Discovering Specifications for Unknown Procedures - Work in Progress

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

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

Abstract

We study automated verification of pointer safety for heap-manipulating imperative programs with unknown procedure calls or code pointers. Given the specification of a procedure whose body contains calls to an unknown procedure, we try to infer the possible specifications for the unknown procedure from its calling contexts. We employ a forward shape analysis with separation logic and an abductive inference mechanism to synthesize both pre- and postconditions for the unknown procedure. The inferred specification is a partial specification of the unknown procedure. Therefore it is subject to a later verification when the code or the complete specification for the unknown procedure are available. Our inferred specifications can also be used for program understanding.

1 Introduction

A program verification system takes as input a given program and a specification of its desired properties and attempts to determine whether or not the program meets the specified properties. The verifier usually requires to have access to the entire given program which, in practice, may not be completely available for various reasons. In general the unknown program fragments can correspond to unknown procedure calls which might be either calls to library procedures without source codes or code pointers with their corresponding meta-programming features like run-time generation/loading of code.

To deal with the verification of programs with unknown procedure calls, current program verifiers either

- stop at the first unknown procedure call and provide an incomplete verification, or
- simply ignore the unknown procedure calls (e.g. replace them by skip) or treat them as nondeterministic assignments without side effects (both methods can be unsound in general) ([4]), or
- assume the variables that are modified by the unknown procedure call and its caller do not overlap (so they have different so-called *footprints* and the hypothetical frame rule [15] can be applied; however this assumption does not hold in most cases), or
- use specification mining [1] to discover possible specifications for the program (which is performed dynamically and is just a check instead of a proof for program correctness), or
- ask for a description of the unknown procedure properties, which might be obtained by analyzing their binary code ([2, 8, 12, 10]). In the case of code pointers it is necessary

to have an analysis that can estimate all possible procedures to which the pointers may refer at run-time. In general this estimation may return too many candidates, making the verification almost impossible at compile time.

In this paper we propose a different approach to the verification of programs with unknown procedure calls. Based on the desired properties specified for the entire program, our solution is to verify the known program fragments, and to postpone the verification of the unknown procedures by inferring partial specifications of their desired properties. The inferred specifications are subject to a later verification when the code or the complete specification of the unknown procedure become known (e.g. at loading time in Java). Our inferred specifications can also be used for program understanding to build partial specifications of the generic procedures (e.g. the methods of Java interfaces). The inferred specifications are derived from the calling contexts. Therefore, in the case of multiple specifications for the same unknown procedure they have to be combined together in order to get a more general (stronger) specification. Even though our approach can not discover the complete specification of the unknown procedure, what we can discover can be used to completely describe the unknown procedure behaviour inside the analyzed module (e.g. the behaviour of the implementation of method `compareTo` of interface `Comparable` inside the current analyzed Java package, `JavaCup`).

We use our approach to verifying the pointer safety of heap-manipulating programs with unknown procedure calls. Our framework is built on the work on shape analysis with separation logic [16, 7, 13]. Program specifications are given as pre and post conditions for each known procedure. We perform a modular verification on a per procedure basis as in the HIP system [13]. Starting with the given precondition, a forward verification is run until the unknown procedure call is met. A precondition for the unknown procedure is derived from the current heap state ([11]). Thus the current heap is split into two disjoint heaps: (1) the unknown procedure's precondition consisting of all of the current heap nodes reachable from the unknown procedure's actual parameters, and (2) a frame. The frame is used as initial state for the forward verification of any program code that follows the unknown procedure call. The verification might find situations where there is not enough information to perform an operation such as a (known) procedure call or a dereferencing. If the missing information refers to the actual arguments or to the result of the unknown procedure call then we perform a similar abductive inference as that in Calcagno et al. [3, 4] to infer what is missing. The inferred information will be part of the unknown procedure postcondition. If the missing information does not refer to the unknown procedure then there is an error in the code.

Our proposal is mainly inspired by the recent work [4] that uses bi-abduction to infer the pre/post conditions of a given procedure by analysing its body. It uses a bottom up approach such that the procedures called in the body of the given procedure are known and their pre/post conditions were inferred prior to the current procedure. The problem we solve in this paper is dual to that in Calcagno et al. [4]. Given the pre/post conditions of a procedure whose body invokes an unknown procedure, we attempt to infer the pre/post conditions of the unknown procedure. A similar problem to ours has been solved in the context of logic programs [9], where, based on the specification and the implementation of a logic programming module, an abductive method infers constraints on undefined literals of that module.

Our proposal can also be regarded as a top-down inference system rather than a bottom-up one [4] that most traditional approaches rely on. In our case the user provides the specification for the top-level procedure and our approach can infer the specifications for the called procedures. This process may be repeated at several levels down. However the inferred specifications may not be complete, therefore they are subjected to a later verification.

A verification system uses an entailment procedure to check whether the program heap state $\Delta_{\mathtt{i}}$ computed at the program instruction i entails the precondition $\mathsf{Pre}_{\mathtt{i+1}}$ (namely the requirements for a correct execution) of the next program instruction $\mathtt{i+1}$ as follows:

$$\Delta_{\mathtt{i}} \vdash \mathsf{Pre}_{\mathtt{i+1}}$$

In general not the entire current program state $\Delta_{\mathtt{i}}$ is required to establish $\mathsf{Pre}_{\mathtt{i+1}}$. Therefore a more general form of the entailment is used:

$$\Delta_{\mathtt{i}} \vdash \mathsf{Pre}_{\mathtt{i+1}} * \mathsf{R}_{\mathtt{i}}$$

where $*$ stands for the separation conjunction [16], and $\mathsf{R}_{\mathtt{i}}$ is the frame, namely the part of the heap state $\Delta_{\mathtt{i}}$ which is not accessed by the program instruction $\mathtt{i+1}$. When the entailment fails

$$\Delta_{\mathtt{i}} \nvdash \mathsf{Pre}_{\mathtt{i+1}}$$

we can use a reasoning with abduction

$$\Delta_{\mathtt{i}} * [\mathsf{M}_{\mathtt{i}}] \triangleright \mathsf{Pre}_{\mathtt{i+1}}$$

to find the missing part $\mathsf{M}_{\mathtt{i}}$ of the heap state $\Delta_{\mathtt{i}}$ such that

$$\Delta_{\mathtt{i}} * \mathsf{M}_{\mathtt{i}} \vdash \mathsf{Pre}_{\mathtt{i+1}} * \mathsf{R}_{\mathtt{i}}$$

In principle the abduction strengthens the current heap state $\Delta_{\mathtt{i}}$. Our approach mainly uses the abduction to discover the postconditions of the unknown procedures. However a naive application of the abduction may generate too strong postconditions for the unknown procedures, in which cases acceptable implementations of the unknown procedures can be rejected as being incorrect. Therefore, in Section 2 we define the property that has to be satisfied by an abductive system in order to be useful for our approach.

The remainder of the paper is organized as follows. Section 2 introduces the key features of our approach by a simple example that consists of one unknown procedure call. Section 3 extends our approach to multiple sequential calls of the same unknown procedure. A brief conclusion is then given at the end.

2 Discovering a Specification for an Unknown Procedure

In this section we illustrate how our analysis computes a precondition and a postcondition for an unknown procedure. Before that, we introduce the abstract domain that we will use, which is similar to the one exploited in Calcagno et al. [4].

We have two disjoint sets of variables. One is a finite set of program variables Var (denoted by $\mathtt{x}, \mathtt{y}, \mathtt{z}, ...$) and a countable set of logical variables $LVar$ (expressed by $\mathtt{x}', \mathtt{y}', ...$). The logical variables are never used in programs; they are for recording some historical values of program variables. Hence they may be existentially quantified at any point. *Loc* is a countably infinite set of locations, and Val is a set of values that includes *Loc*. Our *storage model* is a traditional separation logic one:

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

We regard *symbolic heaps* as special separation logic formulae to represent the abstraction of a set of concrete heaps. They are defined as follows:

$$
\begin{array}{lcll}
E & ::= & \mathtt{x} \mid \mathtt{x}' & \textit{Expressions} \\
\Pi & ::= & E = E \mid E \neq E \mid \mathsf{true} \mid \Pi \wedge \Pi & \textit{Pure formulae} \\
\mathsf{B} & ::= & E \mapsto E \mid \mathtt{list}(E, E) & \textit{Basic separation predicates} \\
\Sigma & ::= & \mathsf{B} \mid \mathsf{true} \mid \mathsf{emp} \mid \Sigma * \Sigma & \textit{Separation formulae} \\
\Delta & ::= & \Pi \wedge \Sigma & \textit{Symbolic heaps}
\end{array}
$$

Expressions can be both kinds of variables. Pure formulae are composed by the conjunction of (dis) equalities between expressions to show (non) alias relationship between references. The basic separation predicates describe simple points-to or list segments, which will be introduced later. Separation formulae stand for basic shape predicate, or empty heap, or a separation conjunction of them. A symbolic heap consists of both pure part and separation (shape) part. We overload the separation conjunction over Δ_1 and Δ_2 to conjoin their shape and pure parts, respectively.

Based on the abstract domain defined, consider the procedure `findLast` whose specification and implementation are shown in Figure 1. The procedure searches for the last (non-null) element of a singly linked list. At line `4` it calls the unknown procedure `unkProc`. We use an imperative language with references. The data structure `node { int val; node next }` defines a list element.

We run a standard forward shape analysis based on separation logic. The results of our analysis are marked as comments in the code. A list segment is described inductively by the following separation predicate:

$$\mathsf{list}(\mathsf{x}, \mathsf{y}) \;=_{df}\; (\mathsf{x}{=}\mathsf{y} \wedge \mathsf{emp}) \vee (\mathsf{x}{\mapsto}\mathsf{y}) \vee (\exists \mathsf{z} \cdot \mathsf{x}{\mapsto}\mathsf{z} * \mathsf{list}(\mathsf{z}, \mathsf{y}) \wedge \mathsf{z} \neq \mathsf{y})$$

where the points-to predicate $\mathtt{x}{\mapsto}\mathtt{y}$ denotes a heap with a single allocated node at address `x` with its next field pointing to the address `y`.

The analysis starts with the procedure `findLast` precondition (line `0a`). The verification is straightforward until just before the unknown procedure call. At line `3`, the current heap Δ_3 is split into two disjoint heaps: the local heap $\mathsf{Local}(\Delta_3, \{\mathtt{x}\})$ to be sent to the unknown procedure and the frame heap $\mathsf{Frame}(\Delta_3, \{\mathtt{x}\})$ that is not accessed by the unknown procedure. The local heap is obtained by taking the part of the current heap Δ_3 reachable in the formula from the actual parameter `x` of the unknown procedure.

In general, at a call site $\mathtt{f}(\mathtt{x}_1, .., \mathtt{x}_n)$ the current heap state Δ can be expressed as $\Delta = \Pi \wedge \Sigma$, where Π is a pure formula insensitive to heap (mainly consisting of aliasing information) and Σ expresses heap shape (the shape of linked data structures located on the heap). The part of the heap to be sent to the procedure is denoted by $\mathsf{Local}(\Pi \wedge \Sigma, \{\mathtt{x}_1, .., \mathtt{x}_n\})$ and is defined as follows:

$$
\begin{array}{rcl}
\mathsf{Local}(\Pi \wedge \Sigma, \{\mathtt{x}_1, .., \mathtt{x}_n\}) & = & \exists \mathtt{fv}(\Pi \wedge \Sigma) \setminus \mathtt{ReachVar}(\Pi \wedge \Sigma, \{\mathtt{x}_1, .., \mathtt{x}_n\}) \cdot \\
& & \Pi * \mathtt{ReachHeap}(\Pi \wedge \Sigma, \{\mathtt{x}_1, .., \mathtt{x}_n\})
\end{array}
$$

where $\mathtt{fv}(\Delta)$ stands for all the free (program and logical) variables occurring in Δ. And the frame, namely the part of the heap that is not accessed by the procedure call is defined as follows:

$$\mathsf{Frame}(\Pi \wedge \Sigma, \{\mathtt{x}_1, .., \mathtt{x}_n\}) = \Pi \wedge \mathtt{UnreachHeap}(\Pi \wedge \Sigma, \{\mathtt{x}_1, .., \mathtt{x}_n\})$$

where $\mathtt{UnreachHeap}(\Pi \wedge \Sigma, \{\mathtt{x}_1, .., \mathtt{x}_n\})$ is the formula consisting of all $*$-conjuncts from Σ which are not in $\mathtt{ReachHeap}(\Pi \wedge \Sigma, \{\mathtt{x}_1, .., \mathtt{x}_n\})$. The formula $\mathtt{ReachHeap}(\Pi \wedge \Sigma, \{\mathtt{x}_1, .., \mathtt{x}_n\})$ denotes the

```
// Given Specification:
        // $\mathsf{Pre}_{\mathtt{findLast}} := \mathsf{list}(\mathtt{x},\mathtt{null}) \wedge \mathtt{x}{\neq}\mathtt{null}$
        // $\mathsf{Post}_{\mathtt{findLast}} := \mathsf{list}(\mathtt{x},\mathtt{res}) * \mathtt{res}{\mapsto}\mathtt{null}$
node findLast(node x) {
0   node w, y, z;
0a   // $\Delta_0 := \mathsf{Pre}_{\mathtt{findLast}}$
0b   // $\Delta_0 \vdash \mathtt{x}{\neq}\mathtt{null}$
1   w := x.next;
1a   // $\Delta_1 := \mathtt{x}{\mapsto}\mathtt{w} * \mathsf{list}(\mathtt{w},\mathtt{null}) \wedge \mathtt{x}{\neq}\mathtt{null}$
2   if (w == null) return x;
2a   // $\Delta_2 := \mathtt{w}{=}\mathtt{null} \wedge \mathtt{res}{=}\mathtt{x} \wedge \mathtt{x}{\mapsto}\mathtt{null} \wedge \mathtt{x}{\neq}\mathtt{null}$
2b   // $\exists \mathtt{w},\mathtt{y},\mathtt{z} \cdot \Delta_2 \vdash \mathsf{Post}_{\mathtt{findLast}}$
3   else {
3a   // $\Delta_3 := \mathtt{x}{\mapsto}\mathtt{w} * \mathsf{list}(\mathtt{w},\mathtt{null}) \wedge \mathtt{x}{\neq}\mathtt{null} \wedge \mathtt{w}{\neq}\mathtt{null}$
3b   // $\mathsf{Local}(\Delta_3,\{\mathtt{x}\}) := \mathtt{x}{\mapsto}\mathtt{w} * \mathsf{list}(\mathtt{w},\mathtt{null}) \wedge \mathtt{x}{\neq}\mathtt{null} \wedge \mathtt{w}{\neq}\mathtt{null}$
3c   // $\mathsf{Frame}(\Delta_3,\{\mathtt{x}\}) := \mathtt{x}{\neq}\mathtt{null} \wedge \mathtt{w}{\neq}\mathtt{null}$
3d   // $\mathsf{Pre}_{\mathsf{unkProc}} := \mathsf{Local}(\Delta_3,\{\mathtt{x}\})$
4   y := unkProc(x);
4a   // $\Delta_4 := (\exists \mathtt{fv}(\mathsf{Pre}_{\mathsf{unkProc}}) \cdot \mathsf{Frame}(\Delta_3,\{\mathtt{x}\})) * (\mathsf{emp} \wedge \mathsf{y}'{=}\mathsf{y} \wedge \mathtt{x}'{=}\mathsf{x})$
4b   // $\mathsf{M} := (\mathtt{res}_{\mathsf{unkProc}}{=}\mathsf{y} \wedge \mathsf{y}'{=}\mathsf{y} \wedge \mathtt{x}'{=}\mathsf{x})$
4c   // $\Delta_4 \nvdash [\mathsf{y}/\mathsf{x}]\ \mathsf{Pre}_{\mathtt{findLast}}$
4d   // $\Delta_4 * [\mathsf{M}_1] \triangleright [\mathsf{y}/\mathsf{x}]\ \mathsf{Pre}_{\mathtt{findLast}}$        $\mathsf{M}_1 := \mathsf{list}(\mathsf{y},\mathtt{null}) \wedge \mathsf{y}{\neq}\mathtt{null}$
4e   // $\mathtt{fv}(\mathsf{M}_1) \subseteq \mathtt{ReachVar}(\Delta_4 * \mathsf{M}_1, \{\mathtt{x}',\mathsf{y}'\})$        $\mathsf{M} := \mathsf{M} * \mathsf{M}_1$
4f   // $\Delta_4 * \mathsf{M}_1 \vdash [\mathsf{y}/\mathsf{x}]\ \mathsf{Pre}_{\mathtt{findLast}} * \mathsf{R}_1$        $\mathsf{R}_1 := \mathsf{y}'{=}\mathsf{y} \wedge \mathtt{x}'{=}\mathtt{x}$
5   z := findLast(y);
5a   // $\Delta_5 := \mathsf{R}_1 * [\mathsf{y}/\mathtt{x},\mathsf{z}/\mathtt{res}]\ \mathsf{Post}_{\mathtt{findLast}}$
6   return z;
6a   // $\Delta_6 := \Delta_5 \wedge \mathtt{res}{=}\mathtt{z}$
6b   // $(\exists \mathsf{z},\mathsf{y},\mathsf{y}',\mathtt{x}' \cdot \Delta_6) \nvdash \mathsf{Post}_{\mathtt{findLast}}$
6c   // $(\exists \mathsf{z},\mathsf{y},\mathsf{y}',\mathtt{x}' \cdot \Delta_6) * [\mathsf{M}_2] \triangleright \mathsf{Post}_{\mathtt{findLast}}$        $\mathsf{M}_2 := \mathsf{list}(\mathtt{x},\mathsf{y})$
6d   // $\mathtt{fv}(\mathsf{M}_2) \subseteq \mathtt{ReachVar}(\Delta_6 * \mathsf{M}_2, \{\mathtt{x}',\mathsf{y}'\})$        $\mathsf{M} := \mathsf{M} * \mathsf{M}_2$
7   } }
8 // $\mathsf{Post}_{\mathsf{unkProc}} := \exists \mathtt{fv}(\mathsf{M}) \setminus (\mathtt{fv}(\mathsf{Pre}_{\mathsf{unkProc}}) \cup \{\mathtt{res}_{\mathsf{unkProc}}\}) \cdot \mathsf{M}$
```

Figure 1: Procedure `findLast` calling an unknown procedure `unkProc`.

part of Σ reachable from $\{\mathtt{x_1},..,\mathtt{x_n}\}$ and is formally defined as the $*$-conjunction of the following set of formulae:

$$\{\Sigma_1 \mid \exists \mathtt{z_1},\mathtt{z_2},\Sigma_2 \cdot \mathtt{z_1} \in \mathtt{ReachVar}(\Pi \wedge \Sigma, \{\mathtt{x_1},..,\mathtt{x_n}\}) \wedge \Sigma = \Sigma_1 * \Sigma_2 \wedge \Sigma_1 = \mathsf{B}(\mathtt{z_1},\mathtt{z_2})\}$$

$\mathtt{ReachVar}(\Pi \wedge \Sigma, \{\mathtt{x_1},..,\mathtt{x_n}\})$ is the minimal set of variables which are aliases or are reachable through the heap such that:

$$\begin{array}{l}\{\mathtt{x_1},..,\mathtt{x_n}\} \cup \{\mathtt{z_2} \mid \exists \mathtt{z_1},\Pi_1 \cdot \mathtt{z_1} \in \mathtt{ReachVar}(\Pi \wedge \Sigma, \{\mathtt{x_1},..,\mathtt{x_n}\}) \wedge \Pi = (\mathtt{z_1} = \mathtt{z_2}) \wedge \Pi_1\} \cup \\ \quad \{\mathtt{z_2} \mid \exists \mathtt{z_1},\Sigma_1 \cdot \mathtt{z_1} \in \mathtt{ReachVar}(\Pi \wedge \Sigma, \{\mathtt{x_1},..,\mathtt{x_n}\}) \wedge \Sigma = \mathsf{B}(\mathtt{z_1},\mathtt{z_2}) * \Sigma_1\} \subseteq \\ \qquad \mathtt{ReachVar}(\Pi \wedge \Sigma, \{\mathtt{x_1},..,\mathtt{x_n}\})\end{array}$$

Note that $\mathsf{B}(\mathtt{z_1},\mathtt{z_2})$ stands for either $\mathtt{z_1}{\mapsto}\mathtt{z_2}$ or $\mathsf{list}(\mathtt{z_1},\mathtt{z_2})$.

Thus the precondition of `unkProc` is set to the local heap $\mathsf{Local}(\Delta_3, \{\mathtt{x}\})$ as follows:

$$\mathsf{Pre}_{\mathtt{unkProc}} := \mathtt{x}{\mapsto}\mathtt{w} * \mathsf{list}(\mathtt{w},\mathtt{null}) \wedge \mathtt{x}{\neq}\mathtt{null} \wedge \mathtt{w}{\neq}\mathtt{null}$$

The above precondition is a safe estimation of that part of the calling context heap that may be accessed by the unknown procedure `unkProc`. Therefore it is not one of the weakest possible preconditions. Our computed precondition may also contain cutpoints. In our example the cutpoint is `w`. As can be seen below, this cutpoint does not occur in the postcondition (`II`) of the procedure `unkProc`. Therefore it can be existentially quantified as follows:

$$\mathsf{Pre}_{\mathtt{unkProc}} := \exists \mathtt{w} \cdot (\mathtt{x}{\mapsto}\mathtt{w} * \mathsf{list}(\mathtt{w},\mathtt{null}) \wedge \mathtt{x}{\neq}\mathtt{null} \wedge \mathtt{w}{\neq}\mathtt{null}) \qquad (\mathtt{I}')$$

Next we perform a weakening over the above formula ($\mathtt{I}'$). Generally it is unsound to weaken a computed precondition in verification; however, we can abstract ($\mathtt{I}'$) to a simpler one for two reasons. First, we inferred this precondition from the calling context (rather than the procedure body). Second, for a possible later verification against the procedure body, this weakening does not allow any incorrect implementation to pass. Therefore we get the weakened result as follows:

$$\mathsf{Pre}_{\mathtt{unkProc}} := \mathsf{list}(\mathtt{x},\mathtt{null}) \wedge \mathtt{x}{\neq}\mathtt{null} \qquad (\mathtt{I})$$

However the precondition ($\mathtt{I}'$) requires a list with at least two nodes while the precondition (`I`) requires a list with at least one node. Thus the precondition (`I`) may reject some possible correct implementations of the unknown procedure `unkProc`. Therefore, to maintain necessary precision, it is not always desired to weaken the precondition inferred from the calling context.

The unknown procedure call is at line `4`. In general the heap state after a procedure call consists of two disjoint parts: the frame (the heap part that is not changed by the procedure call) and the procedure postcondition (the heap part that is changed by the procedure call). In our case the frame is known (the first part of state Δ_4 at line `4a`), but the procedure postcondition is unknown. What we know is that the unknown procedure `unkProc` may change the heap reachable from the program variables `x` and `y`. Therefore initially we set the unknown procedure postcondition to $\mathsf{emp} \wedge \mathtt{y}{=}\mathtt{y}' \wedge \mathtt{x}{=}\mathtt{x}'$ (the second part of state Δ_4 at line `4a`). We use the logical variables $\mathtt{x}'$ and $\mathtt{y}'$ to denote respectively the values of the program variables `x` and `y` when the unknown procedure call returns. Our goal is to discover the postcondition, namely the heap that can be reached from $\mathtt{x}'$ and $\mathtt{y}'$ by analysing the usage of $\mathtt{x}'$ and $\mathtt{y}'$ in the remaining code after the unknown procedure call. Therefore after the unknown procedure call our analysis keeps track of a pair $\langle \Delta_{\mathtt{i}};\ \mathsf{M} \rangle$ where $\Delta_{\mathtt{i}}$ is the current heap state, while M denotes the postcondition discovered so far for the unknown procedure. The notations $\mathsf{M}_{\mathtt{i}}$ are also used to denote parts of the discovered postcondition. Thus the first discovered information

M is that just after the unknown call the value of the program variable `y` and the unknown procedure result are equal (line `4b`). M also contains the information that the logical variables $\mathtt{x}'$ and $\mathtt{y}'$ denote the values of the program variables `x` and `y` respectively, just after the unknown procedure call.

At line `5` the procedure `findLast` is called recursively. Since the current heap state Δ_4 does not entail the precondition of the procedure `findLast` (line `4c`) there is an error. However this error may not be a program error, it may be caused by the incomplete heap denoted by $\mathtt{x}'$ and/or $\mathtt{y}'$. Therefore our analysis performs an abductive reasoning (line `4d`) to infer the missing part M_1 of Δ_4 such that $\mathsf{M}_1 * \Delta_4$ entails the precondition of the procedure `findLast`. At line `4e` our analysis checks whether the inferred M_1 is part of the unknown procedure postcondition, namely whether M_1 refers either to $\mathtt{x}'$ and/or $\mathtt{y}'$ or to aliases of $\mathtt{x}'$ and/or $\mathtt{y}'$ or to a heap reachable from $\mathtt{x}'$ and/or $\mathtt{y}'$. This check succeeds and therefore M_1 is added to the current discovered postcondition M.

The heap state Δ_4 combined with the inferred M_1 entails the precondition of the procedure `findLast` and also generates a residual frame heap $\mathtt{R}_1$ (line `4f`). The heap state Δ_5 after the recursive call consists of the procedure `findLast` postcondition and the frame heap $\mathtt{R}_1$ (line `5a`).

At the end of the procedure body the current heap state Δ_6 (computed at line `6a`) must entail the postcondition of the procedure `findLast`. This entailment fails (line `6b`). We perform another abductive reasoning (line `6c`) to infer the missing M_2 as follows:

$$(\exists \mathsf{z}, \mathsf{y}, \mathsf{y}', \mathsf{x}' \cdot \mathsf{list}(\mathtt{y}, \mathtt{z}) * \mathtt{z} {\mapsto} \mathtt{null} \wedge \mathtt{res}{=}\mathtt{z} \wedge \mathtt{y}{=}\mathtt{y}' \wedge \mathtt{x}{=}\mathtt{x}') * [\mathsf{M}_2] \triangleright \mathsf{list}(\mathtt{x}, \mathtt{res}) * \mathtt{res} {\mapsto} \mathtt{null}$$

where Δ_6 and $\mathsf{Post}_{\mathtt{findLast}}$ are replaced by their formulae. The above judgment can be further simplified as follows:

$$\mathsf{list}(\mathtt{y}, \mathtt{res}) * \mathtt{res} {\mapsto} \mathtt{null} * [\mathsf{M}_2] \triangleright \mathsf{list}(\mathtt{x}, \mathtt{res}) * \mathtt{res} {\mapsto} \mathtt{null}$$

and then to the following:

$$\mathsf{list}(\mathtt{y}, \mathtt{res}) * [\mathsf{M}_2] \triangleright \mathsf{list}(\mathtt{x}, \mathtt{res})$$

Using the abductive inference rules from Calcagno et al. [4] we can obtain the following result $\mathsf{M}_2 := \mathsf{list}(\mathtt{x}, \mathtt{res})$ that passes the check from line `6d`. Using formula from line `8` we obtain the following postcondition for the unknown procedure (where `u` stands for α-renaming of `res`):

$$\mathsf{Post}_{\mathsf{unkProc}} := \exists \mathsf{u} \cdot \mathsf{list}(\mathtt{x}, \mathtt{u}) * \mathsf{list}(\mathtt{res}_{\mathsf{unkProc}}, \mathtt{null}) \wedge \mathtt{res}_{\mathsf{unkProc}} {\neq} \mathtt{null} \quad (\mathtt{II}')$$

The above postcondition $(\mathtt{II}')$ and the precondition $(\mathtt{I})$ might form a candidate specification for the unknown procedure `unkProc`. However, the postcondition $(\mathtt{II}')$ is not strong enough to establish the postcondition of the outer procedure `findLast`. This problem is due to incompleteness of the abductive inference system from Calcagno et al. [4].

It is very difficult to impose the completeness for an abductive inference system. Therefore we define the following weaker property that has to be satisfied by the abductive system. In general, given an abductive judgment:

$$\Delta * [\mathsf{M}] \triangleright \mathsf{H}$$

there are many possible solutions for M. In order to be able to compute the most precise postcondition we are interested in the weakest solution M (namely minimal solution w.r.t. the order $\preceq$ defined in Calcagno et al. [4]) satisfying the following entailment

$$\Delta * [\mathsf{M}] \vdash \mathsf{H} * \mathsf{R} \quad (\mathtt{III})$$

such that R does not refer to the arguments and result of the unknown call, or to the aliases of the arguments and result of the unknown call, or to the heap reachable from the arguments and result of the unknown call. Using the notations of our example with an unknown call having the argument $\mathtt{x}'$ and the result $\mathtt{y}'$ the property of R can be expressed as follows:

$$\mathtt{fv}(\mathsf{R}) \not\subseteq \mathtt{ReachVar}(\Delta * \mathsf{M}, \{\mathtt{x}', \mathtt{y}'\}) \quad (\mathtt{IV})$$

The computation of a solution that satisfies the properties (III) and (IV) depends on the abductive inference rules which implement the abductive judgment. However the existence of this solution does not imply the completeness of the abductive inference system.

In our case for the following abductive judgment:

$$\mathsf{list}(\mathtt{y}, \mathtt{res}) * [\mathsf{M}_2] \triangleright \mathsf{list}(\mathtt{x}, \mathtt{res})$$

there are three possible solutions $\mathsf{M}_2 := \mathtt{x}{=}\mathtt{y}$, $\mathsf{M}_2 := \mathtt{x}{\mapsto}\mathtt{y}$, and $\mathsf{M}_2 := \mathsf{list}(\mathtt{x}, \mathtt{y})$ which satisfy the above properties (III) and (IV). The best solution is $\mathsf{M}_2 := \mathsf{list}(\mathtt{x}, \mathtt{y})$ which is weaker than the other two solutions. However the current abductive inference rules from Calcagno et al. [4] are not able to infer these solutions. Therefore those rules have to be refined such that they can also support matching on the right sides of two separation predicates when their left sides are equal. Using the best solution $\mathsf{M}_2 := \mathsf{list}(\mathtt{x}, \mathtt{y})$ we can obtain a more precise postcondition for the unknown procedure as follows:

$$\mathsf{Post}_{\mathtt{unkProc}} := \mathsf{list}(\mathtt{x}, \mathtt{res}_{\mathtt{unkProc}}) * \mathsf{list}(\mathtt{res}_{\mathtt{unkProc}}, \mathtt{null}) \wedge \mathtt{res}_{\mathtt{unkProc}}{\neq}\mathtt{null} \quad (\mathtt{II})$$

where the best solution $\mathsf{M}_2 := \mathsf{list}(\mathtt{x}, \mathtt{y})$ also satisfies the check from line 6d.

Note that the accumulation of the inferred $\mathsf{M}_\mathtt{i}$ into M does not generate any inconsistency as long as the effect of each $\mathsf{M}_\mathtt{i}$ is reflected in the next state $\Delta_{\mathtt{i+1}}$ through the entailment residual frame (e.g. line 4f).

After each abduction our analysis checks whether the inferred formula can be part of the unknown procedure postcondition (e.g. checks from lines 4e and 6d). A failed check denotes a program error. In Figure 2 we consider the same example from Figure 1 but after line 4 we added a new instruction which dereferences a local variable. The new instruction is a program error. Our analysis discovers this error at line 4e when the check on the new abducted formula M_1 fails.

3 Multiple Invocations of an Unknown Procedure

In this section we illustrate how our analysis computes a precondition and a postcondition for an unknown procedure, which is invoked more than once. There are two cases: (1) the two unknown calls are in sequence and are executed one after another; (2) the two unknown calls are on different program execution paths. For the second case (e.g. two unknown calls are in two different branches of a conditional statement) it is not difficult to use the approach described in the previous section. First we infer the preconditions for both unknown calls by localizing the program state immediately before the calls. Then we do abductions till the end to gain their postconditions, respectively. Since the two specifications are inferred from different calling contexts, they are combined by conjunction [6]. In a later verification of the procedure's body they must be verified to ensure the implementation satisfies requirements from each calling context. Therefore in this section we concentrate on the first case where some source code invokes an unknown procedure many times in sequence.

```
// Given Specification:
       // $\mathsf{Pre}_{\mathtt{findLast}} := \mathsf{list}(\mathtt{x},\mathtt{null}) \wedge \mathtt{x}{\neq}\mathtt{null}$
       // $\mathsf{Post}_{\mathtt{findLast}} := \mathsf{list}(\mathtt{x},\mathtt{res}) * \mathtt{res}{\mapsto}\mathtt{null}$
node findLast(node x) {
0    node w, y, z, a, b;
0a    // $\Delta_0 := \mathsf{Pre}_{\mathtt{findLast}}$
0b    // $\Delta_0 \vdash \mathtt{x}{\neq}\mathtt{null}$
1    w := x.next;
1a    // $\Delta_1 := \mathtt{x}{\mapsto}\mathtt{w} * \mathsf{list}(\mathtt{w},\mathtt{null}) \wedge \mathtt{x}{\neq}\mathtt{null}$
2    if (w == null) return x;
2a    // $\Delta_2 := \mathtt{w}{=}\mathtt{null} \wedge \mathtt{res}{=}\mathtt{x} \wedge \mathtt{x}{\mapsto}\mathtt{null} \wedge \mathtt{x}{\neq}\mathtt{null}$
2b    // $\exists \mathtt{w},\mathtt{y},\mathtt{z} \cdot \Delta_2 \vdash \mathsf{Post}_{\mathtt{findLast}}$
3    else {
3a    // $\Delta_3 := \mathtt{x}{\mapsto}\mathtt{w} * \mathsf{list}(\mathtt{w},\mathtt{null}) \wedge \mathtt{x}{\neq}\mathtt{null} \wedge \mathtt{w}{\neq}\mathtt{null}$
3b    // $\mathsf{Local}(\Delta_3,\{\mathtt{x}\}) := \mathtt{x}{\mapsto}\mathtt{w} * \mathsf{list}(\mathtt{w},\mathtt{null}) \wedge \mathtt{x}{\neq}\mathtt{null} \wedge \mathtt{w}{\neq}\mathtt{null}$
3c    // $\mathsf{Frame}(\Delta_3,\{\mathtt{x}\}) := \mathtt{x}{\neq}\mathtt{null} \wedge \mathtt{w}{\neq}\mathtt{null}$
3d    // $\mathsf{Pre}_{\mathtt{unkProc}} := \mathsf{Local}(\Delta_3,\{\mathtt{x}\})$
4    y := unkProc(x);
4a    // $\Delta_4 := (\exists \mathtt{fv}(\mathsf{Pre}_{\mathtt{unkProc}}) \cdot \mathsf{Frame}(\Delta_3,\{\mathtt{x}\})) * (\mathsf{emp} \wedge \mathsf{y}{=}\mathsf{y}' \wedge \mathsf{x}{=}\mathsf{x}')$
4b    // $\mathsf{M} := (\mathtt{res}_{\mathtt{unkProc}}{=}\mathsf{y} \wedge \mathsf{y}{=}\mathsf{y}' \wedge \mathsf{x}{=}\mathsf{x}')$
4c    // $\Delta_4 \nvdash \mathtt{b}{\neq}\mathtt{null}$
4d    // $\Delta_4 * [\mathsf{M}_1] \triangleright \mathtt{b}{\neq}\mathtt{null}$
4e    // $\mathsf{M}_1 := \mathtt{b}{\neq}\mathtt{null}$
4f    // $\mathtt{fv}(\mathsf{M}_1) \not\subseteq \mathtt{ReachVar}(\Delta_4 * \mathsf{M}_1,\{\mathsf{x}',\mathsf{y}'\})$      Error!!!
5    a := b.next;
6    z := findLast(y);
7    return z;
8    } }
```

Figure 2: An error in the procedure `findLast`.

We start from a simplest case. Figure 3 describes the general scenario consisting of a known procedure, known which calls twice the same unknown procedure unknown. Note that the code blocks b_1, b_2 and b_3 are composed by known program instructions. The specification of the known procedure known is given as the precondition $\mathsf{Pre}_{\mathtt{known}}$ and the postcondition $\mathsf{Post}_{\mathtt{known}}$. We denote by P_1, Q_1, and R_1 (P_2, Q_2, and R_2) the precondition, the postcondition and the frame, respectively for the first (the second) unknown procedure unknown call. Our goal is to infer P_1, Q_1, P_2 and Q_2 from procedure known's codes and specification.

```
$\mathsf{Pre}_{\mathtt{known}}$
procedure known()
  $b_1$;   // Code block one
  // $P_1 * R_1$
  unknown($\vec{x_1}$);
  // $Q_1 * R_1$
  $b_2$;   // Code block two
  // $P_2 * R_2$
  unknown($\vec{x_2}$);
  // $Q_2 * R_2$
  $b_3$;   // Code block three
end
$\mathsf{Post}_{\mathtt{known}}$
```

Figure 3: Two sequential unknown procedure calls.

In order to achieve our goal we assume there exists a most general specification $\{\mathsf{Pre}_{\mathtt{unknown}}\}$ unknown $\{\mathsf{Post}_{\mathtt{unknown}}\}$, such that P_1 implies $\mathsf{Pre}_{\mathtt{unknown}}$, P_2 implies $\mathsf{Pre}_{\mathtt{unknown}}$, $\mathsf{Post}_{\mathtt{unknown}}$ implies Q_1 and $\mathsf{Post}_{\mathtt{unknown}}$ implies Q_2. This assumption may not be useful for all possible unknown procedures unknown, since in the worst case the most general specification corresponds to the trivial specification {true} unknown {false}. However our goal is to avoid the trivial specification and to get more precise result.

```
Algorithm InferTwoSpec
  Do forward analysis from $\mathsf{Pre}_{\mathtt{known}}$ over $b_1$ to get $P_1 * R_1$;
  Distinguish $P_1$ as the local state of $\vec{x_1}$;
  Do forward analysis with abduction from $\langle \mathsf{emp};\ \mathsf{emp} \rangle$ to
    $\langle \mathsf{Post}_{\mathtt{known}};\ \mathsf{M}_1 \rangle$ over $b_3$ to get $\mathsf{M}_1$ as $Q_2$;
  Assume $Q_1 := \sigma\ Q_2$, and $P_2 := \sigma^{-1}\ P_1$;
  Do forward analysis with abduction from $\langle Q_1 * R_1;\ \mathsf{emp} \rangle$ to
    $\langle P_2 * R_2;\ \mathsf{M}_2 \rangle$ over $b_2$ to get $\mathsf{M}_2$;
  return $\{P_1\}$ unknown $\{Q_1 * \mathsf{M}_2\}$;
```

Figure 4: Analysis algorithm for two sequential unknown calls.

The general idea of our abduction-based approach is informally described in Figure 4. First our approach infers the precondition P_1 of the first unknown call and the postcondition Q_2 of the second unknown call in the same way as in the previous section. Afterwards we regard them as a raw specification for the unknown procedure (with necessary substitutions), and attempt to refine them with the codes in between the two unknown calls. Therefore we have $Q_1 := \sigma\ Q_2$ and $P_2 := \sigma^{-1}\ P_1$, and try to verify b_2 with the specification $\{Q_1 * R_1\}\ b_2\ \{P_2 * R_2\}$.

Here is an example for our general approach. We consider the procedure `appendThree` whose specification and implementation are shown in Figure 5. The procedure appends three singly linked lists into one, by updating one's tail to point to another's head. It has two invocations of the same unknown procedure `unkProc`, one followed by the other.

```
// Given Specification:
        // $\mathsf{Pre}_{\texttt{appendThree}} := \mathsf{list}(\mathtt{x}, \mathtt{null}) * \mathsf{list}(\mathtt{y}, \mathtt{null}) * \mathsf{list}(\mathtt{z}, \mathtt{null})$
        // $\mathsf{Post}_{\texttt{appendThree}} := \mathsf{list}(\mathtt{x}, \mathtt{y}) * \mathsf{list}(\mathtt{y}, \mathtt{z}) * \mathsf{list}(\mathtt{z}, \mathtt{null})$
// Goal:
        // To infer unkProc's specification
void appendThree(node x, node y, node z) {
0a   // Step 1:
0b   // $\Delta_0 := \mathsf{list}(\mathtt{x}, \mathtt{null}) * \mathsf{list}(\mathtt{y}, \mathtt{null}) * \mathsf{list}(\mathtt{z}, \mathtt{null})$
0c   // $\mathsf{Local}(\Delta_0, \{\mathtt{x}, \mathtt{y}\}) := \mathsf{list}(\mathtt{x}, \mathtt{null}) * \mathsf{list}(\mathtt{y}, \mathtt{null})$
0d   // $\mathsf{Frame}(\Delta_0, \{\mathtt{x}, \mathtt{y}\}) := \mathsf{list}(\mathtt{z}, \mathtt{null})$
0e   // $\mathsf{Pre}_{\texttt{unkProc}} := \mathsf{Local}(\Delta_0, \{\mathtt{x}, \mathtt{y}\})$
1    unkProc(x, y);
1a   // Step 3:
1b   // Current state $\Delta_3 := \exists \mathtt{u} \cdot \mathsf{list}(\mathtt{x}, \mathtt{u}) * \mathsf{list}(\mathtt{u}, \mathtt{y}) * \mathsf{list}(\mathtt{y}, \mathtt{null}) * \mathsf{list}(\mathtt{z}, \mathtt{null})$
1c   // Check entailment $\Delta_3 \vdash [\mathtt{z}/\mathtt{y}]\ \mathsf{Pre}_{\texttt{unkProc}}$
1d   // Entailment succeeds, and unkProc's specification is approved
2    unkProc(x, z);
2a   // Step 2:
2b   // $\Delta_1 := \mathsf{emp} \wedge \mathtt{x}{=}\mathtt{x}' \wedge \mathtt{z}{=}\mathtt{z}' \quad \mathsf{M} := \mathsf{emp} \wedge \mathtt{x}{=}\mathtt{x}' \wedge \mathtt{z}{=}\mathtt{z}'$
2c   // Do abduction to get $\Delta_1 * [\ \mathsf{M}_1\ ] \triangleright \mathsf{Post}_{\texttt{appendThree}}$
2d   // $\mathsf{M}_1 := \mathsf{list}(\mathtt{x}, \mathtt{y}) * \mathsf{list}(\mathtt{y}, \mathtt{z}) * \mathsf{list}(\mathtt{z}, \mathtt{null})$
2e   // $\Delta_2 := \Delta_1 * \mathsf{M}_1 \quad \mathsf{M} := \mathsf{M} * \mathsf{M}_1$
2f   // $\mathsf{Post}_{\texttt{unkProc}} := \exists \mathtt{x}, \mathtt{z}, \mathtt{u} \cdot [\mathtt{u}/\mathtt{y}]\ \mathsf{M}$
2g   // $\mathsf{Post}_{\texttt{unkProc}} := \exists \mathtt{u} \cdot \mathsf{list}(\mathtt{x}', \mathtt{u}) * \mathsf{list}(\mathtt{u}, \mathtt{z}') * \mathsf{list}(\mathtt{z}', \mathtt{null})$
3    }
```

Figure 5: Procedure `appendThree` calling an unknown procedure twice.

To focus on our algorithm itself, this example has no other codes except for the unknown calls, without losing generality. From the beginning we are provided with $\mathsf{Pre}_{\texttt{appendThree}}$ and $\mathsf{Post}_{\texttt{appendThree}}$, and our aim is to find the unknown procedure `unkProc(x,y)`'s specification.

As shown in line `0a`, the first step is to infer the initial precondition of the unknown procedure call, P_1. This is accomplished by localizing the state before the first unknown call against its parameters `x` and `y`. We get $P_1 := \mathsf{list}(\mathtt{x}, \mathtt{null}) * \mathsf{list}(\mathtt{y}, \mathtt{null})$, with the frame part $\mathsf{list}(\mathtt{z}, \mathtt{null})$ left unchanged and to be carried over to the second unknown call.

The second step beginning from line `2a` is analogous to the inference of the unknown procedure call's postcondition in the last section. It starts with $\mathsf{emp} \wedge \mathtt{x} = \mathtt{x}' \wedge \mathtt{z} = \mathtt{z}'$ to assume the unknown call's effect as emp with logical variables to record the parameters' current values, and utilizes abduction when necessary to get the postcondition Q_2 for the unknown procedure call. In this case it directly gains the abduction result from `appendThree`'s postcondition as $\mathsf{list}(\mathtt{x}, \mathtt{y}) * \mathsf{list}(\mathtt{y}, \mathtt{z}) * \mathsf{list}(\mathtt{z}, \mathtt{null})$. As postprocessing, it then existentially quantifies `y` as it does not occur in the second unknown call's parameter list, and changes its name to `u`. So the postcondition is $\exists \mathtt{u} \cdot \mathsf{list}(\mathtt{x}, \mathtt{u}) * \mathsf{list}(\mathtt{u}, \mathtt{z}) * \mathsf{list}(\mathtt{z}, \mathtt{null})$.

The last step from line `1a` performs a final check for $\{Q_1 * R_1\}\ b_2\ \{P_2 * R_2\}$. Here from context we know $R_1 := \mathsf{list}(\mathtt{z}, \mathtt{null})$ and $R_2 := \mathsf{emp}$. According to our algorithm, Q_1 is assumed the same as Q_2 and P_2 the same as P_1, under some substitutions. Hence we have $Q_1 * R_1 := \exists \mathtt{u} \cdot \mathsf{list}(\mathtt{x}, \mathtt{u}) * \mathsf{list}(\mathtt{u}, \mathtt{y}) * \mathsf{list}(\mathtt{y}, \mathtt{null}) * \mathsf{list}(\mathtt{z}, \mathtt{null})$, and $P_2 * R_2 := \mathsf{list}(\mathtt{x}, \mathtt{null}) * \mathsf{list}(\mathtt{z}, \mathtt{null})$. Then the entailment $Q_1 * R_1 \vdash P_2 * R_2$ is checked to ensure the correctness of the second invocation. As it succeeds, the specification for the unknown procedure $\mathtt{unkProc}(\mathsf{a}_1, \mathsf{a}_2)$ is as follows:

$$\begin{aligned}
\mathsf{Pre}_{\mathtt{unkProc}} &:= \mathsf{list}(\mathsf{a}_1, \mathtt{null}) * \mathsf{list}(\mathsf{a}_2, \mathtt{null}) \\
\mathsf{Post}_{\mathtt{unkProc}} &:= \exists \mathtt{u} \cdot \mathsf{list}(\mathsf{a}_1, \mathtt{u}) * \mathsf{list}(\mathtt{u}, \mathsf{a}_2) * \mathsf{list}(\mathsf{a}_2, \mathtt{null})
\end{aligned}$$

However, it is still possible that $Q_1 * R_1$ is not sufficiently strong in the verification for b_2 to establish $P_2 * R_2$, especially when the specification for the known procedure is imprecise (either the precondition is excessively strong or the postcondition is too weak). For this sake we will use abduction in the verification to collect the heap states (M_2) that Q_1 lacks, and strengthen $\mathsf{Post}_{\mathtt{unknown}}$ to be $Q_1 * \mathsf{M}_2$.

We have yet another example to illustrate this postcondition strengthening. Figure 6 describes a procedure `towardsLast`. According to its specification, the procedure takes the head of a linked list as input, and returns any node in the list. It also calls twice the same unknown procedure in sequence, and our aim is to analyze the procedure's pre- and postconditions.

To start with, we still take `towardsLast`'s precondition to do forward analysis to get the precondition for the first unknown call. As shown in lines `0b` and `0e`, the program state immediately before that call is $\mathsf{list}(\mathtt{y}, \mathtt{null}) \wedge \mathtt{y}{\neq}\mathtt{null} \wedge \mathtt{y}{=}\mathtt{x}$, and the localized precondition for the call is $\mathsf{list}(\mathtt{y}, \mathtt{null}) \wedge \mathtt{y}{\neq}\mathtt{null}$.

In the second step, we use the same approach (forward analysis with abduction) to find out the postcondition of the second unknown procedure call, expressed from lines `3a` to `3g`. After that call we have no knowledge about the heap, and so the result of abduction will be the whole postcondition of `towardsLast`, $\mathsf{list}(\mathtt{x}, \mathtt{z}) * \mathsf{list}(\mathtt{z}, \mathtt{null}) \wedge \mathtt{x}{\neq}\mathtt{null}$. The current discovered postcondition is computed at line `3g`.

Last we also try to verify the codes between the two unknown calls, from the postcondition of the first call (plus its frame part) to the precondition of the second. In this case there are no codes and an entailment checking is performed to guarantee that $[\mathtt{y}/\mathtt{res}_{\mathtt{unkProc}}]\ \mathsf{Post}_{\mathtt{unkProc}} \wedge \mathtt{y}'{=}\mathtt{x} \vdash \mathsf{Pre}_{\mathtt{unkProc}}$, where the logical variable $\mathtt{y}'$ stands for the previous value of the program variable `y`. This check fails, because the first postcondition is not adequately strong. Therefore an abduction is necessary to enhance it, as accomplished in line `1d`. There we achieve an extra requirement for the postcondition such that the final specification for the unknown procedure $\mathtt{unkProc}(\mathsf{a})$ is as follows:

```
// Given Specification:
       // $\mathsf{Pre}_{\texttt{towardsLast}} := \mathsf{list}(\texttt{x},\texttt{null}) \wedge \texttt{x} \neq \texttt{null}$
       // $\mathsf{Post}_{\texttt{towardsLast}} := \mathsf{list}(\texttt{x},\texttt{res}) * \mathsf{list}(\texttt{res},\texttt{null}) \wedge \texttt{x} \neq \texttt{null}$
node towardsLast(node x) {
0    node y := x, z;
0a   //  Step 1:
0b   //  $\Delta_0 := \mathsf{list}(\texttt{y},\texttt{null}) \wedge \texttt{y}{\neq}\texttt{null} \wedge \texttt{y}{=}\texttt{x}$
0c   //  $\mathsf{Local}(\Delta_0,\{\texttt{y}\}) := \mathsf{list}(\texttt{y},\texttt{null}) \wedge \texttt{y}{\neq}\texttt{null}$
0d   //  $\mathsf{Frame}(\Delta_0,\{\texttt{y}\}) := \mathsf{emp} \wedge \texttt{y}{=}\texttt{x}$
0e   //  $\mathsf{Pre}_{\texttt{unkProc}} := \mathsf{Local}(\Delta_0,\{\texttt{y}\})$
1    y := unkProc(y);
1a   //  Step 3:
1b   //  Initialize $\mathsf{M} := \mathsf{emp} \wedge \texttt{res}_{\texttt{unkProc}}{=}\texttt{y}$
1c   //  Current state $\Delta_3 := \exists \texttt{u} \cdot \mathsf{list}(\texttt{u},\texttt{y}) * \mathsf{list}(\texttt{y},\texttt{null}) \wedge \texttt{u}{\neq}\texttt{null} \wedge \texttt{y}'{=}\texttt{x}$
1d   //  Do abduction to get $\Delta_3 * [\ \mathsf{M}_3\ ] \rhd \mathsf{Pre}_{\texttt{unkProc}}$
1e   //  $\mathsf{M}_3 := \texttt{y}{\neq}\texttt{null} \qquad \mathsf{M} := \mathsf{M} * \mathsf{M}_3$
1f   //  So the unknown's postcondition is strengthened to be
1g   //  $\mathsf{Post}_{\texttt{unkProc}} := \exists \texttt{x},\texttt{y},\texttt{z} \cdot \mathsf{Post}_{\texttt{unkProc}} * \mathsf{M}$
1h   //  $\mathsf{Post}_{\texttt{unkProc}} := \exists \texttt{u} \cdot \mathsf{list}(\texttt{u},\texttt{res}_{\texttt{unkProc}}) * \mathsf{list}(\texttt{res}_{\texttt{unkProc}},\texttt{null}) \wedge$
     //                  $\texttt{u}{\neq}\texttt{null} \wedge \texttt{res}_{\texttt{unkProc}}{\neq}\texttt{null}$
2    z := unkProc(y);
3    return z;
3a   //  Step 2:
3b   //  $\Delta_1 := \mathsf{emp}{\wedge}\texttt{y}{=}\texttt{y}''{\wedge}\texttt{z}{=}\texttt{z}'{\wedge}\texttt{z}{=}\texttt{res} \qquad \mathsf{M} := \mathsf{emp}{\wedge}\texttt{y}{=}\texttt{y}''{\wedge}\texttt{z}{=}\texttt{z}'{\wedge}\texttt{z}'{=}\texttt{res}_{\texttt{unkProc}}$
3c   //  Do abduction to get $\Delta_1 * [\ \mathsf{M}_1\ ] \rhd \mathsf{Post}_{\texttt{towardsLast}}$
3d   //  $\mathsf{M}_1 := \mathsf{list}(\texttt{x},\texttt{z}) * \mathsf{list}(\texttt{z},\texttt{null}) \wedge \texttt{x}{\neq}\texttt{null}$
3e   //  $\Delta_2 := \Delta_1 * \mathsf{M}_1 \qquad \mathsf{M} := \mathsf{M} * \mathsf{M}_1$
3f   //  $\mathsf{Post}_{\texttt{unkProc}} := \exists \texttt{x},\texttt{y},\texttt{z} \cdot \mathsf{M}$
3g   //  $\mathsf{Post}_{\texttt{unkProc}} := \exists \texttt{x} \cdot \mathsf{list}(\texttt{x},\texttt{res}_{\texttt{unkProc}}) * \mathsf{list}(\texttt{res}_{\texttt{unkProc}},\texttt{null}) \wedge \texttt{x}{\neq}\texttt{null}$
4   }
```

Figure 6: Procedure `towardsLast` calling an unknown procedure twice.

$$\begin{aligned}\mathsf{Pre}_{\texttt{unkProc}} &:= \mathsf{list}(\mathsf{a},\texttt{null}) \wedge \mathsf{a}{\neq}\texttt{null}\\ \mathsf{Post}_{\texttt{unkProc}} &:= \exists \mathsf{u}\cdot \mathsf{list}(\mathsf{u},\texttt{res}_{\texttt{unkProc}}) * \mathsf{list}(\texttt{res}_{\texttt{unkProc}},\texttt{null}) \wedge \mathsf{u}{\neq}\texttt{null} \wedge \texttt{res}_{\texttt{unkProc}}{\neq}\texttt{null}\end{aligned}$$

In this case our inferred postcondition is strengthened to meet the request of the second unknown call's precondition. It is always safe to do this for two reasons. First, according to previous discussions, it is unsound to weaken a precondition or strengthen a postcondition in an analysis for some known codes' specifications [4]. However, since our aim is to find possible specifications of unknown procedure calls for a possibly later verification, when we weaken the precondition or strengthen the postcondition that we found, the range of "correct" programs is narrowed, which is always sound (safe), although at the cost of possible precision lost. Second, the strengthened postcondition is always capable to entail the known procedure's postcondition. This maintains the consistency of steps 2 and 3 in our analysis algorithm and stands for the other aspect of our approach's soundness.

```
Pre_known
procedure known()
  $b_1$        // Code block one;
  //  $P_1 * R_1$
  unknown($\vec{x_1}$);
  //  $Q_1 * R_1$
  $b_2$        // Code block two;
  //  $P_2 * R_2$
  unknown($\vec{x_2}$);
  //  $Q_2 * R_2$
  ...
  $b_n$        // Code block $n$;
  //  $P_n * R_n$
  unknown($\vec{x_n}$);
  //  $Q_n * R_n$
  $b_{n+1}$    // Code block $n$+1;
end
Post_known
```

Figure 7: n unknown procedure calls in sequence.

At last it is worth noting that, this approach can be extended without difficulty to cater for more invocations of the same unknown procedure. Suppose we have n calls as shown in Figure 7. In this case, first we still infer the precondition for the first unknown call and the postcondition for the last, to gain $\mathsf{Pre}_{\texttt{unknown}} := P_1$ and $\mathsf{Post}_{\texttt{unknown}} := Q_n$, respectively. After that we use abduction to verify $b_2, b_3, \ldots, b_n$ with the post- and preconditions. For b_i we take $\sigma_1\ \mathsf{Post}_{\texttt{unknown}} * R_{i-1}$ as its precondition and $\sigma_2\ \mathsf{Pre}_{\texttt{unknown}} * R_i$ as postcondition, to gather abductions that strengthen $\mathsf{Post}_{\texttt{unknown}}$.

4 Conclusion

To verify the pointer safety for imperative programs with unknown procedure calls (or code pointers), we propose a novel approach to inferring the possible specifications for the unknown procedure from the calling contexts. We employ a forward shape analysis with separation logic and an abductive inference mechanism to synthesize both pre- and postconditions of the unknown procedure. We have defined the property that has to be satisfied by an abductive system in order to be useful for our approach. We have also discussed the solution for multiple calls of the same unknown procedure. The inferred specifications are partial specifications of the unknown procedure therefore they are subject to a later verification when the codes or the complete specifications of the unknown procedures become known.

Currently we are working on the formalization of our framework. We have also started to implement a prototype for an experimental validation of our approach.

Our approach is a general framework such that changing the abstract domain permits our framework to infer specifications within that new abstract domain. Two possible future works are to extend this method to broader shape domains and/or to other shape-related domains. The first adds more shape predicates to the domain to increase expressiveness, like doubly linked lists and trees ([13]). The second extension, for example the size domain ([5]), allows us to reason about properties such as length of lists, sortedness, and so forth. With the extended domains, the abstract semantics and analysis algorithm will remain conceivably the same, but the abduction will be redefined to discover the anti-frames for the newly introduced features. Since one possible application scenario for our approach consists of programs where the unknown procedures are only known at runtime, it might be interesting to combine our method with runtime verification of separation logic specifications [14].

Acknowledgement

This work was supported by the Engineering and Physical Sciences Research Council [grant number EP/E021948/1]. We would like to thank Hongseok Yang for helpful discussion on abduction and footprint analysis.

References

[1] Glenn Ammons, Rastislav Bodik, and James R. Larus. Mining specifications. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 4–16, New York, NY, USA, 2002. ACM.

[2] Michael G. Burke, Paul R. Carini, Jong-Deok Choi, and Michael Hind. Flow-insensitive interprocedural alias analysis in the presence of pointers. In *LCPC '94: Proceedings of the 7th International Workshop on Languages and Compilers for Parallel Computing*, pages 234–250, London, UK, 1995. Springer-Verlag.

[3] Cristiano Calcagno, Dino Distefano, Peter O'Hearn, and Hongseok Yang. Footprint analysis: A shape analysis that discovers preconditions. In *Static Analysis Symposium 2007 (SAS'07)*, Denmark, 2007.

[4] Cristiano Calcagno, Dino Distefano, Peter O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, Savannah, Georgia, USA, January 2009. ACM Press.

[5] Wei-Ngan Chin, Cristina David, Huu Hai Nguyen, and Shengchao Qin. Automated verification of shape, size and bag properties. In *Proc. 12th IEEE International Conference on Engineering Complex Computer Systems*, pages 307–320, 2007.

[6] Wei-Ngan Chin, Cristina David, Huu Hai Nguyen, and Shengchao Qin. Multiple pre/post specifications for heap-manipulating methods. In *Proc. 10th IEEE High Assurance Systems Engineering Symposium (HASE'07)*, Dallas, Texas, November 2007. IEEE CS Press.

[7] Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. A local shape analysis based on separation logic. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 3920 of *LNCS*. Springer, 2006.

[8] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 242–256, New York, NY, USA, 1994. ACM.

[9] R. Giacobazzi. Abductive analysis of modular logic programs. In M. Bruynooghe, editor, *Proc. 1994 Int'l Symposium on Logic Programming (ILPS'94)*, pages 377–391. The MIT Press, 1994.

[10] Denis Gopan and Thomas Reps. Low-level library analysis and summarization. In *Proceedings of International Conference on Computer Aided Verification 2007*, 2007.

[11] Alexey Gotsman, Josh Berdine, and Byron Cook. Interprocedural shape analysis with separated heap abstractions. In *Static Analysis Symposium 2006 (SAS'06)*, 2006.

[12] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Precise call graphs for c programs with function pointers. *Automated Software Engineering*, 11(1):7–26, 2004.

[13] Huu Hai Nguyen, Cristina David, Shengchao Qin, and Wei-Ngan Chin. Automated verification of shape and size properties via separation logic. In *VMCAI 2007: Proceedings of the 8th International Conference on Verification, Model Checking, and Abstract Interpretation*, volume 4349 of *Lecture Notes in Computer Science*, 2007.

[14] Huu Hai Nguyen, Viktor Kuncak, and Wei-Ngan Chin. Runtime checking of separation logic. In *VMCAI 2008: Proceedings of the 9th International Conference on Verification, Model Checking, and Abstract Interpretation*, LNCS. Springer, 2008.

[15] Peter W. O'Hearn, Hongseok Yang, and John C. Reynolds. Separation and information hiding. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, Venice, Italy, January 2004. ACM Press.

[16] J. C. Reynolds. Separation logic: a logic for shared mutable data structures. In *Proceedings of the Seventeenth Annual IEEE Symposium on Logic in Computer Science*, pages 55–74, 2002.