Pre}, \mathsf{Post}) \in \mathsf{Spec}_U\}$
12       $R := \mathsf{Verify}(\mathcal{T}, \mathsf{Spec}_T, V)$
13       if $\exists(\mathsf{Pre}, \mathsf{Post}) \in \mathsf{Spec}_U, \Delta \in [\![C]\!]_{\mathcal{T}}\{\mathsf{Pre} \wedge \neg b\} \cdot$
               $\Delta{=}\top \vee \Delta \not\vdash \mathsf{Post}{*}\mathtt{true}$ then return $\mathsf{fail}$
14       else return $R$ endif
15    case $\mathsf{if\ b\ then}\ C\ \mathsf{else}\ V\ \mathsf{fi}$   (Similar as last case)
16    case $\mathsf{while\ b\ do}\ V\ \mathsf{od}$
17       Denote $V$ as $\{C_1; U_{(\boldsymbol{x}_1;\boldsymbol{y}_1)}; C_2\}_{(\boldsymbol{x};\boldsymbol{y})}$
18       Define $\mathsf{loop}(\boldsymbol{x}; \boldsymbol{y})\{\ \mathsf{if\ b\ then}\ V;\ \mathsf{loop}(\boldsymbol{x}; \boldsymbol{y})\ \mathsf{fi}\}$
19       $\mathcal{T}' := \mathcal{T} \uplus \{\ (\mathsf{loop}, \mathsf{Spec}_U)\ \}$
20       return $\mathsf{Verify}(\mathcal{T}', \mathsf{Spec}_U, \{\mathsf{if\ b\ then}\ V; \mathsf{loop}(\boldsymbol{x}; \boldsymbol{y})\ \mathsf{fi}\}_{(\boldsymbol{x};\boldsymbol{y})})$
21    case $\mathsf{unkFn}(\boldsymbol{x}_0; \boldsymbol{y}_0);\ \{\ ; C_i; \mathsf{unkFn}(\boldsymbol{x}_i; \boldsymbol{y}_i)\}_{i=1}^{n}$
22       return $\{(\mathsf{unkFn}, \mathsf{SeqUnkCalls}(\mathcal{T}, \mathsf{Spec}_U, U))\}$
end Algorithm
```

Fig. 4. The case analysis algorithm.

Pre is one of the preconditions of the whole if. The results obtained from two branches are then combined using the $\uplus$ operator:

$$R_1 \uplus R_2 =_{df} \{(\mathsf{f}, \mathsf{Refine}(\mathsf{Spec}^1_{\mathsf{f}} \cup \mathsf{Spec}^2_{\mathsf{f}})) \mid (\mathsf{f}, \mathsf{Spec}^1_{\mathsf{f}}){\in}R_1 \wedge (\mathsf{f}, \mathsf{Spec}^2_{\mathsf{f}}){\in}R_2\}$$

where Refine is defined as

$$\begin{aligned}
&\mathsf{Refine}\ (\emptyset) &&=_{df} \emptyset\\
&\mathsf{Refine}\ (\{(\mathsf{Pre}, \mathsf{Post})\} \cup \mathsf{Spec}) &&=_{df} \mathbf{if}\ \exists(\mathsf{Pre}', \mathsf{Post}'){\in}\mathsf{Spec} \cdot \mathsf{Pre}'{\preceq}\mathsf{Pre} \wedge \mathsf{Post}{\preceq}\mathsf{Post}'\\
& &&\quad\ \mathbf{then}\ \mathsf{Refine}(\mathsf{Spec})\\
& &&\quad\ \mathbf{else}\ \{(\mathsf{Pre}, \mathsf{Post})\} \cup \mathsf{Refine}(\mathsf{Spec})
\end{aligned}$$

The intuition of Refine is to eliminate any specification $(\mathsf{Pre}', \mathsf{Post}')$ from a set if there exists a 'stronger' one $(\mathsf{Pre}, \mathsf{Post})$ such that $\mathsf{Pre}{\preceq}\mathsf{Pre}'$ and $\mathsf{Post}'{\preceq}\mathsf{Post}$. $\uplus$ is to refine the union of two specification sets.

The third and fourth cases (line 10 and 15) are for if constructs which contain one unknown block in one branch. This is handled in a similar way as in the second case. The only difference is that, for the branch without unknown blocks, we need to verify it with the underlying semantics (line 13).

The fifth case is the while loop. In the motivating example in Section 2, we have shown that our approach is able to handle the verification of a tail-recursive function provided with both pre- and postconditions, our solution here is to translate the while loop into a tail-recursive function to verify it. As can be seen in lines $16-20$, the algorithm generates a new function loop for the while loop, and takes the variables accessed by V to be its parameters. Note that the read-only variables ($\boldsymbol{x}$) become call-by-value parameters and other possibly mutable ones ($\boldsymbol{y}$) become call-by-reference parameters (which is one reason for us to introduce call-by-reference parameters in our language). The algorithm then adds the specifications found for the while loop as loop's specifications into table $\mathcal{T}$, verifying it, and at the same time obtaining specifications for the unknown procedure in the while loop, using the main algorithm Verify.

In the last case (line 21), where U consists of multiple unknown procedure calls in sequence, the algorithm invokes another algorithm, SeqUnkCalls, to deal with it.

Note that CaseAnalysis also has to invoke Verify recursively to discover specifications for the unknown procedures. Meanwhile, to handle the most complicated case, unknown procedure calls in sequence, we still need the SeqUnkCalls algorithm. First we illustrate the brief idea using two sequential unknown procedure calls as an example, followed by the general algorithm which is a bit complicated.

Suppose we have

$$\{\mathsf{Pre}\}\ \{\mathsf{unkFn}_1(\boldsymbol{x}_0;\boldsymbol{y}_0); C; \mathsf{unkFn}_2(\boldsymbol{x}_1;\boldsymbol{y}_1)\}\ \{\mathsf{Post}\}$$

where C is the only known code fragment within the block. Our current (partial) solution assumes that there exist common specifications for the two unknown procedures and attempts to find such specifications.

Our algorithm works in three steps. In the first step, it extracts the precondition for the first procedure, say Pre_U, from the given precondition Pre by extracting the part of heap that may be accessed by the call via $\boldsymbol{x}_0$ and $\boldsymbol{y}_0$, which is similar to the first step of the main algorithm Verify. It then assumes that the second procedure has the same precondition Pre_U (under the assumption mentioned above). In the second step, it symbolically executes the code fragment C with the help of the abductor, to discover a crude postcondition, say Post_U^0, expected from the first unknown call. This is similar to the second step of the main algorithm Verify, except that the postcondition for C is now assumed to be Pre_U. In the third step, the algorithm takes Post_U^0 (with appropriate variable substitutions) as the postcondition of the second unknown call, and checks whether or not the derived post (Post_U^0) satisfies Post. If not, it invokes another abduction to strengthen Post_U^0 to obtain the final postcondition Post_U for the unknown procedures. Note that this strengthening does not affect soundness: the strengthened Post_U can still be used as a postcondition for both unknown procedures.

Figure 5 presents the algorithm to infer specifications for n ($n \geq 2$) unknown calls in sequence.

As aforementioned, for several unknown procedure calls in sequence, we provide a partial solution by assuming that there is a specification (Pre, Post) suit-

```
$\mathsf{SeqUnkCalls} : \mathsf{AllSpec} \times \mathcal{P}(\mathsf{SH} \times \mathsf{SH}) \times U \rightharpoonup \mathcal{P}(\mathsf{SH} \times (\mathsf{SH} \cup \{\top\}))$
Algorithm $\mathsf{SeqUnkCalls}(\mathcal{T}, \mathsf{Spec}_U, U)$
 1  Denote $U$ as $\mathsf{unkFn}(\boldsymbol{x}_0; \boldsymbol{y}_0)\ \{; C_i; \mathsf{unkFn}(\boldsymbol{x}_i; \boldsymbol{y}_i)\}_{i=1}^{n}$
 2  $R := \emptyset$
 3  foreach $(\mathsf{Pre}, \mathsf{Post}) \in \mathsf{Spec}_U$ do
 4    $\mathsf{Pre}_U := \mathsf{Local}(\mathsf{Pre}, \{\boldsymbol{x}_0, \boldsymbol{y}_0\})$
 5    Denote $\boldsymbol{z}_0 = \mathsf{fv}(\mathsf{Pre}_U) \setminus \{\boldsymbol{x}_0, \boldsymbol{y}_0\}$
 6    $\mathsf{Pre}_U := [\boldsymbol{a}/\boldsymbol{x}_0, \boldsymbol{b}/\boldsymbol{y}_0, \boldsymbol{c}/\boldsymbol{z}_0]\mathsf{Pre}_U$
 7    $S'_0 := \{(\mathsf{Pre} \wedge \boldsymbol{y}_0{=}\mathsf{old}(\boldsymbol{y}_0), \mathsf{emp} \wedge \boldsymbol{a}{=}\boldsymbol{x}_0 \wedge \boldsymbol{b}{=}\boldsymbol{y}_0 \wedge \boldsymbol{c}{=}\boldsymbol{z}_0)\}$
 8    for $i := 1$ to $n$ do
 9      $S_i := [\![C_i]\!]^{\mathsf{A}}_{\mathcal{T}}\{\ (\mathsf{Post}_{i-1}{*}[\mathsf{old}(\boldsymbol{b})/\boldsymbol{y}_{i-1}]\ \mathsf{Frame}(\Delta_{i-1}, \{\boldsymbol{x}_{i-1}, \boldsymbol{y}_{i-1}\}), \mathsf{Post}_{i-1}) \mid$
            $(\Delta_{i-1}, \mathsf{M}_{i-1}){\in}S'_{i-1} \wedge \mathsf{Post}_{i-1} = ([\boldsymbol{x}_{i-1}/\boldsymbol{a}, \boldsymbol{y}_{i-1}/\boldsymbol{b}, \boldsymbol{z}_{i-1}/\boldsymbol{c}]\ \mathsf{sub\_alias}($
            $\mathsf{M}_{i-1}, \{\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}\})) \wedge \boldsymbol{a}{=}\boldsymbol{x}_{i-1} \wedge \boldsymbol{b}{=}\boldsymbol{y}_{i-1} \wedge \boldsymbol{c}{=}\boldsymbol{z}_{i-1}\}$ where $\boldsymbol{z}_{i-1}$ is fresh
10      $S'_i := \{(\Delta, \mathsf{M}) \mid (\Delta, \mathsf{M}){\in}S_i \wedge \sigma\Delta \vdash \exists \boldsymbol{c} \cdot \mathsf{Pre}_U{*}\mathtt{true}\} \cup \{(\Delta * \mathsf{M}', \mathsf{M} * \mathsf{M}') \mid$
            $(\Delta, \mathsf{M}){\in}S_i \wedge \sigma\Delta \nvdash \exists \boldsymbol{c} \cdot \mathsf{Pre}_U{*}\mathtt{true} \wedge \sigma\Delta{*}[\mathsf{M}'] \triangleright \exists \boldsymbol{c} \cdot \mathsf{Pre}_U\}$
            where $\sigma{=}[\boldsymbol{a}/\boldsymbol{x}_i, \boldsymbol{b}/\boldsymbol{y}_i]$
11      if $\exists(\Delta, \mathsf{M}){\in}S'_i \cdot \mathsf{fv}(\mathsf{M}) \nsubseteq \mathsf{ReachVar}(\Delta, \{\boldsymbol{a}, \boldsymbol{b}\})$ then
          return $(\mathsf{fail}, \mathsf{Local}(\mathsf{M}, \{\boldsymbol{x}_0, \mathsf{old}(\boldsymbol{y}_0)\}))$ endif
12    end for
13    $S_{n+1} := \{\ (\mathsf{Post}_n{*}[\mathsf{old}(\boldsymbol{b})/\boldsymbol{y}_n]\mathsf{Frame}(\Delta_n, \{\boldsymbol{x}_n, \boldsymbol{y}_n\}), \mathsf{Post}_n) \mid (\Delta_n, \mathsf{M}_n){\in}S'_n \wedge$
            $\mathsf{Post}_n = ([\boldsymbol{x}_n/\boldsymbol{a}, \boldsymbol{y}_n/\boldsymbol{b}, \boldsymbol{z}_n/\boldsymbol{c}]\ \mathsf{sub\_alias}(\mathsf{M}_n, \{\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}\})) \wedge$
            $\boldsymbol{a}{=}\boldsymbol{x}_n \wedge \boldsymbol{b}{=}\boldsymbol{y}_n \wedge \boldsymbol{c}{=}\boldsymbol{z}_n\}$ where $\boldsymbol{z}_n$ is fresh
14    $S'_{n+1} := \{(\Delta, \mathsf{M}) \mid (\Delta, \mathsf{M}){\in}S_{n+1} \wedge \Delta \vdash \mathsf{Post}{*}\mathtt{true}\} \cup \{(\Delta * \mathsf{M}', \mathsf{M} * \mathsf{M}') \mid$
            $(\Delta, \mathsf{M}){\in}S_{n+1} \wedge \Delta \nvdash \mathsf{Post}{*}\mathtt{true} \wedge \Delta{*}[\mathsf{M}'] \triangleright \mathsf{Post}\}$
15    if $\exists(\Delta, \mathsf{M}){\in}S'_{n+1} \cdot \mathsf{fv}(\mathsf{M}) \nsubseteq \mathsf{ReachVar}(\Delta, \{\boldsymbol{a}, \boldsymbol{b}\})$ then
        return $(\mathsf{fail}, \mathsf{Local}(\mathsf{M}, \{\boldsymbol{x}_0, \mathsf{old}(\boldsymbol{y}_0)\}))$ endif
16    foreach $(\Delta, \mathsf{M}) \in S'_{n+1}$ do
17      $\mathsf{Post}_U := \mathsf{sub\_alias}(\mathsf{M}, \{\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c}\})$
18      $g := \mathsf{fv}(\mathsf{Pre}_U) \cap \mathsf{fv}(\mathsf{Post}_U) \setminus \{\boldsymbol{a}, \boldsymbol{b}\}$
19      $R := \mathsf{Refine}(R \cup \{(\exists \mathsf{fv}(\mathsf{Pre}_U) \setminus (g \cup \{\boldsymbol{a}, \boldsymbol{b}\}) \cdot \mathsf{Pre}_U, \mathsf{Post}_U)\})$
20    end foreach
21  end foreach
22  return $R$
end Algorithm
```

Fig. 5. Algorithm for sequential unknown calls.

able for each unknown procedure call, i.e., for any specification of such unknown procedure calls $(\mathsf{Pre}', \mathsf{Post}')$, we have $\mathsf{Pre} \preceq \mathsf{Pre}'$ and $\mathsf{Post}' \preceq \mathsf{Post}$. Then this algorithm will try to find $(\mathsf{Pre}, \mathsf{Post})$.

Given a block of $(n{+}1)$ unknown procedure calls with n pieces of known code blocks sandwiched among them ($\mathsf{unkFn}(\boldsymbol{x}_0; \boldsymbol{y}_0)\ \{; C_i; \mathsf{unkFn}(\boldsymbol{x}_i; \boldsymbol{y}_i)\}_{i=1}^{n}$ in line 1), and the specification $(\mathsf{Pre}, \mathsf{Post})$ (line 3) for such a block, our approach generally works in three steps: first, to compute a precondition for all the unknown calls; second, to verify each code fragment $C_i (i = 1, ..., n)$ with abduction to collect expected behavior of the unknown calls (as part of their postcondition); third, to guarantee that the collected postcondition satisfies Post. If not, then another abduction is conducted to strengthen the gained postcondition to ensure this.

The first step is completed by lines 4 to 6. The local part of Pre is extracted w.r.t. the first unknown call's parameters $\boldsymbol{x}_0$ and $\boldsymbol{y}_0$. Other free variables are distinguished as $\boldsymbol{z}_0$, which may be ghost variables. Finally the precondition is found in terms of special logical variables $\boldsymbol{a}, \boldsymbol{b}$ and $\boldsymbol{c}$.

The second step is performed over each $C_i; \mathsf{unkFn}(\boldsymbol{x}_i; \boldsymbol{y}_i)$. Its main idea is to take the postcondition generated for the last unknown call (Post_{i-1}) plus the frame part as the post-state of $\mathsf{unkFn}(\boldsymbol{x}_{i-1}; \boldsymbol{y}_{i-1})$, and try to verify C_i beginning with such a state, using abduction when necessary (line 9). After the verification we get S_i containing abstract states before $\mathsf{unkFn}(\boldsymbol{x}_i; \boldsymbol{y}_i)$, and we want those states to satisfy its precondition Pre_U w.r.t. substitution. Note that during the verification of C_i and the last satisfaction checking we may use abduction to strengthen the program state, whose results reflect the expected behavior of $\mathsf{unkFn}(\boldsymbol{x}_{i-1}; \boldsymbol{y}_{i-1})$ and are accumulated as its expected postcondition. Hence after this step we achieve a sufficiently strong postcondition for each unknown call.

The third step is similar as the first algorithm: to check the final abstract state can establish the postcondition of the whole block, and to strengthen it with abduction if it cannot. Then the ghost variables are recognized and processed in an analogous way to the first algorithm. Finally the strongest specifications discovered for those unknown procedures are returned.

Remarks: Our current solution to deal with unknown calls in sequence is based on the assumption that there is a common specification ($\mathsf{Pre}, \mathsf{Post}$), suitable for all the unknown procedures. Our current solution is to find out this specification. In general, the unknown procedures should be allowed to have different specifications, which in theory, can always be achieved by analyzing the known code fragments among those unknown calls. The analysis of a code fragment C_i results in the postcondition for the $(i{-}1)$-th procedure and a precondition for the i-th. Currently we are trying to develop new heuristics to solve this. In the case of two unknown calls $\mathsf{unkFn}_0(\boldsymbol{x}_0; \boldsymbol{y}_0); C_1; \mathsf{unkFn}_1(\boldsymbol{x}_1; \boldsymbol{y}_1)$, unkFn_0's precondition and unkFn_1's postcondition are easy to get. For unkFn_0's postcondition Post_0 and unkFn_1's precondition Pre_1, we initialize Post_0 to be emp to start a forward analysis over C_1 with abduction, to accumulate some expected behavior of unkFn_0 as Post_0, and obtain the local part of the abstract state at the end of C_1, which is related to unkFn_1, as Pre_1. However, according to our experience, the Post_0 and Pre_1 are quite weak here (unless C_1 is sufficiently complex to expose much behavior of unkFn_0). In this situation we can employ the frame rule [18] to add part of the symbolic heap to both Post_0 and Pre_1 which is not touched by C_1, in order to tune the gained specifications with more sense. At the moment we are exploring related works and designing heuristics to improve this aspect.

6.2 Soundness and Precision

Informally, in the presence of unknown procedure calls, the soundness of the verification signifies that, a program is successfully verified against its specifications, if all the unknown procedures that it invokes conform to the specifications discovered by the verification algorithm. Therefore, the correctness of the program depends on a (possible) further verification for the unknown procedures. It can be defined as follows:

Definition 1 (Soundness). *Suppose for specification table $\mathcal{T}$, program to be verified $V = \{C_1; U_{(\boldsymbol{x};\boldsymbol{y})}; C_2\}_{(\boldsymbol{x}_0;\boldsymbol{y}_0)}$ and its specifications Spec_V, our verification succeeds and returns $\mathcal{T}_U$ as the specification table for unknown procedures invoked in V. Then we say our verification is sound, if the following holds:*

$$(\forall \Delta \in [\![C_1; U; C_2]\!]_{\mathcal{T} \uplus \mathcal{T}_U} \{[\boldsymbol{x}_0/\boldsymbol{a}, \boldsymbol{y}_0/\boldsymbol{b}]\mathsf{Pre}\} \cdot \Delta \vdash [\boldsymbol{x}_0/\boldsymbol{a}, \boldsymbol{y}_0/\boldsymbol{b}, \mathsf{old}(\boldsymbol{y}_0)/\mathsf{old}(\boldsymbol{b})]\mathsf{Post} * \mathtt{true})$$

which means that, with respect to the underlying semantics, if all the unknown procedures can be verified to satisfy their specifications in $\mathcal{T}_U$, then the whole program V should meet all the specifications in Spec_V.

The soundness definition above is built into the algorithm Post_Check at line 20 in the algorithm Verify. Upon return, the algorithm conducts another forward verification on the whole program, assuming that the unknown procedures are already verified against the discovered specifications. As in Calcagno et al. [5], this final check rules out any potentially unsound pre/post pairs (due to the use of abstraction); therefore, it ensures the soundness of our verification.

Now we have a brief discussion about soundness and precision. This mainly concerns the specifications we obtain for the unknown procedures, and we define a partial order for them first:

$$(\mathsf{Pre}, \mathsf{Post}) \preceq (\mathsf{Pre}', \mathsf{Post}') =_{df} \mathsf{Pre}' \preceq \mathsf{Pre} \wedge \mathsf{Post} \preceq \mathsf{Post}'$$

where we overload the operator $\preceq$ for specifications as well. By saying "specification $(\mathsf{Pre}', \mathsf{Post}')$ is stronger than $(\mathsf{Pre}, \mathsf{Post})$", we denote that Pre' is weaker than Pre and Post' is stronger than Post, according to the partial order for symbolic heaps. Its intuitive meaning is that $(\mathsf{Pre}', \mathsf{Post}')$ is a refinement of $(\mathsf{Pre}, \mathsf{Post})$, namely, any implementation satisfying $(\mathsf{Pre}', \mathsf{Post}')$ will satisfy $(\mathsf{Pre}, \mathsf{Post})$, and if an unknown procedure should meet $(\mathsf{Pre}, \mathsf{Post})$ to complete the verification, then it is always safe if it meets $(\mathsf{Pre}', \mathsf{Post}')$.

Naturally our partial order forms a lattice of specifications, with the top $(\mathtt{true}, \mathtt{false})$ and the bottom $(\mathtt{false}, \mathtt{true})$. For any two comparable specifications $S_1 \preceq S_2$, if both are sound to be chosen as the specification for an unknown call, then a better selection would be S_1, since choosing S_2 will reject some possible implementations for the unknown procedure.

Example 5 (Soundness and precision). Take the unknown procedure `unkProc(a;b)` in our motivating example for instance, its postcondition is $\mathsf{ls}(a, b) * \mathsf{ls}(b, \mathtt{null}) \wedge b \neq \mathtt{null}$. This allows a possible implementation of the unknown procedure to be `b:=a.next; if (b.next!=null) b:=b.next;` . However, If we discovered a stronger postcondition $a{\mapsto}b * \mathsf{ls}(b, \mathtt{null}) \wedge b{\neq}\mathtt{null}$, then the aforementioned codes would be ruled out, although it is correct according to `findLast`.

Thus our goal is to try to maintain as much precision as possible by choosing weaker specifications, as long as it is still sound. This is the reason to improve the abduction as said in Section 4. We use abduction to discover expected postconditions of unknown procedures, and any free variable occurring in the postcondition, which is not a ghost variable (shared with the precondition), will become universally quantified. However, such variables can make the postcondition quite strong, so the specification will also become strong in this case. For example, if we adopted the abduction in Calcagno et al. [5], then the postcondition for `unkProc(a;b)` in the motivating example would be $\forall c \cdot \mathsf{ls}(a, c) * \mathsf{ls}(b, \mathtt{null}) \wedge b{\neq}\mathtt{null}$, which can never be satisfied by any program. Conversely, our abduction always tries to introduce fewer free variables, to preserve as much precision as possible and to guarantee soundness in the mean time.

7 Experiments and Evaluation

We have implemented the two abstract semantics and the verification algorithms with Objective Caml and evaluated them over some list processing programs. The results are in Table 1. The first and second columns denote the programs used for evaluation [4, 7] and their time consumption, respectively. We manually hide some instructions in the original programs as calls to unknown procedures, for which we try to discover specifications during the verification process. The third column shows the given specifications for each program's verification. The last column exhibits the discovered specifications for unknown procedures inside those programs. The programs' codes are listed in Figure 6.

We note down two observations on the experimental results. The first is that the discovered specifications for the unknown procedures are more general than we expect. Bear in mind that we have replaced some instructions from those programs with unknown calls. We have compared the inferred specifications for those unknown calls with the original instructions. The results show that the specifications derived by our algorithm not only fully capture the behaviors of those instructions, but also suggest other possible implementations. A case in point is our motivating example `findLast` given in Sec 2, where the original code replaced by unknown call is `y := x.next` with the post-state $x{\mapsto}y * \mathsf{ls}(y, \mathtt{null}) \wedge y{\neq}\mathtt{null}$. According to our result, the post-state can be more general as $\mathsf{ls}(x, y) * \mathsf{ls}(y, \mathtt{null}) \wedge y{\neq}\mathtt{null}$, suggesting that as long as y is not `null`, it can traverse any number of nodes towards the tail of the list.

The other observation is about the last program listed in the table, a procedure called `iWillFail`, the code of which is given as follows:

```
void iWillFail(node x, node y; ) { unknown(x; ) }
```

Program	Time(s)	Program Specification	Discovered Unknown Specification
`findLast` $(x; z)$	0.00135	$\mathsf{Pre} := \mathsf{ls}(x, \mathtt{null}) \wedge x{\neq}\mathtt{null}$ $\mathsf{Post} := \mathsf{ls}(x, z) * z{\mapsto}\mathtt{null}$	$\mathsf{Pre}{:=}\ \exists w \cdot a{\mapsto}w * \mathsf{ls}(w, \mathtt{null}) \wedge a{\neq}\mathtt{null} \wedge w{\neq}\mathtt{null}$ $\mathsf{Post}{:=}\mathsf{ls}(a, b){*}\mathsf{ls}(b, \mathtt{null}){\wedge}b{\neq}\mathtt{null}$
`append` $(y; x)$	0.00216	$\mathsf{Pre} := \mathsf{ls}(x, \mathtt{null}){*}\mathsf{ls}(y, \mathtt{null})$ $\mathsf{Post} := \mathsf{ls}(x, y) * \mathsf{ls}(y, \mathtt{null})$	$\mathsf{Pre} := \mathsf{ls}(a, \mathtt{null})$ $\mathsf{Post} := \mathsf{ls}(a, b) * \mathsf{ls}(b, \mathtt{null})$
`copy`$(x; y)$	0.00204	$\mathsf{Pre} := \mathsf{ls}(x, \mathtt{null})$ $\mathsf{Post}{:=}\mathsf{ls}(x, \mathtt{null}){*}\mathsf{ls}(y, \mathtt{null})$	$\mathsf{Pre} := \mathsf{ls}(a, \mathtt{null})$ $\mathsf{Post} := \mathsf{ls}(a, \mathtt{null})$
`revCopy` $(x; y)$	0.00107	$\mathsf{Pre} := \mathsf{ls}(x, \mathtt{null})$ $\mathsf{Post}{:=}\mathsf{ls}(x,\mathtt{null}){*}\mathsf{ls}(y,\mathsf{old}(y))$	$\mathsf{Pre} := \mathtt{true}$ $\mathsf{Post} := \mathtt{true}$
`clear`$(; x)$	0.00239	$\mathsf{Pre} := \mathsf{ls}(x, \mathtt{null})$ $\mathsf{Post} := \mathsf{emp} \wedge x{=}\mathtt{null}$	$\mathsf{Pre} := a{\mapsto}b * \mathsf{ls}(b, \mathtt{null})$ $\mathsf{Post} := \mathsf{ls}(b, \mathtt{null})$
`appendThree` $(y, z; x)$	0.00315	$\mathsf{Pre}{:=}\ \mathsf{ls}(x, \mathtt{null}){*}\mathsf{ls}(y, \mathtt{null}){*} \mathsf{ls}(z, \mathtt{null})$ $\mathsf{Post}{:=}\mathsf{ls}(x, \mathtt{null})$	$\mathsf{Pre} := \mathsf{ls}(a, \mathtt{null}) * \mathsf{ls}(b, \mathtt{null})$ $\mathsf{Post} := \mathsf{ls}(a, \mathtt{null})$
`towardsLast` $(x; y)$	0.00428	$\mathsf{Pre}{:=}\ \mathsf{ls}(x, \mathtt{null}) \wedge x{\neq}\mathtt{null}$ $\mathsf{Post}{:=}\mathsf{ls}(x, y) * \mathsf{ls}(y, \mathtt{null}) \wedge x{\neq}\mathtt{null}$	$\mathsf{Pre}{:=}\ \mathsf{ls}(a, \mathtt{null}) \wedge a{\neq}\mathtt{null}$ $\mathsf{Post}{:=}\mathsf{ls}(a, \mathtt{null}) * \mathsf{ls}(b, \mathtt{null}) \wedge a{\neq}\mathtt{null} \wedge b{\neq}\mathtt{null}$
`iWillFail` $(x, y;)$	0.00066	$\mathsf{Pre}{:=}\ \mathsf{ls}(x, \mathtt{null})$ $\mathsf{Post}{:=}\mathsf{ls}(x, \mathtt{null}) * \mathsf{ls}(y, \mathtt{null})$	$(\mathsf{fail}, \mathsf{ls}(y, \mathtt{null}))$

Table 1. Experimental results for list programs.

This program is expected to be verified against the specification ($\mathsf{Pre} = \mathsf{ls}(x, \mathtt{null})$, $\mathsf{Post} = \mathsf{ls}(x, \mathtt{null}) * \mathsf{ls}(y, \mathtt{null})$). Our verification fails and returns an additional formula $\mathsf{ls}(y, \mathtt{null})$ from our abduction process. A further analysis reveals that the failure is actually due to the given specification where the precondition Pre is too weak for the program to establish the postcondition Post: since y is not reachable from the parameters of the unknown call, no implementation of the unknown call can establish the postcondition Post involving y. The returned formula from our verification can then be used to strengthen the given specification. In this case, if we add $\mathsf{ls}(y, \mathtt{null})$ into Pre via separation conjunction, the verification will succeed with the specification ($\mathsf{Pre_u} = \mathsf{ls}(x, \mathtt{null}), \mathsf{Post_u} = \mathsf{ls}(x, \mathtt{null})$) discovered for the procedure `unknown`.

8 Conclusion

It is a challenging problem to automatically verify (even pointer safety of) heap-manipulating imperative programs with unknown procedure calls. We propose a novel approach to this problem, which infers expected specifications for unknown procedures from their calling contexts during the verification process. The program is proven correct on condition that the invoked unknown procedures meet the inferred specifications. We employ a forward shape analysis with separation logic and an enhanced abductive reasoning mechanism to synthesize both pre- and postconditions of the unknown procedure. As a proof of concept, we

```
findLast(x; ref z) {
  node w := [x], y;
  if w = null then z := x
  else
    unknown(x; y);
    findLast(y; z)
  fi
}
```

```
append (y; ref x) {
  if x = null then x := y
  else
    unknown(x; w);
    if w = null then [x] := y
    else append(y; w)
    fi
  fi
}
```

```
copy(x; ref y) {
  if x = null then y := null
  else
    w := [x];
    copy (w; y);
    unknown(; y);
  fi
}
```

```
revCopy(x; ref y) {
  if x = null then skip
  else
    unknown(; y);
    w := [x];
    revCopy (w; y)
  fi
}
```

```
clear(; ref x) {
  if x = null then skip
  else
    w := [x];
    unknown(x, w; );
    x := w;
    clear(; x)
  fi
}
```

```
appendThree(y, z; ref x) {
  unknown(x, y; );
  unknown(x, z; )
}
```

```
towardsLast(x; y) {
  unknown(; x);
  unknown(x; y)
}
```

```
iWillFail(x, y; ) {
  unknown(x; )
}
```

Fig. 6. Codes of experimental examples.

have also implemented a prototype system to test the viability of the proposed approach.

There are two possible future directions. One is to explore more general solution for unknown calls in sequence as discussed, e.g., it might be possible for us to invent some heuristics to strengthen the postcondition for the first one, so that the precondition of the second can be strengthened accordingly, to achieve more reasonable specifications for both. Another direction is to extend this method to an abstract domain combining separation and numerical information [6], so that more general properties, such as memory safety and functional correctness, can be specified and verified. We envisage that, with the combined domain, the abstract semantics and analysis algorithms will remain conceivably the same, but the abduction will be redefined to discover the anti-frames for the newly introduced numerical features.

Acknowledgement This work was supported in part by the EPSRC projects [EP/E021948/1 and EP/G042322/1]. We thank Hongseok Yang for his encouragement and invaluable comments and Dino Distefano for very useful discussions on this work.

References

1. G. Ammons, R. Bodik, and J. R. Larus. Mining specifications. In *Proc. of 29th POPL*, pages 4–16, 2002. ACM Press.
2. M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *Proc. of The CASSIS Workshop*, volume 3362 of *Lecture Notes in Computer Science*, pages 49–69. Springer, 2004.
3. J. Berdine, C. Calcagno, and P. W. O'Hearn. Symbolic execution with separation logic. In *Proc. of APLAS*, pages 52–68. Springer, 2005.
4. C. Calcagno, D. Distefano, P. O'Hearn, and H. Yang. Footprint analysis: A shape analysis that discovers preconditions. In *Proc. of SAS*, 2007.
5. C. Calcagno, D. Distefano, P. O'Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. In *Proc. of 36th POPL*, January 2009. ACM Press.
6. W.-N. Chin, C. David, H. H. Nguyen, and S. Qin. Automated verification of shape, size and bag properties. In *Proc. of 12th ICECCS*, pages 307–320, 2007.
7. F. Craciun, C. Luo, G. He, S. Qin, and W.-N. Chin. Discovering specifications for unknown procedures (work in progress). In *Proc. of Workshop of Invariant Generation (WING)*, 2009.
8. D. Distefano, P. W. O'Hearn, and H. Yang. A local shape analysis based on separation logic. In *Proc. of TACAS*, volume 3920 of *LNCS*. Springer, 2006.
9. M. Emami, R. Ghiya, and L. J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proc. of PLDI*, 1994.
10. R. Giacobazzi. Abductive analysis of modular logic programs. In *Proc. of ILPS*, pages 377–391. The MIT Press, 1994.
11. D. Gopan and T. Reps. Low-level library analysis and summarization. In *Proc. of 19th CAV*, 2007.
12. C. Jones, P. O'Hearn, and J. Woodcock. Verified Software: A Grand Challenge. *IEEE Computer*, 39(4):93–95, April 2006.
13. G. T. Leavens, C. Ruby, K. R. M. Leino, E. Poll, and B. Jacobs. JML: notations and tools supporting detailed design in Java. In *Proc. of OOPSLA*, pages 105–106, 2000. ACM Press.
14. G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: a behavioral interface specification language for Java. *SIGSOFT Software Engineering Notes*, 31(3):1–38, 2006.
15. P. Müller, A. Poetzsch-Heffter, and G. T. Leavens. Modular specification of frame properties in JML. *Concurrency and Computation: Practice and Experience*, 15:117–154, 2003.

16. H. H. Nguyen, C. David, S. Qin, and W.-N. Chin. Automated verification of shape and size properties via separation logic. In *Proc. of 8th VMCAI*, volume 4349 of *LNCS*, 2007.
17. P. W. O'Hearn, H. Yang, and J. C. Reynolds. Separation and information hiding. In *Proc. of 31st POPL*, January 2004. ACM Press.
18. J. C. Reynolds. Separation logic: a logic for shared mutable data structures. In *Proc. of 17th LICS*, pages 55–74, 2002.
19. J. Woodcock. Verified software grand challenge. In *Proc. of 14th FM*, pages 617–617, 2006.
20. H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. O'Hearn. Scalable shape analysis for systems code. In *Proc. of 20th CAV*, volume 5123 of *LNCS*, pages 385–398. Springer, April 2008.