Automated Repair of Heap-Manipulating Programs using Deductive Synthesis

Thanh-Toan Nguyen[1], Quang-Trung Ta[1], Ilya Sergey[2,1], and Wei-Ngan Chin[1]

[1] School of Computing, National University of Singapore
[2] Yale-NUS College, Singapore
{toannt, taqt, ilya, chinwn}@comp.nus.edu.sg

Abstract. We propose a novel method to automatically repairing buggy heap-manipulating programs using constraint solving and deductive synthesis. Given an input program `C` and its formal specification in the form of a Hoare triple: $\{\mathcal{P}\}$ `C` $\{\mathcal{Q}\}$, we use a separation-logic-based verifier to verify if program `C` is correct w.r.t. its specifications. If program `C` is found buggy, we then repair it in the following steps. First, we rely on the verification results to collect a list of suspicious statements of the buggy program. For each suspicious statement, we temporarily replace it with a template patch representing the desired statements. The template patch is also formally specified using a pair of unknown pre- and postcondition. Next, we use the verifier to analyze the temporarily patched program to collect constraints related to the pre- and postcondition of the template patch. Then, these constraints are solved by our constraint solving technique to discover the suitable specifications of the template patch. Subsequently, these specifications can be used to synthesize program statements of the template patch, consequently creating a candidate program. Finally, if the candidate program is validated, it is returned as the repaired program. We demonstrate the effectiveness of our approach by evaluating our implementation and a state-of-the-art approach on a benchmark of 231 buggy programs. The experimental results show that our tool successfully repairs 223 buggy programs and considerably outperforms the compared tool.

1 Introduction

The goal of automated program repair (APR) is to identify fragments of a program that contains bugs and then to discover a patch that can be applied to fix the issue. This intuitive definition of APR and its evident practical utility have aroused a lot of interest and researchers have proposed various approaches to automatically fixing buggy programs, using ideas from mutation testing [21,29,41,52,53], mining of semantic constraints [35,36,37], symbolic analysis of the reference implementations [34,46], and deep learning [15,33].

However, one of the current limitations in APR is that few studies focus on repairing heap-manipulating programs. One of them is a mutation-based approach [28] that combines formal verification and genetic programming to repair buggy programs. Concretely, this approach uses genetic programming operators, such as *mutate*, *insert*, *delete*, to generate mutated programs, and then use a verifier to validate these programs. However, these tactics are insufficient to repair non-trivial bugs in heap-manipulating programs. In another study, Verma and Roy [50] enable users to express their expected program's graphical states at different program points in a debug-and-repair process. Their synergistic method is effective in fixing various bug patterns of heap-manipulating programs, but not fully automated. In contrast, we aim to build a fully automated method that requires only program specifications in the form of pre- and postconditions. Similar to our approach in using formal specifications, the previous approaches [17,49] leverage the static verifier Infer [2,3] to repair buggy programs. Although they target large-scale projects, these tools can fix only memory-related bugs, such as *null dereferences*, *memory leaks*, and *resource leaks*. In contrast, our work aims to repair more complicated bugs related to the functional correctness of heap-manipulating programs.

In this work, we introduce a fully automated approach to repairing heap-manipulating programs using constraint solving and deductive synthesis. It is inspired by recent advances in program synthesis using formal specification [40,42]. However, our usage of program synthesis only applies to buggy statements to leave the repaired program with the least changes. The inputs of our approach are a program C, its precondition $\mathcal{P}$, its postcondition $\mathcal{Q}$. The input program C is first verified w.r.t. its specifications $\mathcal{P}$ and $\mathcal{Q}$, using a separation-logic-based verifier. If the input program does not satisfy its specifications, it is considered buggy, and we start the repair process as follows.

Firstly, our approach localizes a list of suspicious statements using invalid verification conditions in the verification step. Each suspicious statement is subsequently replaced by a template patch TP, consequently making a *template program*. The key idea is to find program statements of TP to make the template program satisfy w.r.t. the specifications $\mathcal{P}$ and $\mathcal{Q}$. Next, the verifier is used to analyze the template program w.r.t. the specifications $\mathcal{P}$ and $\mathcal{Q}$ to generate constraints related to the specifications of the template patch TP. These constraints are then solved using our constraint solving technique to discover the definition of the pre- and postcondition the template patch. In the next step, these specifications are used to synthesize program statements of the template patch TP. Then, synthesized program statements replace TP in the template program to produce a candidate program. The candidate program is validated using the verifier to finally return a repaired program.

Contributions. This paper makes the following contributions.

– We propose a novel approach to repairing buggy heap-manipulating programs. To fix a buggy program, we first use constraint solving to infer the specifications of a patch. Then, from these inferred specifications, we use deductive synthesis to synthesize program statements of the patch.
– We introduce a list of inference rules and an algorithm to formally infer the specifications of a patch. We also present synthesis rules and an algorithm to synthesize program statements of the patch using the inferred specifications.
– We implement the proposed approach in a prototype and evaluate it in a benchmark of buggy heap-manipulating programs. Our tool can repair 223 out of 231 buggy programs and outperforms a state-of-the-art repair tool.

2 Motivation

```
 1: typedef struct node {
 2:     struct node* prev;
 3:     struct node* next;} node;
 4:
 5: // dll(p, q, n) ≜ (p=null ∧ n=0 ∧ emp)
 6: //          ∨ (∃r. p↦{q, r} ∗ dll(r, p, n−1))
 7:
 8: void append(node* x, node* y)
 9: // requires dll(x, a, n) ∗ dll(y, b, m) ∧ n>0
10: // ensures dll(x, a, n+m);
11: {
12:     if (x->next == NULL) {
13:         x->next = y;
14:         if (y != NULL)
15:             y->next = x;
16:     } else append(x->next, y);
17: }
```

Fig. 1: A buggy `dll-append` program.

We illustrate our repair approach using the program `dll-append` in Fig. 1 which implements a buggy version of a function that should append two disjoint doubly-linked lists (DLLs). The function `append` takes as parameters two pointers of the structure `node`. Each `node` is an element of a doubly-linked list and stores pointers to the previous and next elements of the list. Following the definition of the structure `node` (lines 1 - 3) is a separation logic (SL) *inductive predicate* dll, which recursively describes the shape of a symbolic-heap fragment that stores a DLL of length n. That is, a DLL is either a `NULL`-pointer with the empty heap predicate `emp` and zero-length (line 5), or a non-`NULL` pointer p to the head of the structure `node` (denoted via $p{\mapsto}\{q, r\}$) such that the pointer q points to the "tail" of the DLL, with the recursively repeating dll-structure, and a length decremented by one (line 6).

The SL specifications for the function `append` are given by a precondition and a postcondition that follow the syntax of `requires` and `ensures`, respectively. The precondition specifies that x and y both are the heads of two disjoint DLLs (the disjointness is enforced by the *separating conjunction* $*$), and the first DLL's length is positive (line 9). The postcondition expects that the result of `append` is a DLL, starting at x, with a length equal to the sum of the lengths of the initial lists (line 10). Here, the predicate definitions and program specifications are written after the notation //.

An astute reader could have noticed the bug we have planted on line 15 of Fig. 1: upon reaching the end of the x-headed DLL, the implementation does incorrectly set the pointer `y->next` to point to `x`. Let us now present how this mistake can be automatically discovered and fixed using our approach.

Firstly, the starting program state is the precondition $\mathsf{dll}(a, x, n) {*} \mathsf{dll}(y, b, m) \wedge n{>}0$. Then, we use separation logic rules to update a program state. When the condition `x->next == NULL` at line 12 is true, the predicate $\mathsf{dll}(x, a, n)$ is unfolded as $\exists u.\ x{\mapsto}\{a, u\} \wedge n{=}1 \wedge u{=}\mathsf{null}$. Consequently, the program state is $\exists u.\ x{\mapsto}\{a, u\} * \mathsf{dll}(y, b, m) \wedge n{=}1 \wedge u{=}\mathsf{null}$. Next, at line 13, we have the following Hoare triple $\{\exists u.\ x{\mapsto}\{a, u\}\}$ `x->next = y;` $\{\exists u.\ x{\mapsto}\{a, y\}\}$, leading to the program state of $\exists u.\ x{\mapsto}\{a, y\} * \mathsf{dll}(y, b, m) \wedge n{=}1 \wedge u{=}\mathsf{null}$. This program state could be simplified as $x{\mapsto}\{a, y\} * \mathsf{dll}(y, b, m) \wedge n{=}1$. Then, when the condition `y != NULL` at line 14 is `true`, the predicate $\mathsf{dll}(y, b, m)$ is unfolded as $\exists v.\ y{\mapsto}\{b, v\} * \mathsf{dll}(v, y, m{-}1)$. As a result, the program state upon reaching line 15 is $\exists v.\ x{\mapsto}\{a,y\} * y{\mapsto}\{b,v\} * \mathsf{dll}(v,y,m{-}1) \wedge n{=}1$. After executing the statement at line 15, the state is $\exists v.\ x{\mapsto}\{a,y\} * y{\mapsto}\{b,x\} * \mathsf{dll}(v,y,m{-}1) \wedge n{=}1$. Then, this state has to entail the postcondition, but the following entailment is invalid: $\exists v.\ x{\mapsto}\{a,y\} * y{\mapsto}\{b,x\} * \mathsf{dll}(v,y,m{-}1) \wedge n{=}1 \nvdash \mathsf{dll}(x,a,n{+}m)$. Hence, the function `append` is buggy.

Our approach repairs this buggy function by replacing a buggy statement with a template patch. The idea is to infer the specifications the template patch. Then, these specifications are used to synthesize program statements of the template patch. We will elaborate on the details of our approach in the next sections. We also use the motivating example to illustrate each step of our approach.

Note that the mutation-based approach [28] is not able to repair this motivating example. That approach relies on mutation operators, such as *mutate*, *delete*, *insert*. However, there is no expression `y->prev` or statement `y->prev = x` available in the program `dll-append` to replace the buggy expression or statement at line 15. Besides, as discussed in the Introduction (Sec. 1), the semi-automated approach [50] requires a user to specify program states at various points to repair this motivating example while our automated approach only requires a pair of pre- and postcondition of the input program.

3 Overview of Program Repair using Deductive Synthesis

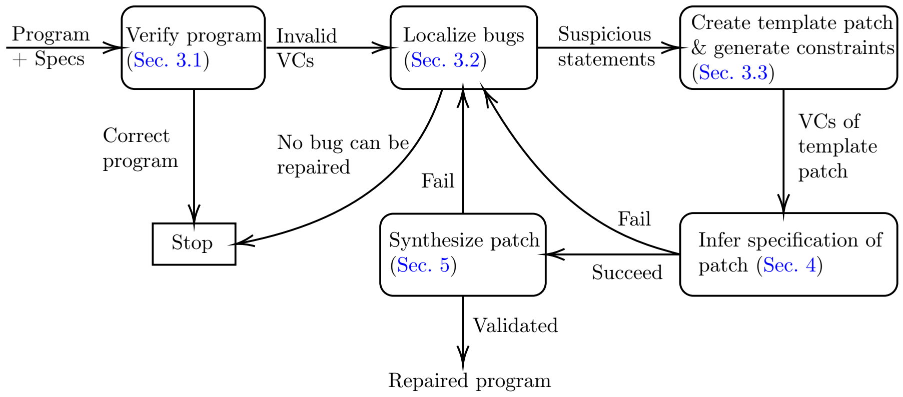

Fig. 2: Our automated program repair workflow.

Fig. 2 presents an overview of our program repair approach. An input program is first verified w.r.t. its specifications using a separation-logic-based verifier (Sec. 3.1). Concretely, the HIP/SLEEK verifier [7] is used. If the input program does not satisfy its specifications, it is considered buggy, and we start our repair process. Firstly, our approach collects a list of suspicious statements and rank them by their likelihood to trigger the bug (Sec. 3.2), based on the invalid verification conditions (VCs) and program traces generated during the verification step. Secondly, each suspicious statement `stmt`, starting from the highest-ranked one, is replaced by a template patch $\mathtt{TP}_{\mathtt{stmt}}$, consequently creating a *template program* (Sec. 3.3). This template patch $\mathtt{TP}_{\mathtt{stmt}}$ is accompanied by an initially unknown precondition $\mathcal{P}_{\texttt{tp}}$ and a postcondition $\mathcal{Q}_{\texttt{tp}}$. Thirdly, the approach invokes the verifier to analyze the template program to generate VCs related to unknown predicates $\mathcal{P}_{\texttt{tp}}$ and $\mathcal{Q}_{\texttt{tp}}$. These VCs are subsequently solved by our constraint solving technique to discover the definition of $\mathcal{P}_{\texttt{tp}}$ and $\mathcal{Q}_{\texttt{tp}}$ (Sec. 4). Then, these specifications $\mathcal{P}_{\texttt{tp}}$ and $\mathcal{Q}_{\texttt{tp}}$ is used to synthesize program statements of the template patch $\texttt{TP}_{\texttt{stmt}}$ (Sec. 5). If the synthesis step succeeds, a candidate program is created by replacing the template $\texttt{TP}_{\texttt{stmt}}$ with the synthesized statements. Finally, the candidate program is validated using the verifier to return a repaired program. Note that HIP/SLEEK is able to prove program termination [25,26]. Hence, the repaired program always terminates.

We will elaborate on the details of our framework in the rest of Sec. 3 and Sections 4 and 5. We also formalize our repair algorithm in Sec. 6.

3.1 Program Verification using Separation Logic

Fig. 3 presents the syntax of the formula of our specification language. They are formulae which follow the pre- and postcondition syntax (`requires` and `ensures`) as introduced in the motivating example (Fig. 1). Our separation logic fragment is called SL_{R} and contains inductive heap predicates and linear arithmetic. In this fragment, x, k, null denote a variable, an integer constant, and a null pointer, respectively. A term t can be an arithmetic expression e or a memory address expression a. Moreover, emp is the predicate describing the empty memory, and $x \overset{\iota}{\mapsto} \{t_1, ..., t_n\}$ is a singleton predicate representing a single data structure of type ι [3], pointed to by x, having n fields $t_1, ..., t_n$. Besides, $\mathsf{P}(t_1, ..., t_n)$ is an inductive predicate modeling a recursive data structure (Definition 1). These predicates compose a *spatial formula* Σ via the separating conjunction operator $*$. Moreover, Π denotes a *pure formula* in the first-order theory of equality and linear arithmetic. Finally, F is a *symbolic-heap formula.*

$$
\begin{array}{lll}
t \ ::= e \mid a & e ::= k \mid x \mid -e \mid e_1{+}e_2 \mid e_1{-}e_2 \mid k{\cdot}e & a ::= \mathsf{null} \mid x \\
\Pi \ ::= \mathsf{true} \mid \mathsf{false} \mid a_1{=}a_2 \mid a_1{\neq}a_2 \mid e_1{=}e_2 \mid e_1{\neq}e_2 \mid e_1{>}e_2 \mid e_1{\geq}e_2 \mid e_1{<}e_2 \mid e_1{\leq}e_2 \mid & & \\
\qquad \neg\Pi \mid \Pi_1{\wedge}\Pi_2 \mid \Pi_1{\vee}\Pi_2 \mid \Pi_1{\rightarrow}\Pi_2 \mid \forall x.\Pi \mid \exists x.\Pi & & \\
\Sigma \ ::= \mathsf{emp} \mid x \overset{\iota}{\mapsto} \{t_1, ..., t_n\} \mid \mathsf{P}(t_1, ..., t_n) \mid \Sigma_1 * \Sigma_2 & & F ::= \Sigma \mid \Pi \mid \Sigma \wedge \Pi \mid \exists x.F
\end{array}
$$

Fig. 3: Syntax of formulae in SL_{R}.

Definition 1 (Inductive heap predicate). *A system of k inductive heap predicates P_i, with $i = 1, ..., k$, is defined as follows, where each F^i_j is called a definition case of P_i, and is denoted as $F^i_j \overset{\text{def}}{\Rightarrow} \mathsf{P}_i$:*

$$\left\{\mathsf{P}_i(x^i_1, ..., x^i_{n_i}) \ \overset{\text{def}}{=}\ F^i_1(x^i_1, ..., x^i_{n_i}) \vee \ldots \vee F^i_{m_i}(x^i_1, ..., x^i_{n_i})\right\}_{i=1}^{k}$$

Example 1 (Doubly linked-list). The doubly-linked list in Sec. 2 is an example of an inductive heap predicate, which has one base case and one inductive case.

$$\mathsf{dll}(p, q, n) \ \overset{\text{def}}{=}\ (\mathsf{emp} \wedge p{=}\mathsf{null} \wedge n{=}0) \ \vee\ \exists r.\,(p{\mapsto}\{q, r\} * \mathsf{dll}(r, p, n{-}1))$$

Fig. 4 presents the semantics of formulae in our separation logic SL_{R}. Given a set `Var` of variables, `Sort` of sorts, `Val` of values, `Loc` of memory addresses

[3] For brevity, we omit ι when presenting examples.

($\texttt{Loc} \subset \texttt{Val}$), a model of a formula consists of: a *stack* model s, which is a function $s\colon \texttt{Var} \to \texttt{Val}$, and a *heap* model h, which is a partial function $h\colon (\texttt{Loc} \times \texttt{Sort}) \rightharpoonup \texttt{Val}^+$. In this model, $[\![\Pi]\!]_s$ denotes the value of a pure formula Π under the stack model s. Likewise, $\mathrm{dom}(h)$ is the domain of h; $h \# h'$ shows that h and h' have disjoint domains, i.e., $\mathrm{dom}(h) \cap \mathrm{dom}(h') = \varnothing$; and $h \circ h'$ is the union of two disjoint heap models h and h'. In addition, $[f \,|\, x{:}y]$ is a function like f except that it returns y for the input x. Regarding the semantics of an inductive heap predicate, we follow the standard *least fixed point semantics* [1] by interpreting an inductive predicate symbol P as the least prefixed point $[\![\mathsf{P}]\!]$ of a monotone operator constructed from its inductive definition.

$$
\begin{array}{lll}
s, h \models \Pi & \text{iff} & [\![\Pi]\!]_s = \mathsf{true} \text{ and } \mathrm{dom}(h) = \varnothing \\
s, h \models \mathsf{emp} & \text{iff} & \mathrm{dom}(h) = \varnothing \\
s, h \models x \overset{\iota}{\mapsto} \{x_1, ..., x_n\} & \text{iff} & \mathrm{dom}(h) = \{s(x)\} \text{ and } h(s(x), \iota) = (s(x_1), ..., s(x_n)) \\
s, h \models \mathsf{P}(x_1, ..., x_n) & \text{iff} & (h, [\![x_1]\!]_s, ..., [\![x_n]\!]_s) \in [\![\mathsf{P}]\!] \\
s, h \models \Sigma_1 * \Sigma_2 & \text{iff} & \exists h_1, h_2 : h_1 \# h_2;\ h_1 \circ h_2 = h;\ s, h_1 \models \Sigma_1;\ s, h_2 \models \Sigma_2 \\
s, h \models \Sigma \wedge \Pi & \text{iff} & [\![\Pi]\!]_s = \mathsf{true} \text{ and } s, h \models \Sigma \\
s, h \models \exists x.\, F & \text{iff} & \exists v \in \texttt{Val} : [s \,|\, x{:}v], h \models F
\end{array}
$$

Fig. 4: Semantics of formulae in SL_{R}.

We follow the literature to use separation logic [5,19,44] to verify the functional correctness of a program w.r.t. its specification. Separation logic follows Hoare logic in using a triple $\{\mathcal{P}\}\ \texttt{C}\ \{\mathcal{Q}\}$ to describe how the program state is updated during the execution of the program $\texttt{C}$. Here, $\mathcal{P}$ and $\mathcal{Q}$ represents the precondition and postcondition of the program $\texttt{C}$, respectively. The triple $\{\mathcal{P}\}\ \texttt{C}\ \{\mathcal{Q}\}$ expresses that given a starting program state of a program $\texttt{C}$ satisfying $\mathcal{P}$, if the program $\texttt{C}$ executes and terminates, then the resulting program state would satisfy $\mathcal{Q}$. Hence, a program $\texttt{C}$ is verified w.r.t. its specifications $\mathcal{P}$ and $\mathcal{Q}$ if the triple $\{\mathcal{P}\}\ \texttt{C}\ \{\mathcal{Q}\}$ holds. For instance, in Sec. 2, we have used separation logic rules to update program states and found that the motivating example in Fig. 1 is buggy because there exists an invalid entailment that consequently makes the Hoare triple of the function `append` not valid. Technically, we use the HIP/SLEEK verifier [7] to update program states. In this section, we do not present separation logic rules due to page limit. Interested readers could refer to [5,19,44].

3.2 Bug Localization

We localize suspicious statements and rank them according to their likelihood to cause a bug by utilizing invalid VCs and program traces collected during theg verification step (Sec. 3.1). Firstly, we collect a list of statements belonging to buggy traces. Then, we rank these statements by (i) how many times they appear in the buggy and correct traces and (ii) the distance from it to invalid VCs using program positions. Concretely, a statement is ranked higher if it appears more times in buggy traces and fewer times in the correct traces. Then, if two statements are the same in the first measure, the second measure is used.

For example, the only buggy trace in the motivating example (Fig. 1) is from taking the *if* branches of the two conditional statements. Hence, we collect two statements `x->next = y` at line 13 and `y->next = x` at line 15. Moreover, the statement `x->next = y` also appears in the correct trace when the conditional expression `y != NULL` at line 14 is `false`. Therefore, the second statement is more likely to cause the bug than the first one, consequently being ranked higher. In summary, we localize two suspicious statements with their corresponding ranking that are subsequently used as inputs of the next phase.

3.3 Template Patch Creation and Constraint Generation

In this phase, each suspicious statement is substituted by a template patch. This replacement will generate a program that our approach regards as a *template program*. Intuitively, the specifications, the pre- and postcondition, of the template patch will be inferred and later used to synthesize program statements of the template patch.

For example, Fig. 5 shows a *template program* created by replacing the highest-ranked suspicious statement `y->next = x` with the template patch $\texttt{TP}(x, y)$ at line 11. The special statement $\texttt{TP}(x, y)$ is currently encoded as a function call with parameters of all program variables available at that program location. We also encode the pre- and postcondition of this function call using unknown predicates $\mathsf{P}(x, y, a, b, n, m)$ and $\mathsf{Q}(x, y, a, b, n, m)$, respectively (lines 2, 3). The parameters of these predicates are parameters x, y of the template patch $\texttt{TP}(x, y)$ and other variables a, b, n, m in the precondition of `append`.

```
1:  void TP(node* x, node* y);
2:  // requires P(x, y, a, b, n, m)
3:  // ensures Q(x, y, a, b, n, m);
4:  void append(node* x, node* y)
5:  // requires dll(x, a, n) ∗ dll(y, b, m) ∧ n>0
6:  // ensures dll(x, a, n+m);
7:  {
8:     if (x->next == NULL) {
9:        x->next = y;
10:       if (y != NULL)
11:          TP(x,y);
12:    } else append(x->next, y);
13: }
```

Fig. 5: A template program.

To generate constraints related to the specifications of a template patch, the separation-logic-based verifier is called to verify the template program. All the entailments related to the specifications (unknown predicates) of the template patch are collected. The aim is to infer the definition of unknown predicates to make all VCs correct, and then use inferred specifications to synthesize statements of the template patch $\texttt{TP}(x, y)$. For example, we collect all VCs containing predicates P and Q in Fig. 5. These VCs are later used in Sec. 4 to infer the definition of predicates P and Q. Finally, these specifications are subsequently used in Sec. 5 to synthesize program statements of the template patch $\texttt{TP}(x, y)$.

Example 2 (VCs of template patch). The entailments related to unknown predicates P and Q of the template patch in Fig. 5 are as follows.

$$x{\mapsto}\{a, y\} * \mathsf{dll}(y, b, m) \land n{=}1 \land y{\neq}\mathsf{null} \vdash \mathsf{P}(x, y, a, b, n, m) * \mathsf{K}(x, y, a, b, n, m)$$

$$\mathsf{Q}(x, y, a, b, n, m) * \mathsf{K}(x, y, a, b, n, m) \quad \vdash \mathsf{dll}(x, a, m{+}n)$$

where the predicate $\mathsf{K}(x, y, a, b, n, m)$ is the frame formula [4] which is obtained after analyzing the function call $\mathtt{TP}(x, y)$ with the precondition $\mathsf{P}(x, y, a, b, n, m)$.

4 Specification Inference

In Sec. 3.3, we explain how to create a template program and collect entailments related to the specifications of the template patch. In this section, we will describe how our approach solves theses entailments to discover the definition of the specifications of the template patch.

4.1 Inference Rules

Our inference rules to discover the definition of unknown predicates are presented in Fig. 6. Each inference rule has zero or more premises, a conclusion, and possibly a side condition. A premise or a conclusion is of the form $\mathsf{S};\, \Delta$, where Δ is a set of entailments, and S is the current discovered solution (a set of definitions of unknown predicates). Furthermore, we write $\Delta,\, \{F_1 \vdash F_2\}$ to denote a new entailment set obtained by extending Δ with the entailment $F_1 \vdash F_2$. When F is a symbolic-heap formula of the form $\exists\vec{x}.(\Sigma \wedge \Pi)$, we define $F * \Sigma' \triangleq \exists\vec{x}.(\Sigma * \Sigma' \wedge \Pi)$ and $F \wedge \Pi' \triangleq \exists\vec{x}.(\Sigma \wedge \Pi \wedge \Pi')$, given that $\mathsf{fv}(\Sigma') \cap \vec{x} = \varnothing$ and $\mathsf{fv}(\Pi') \cap \vec{x} = \varnothing$. Here, $\mathsf{fv}(F)$ denotes the set of free variables in the formula F. We also write $\vec{u}{=}\vec{v}$ to denote $(u_1{=}v_1) \wedge \ldots \wedge (u_n{=}v_n)$, given that $\vec{u} \triangleq u_1, \ldots, u_n$ and $\vec{v} \triangleq v_1, \ldots, v_n$ are two variable lists of the same size. Finally, $\vec{u} \,\#\, \vec{v}$ indicates that the two lists $\vec{u}$ and $\vec{v}$ are disjoint, i.e., $\nexists w.(w \in \vec{u} \wedge w \in \vec{v})$.

$$\top\ \frac{}{\mathsf{S};\ \Delta}\ \Delta = \varnothing \qquad\qquad \vdash_\pi\ \frac{\mathsf{S};\ \Delta}{\mathsf{S};\ \Delta,\ \{\Pi_1 \vdash \Pi_2\}}\ \Pi_1{\rightarrow}\Pi_2$$

$$\bot_\pi\ \frac{\mathsf{S};\ \Delta}{\mathsf{S};\ \Delta,\ \{\Sigma_1 \wedge \Pi_1 \vdash F_2\}}\ \Pi_1{\rightarrow}\mathsf{false} \qquad\qquad \bot_\sigma\ \frac{\mathsf{S};\ \Delta}{\mathsf{S};\ \Delta,\ \{\Sigma_1 * u \overset{\iota_1}{\mapsto} \{\vec{t}\} * u \overset{\iota_2}{\mapsto} \{\vec{r}\} \wedge \Pi_1 \vdash F_2\}}$$

$$\mathsf{P_L}\ \frac{\mathsf{S};\ \Delta,\ \{F_1 * F_1^{\mathsf{P}}(\vec{t}) \vdash F_2\}, \ldots, \{F_1 * F_n^{\mathsf{P}}(\vec{t}) \vdash F_2\}}{\mathsf{S};\ \Delta,\ \{F_1 * \mathsf{P}(\vec{t}) \vdash F_2\}}\ \mathsf{P}(\vec{t}) \overset{\text{def}}{=} F_1^{\mathsf{P}}(\vec{t}) \vee \ldots \vee F_n^{\mathsf{P}}(\vec{t})$$

$$\mathsf{P_R}\ \frac{\mathsf{S};\ \Delta,\ \{F_1 \vdash \exists\vec{x}(F_2 * F_i^{\mathsf{P}}(\vec{t}))\}}{\mathsf{S};\ \Delta,\ \{F_1 \vdash \exists\vec{x}.(F_2 * \mathsf{P}(\vec{t}))\}}\ F_i^{\mathsf{P}}(\vec{t}) \overset{\text{def}}{\Rightarrow} \mathsf{P}(\vec{t}) \qquad\qquad =_{\mathsf{L}}\ \frac{\mathsf{S};\ \Delta,\ \{F_1[t/u] \vdash F_2[t/u]\}}{\mathsf{S};\ \Delta,\ \{F_1 \wedge u{=}t \vdash F_2\}}$$

$$*{\mapsto}\ \frac{\mathsf{S};\ \Delta,\ \{F_1 \vdash \exists\vec{x}.(F_2 \wedge u{=}v \wedge \vec{t}{=}\vec{r})\}}{\mathsf{S};\ \Delta,\ \{F_1 * u \overset{\iota}{\mapsto} \{\vec{t}\} \vdash \exists\vec{x}.(F_2 * v \overset{\iota}{\mapsto} \{\vec{r}\})\}}\ \mathsf{fv}(v, \vec{r}) \,\#\, \vec{x} \qquad\qquad \exists_{\mathsf{L}}\ \frac{\mathsf{S};\ \Delta,\ \{F_1[u/v] \vdash F_2\}}{\mathsf{S};\ \Delta,\ \{\exists v.F_1 \vdash F_2\}}\ u \notin \mathsf{fv}(F_1, F_2)$$

$$*\mathsf{P}\ \frac{\mathsf{S};\ \Delta,\ \{F_1 \vdash \exists\vec{x}.(F_2 \wedge \vec{t}{=}\vec{r})\}}{\mathsf{S};\ \Delta,\ \{F_1 * \mathsf{P}(\vec{t}) \vdash \exists\vec{x}.(F_2 * \mathsf{P}(\vec{r}))\}}\ \mathsf{fv}(\vec{r}) \,\#\, \vec{x} \qquad\qquad \exists_{\mathsf{R}}\ \frac{\mathsf{S};\ \Delta,\ \{F_1 \vdash \exists\vec{x}.F_2[t/u]\}}{\mathsf{S};\ \Delta,\ \{F_1 \vdash \exists\vec{x}, u.(F_2 \wedge u{=}t)\}}$$

$$\mathsf{U_L}\ \frac{\mathsf{S} \cup \{\mathsf{U}(\vec{t}) \overset{\text{def}}{=} F\};\ \Delta[F/\mathsf{U}(\vec{t})],\ \{F_1 \vdash F_2\}}{\mathsf{S};\ \Delta,\ \{F_1 * \mathsf{U}(\vec{t}) \vdash F_2 * F\}}\ \mathsf{U} \notin F_1, F_2 \qquad\qquad \mathsf{E_L}\ \frac{\mathsf{S};\ \Delta,\ \{F_1 \vdash F_2\}}{\mathsf{S};\ \Delta,\ \{F_1 * \mathsf{emp} \vdash F_2\}}$$

$$\mathsf{U_R}\ \frac{\mathsf{S} \cup \{\mathsf{U}(\vec{t}) \overset{\text{def}}{=} F\};\ \Delta[F/\mathsf{U}(\vec{t})],\ \{F_1 \vdash F_2\}}{\mathsf{S};\ \Delta,\ \{F_1 * F \vdash F_2 * \mathsf{U}(\vec{t})\}}\ \mathsf{U} \notin F_1, F_2 \qquad\qquad \mathsf{E_R}\ \frac{\mathsf{S};\ \Delta,\ \{F_1 \vdash \exists\vec{x}.F_2\}}{\mathsf{S};\ \Delta,\ \{F_1 \vdash \exists\vec{x}.(F_2 * \mathsf{emp})\}}$$

Fig. 6: Specification Inference Rules.

Most of our proposed rules are inspired by the standard entailment checking rules in separation logic literature [47,48]. However, there are two main differences. Firstly, they need to handle multiple entailments generated from the verification of a temporarily patched program. Secondly, they also have to deal with unknown heap predicates. We will explain the details of our rules as follows.

– *Axiom rule* $\top$. This rule will return the current set of discovered specification S if no entailment needs to be handled ($\Delta = \varnothing$).
– *Elimination rules* $\bot_\pi$, $\bot_\sigma$, $\vdash_\pi$. These rules eliminate a valid entailment from the entailment set Δ in their conclusions. Here, we utilize three simple checks for the validity of the candidate entailment when (i) it has a contradiction in the antecedent ($\bot_\pi$), (ii) or it contains overlaid singleton heaps ($\bot_\sigma$), (iii) or it is a pure entailment ($\vdash_\pi$). In the last case, an off-the-shelf prover like Z3 [10] will be invoked to prove the pure entailment.
– *Normalization rules* $\exists_\mathsf{L}$, $\exists_\mathsf{R}$, $=_\mathsf{L}$, $\mathsf{E_L}$, $\mathsf{E_R}$. These rules simplify an entailment in Δ by either eliminating existentially quantified variables ($\exists_\mathsf{L}$, $\exists_\mathsf{R}$), or removing equalities ($=_\mathsf{L}$) or empty heap predicates ($\mathsf{E_L}$, $\mathsf{E_R}$) from the entailment.
– *Unfolding rules* $\mathsf{P_L}$, $\mathsf{P_R}$. These rules derive new entailments from a goal entailment in Δ by unfolding a heap predicate in its antecedent or its consequent. Note that there is a slight difference between these two rules. When a heap predicate in the antecedent is unfolded ($\mathsf{P_L}$), all derived entailments will be added to Δ. In contrast, only one derived entailment will be added to Δ when a heap predicate in the consequent is unfolded ($\mathsf{P_R}$).
– *Matching rules* $*\!\mapsto$, $*\mathsf{P}$. These rules remove identical instances of singleton heap predicates ($*\!\mapsto$) or inductive heap predicates ($*\mathsf{P}$) from two sides of a goal entailment in Δ. Here, we ensure that these instances of predicates are identical by adding equality constraints about their parameters into the consequent of the derived entailment.
– *Solving rules* $\mathsf{U_L}$, $\mathsf{U_R}$. These rules discover the definition of an unknown heap predicate $\mathsf{U}(\vec{t})$ in a goal entailment of Δ and update it to the solution set S. More specifically, if $\mathsf{U}(\vec{t})$ appears in the entailment's antecedent, then the rule $\mathsf{U_L}$ chooses sub-formula of the consequent as the definition of $\mathsf{U}(\vec{t})$. Similarly, when $\mathsf{U}(\vec{t})$ appears in the consequent, then the rule $\mathsf{U_R}$ assigns $\mathsf{U}(\vec{t})$ to a sub-formula of the antecedent. In practice, these rules are often used when the entailment contains $\mathsf{U}(\vec{t})$ as its only heap predicate the antecedent (or the consequent). Then, the rule $\mathsf{U_L}$ (or $\mathsf{U_R}$) can simply choose the entire consequent (or the entire antecedent) as the definition of $\mathsf{U}(\vec{t})$.

4.2 Inference Algorithm

Fig. 7 presents our proof search procedure InferUnknPreds, which is implemented recursively to infer specifications from the unknown entailment set. Its inputs include a set Δ of unknown entailments and a set S of the currently discovered unknown heap predicates. This input pair correlates to a conclusion or a premise of an inference rule. Its output is a set that contains the definitions of the unknown heap predicates. When InferUnknPreds is invoked for the first time the input S is set to an empty list ($\varnothing$).

Given the predicate set S and the unknown entailment set Δ, the algorithm InferUnknPreds considers two cases. The first case is when there exists an entailment $F \vdash G$ that has more than one unknown predicate in G and no unknown predicate in F. Then, the definitions of unknown predicates in G are

Procedure: $\mathsf{InferUnknPreds}(\mathsf{S}, \mathit{\Delta})$

Input: $\mathit{\Delta}$, S are sets of unknown entailments and discovered heap predicates.
Output: The solution set of unknown predicates.

```
1:  if $F \vdash G \in \mathit{\Delta} \wedge \mathsf{NumOfUnknPred}(F) = 0 \wedge \mathsf{NumOfUnknPred}(G) > 1$ then
2:      $\mathit{\Omega} \leftarrow \mathsf{DivideHeapFormula}(F)$                ▷ devide the spatial formula of $F$
3:      for each $(\mathsf{S_{sub}}, \mathit{\Delta}_{\mathsf{sub}})$ in $\mathit{\Omega}$ do                          ▷ One way of dividing
4:          $\mathsf{res} \leftarrow \mathsf{InferUnknPreds}(\mathsf{S_{sub}}, \mathit{\Delta}_{\mathsf{sub}})$
5:          if $\mathsf{res} \neq \varnothing$ then return $\mathsf{res}$                               ▷ Discover a solution
6:  else
7:      $\mathcal{R} \leftarrow \mathsf{FindInferRules}(\mathsf{S}, \mathit{\Delta})$                              ▷ find inference rules
8:      for each $\mathsf{R}$ in $\mathcal{R}$ do
9:          if $\mathsf{R} = \top$ then return $\mathsf{S}$                                            ▷ axiom rule
10:         else                                                          ▷ other inference rules
11:             $(\mathsf{S}', \mathit{\Delta}') \leftarrow \mathsf{GetPremise}(\mathsf{R})$                         ▷ apply the chosen rule
12:             $\mathsf{res} \leftarrow \mathsf{InferUnknPreds}(\mathsf{S}', \mathit{\Delta}')$
13:             if $\mathsf{res} \neq \varnothing$ then return $\mathsf{res}$                            ▷ discover a solution
14:     return $\varnothing$                                                         ▷ fail to solve $\mathit{\Delta}$
```

Fig. 7: Proof search algorithm for unknown entailments.

discovered by defining their spatial and pure formulae (line 2). Firstly, their spatial formulae of unknown predicates are defined by dividing the spatial formula of the antecedent F using the procedure $\mathsf{DivideHeapFormula}$. For instance, the first entailment in Example 2 has two unknown predicate $\mathsf{P}(x, y, a, b, n, m)$ and $\mathsf{K}(x, y, a, b, n, m)$ in its consequent, and has no unknown predicate in its antecedent. Hence, the pair $(\mathsf{P_S}, \mathsf{K_S})$ that contains the corresponding spatial formulae of $\mathsf{P}(x, y, a, b, n, m)$ and $\mathsf{K}(x, y, a, b, n, m)$ could be either $(\mathsf{emp}, x{\mapsto}\{a, y\} * \mathsf{dll}(y, b, m))$, or $(x{\mapsto}\{a, y\}, \mathsf{dll}(y, b, m))$, or pairs in the reverse order. Then, the pure part of an unknown predicate is defined by using constraints in the antecedent F such that the constraints are related to variables in the spatial formula and the parameters of the predicate. For example, if we have $\mathsf{P_S} \overset{\text{def}}{=} x{\mapsto}\{a, y\}$, then we have the following definition: $\mathsf{P}(x, y, a, b, n, m) \overset{\text{def}}{=} x{\mapsto}\{a, y\} \wedge n{=}1 \wedge y{\neq}\mathsf{null}$. Each way of dividing the spatial formula of the antecedent F results in a pair $(\mathsf{S_{sub}}, \mathit{\Delta}_{\mathsf{sub}})$ where the definitions of unknown predicates are added to the set S to generate $\mathsf{S_{sub}}$. Next, $\mathit{\Delta}_{\mathsf{sub}}$ is obtained by substituting unknown predicates by their corresponding definitions in $\mathit{\Delta}$. Then, the algorithm with new arguments $(\mathsf{S_{sub}}, \mathit{\Delta}_{\mathsf{sub}})$ continues recursively (lines 4, 5).

In the second case, when there is no such entailment $F \vdash G$, the algorithm first finds from all inference rules presented in Fig. 6 a set of rules $\mathcal{R}$ whose conclusion can be unified with the entailment set $\mathit{\Delta}$ (line 7). Then $\mathsf{InferUnknPreds}$ subsequently applies each of the selected rules in $\mathcal{R}$ to solve the unknown entailment set $\mathit{\Delta}$. In particular, if the selected rule R is an axiom rule $\top$, the procedure $\mathsf{InferUnknPreds}$ immediately returns the current solution set S, which does not derive any new entailment set (line 9). Otherwise, it continues to solve the new set of unknown entailments obtained from the premise of the rule R (lines 11, 12) to discover the definitions of the unknown predicates (line 13). Finally, $\mathsf{InferUnknPreds}$ returns an empty set ($\varnothing$) if all selected rules fail to solve the unknown entailment set Δ (line 14).

In practice, to make the proof search more efficient, we also rank the discovered inference rules in $\mathcal{R}$ by their likelihood to solve the unknown entailments. These heuristics are as follows:

– The axiom rule ($\top$) is the most important since it immediately returns the solution set.
– The elimination rules ($\bot_\sigma, \bot_\pi, \vdash_\pi$) are the second most important since they can remove valid entailments from the entailment set Δ.
– The normalization rules ($\exists_\mathsf{L}, \exists_\mathsf{R}, \mathsf{E_L}, \mathsf{E_R}, =_\mathsf{L}$) are the third most important since they can simplify and make all the entailments more concise.
– Other rules ($\mathsf{P_L}, \mathsf{P_R}, {*}{\mapsto}, {*}\mathsf{P}, \mathsf{P_L}, \mathsf{P_R}$) generally have the same priority. However, in several special cases, the priority of these rules change as follows:
  - The rules ${*}{\mapsto}$, ${*}\mathsf{P}$, $\mathsf{P_L}$, $\mathsf{P_R}$ have high priority if the following conditions are satisfied. (i) The rule ${*}{\mapsto}$ matches and removes singleton heap predicates of the same root. (ii) The rule ${*}\mathsf{P}$ matches and removes identical instances of inductive heap predicates. (iii) In the rule $\mathsf{P_L}$, F_1 is a pure formula and F_2 is emp. (iv) In the rule $\mathsf{P_R}$, F_1 is emp and F_2 is a pure formula.
  - The rules ${*}{\mapsto}$, ${*}\mathsf{P}$ have high priority when they match and remove heap predicates that have some identical arguments.
  - Finally, the rules $\mathsf{P_L}$, $\mathsf{P_R}$ are more important if after unfolding, they can introduce heap predicates that have some identical arguments, which can be removed latter by the two rules ${*}{\mapsto}$, ruleInferStarPred.

$$\dfrac{\dfrac{\dfrac{}{\mathsf{S}\cup\{\mathsf{Q}(x,y,a,b,n,m)\overset{\text{def}}{=}\mathsf{dll}(x,a,m{+}1)\};\ \varnothing}\ \top}{\mathsf{S};\ \mathsf{Q}(x,y,a,b,n,m)\wedge y{\neq}\mathsf{null}\vdash\mathsf{dll}(x,a,m{+}1)}\ \mathsf{U_L}}{\mathsf{S};\ \mathsf{Q}(x,y,a,b,n,m)\wedge n{=}1\wedge y{\neq}\mathsf{null}\vdash\mathsf{dll}(x,a,m{+}n)}\ =_\mathsf{L}$$

Fig. 8: A proof tree of applying specification inference rules.

Example 3 (Specification inference). We illustrate how to apply specification inference rules to solve to the below unknown entailments, given in Example 2.

$$x{\mapsto}\{a,y\} * \mathsf{dll}(y,b,m)\wedge n{=}1\wedge y{\neq}\mathsf{null}\vdash\mathsf{P}(x,y,a,b,n,m) * \mathsf{K}(x,y,a,b,n,m)$$
$$\mathsf{Q}(x,y,a,b,n,m) * \mathsf{K}(x,y,a,b,n,m)\quad\vdash\mathsf{dll}(x,a,m{+}n)$$

The first entailment has two unknown predicates, namely $\mathsf{P}(x,y,a,b,n,m)$ and $\mathsf{K}(x,y,a,b,n,m)$, in its consequent, and no unknown predicate in its antecedent. Hence, the definitions of $\mathsf{P}(x,y,a,b,n,m)$ and $\mathsf{K}(x,y,a,b,n,m)$ are discovered using $\mathsf{DivideHeapFormula}$ to partition the spatial part of the antecedent. One possible solution is that the spatial part of $\mathsf{P}(x,y,a,b,n,m)$ is $x{\mapsto}\{a,y\}*\mathsf{dll}(y,b,m)$ while the spatial part of $\mathsf{K}(x,y,a,b,n,m)$ is emp, as follows:

$$\mathsf{P}(x,y,a,b,n,m)\overset{\text{def}}{=}\ x{\mapsto}\{a,y\} * \mathsf{dll}(y,b,m)\wedge y{\neq}\mathsf{null}$$
$$\mathsf{K}(x,y,a,b,n,m)\overset{\text{def}}{=}\ \mathsf{emp}\wedge n{=}1\wedge y{\neq}\mathsf{null}$$

Now, we can replace $\mathsf{K}(x,y,a,b,n,m)$ in the second entailment with its actual definition to obtain the following entailment.

$$\mathsf{Q}(x,y,a,b,n,m) \wedge n{=}1 \wedge y{\neq}\mathsf{null} \vdash \mathsf{dll}(x,a,m{+}n)$$

The above entailment can be solved by our inference rules, as presented in the proof tree in Fig. 8 where S contains the definition of predicates $\mathsf{P}(x,y,a,b,n,m)$ and $\mathsf{K}(x,y,a,b,n,m)$.

5 Deductive Program Synthesis

In this section, we show how program statements of a template patch are synthesized from the specifications inferred in Sec. 4. We first define the notion of synthesis goal, then explain all synthesis rules, and finally introduce an algorithm that synthesizes program statements using synthesis rules.

$$\textsf{Exists}_{\textsf{L}}\ \frac{\Gamma; V; \{F_1[t/u]\} \rightsquigarrow \{F_2\} \mid \texttt{C}}{\Gamma; V; \{\exists u.F_1\} \rightsquigarrow \{F_2\} \mid \texttt{C}}\ t{\notin}\mathsf{fv}(V,F_1,F_2) \qquad \textsf{Exists}_{\textsf{R}}\ \frac{\Gamma; V; \{F_1\} \rightsquigarrow \{\exists \vec{z}.F_2[t/u]\} \mid \texttt{C}}{\Gamma; V; \{F_1\} \rightsquigarrow \{\exists \vec{z},u.(F_2 \wedge u{=}t)\} \mid \texttt{C}}$$

$$\textsf{Frame}_{\mapsto}\ \frac{\Gamma; V; \{\Sigma_1 \wedge \Pi_1\} \rightsquigarrow \{\exists \vec{z}.(\Sigma_2 \wedge \Pi_2)\} \mid \texttt{C}}{\Gamma; V; \{\Sigma_1 * u\overset{\iota}{\mapsto}\{\vec{t}\} \wedge \Pi_1\} \rightsquigarrow \{\exists \vec{z}.(\Sigma_2 * u\overset{\iota}{\mapsto}\{\vec{t}\} \wedge \Pi_2)\} \mid \texttt{C}}\ \mathsf{fv}(u,\vec{t}) \,\#\, \vec{z}$$

$$\textsf{Frame}_{\textsf{P}}\ \frac{\Gamma; V; \{\Sigma_1 \wedge \Pi_1\} \rightsquigarrow \{\exists \vec{z}.(\Sigma_2 \wedge \Pi_2)\} \mid \texttt{C}}{\Gamma; V; \{\Sigma_1 * \mathsf{P}(\vec{t}) \wedge \Pi_1\} \rightsquigarrow \{\exists \vec{z}.(\Sigma_2 * \mathsf{P}(\vec{t}) \wedge \Pi_2)\} \mid \texttt{C}}\ \mathsf{fv}(\vec{t}) \,\#\, \vec{z}$$

$$\textsf{Unfold}_{\textsf{L}}\ \frac{\Gamma; V; \{F_1 * F_P^i(\vec{t})\} \rightsquigarrow \{F_2\} \mid \texttt{C}}{\Gamma; V; \{F_1 * \mathsf{P}(\vec{t})\} \rightsquigarrow \{F_2\} \mid \texttt{C}}\ \begin{array}{l} \mathsf{P}(\vec{t}) \overset{\text{def}}{=} F_P^1(\vec{t}) \vee \ldots \vee F_P^n(\vec{t}) \\ 1{\leq}i{\leq}n, \forall j{\neq}i, F_1 * F_P^j(\vec{t}) \equiv \mathsf{false} \end{array}$$

$$\textsf{Unfold}_{\textsf{R}}\ \frac{\Gamma; V; \{F_1\} \rightsquigarrow \{\exists \vec{z}.(F_2 * F_P(\vec{t}))\} \mid \texttt{C}}{\Gamma; V; \{F_1\} \rightsquigarrow \{\exists \vec{z}.(F_2 * \mathsf{P}(\vec{t}))\} \mid \texttt{C}}\ F_P(\vec{t}) \overset{\text{def}}{\Rightarrow} \mathsf{P}(\vec{t})$$

$$\textsf{Call}\ \frac{\Gamma \cup \{\{G_1\}\texttt{fname(}\vec{\texttt{u}}\texttt{)}\{G_2\}\}; V \cup \{v\}; \{F_1 * G_2\theta[v/\mathsf{res}]\} \rightsquigarrow \{F_2\} \mid \texttt{C}}{\Gamma \cup \{\{G_1\}\texttt{fname(}\vec{\texttt{u}}\texttt{)}\{G_2\}\}; V; \{F_1 * F\} \rightsquigarrow \{F_2\} \mid \texttt{typ v = fname(}\vec{u}\theta\texttt{); C}}\ \begin{array}{l} G_1\theta = F, \\ \mathsf{fv}(\vec{u}\theta) \in V \end{array}$$

$$\textsf{Assign}\ \frac{\Gamma; V \cup \{u\}; \{F_1 \wedge u{=}e\} \rightsquigarrow \{\exists \vec{z}.(F_2 \wedge u{=}e)\} \mid \texttt{C}}{\Gamma; V \cup \{u\}; \{F_1\} \rightsquigarrow \{\exists \vec{z}.(F_2 \wedge u{=}e)\} \mid \texttt{u=e; C}}\ \begin{array}{l} u \notin \vec{z}, u \notin \mathsf{fv}(F_1), \\ \mathsf{fv}(e) \subseteq V \end{array}$$

$$\textsf{Skip}\ \frac{}{\Gamma; V; \{F_1\} \rightsquigarrow \{F_2\} \mid \texttt{skip}}\ F_1 \vdash F_2 \qquad \textsf{Ret}\ \frac{}{\Gamma; V; \{F_1\} \rightsquigarrow \{\exists \vec{z}.(F_2 \wedge \mathsf{res}{=}e)\} \mid \texttt{return e;}}\ \begin{array}{l} F_1 \vdash \exists \vec{z}.F_2, \\ \mathsf{fv}(e) \subseteq V \end{array}$$

$$\textsf{Read}\ \frac{\Gamma; V \cup \{v\}; \{\Sigma_1 * u\overset{\iota}{\mapsto}(\texttt{fld}:t) \wedge \Pi_1 \wedge v{=}t\} \rightsquigarrow \{F_2\} \mid \texttt{C}}{\Gamma; V; \{\Sigma_1 * u\overset{\iota}{\mapsto}(\texttt{fld}:t) \wedge \Pi_1\} \rightsquigarrow \{F_2\} \mid \texttt{typ v = u->fld; C}}\ \begin{array}{l} u \in V, \\ v \notin V \end{array}$$

$$\textsf{Write}\ \frac{\Gamma; V; \{F_1 * u\overset{\iota}{\mapsto}(\texttt{fld}:t)\} \rightsquigarrow \{F_2 * u\overset{\iota}{\mapsto}(\texttt{fld}:t)\} \mid \texttt{C}}{\Gamma; V; \{F_1 * u\overset{\iota}{\mapsto}(\texttt{fld}:r)\} \rightsquigarrow \{F_2 * u\overset{\iota}{\mapsto}(\texttt{fld}:t)\} \mid \texttt{u->fld = t; C}}\ \begin{array}{l} \mathsf{fv}(u,t) \subseteq V, \\ r{\neq}t \end{array}$$

$$\textsf{Alloc}\ \frac{\Gamma; V \cup \{u\}; \{\Sigma_1 * u\overset{\iota}{\mapsto}\{\vec{v}\} \wedge \Pi_1\} \rightsquigarrow \{\Sigma_2 * u\overset{\iota}{\mapsto}\{\vec{t}\} \wedge \Pi_2\} \mid \texttt{C}}{\Gamma; V; \{\Sigma_1 \wedge \Pi_1\} \rightsquigarrow \{\Sigma_2 * u\overset{\iota}{\mapsto}\{\vec{t}\} \wedge \Pi_2\} \mid \texttt{struct}\ \iota\ \texttt{u = malloc(sizeof(struct}\ \iota\texttt{)); C}}\ \begin{array}{l} u \notin \mathsf{fv}(V,\Sigma_1) \\ \vec{v} \text{ are fresh} \\ \vec{t} \subseteq V \end{array}$$

$$\textsf{Free}\ \frac{\Gamma; V; \{F_1 \wedge \Pi_1\} \rightsquigarrow \{\exists \vec{z}.(\Sigma_2 \wedge \Pi_2)\} \mid \texttt{C}}{\Gamma; V; \{F_1 * u\mapsto\{\vec{t}\} \wedge \Pi_1\} \rightsquigarrow \{\exists \vec{z}.(\Sigma_2 \wedge \Pi_2)\} \mid \texttt{free(u); C}}\ \begin{array}{l} u \notin \mathsf{fv}(\Sigma_2) \\ u, \vec{t} \subseteq V \end{array}$$

Fig. 9: Deductive Synthesis Rules.

A *synthesis goal* is written as $\Gamma; V; \{F_1\} \rightsquigarrow \{F_2\} \mid \texttt{C}$, where Γ is a list of declared functions that supports to synthesize function call statements, V consists of all available variables that could be used during the synthesis algorithm, F_1

is a precondition, F_2 is a postcondition, and `C` is a list of program statements that will be synthesized. Hence, solving a synthesis goal is equivalent to finding program statements `C` such that the Hoare triple $\{F_1\}$ `C` $\{F_2\}$ holds.

5.1 Synthesis Rules

Fig. 9 presents our synthesis rules to synthesize program statements. A synthesis rule contains zero or more premises, a conclusion, and possible side conditions. A premise or a conclusion of a synthesis rule is a synthesis goal. Here, `typ`, `fld`, and `fname` indicate a variable type, a field of a data structure, and a function name, respectively. Besides, res is a keyword in our specification language to indicate the returned result of a function. Other notations are introduced previously in Sec. 4.1 and Sec. 3.1. All synthesis rules are described as follows.

- *Simplification rules* $\mathsf{Exists_L}$ and $\mathsf{Exists_R}$. These rules simplify a synthesis goal by removing an existential variable in its precondition ($\mathsf{Exists_L}$) or its postcondition ($\mathsf{Exists_R}$).
- *Frame rules* $\mathsf{Frame_{\mapsto}}$ and $\mathsf{Frame_P}$. These rules remove an identical singleton heap predicate ($\mathsf{Frame_{\mapsto}}$) or inductive heap predicate ($\mathsf{Frame_P}$) from the pre- and postcondition of a synthesis goal.
- *Unfolding rules* $\mathsf{Unfold_L}$ and $\mathsf{Unfold_R}$. These rules produce a new synthesis goal by unfolding an inductive heap predicate in the pre- or postcondition of a synthesis goal. When an inductive heap predicate is unfolded in the precondition ($\mathsf{Unfold_L}$), the side conditions ensure that only one definition case $F^i_P(\vec{t})$ of $\mathsf{P}(\vec{t})$ is satisfiable. In contrast, unfolding an inductive heap predicate $\mathsf{P}(\vec{t})$ in the postcondition ($\mathsf{Unfold_R}$) creates multiple subgoals, but solving the current synthesis goal requires only one subgoal to succeed.
- *Rule* Call. This rule invokes a function call $\mathtt{fname}(\vec{u})$ which has the specification of $\{G_1\}\mathtt{fname}(\vec{u})\{G_2\}$ when all chosen input arguments $\vec{u}\theta$ satisfy the specification of its corresponding parameters. Here, θ is a substitution of arguments to the parameters $\vec{u}$ of the function, i.e., replacing the function's formal parameters with the corresponding actual arguments. Then, updating the precondition of the synthesis goal is similar to update a program state in formal verification when a function call $\mathtt{fname}(\vec{u}\theta)$ is encountered.
- *Rule* Assign. The rule Assign assigns a value e to a variable u that is in the list V ($u \in V$) when a constraint $u{=}e$ appears in the postcondition but the variable u is not assigned any value in the precondition ($u \notin \mathsf{fv}(F_1)$). Consequently, an assignment statement `u = e;` is generated.
- *Rules* Skip and Ret. The rule Skip is applicable when there exists a valid entailment $F_1 \vdash F_2$. It also marks that a synthesis goal is solved by producing a `skip` statement. Meanwhile, the rule Ret generates a statement `return e;`. It is similar to combining the rule Assign (with `u` is res) and the rule Skip. These rules have no premises, meaning that they are terminating rules.
- *Rules* Read and Write. The rule Read assigns the value of a field of a heap variable to a new variable. For instance, if a statement `append(x->next, y)` needs to be synthesized when repairing a buggy `dll-append` program, Read is executed to create the statement `node* nx = x->next;`. Then, Call is used to

synthesize the statement `append(nx, y);`. Meanwhile, the rule Write assigns a new value to a field of a heap variable is if the values of the field of the variable in the pre- and postcondition differ (see Example 4).

– *Rules* Alloc and Free. The rule Alloc allocates a new variable u of the data structure ι if all arguments $\vec{t}$ are in the list V. This rule is called when a new heap variable u appears in the spatial formula of the postcondition but not the precondition ($u \notin \mathsf{fv}(\Sigma_1)$). On the other hand, the rule Free deallocates a heap variable u if it is in the spatial formula of the precondition but not in the spatial formula of the postcondition.

5.2 Synthesis Algorithm

Fig. 10 shows our synthesis algorithm SynthesizeStmts. Given a list of declared functions Γ, a list of available variables V, a precondition F_1, and a postcondition F_2, the algorithm SynthesizeStmts aims to produce a program statement `C` that solves the synthesis goal $\Gamma; V; \{F_1\} \rightsquigarrow \{F_2\} \mid$ `C`. Hence, the program statement `C` is the patch that will replace the template patch to create a candidate program.

Algorithm: SynthesizeStmts(Γ, V, F_1, F_2)

Input: A list of function declarations Γ, a list of available variables V, a precondition F_1, and a postcondition F_2.

Output: A list of synthesized statements, or Fail if no statement is synthesized.

```
1:  $\mathcal{R} \leftarrow$ FindSynRules($\Gamma, V, F_1, F_2$)                         ▷ find applicable rules
2:  for each rule R in $\mathcal{R}$ do
3:    if R $\in$ {Skip, Ret} then return DeriveStmt(R)              ▷ a terminating rule
4:    else if R $\in$ {Exists$_L$, Exists$_R$, Frame$_{\mapsto}$, Frame$_P$, Unfold$_L$, Unfold$_R$} then
5:      ($\Gamma', V', F_1', F_2'$) $\leftarrow$ DeriveNewGoal(R)                ▷ a normalization rule
6:      return SynthesizeStmts($\Gamma, V', F_1', F_2'$)
7:    else                                                              ▷ other rules
8:      ($\Gamma', V', F_1', F_2'$), stmt $\leftarrow$ DeriveNewGoalAndStmt(R)
9:      res $\leftarrow$ SynthesizeStmts($\Gamma', V', F_1', F_2'$)
10:     if res $\neq$ Fail then return AppendStmts(stmt, res)     ▷ found a solution
11: return Fail                                                         ▷ fail to synthesize
```

Fig. 10: The SynthesizeStmts algorithm.

The algorithm SynthesizeStmts first finds from all synthesis rules presented in Sec. 5.1 a list of rules $\mathcal{R}$ that are applicable to the current tuple (Γ, V, F_1, F_2) (line 1). If there exists a terminating rule (Skip or Ret), then SynthesizeStmts returns a `skip` statement (Skip) or a return statement (Ret) (line 3). If a normalization rule is selected, then SynthesizeStmts will immediately apply it to derive a new goal and continue the synthesis (lines 4–6). For other synthesis rules, SynthesizeStmts subsequently executes each of them to derive both a new goal $(\Gamma', V', F_1', F_2')$ and a program statement `stmt` (line 8). Our algorithm also checks if the rule Call is executed on a smaller sub-heap of the precondition F_1 that consequently ensures termination of a patched program. The algorithm will continue the synthesis process on the new goal (line 9) and will append the previously synthesized statement `stmt` on the new result `res` to return all synthesized statements (line 10). Besides, it also returns Fail if all selected rules fail to synthesize program statements from the current inputs (line 11).

In practice, we also ranks synthesis rules collected by FindSynRules to make the algorithm SynthesizeStmts more effective. Firstly, two terminating rules Skip and Ret are the most important since they immediately return a list of synthesized statements. Then, the rules $\textsf{Exists}_\textsf{L}$, $\textsf{Exists}_\textsf{R}$, $\textsf{Unfold}_\textsf{L}$ are the second most important rules because they simplify the current synthesis goal and generate a more concise synthesis goal. Finally, other synthesis rules are ranked equally.

$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\begin{array}{c}\dfrac{\Gamma;V;\{x\mapsto\{a,y\} * y\mapsto\{x,t\}\} \rightsquigarrow \{x\mapsto\{a,y\} * y\mapsto\{x,t\}\} \mid \texttt{C'} \qquad \texttt{C'} = \texttt{skip}}{\Gamma;V;\{x\mapsto\{a,y\} * y\mapsto\{x,t\}\} \rightsquigarrow \{x\mapsto\{a,y\} * y\mapsto\{x,t\}\} \mid \texttt{C}}\,\textsf{Skip} \\ \texttt{C = y-> prev = x; C'}\end{array}}{\Gamma;V;\{x\mapsto\{a,y\} * y\mapsto\{b,t\}\} \rightsquigarrow \{x\mapsto\{a,y\} * y\mapsto\{x,t\}\} \mid \texttt{C}}\,\textsf{Write}}{\Gamma;V;\{x\mapsto\{a,y\} * y\mapsto\{b,t\} * \mathsf{dll}(t,y,m{-}1)\} \rightsquigarrow \{\exists k,h.\ x\mapsto\{a,k\} * k\mapsto\{x,h\} * \mathsf{dll}(h,k,m{-}1)\} \mid \texttt{C}}\,\textsf{Frame}_\textsf{P}}{\Gamma;V;\{x\mapsto\{a,y\} * y\mapsto\{b,t\} * \mathsf{dll}(t,y,m{-}1)\} \rightsquigarrow \{\exists k.\ x\mapsto\{a,k\} * \mathsf{dll}(k,x,m)\} \mid \texttt{C}}\,\textsf{Unfold}_\textsf{R}}{\Gamma;V;\{\ x\mapsto\{a,y\} * y\mapsto\{b,t\} * \mathsf{dll}(t,y,m{-}1)\} \rightsquigarrow \{\mathsf{dll}(x,a,m{+}1)\} \mid \texttt{C}}\,\textsf{Unfold}_\textsf{R}}{\Gamma;V;\{\exists t.\ x\mapsto\{a,y\} * y\mapsto\{b,t\} * \mathsf{dll}(t,y,m{-}1)\} \rightsquigarrow \{\mathsf{dll}(x,a,m{+}1)\} \mid \texttt{C}}\,\textsf{Exists}_\textsf{L}}{\Gamma;V;\{x\mapsto\{a,y\} * \mathsf{dll}(y,b,m)\wedge y\neq\mathsf{null}\} \rightsquigarrow \{\mathsf{dll}(x,a,m{+}1)\} \mid \texttt{C}}\,\textsf{Unfold}_\textsf{L}
$$

Fig. 11: Synthesis rules are applied to specifications inferred in Example 3.

Example 4 (The patch to repair the motivating example). Fig. 11 explains how our synthesis algorithm applies synthesis rules to synthesize a program statement from the specifications inferred in Example 3. Here, Γ contains a declaration of function `append` while the list V is $\{x, y\}$. The algorithm SynthesizeStmts first uses the rule $\textsf{Unfold}_\textsf{L}$ to remove the constraint $y{\neq}\mathsf{null}$ from the precondition. Next, the rule $\textsf{Exists}_\textsf{L}$ is used to remove the existential variable t from the precondition. Then, the postcondition is unfolded twice using the rule $\textsf{Unfold}_\textsf{R}$ to have a predicate `dll` of length $m{-}1$ like in the precondition. Next, the predicate `dll` of length $m{-}1$ is removed from both the pre- and postcondition using the rule $\textsf{Frame}_\textsf{P}$. Finally, the field `prev` of the variable `y` is updated according to the rule Write to terminate the synthesis algorithm with the rule Skip. Therefore, a program statement `y->prev = x` is synthesized. This statement will replace the template patch `TP(x,y)` at line 11 in Fig. 5 to produce a candidate program. Finally, the candidate program is validated w.r.t. the specifications of the function `append` and then returned as the repaired program of the motivating example.

6 Repair Algorithm

We formally introduce the algorithm Repair in Fig. 12. The inputs of the algorithm include a program `C`, its precondition $\mathcal{P}$, its postcondition $\mathcal{Q}$, and an environment variable Γ containing all declared functions that can be invoked during the repair process.

The algorithm first verifies if program `C` is correct against its specifications (line 1). If the verification fails, then `C` is considered buggy. In this case, all the invalid VCs generated during the verification step are collected (line 2). Next, the algorithm Repair utilizes these VCs to localize a list of suspicious statements

S (line 3). It also ranks these suspicious statements according to their likelihood to cause the bug. Then, it attempts to repair each suspicious statement in S, starting from the highest-ranked one (lines 4–14).

More specifically, the algorithm creates a template program $\mathtt{C}_{\mathtt{stmt}}$ for each suspicious statement stmt (line 5): it replaces that statement with a template patch $\mathtt{TP}_{\mathtt{stmt}}$, which is specified by a pair of unknown pre- and postcondition $\mathcal{P}_{\mathtt{tp}}$, $\mathcal{Q}_{\mathtt{tp}}$. Then, Repair verifies the template patched program to collect all VCs related to $\mathcal{P}_{\mathtt{tp}}$, $\mathcal{Q}_{\mathtt{tp}}$ (line 6). These VCs will be solved by the algorithm InferUnknPreds (described in Sec. 4.2) to infer the actual definition of $\mathcal{P}_{\mathtt{tp}}$, $\mathcal{Q}_{\mathtt{tp}}$ (line 7). If this inference succeeds (lines 8, 9), the inferred pre- and postcondition $\overline{\mathcal{P}}_{\mathtt{tp}}$, $\overline{\mathcal{Q}}_{\mathtt{tp}}$ will be used to synthesize correct program statements of the template patch (line 11), by using the algorithm SynthesizeStmts as explained in Sec. 5.2. When SynthesizeStmts can synthesize a list of program statements, our algorithm Repair will replace the template patch $\mathtt{TP}_{\mathtt{stmt}}$ with synthesized statements to create a candidate program $\bar{\mathtt{C}}$ (line 13). If $\bar{\mathtt{C}}$ can be validated by the procedure Verify, it will be returned as the repaired program (line 14). Otherwise, the algorithm Repair returns Fail if it is unable to fix the input buggy program (line 15). It also returns None if the input program C is correct w.r.t. its specifications (line 16).

Algorithm: Repair($\mathit{\Gamma}, \mathcal{P}, \mathtt{C}, \mathcal{Q}$)

Input: $\mathit{\Gamma}$ is a list of declared functions that can be used, and C, $\mathcal{P}$, $\mathcal{Q}$ are the program to be repaired, and its corresponding pre- and postcondition.
Output: None if the Hoare triple $\{\mathcal{P}\}$ C $\{\mathcal{Q}\}$ holds, $\bar{\mathtt{C}}$ if C is buggy and $\bar{\mathtt{C}}$ is the repaired solution, or Fail if C is buggy but cannot be repaired.

```
1:  if Verify($\mathcal{P}$, C, $\mathcal{Q}$) = Fail then                                    ▷ C is buggy
2:    VCs ← GetInvalidVCs($\mathcal{P}$, C, $\mathcal{Q}$)
3:    S ← LocalizeBuggyStmts(C, VCs)                          ▷ suspicious statements
4:    for each stmt in S do
5:      ($\mathtt{C}_{\mathtt{stmt}}$, $\mathtt{TP}_{\mathtt{stmt}}$, $\mathcal{P}_{\mathtt{tp}}$, $\mathcal{Q}_{\mathtt{tp}}$) ← CreateTemplatePatchedProg(C, stmt)
6:      VCs′ ← GetVCs($\mathtt{C}_{\mathtt{stmt}}$, $\mathcal{P}$, $\mathcal{Q}$)                             ▷ collect entailments
7:      $\mathcal{D}$ ← InferUnknPreds($\varnothing$, VCs′)                        ▷ specification inference
8:      if $\mathcal{D} \neq \varnothing$ then
9:        ($\overline{\mathcal{P}}_{\mathtt{tp}}$, $\overline{\mathcal{Q}}_{\mathtt{tp}}$) ← GetPredDefn($\mathcal{D}$)
10:       $V$ ← GetAvailableVars($\mathtt{C}_{\mathtt{stmt}}$)
11:       $\overline{\mathtt{TP}}_{\mathtt{stmt}}$ ← SynthesizeStmts($\mathit{\Gamma}$, $V$, $\overline{\mathcal{P}}_{\mathtt{tp}}$, $\overline{\mathcal{Q}}_{\mathtt{tp}}$)            ▷ deductive synthesis
12:       if $\overline{\mathtt{TP}}_{\mathtt{stmt}} \neq$ Fail then
13:         $\bar{\mathtt{C}}$ ← GetCandidateProg($\mathtt{C}_{\mathtt{stmt}}$, $\overline{\mathtt{TP}}_{\mathtt{stmt}}$)
14:         if Verify($\mathcal{P}$, $\bar{\mathtt{C}}$, $\mathcal{Q}$) = Valid then return $\bar{\mathtt{C}}$          ▷ discover a patch
15:   return Fail                                                 ▷ fail to repair
16: else return None                                              ▷ C is correct
```

Fig. 12: The Repair algorithm.

We claim that our program repair algorithm in Fig. 12 is sound. We formally state that soundness in the following Theorem 1.

Theorem 1 (Soundness). *Given a program* $\mathtt{C}$*, a precondition* $\mathcal{P}$*, and a postcondition* $\mathcal{Q}$*, if the Hoare triple* $\{\mathcal{P}\}\ \mathtt{C}\ \{\mathcal{Q}\}$ *does not holds, and the algorithm* Repair *returns a program* $\bar{\mathtt{C}}$*, then the Hoare triple* $\{\mathcal{P}\}\ \bar{\mathtt{C}}\ \{\mathcal{Q}\}$ *holds.*

Proof. In our repair algorithm Repair (Fig. 12), an input program $\mathtt{C}$ is buggy when $\mathsf{Verify}(\mathcal{P}, \mathtt{C}, \mathcal{Q}) = \mathsf{Fail}$ or the Hoare triple $\{\mathcal{P}\}\ \mathtt{C}\ \{\mathcal{Q}\}$ does not hold. Then, if a candidate program $\bar{\mathtt{C}}$ is produced (line 13), the algorithm Repair always verifies $\bar{\mathtt{C}}$ w.r.t. the precondition $\mathcal{P}$ and the postcondition $\mathcal{Q}$ before returning $\bar{\mathtt{C}}$ (line 14). Hence, if $\{\mathcal{P}\}\ \mathtt{C}\ \{\mathcal{Q}\}$ does not hold and the algorithm Repair returns a program $\bar{\mathtt{C}}$, the Hoare triple $\{\mathcal{P}\}\ \bar{\mathtt{C}}\ \{\mathcal{Q}\}$ holds. □

7 Evaluation

We implemented our prototype tool, called NEM, on top of the HIP/SLEEK verification framework [6,7,38]. The specification inference in Sec. 4 was implemented on top of the Songbird prover [47,48]. Because our approach currently does not synthesize conditional statements, we apply mutation operators, e.g., changing from `x->next != NULL` to `x->prev != NULL`, to repair buggy conditional expressions of the conditional statements. We conducted experiments on a computer with CPU Intel® Core™ i7-6700 (3.4GHz), 8GB RAM, and Ubuntu 16.04 LTS. The details of our tool NEM and experiments are available online at `https://nem-repair-tool.github.io/`.

Table 1: Evaluation of NEM and a mutation-based tool [28] on repairing buggy heap-manipulating programs. Programs denoted with * are from [50].

Program	#LoC	#Buggy	NEM		Mutation-Based Tool [28]	
			#Repaired	Avg.Time (s)	#Repaired	Avg.Time (s)
sll-length	12	11	**11**	3.31	0	-
sll-copy	13	9	**9**	5.4	0	-
sll-append	11	18	**18**	5.84	1	2.13
sll-insert*	11	11	**11**	4.86	1	2.1
sll-delete*	13	17	**17**	5.32	0	-
dll-length	12	12	**12**	3.7	0	-
dll-append	16	20	**16**	10.12	1	5.36
dll-insert	12	11	**11**	4.58	1	2.11
dll-delete	20	20	**20**	15.72	0	-
srtll-insert*	19	20	**20**	18.35	2	6.13
tree-size	13	16	**16**	9.46	0	-
tree-height	16	20	**20**	13.67	0	-
avl-size	17	16	**16**	32.63	0	-
bst-size	13	16	**16**	11.46	0	-
bst-height	16	14	**10**	40.54	0	-
Summary		231	**223**	12.59	6	3.99

To evaluate our repair approach, we first selected a list of heap-manipulating programs written in a C-like language that was formally defined in [7]. These programs include algorithms of various data structures, such as singly-linked list (sll), doubly-linked list (dll), sorted linked list (srtll), binary tree (tree), binary search tree (bst), and AVL tree (avl). They include popular algorithms like *insert*, *append*, *delete*, *copy* for linked-lists. Some of these programs are taken from the benchmark used in [50] that are annotated accordingly. Note that our programs are tail-recursive while these in [50] use *while* loops.

To demonstrate the effectiveness of our method, we compared our tool with a state-of-the-art repair tool [28]. This tool uses genetic programming operators, e.g., *mutate*, *delete*, *insert*, to generate candidate programs, and then verifies these programs using a verifier. Both program repair tools verify and repair programs according to their provided specifications. Moreover, these two tools also use the HIP/SLEEK verifier to verify input programs and validate candidate programs. Each tool is configured to repair a buggy program within a timeout of 300 seconds. Regarding our tool NEM, we set both the timeouts of specification inference (Sec. 4) and deductive synthesis (Sec. 5) to 20 seconds. Besides, we did not include the semi-automated program repair tool Wolverine [50] since we could not obtain the implementation.

We followed a previous approach [50] in building a bug injection tool. This tool modifies program statements of a verified program to introduce errors at various program locations. Our tool modifies directly on the input program and generates readable buggy versions that are close to the input program. In contrast, the bug injection tool in Wolverine creates bugs in the intermediate representation code, consequently not enabling users to compare the buggy versions with the original correct program. Our bug injection tool currently modifies *one* statement or *two* statements in different branches of conditional statements. We also limit the maximum buggy versions of each program to 20.

Table 1 shows the results of running our prototype NEM and the mutation-based tool on the chosen benchmark. For each program, # LoC is the number of lines of code while # Buggy is the number of buggy versions created by the bug injection tool. Regarding the last 4 columns, we compared two evaluated tools in the number of buggy versions repaired, and the average time one tool needs to fix a buggy version. The results show that our tool outperforms the mutation-based one in the number of buggy programs repaired. Concretely, NEM was able to repair 223 buggy versions out of a total of 231 cases while the mutation-based tool only generated 6 correct patches. It is also noteworthy that NEM could handle buggy cases in all 15 programs. On the other hand, the mutation-based tool only gave correct patches for 5/15 programs. This is because our approach synthesizes patch candidates based on constraints collected from program semantics. Meanwhile, the mutation-based approach only uses a list of genetic programming operators, such as *mutate*, *delete*, *insert*, which limits the pattern of candidate patches.

Regarding the running time, our tool NEM needed 12.59 seconds on average to generate one correct patch while the mutation-based approach requires 3.99 seconds on average. This is because our approach of using specification inference and deductive synthesis is more expensive than mutation operators. However, it is much more effective when it can repair substantially more buggy programs.

There are 8 buggy versions of the 2 programs dll-append and bst-height that NEM could not repair. This is because the constraints collected when repairing these programs are highly complicated that they could not be solved by the current proof search heuristics of our constraint solver. This is a limitation of our work and we aim to resolve it in the future.

8 Related Work

Similar to our approach, a mutation-based approach [28] also aims to enhance APR with deductive verification. Technically, this method requires an iteration of (*i*) generating a patch by employing code mutation operators via GenProg [53] and (*ii*) verifying the patched program via HIP/SLEEK [7]. In contrast, we use specification inference and deductive synthesis to generate program statements of a patch. Hence, our approach is more effective than the mutation-based approach [28] in repairing buggy heap-manipulating programs as shown in Sec. 7.

The initial approaches that repair buggy heap-manipulating programs [11,12] rely on first-order logic formulae for specifications. However, these approaches are limited to detecting and fixing violations of user-provided data structure invariants. Meanwhile, our approach handles a broader class of errors, manifested as violations of specifications for arbitrary programs. The work [13] uses an input program and its specifications to generate constraints, which are then solved using the Alloy solver [20]. The obtained solutions are then translated into a repaired program. However, the repair procedure in [13] is restricted to a specific number of templates. In contrast, our approach uses a fully-fledged deductive synthesis framework, thus, allowing for fixing a larger class of bugs.

Recently, the tool FootPatch [49] relies on the Infer analysis tool [2,3] to detect errors and emit fixes for them. While also grounded in separation logic, this approach is less general than ours, as it only considers a fixed number of classes of bugs. However, it is more scalable thanks to the ability of Infer to detect unsafe memory accesses in large codebases without any user input. Likewise, Logozzo and Ball [31] use abstract interpretation [9] as a way to detect and fix mistakes in programs, but only for a limited number of issues that are captured by the employed analyzer. Similar to our approach, Maple [39] uses program specifications to detect bugs and validate candidate patches in numerical programs.

Our idea of generating correct-by-construction patches is similar to synthesizing programs from Hoare-style specifications [8,40,42]. However, it is applied in the context of program repair where the minimum number of statements is synthesized, leading to patches that are close to the original programs. Similar to our approach, deductive program repair [22] fixes buggy *functional* programs using the specifications from both symbolically executed tests and pre/postconditions and verifying the resulting program using the Leon tool [23].

Traditional APR approaches rely on *test suites* in checking the correctness of programs. Two main approaches of test-based APR are heuristic repair and constraint-based repair [30]. The *heuristic repair* identifies the bugs and the patches in the programs employing the insights that similar code patterns might be observed in sufficiently large codebases [16,43] while the *constraint-based* repair uses the provided test suite to infer symbolic constraints, and then solves those constraints to generate a patch [32,35,36,37,54]. Test-based APR approaches have been previously applied for fixing bugs in programs with pointers. For instance, a recent approach [50] involves a programmer in the debug-repair process to define correct program states at run-time. In contrast, our approach repairs buggy programs without the involvement of programmers.

9 Limitation and Future Work

We now discuss the limitations of our current approach and our plans to address them. Firstly, our approach mainly focuses on fixing program statements. Consequently, it may remove correct expressions, e.g., the left-hand side of an assignment or a parameter of a function call. This is because it aims to ease the bug localization step in limiting the number of suspicious statements as the number of suspicious expressions would be considerably larger than that of suspicious statements. In the future, we plan to add expression-level program repair by expressing correct expressions as *holes* in program sketches as in ImpSynt [42]. This method could also enhance our approach in repairing multiple locations as our approach currently only repairs *one* buggy statement or *two* buggy statements in different branches of a conditional statement.

Secondly, our approach does not handle omission errors. The debug-and-repair approach [50] can repair these cases by adding program states at various program points. Likewise, we can improve the capabilities of our repair method by inserting template patches at various program locations, and then synthesizing program statements of these patches. Thirdly, our approach could not repair buggy programs using the structure *list segment* in their specifications, e.g., the program schedule3 in the benchmark of [28]. Our early inspections indicate that these cases need *lemmas* in synthesizing program statements. Therefore, we aim to incorporate *lemma synthesis*, e.g., [48], to our repair framework. Fourthly, the buggy programs in our benchmark (Sec. 7) are produced using a bug injection tool. Therefore, we plan to evaluate our approach on student submissions in programming courses, similar to previous approaches [14,18,45,51].

Furthermore, we aim to ease the requirement of providing program specifications from users. To do that, we could either leverage static analyzers that do not require program specifications, e.g., Infer [2,3], or incorporate specification inference techniques, such as [24,27], to automatically infer program specifications. Finally, as discussed in Sec. 7, we plan to improve our constraint-solving technique to handle buggy programs that our approach currently fails to repair.

10 Conclusion

We have proposed a novel approach to fix buggy heap-manipulating programs. If a program is found buggy, we first encode program statements to fix this program in a template patch. Then, the specifications of the template patch are inferred using a constraint solving technique. Finally, from the inferred specifications, we use deductive synthesis to synthesize program statements of the template patch. The experimental results showed that our approach substantially outperformed a mutation-based approach in repairing buggy heap-manipulating programs.

Acknowledgments. We are grateful to the anonymous reviewers for their valuable and detailed comments. The first author would like to thank Cristina David for encouraging discussions, and Xuan-Bach D. Le for sharing the implementation of [28]. Thanh-Toan Nguyen and Wei-Ngan Chin are supported by the Singapore NRF grant R-252-007-A50-281. Ilya Sergey's work has been supported by the grant of Singapore NRF National Satellite of Excellence in Trustworthy Software Systems (NSoE-TSS) and by Crystal Centre at NUS School of Computing. Quang-Trung Ta is supported by the grant of the Singapore NRF under its Emerging Areas Research Projects (EARP) Funding Initiative in Singapore Blockchain Innovation Programme (SBIP).

References

1. James Brotherston and Alex Simpson. Sequent calculi for induction and infinite descent. *Journal of Logic and Computation*, 21(6):1177–1216, 2011.
2. Cristiano Calcagno and Dino Distefano. Infer: An automatic program verifier for memory safety of C programs. In *NASA International Symposium on Formal Methods (NFM)*, pages 459–465, 2011.
3. Cristiano Calcagno, Dino Distefano, Jérémy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter W. O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. Moving fast with software verification. In *NASA International Symposium on Formal Methods (NFM)*, pages 3–11, 2015.
4. Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. *Journal of the ACM*, 58(6):26:1–26:66, 2011.
5. Arthur Charguéraud. Separation logic for sequential programs (functional pearl). In *International Conference on Functional Programming (ICFP)*, pages 116:1–116:34, 2020.
6. Wei-Ngan Chin, Cristina David, Huu Hai Nguyen, and Shengchao Qin. Enhancing modular OO verification with separation logic. In *Symposium on Principles of Programming Languages (POPL)*, pages 87–99, 2008.
7. Wei-Ngan Chin, Cristina David, Huu Hai Nguyen, and Shengchao Qin. Automated verification of shape, size and bag properties via user-defined predicates in Separation Logic. *Science of Computer Programming (SCP)*, 77(9):1006–1036, 2012.
8. Andreea Costea, Amy Zhu, Nadia Polikarpova, and Ilya Sergey. Concise read-only specifications for better synthesis of programs with pointers. In *European Symposium on Programming (ESOP)*, pages 141–168, 2020.
9. Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Symposium on Principles of Programming Languages (POPL)*, pages 238–252, 1977.
10. Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, pages 337–340, 2008.
11. Brian Demsky and Martin C. Rinard. Automatic detection and repair of errors in data structures. In *International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*, pages 78–95, 2003.

12. Brian Demsky and Martin C. Rinard. Data structure repair using goal-directed reasoning. In *International Conference on Software Engineering (ICSE)*, pages 176–185, 2005.
13. Divya Gopinath, Muhammad Zubair Malik, and Sarfraz Khurshid. Specification-based program repair using SAT. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, pages 173–188, 2011.
14. Sumit Gulwani, Ivan Radicek, and Florian Zuleger. Automated clustering and program repair for introductory programming assignments. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 465–480, 2018.
15. Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish K. Shevade. Deepfix: Fixing common C language errors by deep learning. In *AAAI Conference on Artificial Intelligence (AAAI)*, pages 1345–1351, 2017.
16. Mark Harman. Automated patching techniques: the fix is in: technical perspective. *Communications of the ACM*, 53(5):108, 2010.
17. Seongjoon Hong, Junhee Lee, Jeongsoo Lee, and Hakjoo Oh. Saver: Scalable, precise, and safe memory-error repair. In *International Conference on Software Engineering (ICSE)*, 2020.
18. Yang Hu, Umair Z. Ahmed, Sergey Mechtaev, Ben Leong, and Abhik Roychoudhury. Re-factoring based program repair applied to programming assignments. In *International Conference on Automated Software Engineering (ASE)*, pages 388–398, 2019.
19. Samin S. Ishtiaq and Peter W. O'Hearn. BI as an assertion language for mutable data structures. In *Symposium on Principles of Programming Languages (POPL)*, pages 14–26, 2001.
20. Daniel Jackson and Mandana Vaziri. Finding bugs with a constraint solver. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 14–25, 2000.
21. Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. Automatic patch generation learned from human-written patches. In *International Conference on Software Engineering (ICSE)*, pages 802–811, 2013.
22. Etienne Kneuss, Manos Koukoutos, and Viktor Kuncak. Deductive program repair. In *International Conference on Computer Aided Verification (CAV)*, pages 217–233, 2015.
23. Etienne Kneuss, Ivan Kuraj, Viktor Kuncak, and Philippe Suter. Synthesis modulo recursive functions. In *International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*, pages 407–426, 2013.
24. Quang Loc Le, Cristian Gherghina, Shengchao Qin, and Wei-Ngan Chin. Shape analysis via second-order bi-abduction. In *International Conference on Computer Aided Verification (CAV)*, pages 52–68, 2014.
25. Ton Chanh Le, Cristian Gherghina, Aquinas Hobor, and Wei-Ngan Chin. A resource-based logic for termination and non-termination proofs. In *International Conference on Formal Engineering Methods (ICFEM)*, pages 267–283, 2014.
26. Ton Chanh Le, Shengchao Qin, and Wei-Ngan Chin. Termination and non-termination specification inference. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 489–498, 2015.
27. Ton Chanh Le, Guolong Zheng, and ThanhVu Nguyen. SLING: using dynamic analysis to infer program invariants in separation logic. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 788–801, 2019.
28. Xuan-Bach D. Le, Quang Loc Le, David Lo, and Claire Le Goues. Enhancing automated program repair with deductive verification. In *IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2016.

29. Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. Genprog: A generic method for automatic software repair. *IEEE Transactions on Software Engineering (TSE)*, 38(1):54–72, 2012.
30. Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. Automated Program Repair. *Communications of the ACM*, 2019.
31. Francesco Logozzo and Thomas Ball. Modular and verified automatic program repair. In *International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*, pages 133–146, 2012.
32. Fan Long and Martin Rinard. Staged program repair with condition synthesis. In *Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 166–178, 2015.
33. Fan Long and Martin Rinard. Automatic Patch Generation by Learning Correct Code. In *Symposium on Principles of Programming Languages (POPL)*, pages 298–312, 2016.
34. Sergey Mechtaev, Manh-Dung Nguyen, Yannic Noller, Lars Grunske, and Abhik Roychoudhury. Semantic program repair using a reference implementation. In *International Conference on Software Engineering (ICSE)*, pages 129–139, 2018.
35. Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. Directfix: Looking for simple program repairs. In *International Conference on Software Engineering (ICSE)*, pages 448–458, 2015.
36. Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. Angelix: scalable multiline program patch synthesis via symbolic analysis. In *International Conference on Software Engineering (ICSE)*, pages 691–701, 2016.
37. Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. Semfix: program repair via semantic analysis. In *International Conference on Software Engineering (ICSE)*, pages 772–781, 2013.
38. Huu Hai Nguyen and Wei-Ngan Chin. Enhancing Program Verification with Lemmas. In *International Conference on Computer Aided Verification (CAV)*, pages 355–369, 2008.
39. Thanh-Toan Nguyen, Quang-Trung Ta, and Wei-Ngan Chin. Automatic program repair using formal verification and expression templates. In *International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 70–91, 2019.
40. Nadia Polikarpova and Ilya Sergey. Structuring the synthesis of heap-manipulating programs. In *Symposium on Principles of Programming Languages (POPL)*, pages 72:1–72:30, 2019.
41. Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. The strength of random search on automated program repair. In *International Conference on Software Engineering (ICSE)*, pages 254–265, 2014.
42. Xiaokang Qiu and Armando Solar-Lezama. Natural synthesis of provably-correct data-structure manipulations. In *International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*, pages 65:1–65:28, 2017.
43. Baishakhi Ray, Vincent Hellendoorn, Saheel Godhane, Zhaopeng Tu, Alberto Bacchelli, and Premkumar T. Devanbu. On the "naturalness" of buggy code. In *International Conference on Software Engineering (ICSE)*, pages 428–439, 2016.
44. John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Symposium on Logic in Computer Science (LICS)*, pages 55–74, 2002.
45. Georgios Sakkas, Madeline Endres, Benjamin Cosman, Westley Weimer, and Ranjit Jhala. Type error feedback via analytic program repair. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 16–30, 2020.

46. Stelios Sidiroglou-Douskos, Eric Lahtinen, Fan Long, and Martin Rinard. Automatic error elimination by horizontal code transfer across multiple applications. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 43–54, 2015.
47. Quang-Trung Ta, Ton Chanh Le, Siau-Cheng Khoo, and Wei-Ngan Chin. Automated mutual explicit induction proof in separation logic. In *International Symposium on Formal Methods (FM)*, pages 659–676, 2016.
48. Quang-Trung Ta, Ton Chanh Le, Siau-Cheng Khoo, and Wei-Ngan Chin. Automated lemma synthesis in symbolic-heap separation logic. In *Symposium on Principles of Programming Languages (POPL)*, pages 9:1–9:29, 2018.
49. Rijnard van Tonder and Claire Le Goues. Static automated program repair for heap properties. In *International Conference on Software Engineering (ICSE)*, pages 151–162, 2018.
50. Sahil Verma and Subhajit Roy. Synergistic debug-repair of heap manipulations. In *Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 163–173, 2017.
51. Ke Wang, Rishabh Singh, and Zhendong Su. Search, align, and repair: data-driven feedback generation for introductory programming exercises. In *Conference on Programming Language Design and Implementation (PLDI)*, pages 481–495, 2018.
52. Westley Weimer, Zachary P. Fry, and Stephanie Forrest. Leveraging program equivalence for adaptive program repair: Models and first results. In *International Conference on Automated Software Engineering (ASE)*, pages 356–366, 2013.
53. Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. Automatically finding patches using genetic programming. In *International Conference on Software Engineering (ICSE)*, pages 364–374, 2009.
54. Jifeng Xuan, Matias Martinez, Favio Demarco, Maxime Clement, Sebastian R. Lamelas Marcote, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. Nopol: Automatic repair of conditional statement bugs in Java programs. *IEEE Transactions on Software Engineering (TSE)*, 43(1):34–55, 2017.