Automated Specification Inference in a Combined Domain via User-Defined Predicates

Shengchao Qin[a,e], Guanhua He[b], Wei-Ngan Chin[c], Florin Craciun[d], Mengda He[a], Zhong Ming[e]

[a]*School of Computing, Teesside University, Tees Valley, TS1 3BX, UK*
[b]*Postal Savings Bank of China, Beijing, China*
[c]*School of Computing, National University of Singapore, Singapore*
[d]*Department of Computer Science, Babes-Bolyai University, Cluj, Romania*
[e]*College of Computer Science and Software Engineering, Shenzhen University, China*

Abstract

Discovering program specifications automatically for heap-manipulating programs is a challenging task due to the complexity of aliasing and mutability of data structures. This task is further complicated by an expressive domain that combines shape, numerical and bag information. In this paper, we propose a compositional analysis framework that would derive the summary for each method in the expressive abstract domain, independently from its callers. We propose a novel abstraction method with a bi-abduction technique in the combined domain to discover pre-/post-conditions that could not be automatically inferred before. The analysis does not only infer memory safety properties, but also finds relationships between pure and shape domains towards full functional correctness of programs. A prototype of the framework has been implemented and initial experiments have shown that our approach can discover interesting properties for non-trivial programs.

1. Introduction

In automated program analysis, certain kinds of program properties have been well explored over the last decades, such as numerical properties in the linear abstraction domain, and shape properties for list-manipulating programs in the separation domain. However, previous works have not yet automatically analysed program properties involving complex mixed domains, particularly for programs with sophisticated data structures and strong invariants involving both structural and pure (numerical and content) information. For example, it is still non-trivial to discover program properties, such as a list becoming sorted during the execution of a program, a binary search tree remaining balanced before and after the execution of a procedure, or the elements of a list remain unchanged after reversing the list. This difficulty is not only due to sharing and mutability of data structures under manipulation, but is also due to closely intertwined program properties, such as structural numerical information (length and height), symbolic contents of data structures (bag of values), and relational numerical information (sortedness and balancedness).

In addition to classical shape analyses (e.g. [5, 22, 39]), separation logic [36] has been applied to analyse shape properties in recent years [7, 13, 42]. These works can automatically infer method specifications in the shape domain. Some other works such as [28, 29] also incorporate simple numerical information into the shape domain to allow automated synthesis of properties like data structure size information.

However, these previous analyses mainly deal with predesignated data structure properties with fixed numerical templates, such as pointer safety for lists and list length information. To analyse a wider range of properties of heap-manipulating programs with flexibility, our previous work [34] offers a template-based approach, whereby users supply shape templates in pre-/post-conditions of procedures, and then the analysis

Email addresses: `s.qin@tees.ac.uk` (Shengchao Qin), `m.he@tees.ac.uk` (Mengda He), `mingz@szu.edu.cn` (Zhong Ming)

infers the missing pure information to complete the given templates. While that approach simplifies fix-point computation to a single domain, one serious limitation is that it relies on users to supply the pre-/post-shape templates. If the supplied templates do not cover all the required heap portions, or are not precise enough, it could fail to discover suitable specifications.

To overcome this limitation, we propose in this paper a compositional program analysis in a combined abstract domain with *shape*, *numerical* and *bag* (i.e. multi-set) information. Our analysis not only handles functional correctness and memory safety together, but can also discover relationships between shape and pure (numerical and bag) domains. Unlike traditional approaches [29] which usually analyse the shape first before turning to pure properties, our approach analyses programs over both domains at the same time. This is very necessary as verifying functional correctness for certain programs may require us to consider both shape and pure information at the same time. Without pure information, a shape analysis may not be able to find useful program specifications (an example is the `merge` procedure discussed in [7]). Our approach can handle this kind of programs smoothly, and we will illustrate our method using the `merge` example (and another example) in Section 2.

Our analysis is compositional. It analyses a program fragment without any given contextual information, and it analyses each method in a modular way independent of its callers. To generate the summary (pre-/post-conditions) for each method, our analysis adopts a new bi-abduction mechanism over the combined domain, which generalises the bi-abduction technique proposed by Calcagno et al. [7] to a more expressive abstract domain. In summary, this paper[1] makes the following contributions:

- We have designed a compositional analysis to discover *program specifications* (in the form of pre-/post-conditions involving shape, numerical and bag properties) guided by user-given data structure predicates.
- For such an analysis, we have designed a *bi-abductive abstract semantics* which incorporates a generalised bi-abduction procedure to facilitate specification discovery over the combined abstract domain.
- In addition to a normal abstraction function, we have also proposed a novel *abductive abstraction* function over the combined domain. This new abstraction function allows us to find stronger method specifications that are often necessary for the successful verification for higher functional correctness.
- We have built a prototype system and conducted some initial experiments, which help confirm the viability and precision of our solution in inferring non-trivial program specifications.

Outline. In what follows, we illustrate our approach informally via two examples (Sec. 2), and then give our programming and specification languages (Sec. 3) as well as our bi-abduction mechanism (Sec. 3.1). Formal details about specification discovery are presented in Sec. 4, followed by experimental results in Sec. 5. Finally, related work and conclusion are given in Sec. 6.

2. The Approach

This section first introduces the preliminaries of our analysis, followed by illustrative examples of our analysis.

2.1. Preliminaries

Separation Logic. Separation logic [21, 36] extends Hoare logic to support reasoning about shared mutable data structures. Separation conjunction $*$ is a new connective introduced in separation logic. A separation logic formula $\mathtt{p}_1 * \mathtt{p}_2$ asserts that two heaps described by $\mathtt{p}_1$ and $\mathtt{p}_2$ are domain-disjoint. The formula `emp` denotes an empty heap. While a formula of the form $\mathtt{x}{\mapsto}\mathtt{y}$ represents a singleton heap referred to by `x`.

[1]which extends an earlier conference paper [19].

User-defined Predicates. In our analysis, users are allowed to define inductive predicates in separation logic to specify both separation and pure properties of recursive data structures. For example, given a data structure `data Node { int val; Node next; }`, one can define a predicate for a list with its content as

$$\begin{array}{l}\mathtt{llB(root,n,S)} \equiv (\mathtt{root{=}null} \wedge \mathtt{n{=}0} \wedge \mathtt{S{=}\emptyset}) \vee \\ \quad (\exists \mathtt{v,q,n_1,S_1} \cdot \mathtt{root{\mapsto}Node(v,q){*}llB(q,n_1,S_1)} \wedge \mathtt{n_1{=}n{-}1} \wedge \mathtt{S{=}S_1{\sqcup}\{v\}})\end{array}$$

The parameter `root` for the predicate `llB` is the root pointer referring to the list. The length and content of the list are denoted resp. by `n` and the bag `S`, and $\sqcup$ indicates multi-set (bag) union.

If one wants to verify a sorting algorithm, they can specify a non-empty sorted list as follows:

$$\begin{array}{l}\mathtt{sllB(root,mi,mx,S)} \equiv (\mathtt{root{\mapsto}Node(mi,null)} \wedge \mathtt{mi{=}mx} \wedge \mathtt{S{=}\{mi\}}) \vee \\ \quad (\mathtt{root{\mapsto}Node(v,q){*}sllB(q,m_1,mx,S_1)} \wedge \mathtt{v{=}mi} \wedge \mathtt{v{\leq}m_1} \wedge \mathtt{m_1{\leq}mx} \wedge \mathtt{S{=}S_1{\sqcup}\{v\}})\end{array}$$

where it keeps track of the minimum (`mi`) and maximum (`mx`) values in the list as well as the bag of all values (`S`). Note that we use a shortened notation that unbound variables, such as `q`, `v`, $\mathtt{m_1}$ and $\mathtt{S_1}$, are implicitly existentially quantified.

Such predicates play an important role in our analysis as (i) they are used to help specify desired properties about data structures under manipulation, and (ii) they serve as a guide for our analysis to discover desired program specifications. To reduce the burden of supplying such predicates, we have defined a library of predicates covering popular data structures and variety of properties.

Entailment. In our work we make use of the separation logic prover Sleek [11, 10] to prove whether one formula Δ' in the combined abstract domain entails another one Δ: $\Delta' \vdash \Delta * \mathtt{R}$. `R` is called the *frame* which is useful for our analysis. For instance, by the entailment proof

$$\exists \mathtt{y} \cdot \mathtt{x{\mapsto}Node(vx,y){*}llB(y,n,S)} \vdash \mathtt{llB(x,m,S_1){*}R}$$

We can generate the frame `R` as $\mathtt{emp} * (\mathtt{m{=}n{+}1} \wedge \mathtt{S_1{=}S{\sqcup}\{vx\}})$, which can be represented as $\mathtt{m{=}n{+}1} \wedge \mathtt{S_1{=}S{\sqcup}\{vx\}}$ since $\mathtt{emp{*}p} \equiv \mathtt{p}$ holds for all separation logic formula `p`. We follow classical separation logic, so a pure formula π in our logic implicitly denotes $\mathtt{emp} \wedge \pi$, which is equivalent to $\mathtt{emp} * \pi$.

Bi-Abduction. In an earlier work [7], a bi-abductive entailment is proposed for the *shape*-only domain: given two shape formulae `G, H`, the bi-abduction $\mathtt{G} * [\mathtt{A}] \rhd \mathtt{H} * [\mathtt{F}]$ infers the *anti-frame* `A` and the *frame* `F` along the entailment proof. An example taken from [7] is

$$\mathtt{x{\mapsto}null{*}z{\mapsto}null}{*}[\underline{\mathtt{list(y)}}] \rhd \mathtt{list(x){*}list(y)} * [\underline{\mathtt{z{\mapsto}null}}]$$

where the $\mathtt{list}(\cdot)$ predicate describes acyclic, `null`-terminated singly-linked lists. In the current work, we will generalise such bi-abductive reasoning to the combined abstract domain involving shape, user-defined predicates, numerical and bag information. A simple example of the generalised bi-abductive reasoning is

$$\exists \mathtt{y} \cdot \mathtt{x{\mapsto}Node(vx,y){*}y{\mapsto}Node(vy,null)}{*}[\underline{\mathtt{A}}] \rhd \mathtt{sllB(x,mi,mx,S)}{*}[\underline{\mathtt{F}}]$$

where $\underline{\mathtt{A}} \equiv (\mathtt{vx{\leq}vy})$ and $\underline{\mathtt{F}} \equiv (\mathtt{mi{=}vx} \wedge \mathtt{mx{=}vy} \wedge \mathtt{S{=}\{vx,vy\}})$.

2.2. Illustrative Examples

2.2.1. Example of Filter.

Firstly, we illustrate our analysis approach via a simple example called `filter` (Fig. 1), which selects elements from a list that satisfy a certain condition ($\leq$ `k`). The example is based on the data structure `Node`, and the shape predicate is `llB` as we defined earlier.

Our analysis aims at finding a sound and precise specification (summary) (Pre, Post) of the method. Before the analysis, we use a pair $(\mathsf{Pre}_0, \mathsf{Post}_0) := (\mathtt{emp}, \mathtt{false})$ as an initial pre-/post-condition of the method, which means we have no knowledge about the program's requirement or effect yet. During the analysis, we use a pair (infP, Curr) to keep trace of the precondition we have discovered and the current state

```
1  Node filter(Node x, int k)
2  {
3    if (x == null) {
4      return x;
5    } else if (x.val <= k) {
6      Node t = x.next;
7      x.next = filter(t, k);
8      return x;
9    } else {
10     Node t = x.next;
11     free(x);
12     x = filter(t, k);
13     return x;
14 } }
```

Figure 1: Filtering the elements of a list.

we have reached, respectively. If the current precondition is not sufficient to operate the program command, we use a bi-abductive inference mechanism (described later in Sec 3.1) to synthesise a candidate precondition as the missing precondition. At the beginning of the analysis, this pair is set to $(\mathsf{infP},\mathsf{Curr}){:=}(\texttt{emp},\ \texttt{emp})$. We iterate the method body by symbolic execution for a number of passes until the pre-/post-condition reaches a fixed point. To ensure convergence, we have designed operations of abstraction, join and widening over both shape and pure domains to achieve the fixed point.

For the example `filter`, the analysis starts with $(\mathsf{infP},\mathsf{Curr}){:=}(\texttt{emp}{\wedge}\texttt{x}{=}\texttt{x}'{\wedge}\texttt{k}{=}\texttt{k}',\ \texttt{emp})$ before line 2 in the first iteration, where we use primed variables to keep track of the current values of program variables, and unprimed variables for the initial values in the precondition. (Since the value of `k` is not changed during the program execution, we omit $\texttt{k}{=}\texttt{k}'$ in the presentation.) The branch from line 5 to line 13 is "short-circuited" in the first iteration, as the current postcondition of the method ($\mathsf{Post}_0{\equiv}\texttt{false}$) is applied as the effect of the recursive call. To enter line 4, the condition `x == null` needs to be satisfied by the precondition. We apply the bi-abduction mechanism and discover $\texttt{x}{=}\texttt{null}$, which will then be added into the precondition. After executing `return x`, we have an initial summary of the method:

$$(\mathsf{Pre}_1, \mathsf{Post}_1) := (\texttt{emp}{\wedge}\texttt{x}{=}\texttt{null}{\wedge}\texttt{x}{=}\texttt{x}',\ \texttt{emp}{\wedge}\texttt{res}{=}\texttt{null}{\wedge}\texttt{res}{=}\texttt{x}'{\wedge}\texttt{x}{=}\texttt{x}'), \tag{1}$$

where `res` denotes the value returned by the program.

When we start the second iteration, the specification (1) is assumed as the new summary of the method. By executing the branch in line 4, we have the same result as summary (1). In Line 5, the expression `x.val` tries to access a field of `x`, by abduction, $\texttt{x}{\mapsto}\texttt{Node}(\texttt{fv}_0,\texttt{fp}_0)$ is added to infP, where $\texttt{fv}_0$ and $\texttt{fp}_0$ are fresh logical variables. After line 6, the paired state $(\mathsf{infP},\mathsf{Curr})$ is $(\texttt{x}{\mapsto}\texttt{Node}(\texttt{fv}_0,\texttt{fp}_0){\wedge}\texttt{fv}_0{\leq}\texttt{k}{\wedge}\texttt{x}{=}\texttt{x}',$ $\texttt{x}{\mapsto}\texttt{Node}(\texttt{fv}_0,\texttt{fp}_0){\wedge}\texttt{t}{=}\texttt{fp}_0{\wedge}\texttt{fv}_0{\leq}\texttt{k}{\wedge}\texttt{x}{=}\texttt{x}')$. Now we can use the summary (1) for the method call, which requires $\texttt{t}{=}\texttt{null}$, i.e. $\texttt{fp}_0{=}\texttt{null}$ to be added to infP and returns $\texttt{t}{=}\texttt{fp}_0$ to be added to Curr. Note that $\texttt{x}{\mapsto}\texttt{Node}(\texttt{fv}_0,\texttt{fp}_0)$ as the frame part is discovered by bi-abduction. The frame part is not altered by the method call and passed to the post-state of this call. As $\texttt{fp}_0$ is a reachable variable from program variable `x`, we add $\texttt{fp}_0{=}\texttt{null}$ to infP, instead of $\texttt{t}{=}\texttt{null}$. After line 8, the summary of the branch from lines 5 to 8 is found:

$$(\texttt{x}{\mapsto}\texttt{Node}(\texttt{fv}_0,\texttt{null}){\wedge}\texttt{fv}_0{\leq}\texttt{k}{\wedge}\texttt{x}{=}\texttt{x}',\ \texttt{res}{\mapsto}\texttt{Node}(\texttt{fv}_0,\texttt{null}){\wedge}\texttt{fv}_0{\leq}\texttt{k}{\wedge}\texttt{res}{=}\texttt{x}'{\wedge}\texttt{x}{=}\texttt{x}') \tag{2}$$

Similarly, the summary of line 9 to 14 is calculated as

$$(\texttt{x}{\mapsto}\texttt{Node}(\texttt{fv}_0,\texttt{null}){\wedge}\texttt{fv}_0{>}\texttt{k}{\wedge}\texttt{x}{=}\texttt{x}',\ \texttt{emp}{\wedge}\texttt{fv}_0{>}\texttt{k}{\wedge}\texttt{res}{=}\texttt{null}{\wedge}\texttt{x}'{=}\texttt{null}) \tag{3}$$

By joining pre-formulae and post-formulae in (1), (2) and (3) respectively, and eliminating intermediate logical variables, we obtain a new summary for the method

$$\begin{aligned}&(\mathsf{Pre}_2, \mathsf{Post}_2) := (\mathsf{Pre}_1 \vee \mathtt{x}{\mapsto}\mathtt{Node}(\mathtt{fv}_0, \mathtt{null}) \wedge \mathtt{x}{=}\mathtt{x}',\\ &\mathsf{Post}_1 \vee \mathtt{emp} \wedge \mathtt{fv}_0{>}\mathtt{k} \wedge \mathtt{res}{=}\mathtt{null} \wedge \mathtt{x}'{=}\mathtt{null} \vee \mathtt{res}{\mapsto}\mathtt{Node}(\mathtt{fv}_0, \mathtt{null}) \wedge \mathtt{fv}_0{\leq}\mathtt{k} \wedge \mathtt{res}{=}\mathtt{x}')\end{aligned} \quad (4)$$

Based on specification (4), a third iteration of symbolic execution is accomplished, with abstract specification as

$$\begin{aligned}&(\mathsf{Pre}_3, \mathsf{Post}_3) := (\mathsf{Pre}_2 \vee \mathtt{x}{\mapsto}\mathtt{Node}(\mathtt{fv}_0, \mathtt{fp}_0) * \mathtt{fp}_0{\mapsto}\mathtt{Node}(\mathtt{fv}_1, \mathtt{null}) \wedge \mathtt{x}{=}\mathtt{x}',\\ &\quad \mathsf{Post}_2 \vee \mathtt{res}{\mapsto}\mathtt{Node}(\mathtt{fv}_0, \mathtt{null}) \wedge \mathtt{fv}_0{\leq}\mathtt{k} \wedge \mathtt{fv}_1{>}\mathtt{k} \wedge \mathtt{res}{=}\mathtt{x}'\\ &\quad \mathtt{res}{\mapsto}\mathtt{Node}(\mathtt{fv}_0, \mathtt{fp}_0) * \mathtt{fp}_0{\mapsto}\mathtt{Node}(\mathtt{fv}_1, \mathtt{null}) \wedge \mathtt{fv}_0{\leq}\mathtt{k} \wedge \mathtt{fv}_1{\leq}\mathtt{k} \wedge \mathtt{res}{=}\mathtt{x}')\end{aligned} \quad (5)$$

Comparing with summary (4), we discover it is possible that x points to a list with two nodes in the precondition. If we continue with this trend, we will get longer formulae to cover successive iterations and more additional logical variables. Following such trend, the analysis will be an infinite regress. Therefore, we first apply abstraction to the precondition with the help of the given predicate llB to eliminate the logical variables, so the heap part $\mathtt{x}{\mapsto}\mathtt{Node}(\mathtt{fv}_0, \mathtt{fp}_0) * \mathtt{fp}_0{\mapsto}\mathtt{Node}(\mathtt{fv}_1, \mathtt{null})$ is abstracted as $\mathtt{llB}(\mathtt{x}, \mathtt{n}_1, \mathtt{S}_1) \wedge \mathtt{n}_1{=}2 \wedge \mathtt{S}_1{=}\{\mathtt{fv}_0, \mathtt{fv}_1\}$. Before we join this with Pre_2, the heap formula $\mathtt{x}{\mapsto}\mathtt{Node}(\mathtt{fv}_0, \mathtt{null})$ in Pre_2 can be unified as $\mathtt{llB}(\mathtt{x}, \mathtt{n}_1, \mathtt{S}_1) \wedge \mathtt{n}_1{=}1 \wedge \mathtt{S}_1{=}\{\mathtt{fv}_0\}$ and $\mathtt{x}{=}\mathtt{null}$ be $\mathtt{llB}(\mathtt{x}, \mathtt{n}_1, \mathtt{S}_1) \wedge \mathtt{n}_1{=}0 \wedge \mathtt{S}_1{=}\emptyset$. Then we join the disjunctive formulae if they have the same shape and widening with the Pre_2 to have the precondition as $\mathtt{llB}(\mathtt{x}, \mathtt{n}_1, \mathtt{S}_1)$. By applying similar operators to postcondition, a new summary is produced:

$$\begin{aligned}&(\mathsf{Pre}_3, \mathsf{Post}_3) := (\mathtt{llB}(\mathtt{x}, \mathtt{n}_1, \mathtt{S}_1) \wedge 0{\leq}\mathtt{n}_1,\\ &\quad \mathtt{llB}(\mathtt{res}, \mathtt{n}_2, \mathtt{S}_2) \wedge 0{\leq}\mathtt{n}_2{\leq}\mathtt{n}_1 \wedge (\forall \mathtt{v}{\in}\mathtt{S}_2 \cdot \mathtt{v} \leq \mathtt{k}) \wedge (\forall \mathtt{v}{\in}(\mathtt{S}_1{-}\mathtt{S}_2) \cdot \mathtt{v}{>}\mathtt{k}) \wedge \mathtt{S}_2{\subseteq}\mathtt{S}_1)\end{aligned} \quad (6)$$

Following a similar symbolic execution process with the newly assumed method summary (6), we obtain a result for the fourth iteration to be the same as the last one, namely (6), which is thus a fixed point desired for our method summary. Note that we eliminate x′ as an existentially quantified variable in the postcondition, since x is a call-by-value parameter.

The essential steps to terminate the search for suitable preconditions are *abstraction* and *widening*. Both operators are tantamount to weakening a state, and they are over-approximation and sound for synthesis of postcondition. However, when such steps are applied to synthesis of precondition, it may make the precondition too weak to be sound. So after the analysis, similar to the post-analysis check in Abductor [7], we shall use a forward analysis process (in our case the HIP verifier [11]) to check the discovered summary.

2.2.2. Example of Merge.

Another motivating example we shall present is the merge method used in merge-sort, which has been mentioned as an unverifiable example in [7], since their analysis does not keep track of data values stored in the list during their shape analysis. The method (Fig. 2) merges two sorted lists into one sorted list, as illustrated by an instance in Fig. 3. If either of the input lists is empty, the other list is returned; if not, it picks up the node with the smallest element from both lists, and lets the next field of this node point to the result list obtained by merging the tail list of this node with the other list.

Automated specification discovery for merge is rather challenging due to two facts: (1) only one input list is fully traversed; (2) both input lists are required to be sorted. For (1), even if we apply the state-of-the-art shape abduction [7], we can only discover as precondition two disjoint lists - one ending with null and one ending with an unknown pointer. Such a precondition unfortunately cannot guarantee the memory safety of the merge method. For (2), if an analysis cannot infer that the two input lists are sorted, it will not be able to discover that the output list is sorted, which would not be sufficient for one to verify the functional correctness of the enclosing merge-sort method. The two input lists being unsorted also causes the unknown pointer problem mentioned above. Our proposed compositional analysis (in a combined shape and pure domain) will be able to overcome these difficulties, where program properties over the combined

```
Node merge(Node x, Node y)
{
  if (x == null) {
    return y;
  } else if (y == null) {
    return x;
  } else
  if (x.val <= y.val) {
    Node t = x.next;
    x.next = merge(t, y);
    return x;
  } else {
    Node t = y.next;
    y.next = merge(x, t);
    return y;
} }
```

Figure 2: Merging two sorted lists.

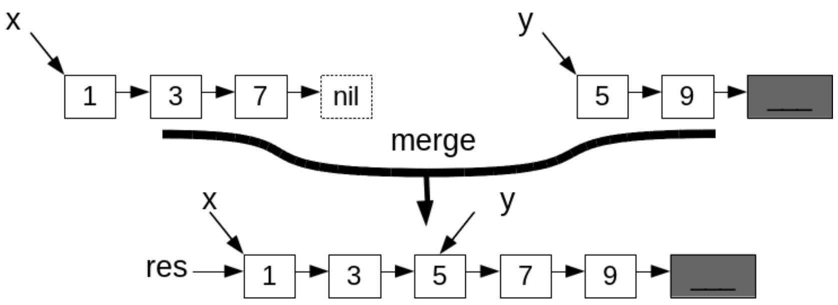

Figure 3: An Instance of Merge

domain are processed at the same time during the analysis. As shown in the previous example, our analysis adopts a novel bi-abduction mechanism to help discover program preconditions in the combined domain. To help infer the precondition that the two input lists are sorted, we introduce another novel mechanism called *abductive abstraction* to apply abductive reasoning during the abstraction process, allowing us to infer the required precondition for this example that previous analyses are unable to achieve.

For the `merge` example, the shape predicate selected for our analysis is `slsB` which keeps track of the minimal (`mi`) and maximal (`mx`) values, a bag of values (`S`) and the tail pointer (`p`) of a (non-empty) sorted list segment.

$$\texttt{slsB}(\texttt{root}, \texttt{mi}, \texttt{mx}, \texttt{S}, \texttt{p}) \equiv (\texttt{root}{\mapsto}\texttt{Node}(\texttt{mi}, \texttt{p}) \wedge \texttt{mi}{=}\texttt{mx} \wedge \texttt{S}{=}\{\texttt{mi}\}) \vee$$
$$(\texttt{root}{\mapsto}\texttt{Node}(\texttt{mi}, \texttt{q}) * \texttt{sllB}(\texttt{q}, \texttt{m}_1, \texttt{mx}, \texttt{S}_1, \texttt{p}) \wedge \texttt{mi}{\le}\texttt{m}_1 \wedge \texttt{m}_1{\le}\texttt{mx} \wedge \texttt{S}{=}\texttt{S}_1 {\sqcup} \{\texttt{mi}\})$$

Same as the previous example,we use a pair of states (infP, Curr) to keep track of the precondition that the analysis has discovered (infP) so far and the current state the execution has reached (Curr), respectively. If the current abstract state does not meet the precondition required by the current program command, we use an abductive inference mechanism (mentioned in the previous subsection) to synthesise a candidate precondition as the missing precondition.

For the `merge` example, the initial specification ($\mathsf{Pre}_0 \equiv \mathtt{emp}, \mathsf{Post}_0 \equiv \mathtt{false}$) allows the analysis to skip the branches with recursive calls to `merge`. The symbolic execution in the first fixpoint iteration starts from state ($\mathsf{infP} \equiv \mathtt{emp}, \mathsf{Curr} \equiv \mathtt{emp}$), since the analysis assumes no prior knowledge about the starting program state. To enter line 4, the condition `x==null` needs to be met by the current abstract state. We apply abduction and discover $\mathtt{x}{=}\mathtt{null}$ which is then added to the precondition. Similarly, we have $\mathtt{y}{=}\mathtt{null}$ from the second branch. After the first iteration, a summary is found as

$$(\mathsf{Pre}_1 \equiv (\mathtt{x}{=}\mathtt{null} \vee \mathtt{y}{=}\mathtt{null}),\ \mathsf{Post}_1 \equiv (\mathtt{x}{=}\mathtt{null} \wedge \mathtt{res}{=}\mathtt{y} \vee \mathtt{y}{=}\mathtt{null} \wedge \mathtt{res}{=}\mathtt{x})) \tag{7}$$

where `res` denotes the return value. Using this new summary for recursive calls to `merge`, symbolically executing the method body again (but with an updated starting state ($\mathsf{infP} \equiv \mathsf{Pre}_1, \mathsf{Curr} \equiv \mathsf{Pre}_1$) yields the summary ($\mathsf{Pre}_2$, Post_2):

$$\begin{array}{rcl} (\mathsf{Pre}_2 & \equiv & \mathtt{x}{=}\mathtt{null} \vee \mathtt{y}{=}\mathtt{null} \vee \mathtt{x} {\mapsto} \mathtt{Node}(\mathtt{xv}_1, \mathtt{xp}_1) {*} \mathtt{y} {\mapsto} \mathtt{Node}(\mathtt{yv}_1, \mathtt{yp}_1) \\ & & \quad \wedge (\mathtt{xv}_1 {\le} \mathtt{yv}_1 \wedge \mathtt{xp}_1 {=} \mathtt{null} \vee \mathtt{xv}_1 {>} \mathtt{yv}_1 \wedge \mathtt{yp}_1 {=} \mathtt{null}), \\ \mathsf{Post}_2 & \equiv & \mathtt{x}{=}\mathtt{null} \wedge \mathtt{res}{=}\mathtt{y} \vee \mathtt{y}{=}\mathtt{null} \wedge \mathtt{res}{=}\mathtt{x} \vee \mathtt{x} {\mapsto} \mathtt{Node}(\mathtt{xv}_1, \mathtt{xp}_1) {*} \mathtt{y} {\mapsto} \mathtt{Node}(\mathtt{yv}_1, \mathtt{yp}_1) \\ & & \quad \wedge (\mathtt{xv}_1 {\le} \mathtt{yv}_1 \wedge \mathtt{res}{=}\mathtt{x} \wedge \mathtt{xp}_1{=}\mathtt{y} \vee \mathtt{xv}_1 {>} \mathtt{yv}_1 \wedge \mathtt{res}{=}\mathtt{y} \wedge \mathtt{yp}_1{=}\mathtt{x})) \end{array} \tag{8}$$

After the third iteration of symbolic execution, we generate a precondition as:

$$\begin{array}{l} \mathtt{x}{=}\mathtt{null} \vee \mathtt{y}{=}\mathtt{null} \vee \mathtt{x} {\mapsto} \mathtt{Node}(\mathtt{xv}_1, \mathtt{xp}_1) {*} \mathtt{y} {\mapsto} \mathtt{Node}(\mathtt{yv}_1, \mathtt{yp}_1) \\ \quad \wedge\ (\mathtt{xv}_1 {\le} \mathtt{yv}_1 \wedge \mathtt{xp}_1{=}\mathtt{null} \vee \mathtt{xv}_1 {>} \mathtt{yv}_1 \wedge \mathtt{yp}_1{=}\mathtt{null}) \end{array} \tag{9}$$

$$\begin{array}{l} \vee \mathtt{x} {\mapsto} \mathtt{Node}(\mathtt{xv}_1, \mathtt{xp}_1) {*} \mathtt{xp}_1 {\mapsto} \mathtt{Node}(\mathtt{xv}_2, \mathtt{xp}_2) {*} \mathtt{y} {\mapsto} \mathtt{Node}(\mathtt{yv}_1, \mathtt{yp}_1) \\ \quad \wedge\ (\mathtt{xv}_1 {\le} \mathtt{yv}_1 \wedge (\mathtt{xv}_2 {\le} \mathtt{yv}_1 \wedge \mathtt{xp}_2{=}\mathtt{null} \vee \mathtt{xv}_2 {>} \mathtt{yv}_1 \wedge \mathtt{yp}_1{=}\mathtt{null})) \end{array} \tag{10}$$

$$\begin{array}{l} \vee \mathtt{x} {\mapsto} \mathtt{Node}(\mathtt{xv}_1, \mathtt{xp}_1) {*} \mathtt{y} {\mapsto} \mathtt{Node}(\mathtt{yv}_1, \mathtt{yp}_1) {*} \mathtt{yp}_1 {\mapsto} \mathtt{Node}(\mathtt{yv}_2, \mathtt{yp}_2) \\ \quad \wedge\ (\mathtt{xv}_1 {>} \mathtt{yv}_1 \wedge (\mathtt{xv}_1 {\le} \mathtt{yv}_2 \wedge \mathtt{xp}_1{=}\mathtt{null} \vee \mathtt{xv}_1 {>} \mathtt{yv}_2 \wedge \mathtt{yp}_2{=}\mathtt{null})) \end{array} \tag{11}$$

The branch (10) says that the program only touches the second node of the `x` list (the list referred to by `x`) if $\mathtt{xv}_1 {\le} \mathtt{yv}_1$. Furthermore, if $\mathtt{xv}_2 {\le} \mathtt{yv}_1$, $\mathtt{xp}_2$ should be `null`; otherwise $\mathtt{yp}_1$ must be `null` to guarantee the termination of the method and memory safety. The branch (11) states a similar condition when touching the second node of the `y` list. The information kept in this formula is very precise, but keeping such a level of details will not allow the analysis to scale up. According to the given predicate `slsB`, we could abstract the shape of the `x` list (and that of the `y` list) to be a sorted list segment. However, the formula itself does not contain sufficient information for us to carry out this abstraction, i.e. *the sortedness information* about the `x` list (and the `y` list) is missing. This missing information is the numerical relation between $\mathtt{xv}_1$ and $\mathtt{xv}_2$ in the `x` list (and that between $\mathtt{yv}_1$ and $\mathtt{yv}_2$ in the `y` list). In other words, we need to use bi-abduction to discover $\mathtt{xv}_1 {\le} \mathtt{xv}_2$ (resp. $\mathtt{yv}_1 {\le} \mathtt{yv}_2$) *during the abstraction* from the shape of the `x` list (resp. the `y` list) to a sorted list segment in the branch (10) (resp. (11)), e.g. for the `x` list:

$$\mathtt{x} {\mapsto} \mathtt{Node}(\mathtt{xv}_1, \mathtt{xp}_1) {*} \mathtt{xp}_1 {\mapsto} \mathtt{Node}(\mathtt{xv}_2, \mathtt{xp}_2) {*} [\underline{\mathtt{xv}_1 {\le} \mathtt{xv}_2}] \rhd \mathtt{slsB}(\mathtt{x}, \mathtt{xv}_1, \mathtt{xv}_2, \mathtt{xS}_1, \mathtt{xp}_2) {*} \mathtt{R} \tag{†}$$

This bi-abduction procedure (details delayed to Sec 3.1) infers the information $\mathtt{xv}_1 {\le} \mathtt{xv}_2$ so that the following entailment holds (signifying an abstraction from LHS to RHS):

$$\mathtt{x} {\mapsto} \mathtt{Node}(\mathtt{xv}_1, \mathtt{xp}_1) {*} \mathtt{xp}_1 {\mapsto} \mathtt{Node}(\mathtt{xv}_2, \mathtt{xp}_2) {*} [\underline{\mathtt{xv}_1 {\le} \mathtt{xv}_2}] \vdash \mathtt{slsB}(\mathtt{x}, \mathtt{xv}_1, \mathtt{xv}_2, \mathtt{xS}_1, \mathtt{xp}_2) {*} \mathtt{R}$$

The inspiration for this *abductive abstraction* (details to follow in Sec 4.2) comes from the definition of the predicate `slsB`. We use such predicates to help infer data structure properties that are anticipated from some program code. Note that a standard abstraction (required in the previous example) would only be able to obtain an abstraction of an ordinary list segment without any sortedness information.

By applying such an *abductive abstraction* against the predicate `slsB` and then joining the branches with the same shape, the precondition obtained after two iterations becomes:

$$\begin{array}{l} \mathtt{x}{=}\mathtt{null} \vee \mathtt{y}{=}\mathtt{null} \vee \mathtt{slsB}(\mathtt{x}, \mathtt{xmi}_0, \mathtt{xmx}_0, \mathtt{xS}_0, \mathtt{xp}_0) * \mathtt{slsB}(\mathtt{y}, \mathtt{ymi}_0, \mathtt{ymx}_0, \mathtt{yS}_0, \mathtt{yp}_0) \\ \quad \wedge\ (\mathtt{xmx}_0 {\le} \mathtt{ymx}_0 \wedge \mathtt{xp}_0{=}\mathtt{null} \vee \mathtt{xmx}_0 {>} \mathtt{ymx}_0 \wedge \mathtt{yp}_0{=}\mathtt{null}) \end{array}$$

$$
\begin{array}{ll}
\mathit{Prog} & ::= \mathit{tdecl}^{*}\ \mathit{meth}^{*} \\
\mathit{tdecl} & ::= \mathit{datat} \mid \mathit{spred} \\
\mathit{datat} & ::= \texttt{data}\ c\ \{\ \mathit{field}^{*}\ \} \\
\mathit{field} & ::= t\ v \\
t & ::= c \mid \tau \\
\mathit{meth} & ::= t\ \mathit{mn}\ ((t\ v)^{*};(t\ v)^{*})\ \mathit{mspec}^{*}\ \{e\} \\
\tau & ::= \texttt{int} \mid \texttt{bool} \mid \texttt{void} \\
e & ::= d \mid d[v] \mid v{:=}e \mid e_1;e_2 \mid t\ v;\ e \mid \texttt{if}\ (v)\ e_1\ \texttt{else}\ e_2 \\
d & ::= \texttt{null} \mid k^{\tau} \mid v \mid \texttt{new}\ c(v^{*}) \mid \mathit{mn}(u^{*};v^{*}) \\
d[v] & ::= v.f \mid v_1.f{:=}v_2 \mid \texttt{free}(v)
\end{array}
$$

Figure 4: A Core (C-like) Imperative Language.

Continuing the analysis, the fixed point of the program summary (Pre,Post) is reached:

$$
\begin{array}{l}
\mathsf{Pre}\ \equiv \texttt{x}{=}\texttt{null} \vee \texttt{y}{=}\texttt{null} \vee \texttt{slsB}(\texttt{x},\texttt{xmi}_0,\texttt{xmx}_0,\texttt{xS}_0,\texttt{xp}_0){*} \\
\quad \texttt{slsB}(\texttt{y},\texttt{ymi}_0,\texttt{ymx}_0,\texttt{yS}_0,\texttt{yp}_0){\wedge}(\texttt{xmx}_0{\le}\texttt{ymx}_0{\wedge}\texttt{xp}_0{=}\texttt{null}{\vee}\texttt{xmx}_0{>}\texttt{ymx}_0{\wedge}\texttt{yp}_0{=}\texttt{null}), \\
\mathsf{Post} \equiv \texttt{x}{=}\texttt{null}{\wedge}\texttt{res}{=}\texttt{y} \vee \texttt{y}{=}\texttt{null}{\wedge}\texttt{res}{=}\texttt{x} \vee \texttt{slsB}(\texttt{x},\texttt{xmi}_1,\texttt{xmx}_1,\texttt{xS}_1,\texttt{xp}_1) \\
\quad {*}\texttt{slsB}(\texttt{y},\texttt{ymi}_1,\texttt{ymx}_1,\texttt{yS}_1,\texttt{yp}_1){\wedge}\texttt{xS}_0{\sqcup}\texttt{yS}_0{=}\texttt{xS}_1{\sqcup}\texttt{yS}_1{\wedge}\texttt{xmi}_1{=}\texttt{xmi}_0{\wedge}\texttt{ymi}_1{=}\texttt{ymi}_0{\wedge} \\
\quad (\texttt{xmi}_0{\le}\texttt{ymi}_0{\wedge}\texttt{res}{=}\texttt{x}{\wedge}\texttt{xp}_1{=}\texttt{y}{\wedge}\texttt{xmx}_1{\le}\texttt{ymi}_1{\vee}\texttt{xmi}_0{>}\texttt{ymi}_0{\wedge}\texttt{res}{=}\texttt{y}{\wedge}\texttt{yp}_1{=}\texttt{x}{\wedge}\texttt{ymx}_1{\le}\texttt{xmi}_1)
\end{array}
$$

From this example, we observe that memory safety can not only be related to the shape of data structures, but may also be related to data values stored in them (and also the relation of such values). For the `merge` example, our analysis can find that one input list is traversed to its end, i.e. until `null` is reached, and the other input list is partially traversed till it reaches an element that is larger than the maximal value of the former list. As captured in the inferred precondition, the rest of the list will not be accessed by the program. Similarly, the inferred postcondition captures a fairly precise specification that represents the merged list using two list segments that either begins from `x` or from `y`, depending on which of the two input lists contains the smallest element.

3. Language and Abstract Domain

To simplify the presentation, we employ a strongly-typed C-like imperative language in Fig. 4 to demonstrate our approach. A program *Prog* written in this language consists of declarations *tdecl*, which can be data type declarations *datat* (e.g. `Node` in Section 2), predicate definitions *spred* (e.g. `llB` and `slsB`), as well as method declarations *meth*. The definitions for *spred* and *mspec* are given later in Fig. 5. We assume that methods come with no specifications (i.e. no mspec^{*} part), and our proposed analysis will discover them. Our language is expression-oriented, and thus the body of a method (e) is an expression formed by program constructors. Note that d and $d[v]$ represent respectively heap-insensitive and heap sensitive commands. k^{τ} is a constant of type τ. The language allows both call-by-value and call-by-reference method parameters, separated with a semicolon (;). These parameters allow each iterative loop to be directly converted to an equivalent tail-recursive method, where mutations on parameters are made visible to the caller via pass-by-reference. This technique of translating away iterative loops is standard and is helpful in further minimising our core language.

Our specification language (in Fig. 5) allows (user-defined) shape predicates *spred* to specify program properties in our combined domain. Note that such predicates are constructed with disjunctive constraints Φ. We require that the predicates be well-formed [11]. The first parameter of a predicate is the pointer referring to the data structures itself. A conjunctive abstract program state σ has mainly two parts: the

$$\begin{array}{ll}
\textit{spred} & ::= p(\texttt{root}, v^*) \equiv \Phi \\
\Phi & ::= \bigvee \sigma^* \\
\sigma & ::= \exists v^* {\cdot} \kappa \wedge \pi \\
\textit{mspec} & ::= \textit{requires}\ \Phi_{pr}\ \textit{ensures}\ \Phi_{po} \\
\Delta & ::= \Phi \mid \Delta_1 \vee \Delta_2 \mid \Delta \wedge \pi \mid \Delta_1 {*} \Delta_2 \mid \exists v {\cdot} \Delta \\
\kappa & ::= \texttt{emp} \mid v {\mapsto} c(v^*) \mid p(v^*) \mid \kappa_1 * \kappa_2 \\
\pi & ::= \gamma \wedge \phi \\
\gamma & ::= v_1{=}v_2 \mid v{=}\texttt{null} \mid v_1{\neq}v_2 \mid v{\neq}\texttt{null} \mid \gamma_1{\wedge}\gamma_2 \\
\phi & ::= \varphi \mid b \mid a \mid \phi_1{\wedge}\phi_2 \mid \phi_1{\vee}\phi_2 \mid \neg\phi \mid \exists v \cdot \phi \mid \forall v \cdot \phi \\
b & ::= \texttt{true} \mid \texttt{false} \mid v \mid b_1 = b_2 \\
a & ::= s_1{=}s_2 \mid s_1{\leq}s_2 \\
s & ::= k^{\texttt{int}} \mid v \mid k^{\texttt{int}}{\times}s \mid s_1{+}s_2 \mid -s \mid \mathit{max}(s_1,s_2) \mid \mathit{min}(s_1,s_2) \mid |\texttt{B}| \\
\varphi & ::= v{\in}\texttt{B} \mid \texttt{B}_1{=}\texttt{B}_2 \mid \texttt{B}_1{\sqsubset}\texttt{B}_2 \mid \texttt{B}_1{\sqsubseteq}\texttt{B}_2 \mid \forall v{\in}\texttt{B}{\cdot}\phi \mid \exists v{\in}\texttt{B}{\cdot}\phi \\
\texttt{B} & ::= \texttt{B}_1{\sqcup}\texttt{B}_2 \mid \texttt{B}_1{\sqcap}\texttt{B}_2 \mid \texttt{B}_1{-}\texttt{B}_2 \mid \emptyset \mid \{v\}
\end{array}$$

Figure 5: The Specification Language.

heap (shape) part κ in the separation domain and the pure part π in the convex polyhedra domain and bag (multi-set) domain, where π consists of γ, ϕ and φ as aliasing, numerical and multi-set information, respectively. $k^{\texttt{int}}$ is an integer constant. The square symbols like $\sqsubset$, $\sqsubseteq$, $\sqcup$ and $\sqcap$ are multi-set operators. The set of all σ formulae is denoted as SH (*symbolic heap*). During the symbolic execution, the abstract program state at each program point will be a disjunction of σ's, denoted by Δ. Its set is defined as $\mathcal{P}_{\mathsf{SH}}$. An abstract state Δ can be normalised to the Φ form [11].

Using entailment [11], we define a partial order over these abstract states:

$$\Delta \preceq \Delta' =_{df} \Delta' \vdash \Delta * \texttt{R}$$

where $\texttt{R}$ is the (computed) residue part. And we also have an induced lattice over these states as the base of fixpoint calculation for our analysis.

The memory model of our specification formulae can be found in [11]. In our analysis, variables include both program and logical variables.

3.1. Generalised Bi-Abduction for the Combined Domain

We present a new bi-abduction procedure over the combined domain (which generalises the previous bi-abduction [7] over only the shape domain).

Given σ and σ_1, the bi-abduction procedure $\sigma * [\sigma'] \rhd \sigma_1 * \sigma_2$ (shown in Fig. 6) aims to find the anti-frame part σ' and the frame part σ_2 such that

$$\sigma * \sigma' \vdash \sigma_1 * \sigma_2$$

where σ and σ_1 can be the current program state and the precondition of the next instruction, respectively. Our abduction procedure can handle more than one predicates in the analysis, while the shape abduction [7] caters for only one specified shape predicate domain. Another advance is that we can infer numerical and bag properties together with the shape formulae as the anti-frame to improve the precision of the analysis.

During the analysis of a program (via symbolic execution), if the current abstract state cannot meet the requirement for the next program command due to the lack of knowledge about the program's precondition, we need to infer such missing information with the bi-abduction procedure. The inferred anti-frame σ' will be propagated back to the precondition of the program to allow the analysis to continue; the inferred frame σ_2 is the unconsumed part from σ and will be carried on. For instance, the entailment $\texttt{emp} \vdash \texttt{x}{\mapsto}\texttt{Node}(\texttt{xv}, \texttt{xp})$

$$\frac{\sigma \nvdash \sigma_1 * \texttt{true} \quad \sigma_1 \vdash \sigma * \sigma' \quad \sigma * \sigma' \vdash \sigma_1 * \sigma_2}{\sigma * [\sigma'] \rhd \sigma_1 * \sigma_2}\ \textbf{Residue}$$

$$\frac{\begin{array}{c}\sigma \nvdash \sigma_1 * \texttt{true} \quad \sigma_1 \nvdash \sigma * \texttt{true} \quad \sigma_0 \in \mathsf{unroll}(\sigma) \quad \mathsf{data\_no}(\sigma_0) \leq \mathsf{data\_no}(\sigma_1) \\ \sigma_0 \vdash \sigma_1 * \sigma' \text{ or } \sigma_0 * [\sigma_0'] \rhd \sigma_1 * \sigma' \qquad \sigma * \sigma' \vdash \sigma_1 * \sigma_2\end{array}}{\sigma * [\sigma'] \rhd \sigma_1 * \sigma_2}\ \textbf{Unroll}$$

$$\frac{\sigma \nvdash \sigma_1 * \texttt{true} \quad \sigma_1 \nvdash \sigma * \texttt{true} \quad \texttt{no\_Reverse} : \sigma_1 * [\sigma_1'] \rhd \sigma * \sigma' \quad \sigma * \sigma' \vdash \sigma_1 * \sigma_2}{\sigma * [\sigma'] \rhd \sigma_1 * \sigma_2}\ \textbf{Reverse}$$

$$\frac{\sigma \nvdash \sigma_1 * \texttt{true} \quad \sigma_1 \nvdash \sigma * \texttt{true} \quad \sigma * \sigma_1 \vdash \sigma_1 * \sigma_2}{\sigma * [\sigma_1] \rhd \sigma_1 * \sigma_2}\ \textbf{Missing}$$

$$\frac{\begin{array}{c}\sigma \nvdash \sigma_1 * \sigma_1' * \texttt{true} \quad \sigma_1 * \sigma_1' \nvdash \sigma * \texttt{true} \quad \sigma \vdash \sigma_1' * \texttt{true} \\ \sigma * [\sigma'] \rhd \sigma_1 * \sigma_2' \quad \sigma * \sigma' \vdash \sigma_1 * \sigma_1' * \sigma_2\end{array}}{\sigma * [\sigma'] \rhd (\sigma_1 * \sigma_1') * \sigma_2}\ \textbf{Remove}$$

Figure 6: Bi-Abduction rules.

fails as the antecedent contains an empty heap. In this case $\texttt{x}{\mapsto}\texttt{Node}(\texttt{xv}, \texttt{xp})$ will be found by the abductive reasoning to strengthen the antecedent so that the entailment $\texttt{emp} * \texttt{x}{\mapsto}\texttt{Node}(\texttt{xv}, \texttt{xp}) \vdash \texttt{x}{\mapsto}\texttt{Node}(\texttt{xv}, \texttt{xp})$ becomes valid.

The first rule **Residue** triggers when the LHS (σ) does not entail the RHS (σ_1) but the RHS entails the LHS with some formula (σ') as the residue. This rule is quite general and applies in many cases. For instance, if LHS is $\texttt{emp}$ (σ), RHS is $\texttt{x}{\mapsto}\texttt{Node}(\texttt{xv}, \texttt{xp})(\sigma_1)$, the RHS can entail the LHS with the frame $\texttt{x}{\mapsto}\texttt{Node}(\texttt{xv}, \texttt{xp})(\sigma')$. The abduction then checks whether σ conjoined with σ' entails $\sigma_1 * \sigma_2$ for some σ_2 ($\texttt{emp}$ in this example), and returns $\texttt{x}{\mapsto}\texttt{Node}(\texttt{xv}, \texttt{xp})$ as the anti-frame.

The second rule **Unroll** deals with the case where neither side entails the other, for example let us take $\texttt{slsB}(\texttt{x}, \texttt{xmi}, \texttt{xmx}, \texttt{xS}, \texttt{null})$ as LHS and $\exists \texttt{p}, \texttt{u}, \texttt{v}{\cdot}\texttt{x}{\mapsto}\texttt{Node}(\texttt{u}, \texttt{p}) * \texttt{p}{\mapsto}\texttt{Node}(\texttt{v}, \texttt{null})$ as RHS. As the shape predicates in the antecedent σ are formed by disjunctions according to their definitions (like $\texttt{slsB}$), its certain disjunctive branches may imply σ_1. As the rule suggests, to accomplish abduction $\sigma * [\sigma'] \rhd \sigma_1 * \sigma_2$, we first unfold σ ($\sigma_0 \in \mathsf{unroll}(\sigma)$) and try entailment or further abduction with the results (σ_0) against σ_1. If it succeeds with a frame σ', then we confirm the abduction by ensuring $\sigma * \sigma' \vdash \sigma_1 * \sigma_2$. For the example above, the abduction returns $\exists \texttt{u}, \texttt{v}{\cdot}\texttt{xS}{=}\{\texttt{u}, \texttt{v}\}$ as the anti-frame σ' and discovers the nontrivial frame $\texttt{u}{=}\texttt{xmi} \wedge \texttt{v}{=}\texttt{xmx} \wedge \texttt{u}{\leq}\texttt{v}$ as σ_2. The function $\mathsf{data\_no}$ returns the number of data nodes in an abstract state, e.g. it returns 1 for $\texttt{x}{\mapsto}\texttt{Node}(\texttt{v}, \texttt{p}) * \texttt{llB}(\texttt{p}, \texttt{n}, \texttt{T})$. This syntactic check prevents unlimited number of times of unrolling from happening when the abduction procedure invokes this rule recursively. The unroll unfolds all shape predicates once in σ, normalises the result to a disjunctive form ($\bigvee_{i=1}^{n} \sigma_i$), and returns the result as a set of formulae ($\{\sigma_1, ..., \sigma_n\}$). An instance is that it expands $\texttt{x}{\mapsto}\texttt{Node}(\texttt{v}, \texttt{p}) * \texttt{llB}(\texttt{p}, \texttt{n}, \texttt{T})$ to be $\{\texttt{x}{\mapsto}\texttt{Node}(\texttt{v}, \texttt{p}) \wedge \texttt{p}{=}\texttt{null} \wedge \texttt{n}{=}0 \wedge \texttt{T}{=}\emptyset, \exists \texttt{u}_0, \texttt{p}_0, \texttt{n}_0, \texttt{T}_0 \cdot \texttt{x}{\mapsto}\texttt{Node}(\texttt{v}, \texttt{p}) * \texttt{p}{\mapsto}\texttt{Node}(\texttt{u}_0, \texttt{p}_0) * \texttt{llB}(\texttt{p}_0, \texttt{n}_0, \texttt{T}_0) \wedge \texttt{n}{=}\texttt{n}_0{+}1 \wedge \texttt{T}{=}\texttt{T}_0{\cup}\{\texttt{u}_0\}\}$.

The third rule **Reverse** handles the case where neither side entails the other, and the 2nd rule does not apply, e.g. $\exists \texttt{p}, \texttt{u}, \texttt{v}, \texttt{q} \cdot \texttt{x}{\mapsto}\texttt{Node}(\texttt{u}, \texttt{p}) * \texttt{p}{\mapsto}\texttt{Node}(\texttt{v}, \texttt{q})$ as LHS and $\exists \texttt{xS} \cdot \texttt{slsB}(\texttt{x}, \texttt{xmi}, \texttt{xmx}, \texttt{xS}, \texttt{xp})$ as RHS. In this case the antecedent cannot be unfolded as it contains only data nodes. As the rule suggests, it reverses two sides of the entailment and applies the second rule to uncover the constraints σ_1' and σ'. It then checks that the LHS (σ), with σ' conjoined, does entail the RHS (σ_1) before it returns σ'. For the example above, the anti-frame is inferred as $\texttt{u}{\leq}\texttt{v}$.

Our bi-abduction procedure (built on the rules in Fig. 6) would attempt the first three rules exhaustively in the given order; if they do not succeed in finding a solution, then the rule **Missing** is invoked to add the consequence to the antecedent, provided that they are consistent. It is effective for situations like $\mathtt{x}{\mapsto}\mathtt{Node}(\_,\_) \nvdash \mathtt{y}{\mapsto}\mathtt{Node}(\_,\_)$, where we should add $\mathtt{y}{\mapsto}\mathtt{node}(\_,\_)$ to the LHS directly. In our analysis, we assume that different variables refer to different nodes unless aliasing is suggested in the program code. For example, the if-statement `if (x == y){c}` suggests that `x` and `y` are aliased in code `c`. Note that when the third rule is applied, the bi-abduction procedure invoked in the premise, namely $\mathtt{no\_Reverse}{:}\,\sigma_1 * [\sigma_1'] \triangleright \sigma * \sigma'$, is not allowed to apply the third rule again, as indicated by the `no_Reverse` prefix. This is to prevent an infinite number of applications of the third rule.

If the first four rules fail, the **Remove** rule then tries to find a part of consequent (σ_1') which is entailed by the antecedent. The abduction is then applied to the remaining part of the consequent (σ_1) to discover the anti-frame (σ'). For example, the bi-abduction question $\mathtt{llB}(\mathtt{x},\mathtt{n},\mathtt{S}){\wedge}\mathtt{n}{>}2{*}[\sigma'] \triangleright \mathtt{x}{\mapsto}\mathtt{Node}(\mathtt{v}_1,\mathtt{p}_1) * \mathtt{y}{\mapsto}\mathtt{Node}(\mathtt{v}_2,\mathtt{p}_2) * \sigma_2$ needs this rule to remove $\mathtt{x}{\mapsto}\mathtt{Node}(\mathtt{v}_1,\mathtt{p}_1)$ from consequent before applying the **Missing** rule to find the anti-frame $\sigma' = \mathtt{y}{\mapsto}\mathtt{Node}(\mathtt{v}_2,\mathtt{p}_2)$.

Our earlier work [34] gives a restricted form of abduction focusing on discovering pure information with the assumption that either complete or partial shape information is available. Our bi-abduction algorithm presented here generalises it to cater for full specification discovery scenarios, whereby, we do not have the hints to guide the analysis anymore due to the absence of shape information in pre/post-conditions; but at the same time we can have more freedom as to what missing information to discover. One observation on abduction is that there can be many solutions of the anti-frame σ' for the entailment $\sigma * \sigma' \vdash \sigma_1 * \sigma_2$ to succeed. Therefore, we define "quality" of anti-frame solutions with the partial order $\preceq$ given in the previous section, i.e. the smaller (weaker[2]) one is regarded as better, similar to the ordering given for the shape domain in [7]. We prefer to find solutions that are (potentially locally) minimal with respect to $\preceq$ and consistent. However, such solutions are generally not easy to compute and could incur excess cost (with additional disjunction in the analysis). Therefore, our abductive inference is designed more from a practical perspective to discover anti-frames that should be suitable as preconditions for programs, and the partial order $\preceq$ sounds more like a guidance of the decision choices of our abduction implementation, rather than a guarantee to find the theoretically best solution.

4. Analysis Algorithm

Our proposed analysis algorithm is given in Fig. 7. It takes three input parameters: $\mathcal{T}$ as the set of method specifications that are already inferred, the procedure to be analysed $t\ mn\ ((t\ x)^*; (t\ y)^*)\ \{e\}$, and a pre-set upper bound n on the number of shared logical variables that we keep during the analysis.

As in a standard abstract interpretation framework, our analysis carries out the fixed-point iteration until a fixed-point $(\mathsf{Pre}_i, \mathsf{Post}_i)$ (for some i) is reached. To infer the pre-conditions, our abstract semantics is equipped with bi-abduction over the combined domain. To allow the discovery of more precise preconditions, our abstraction procedure is also equipped with abduction, yielding the novel *abductive abstraction* $(\mathsf{abs_a})$ for precondition discovery. The postcondition inference still employs the normal abstraction mechanism (abs).

To allow the fixed-point iteration to converge, in addition to the abstraction operators, specifically designed join and widen operators over the combined domain are also proposed.[3] At the beginning, we initialise the iteration variable (i) and the states to record the computed pre- and postconditions (Pre_i and Post_i). We use `emp` as the initial precondition because we know nothing about the footprint of the code. The initial postcondition is set to `false` which denotes the top element of the lattice of our abstraction domain.

We first set the method precondition as `emp` and postcondition as `false` which signifies that we know nothing about the method (line 1). Then for each iteration, a forward bi-abductive analysis is employed to compute a new pre-/post-condition (line 4) based on the current specification. The analysis performs abstraction on both pre-/post-conditions obtained to maintain the finiteness of the shape domain. The obtained results are joined with the results from the previous iteration (line 6), and widening is conducted over both to ensure termination of the analysis (line 7). If the analysis cannot continue due to a program

[2]A formula $\mathtt{p}_2$ is weaker than a formula $\mathtt{p}_2$ if the following entailment holds: $\mathtt{p}_1 \vdash \mathtt{p}_2{*}\mathtt{true}$.
[3]Our analysis uses lifted versions of these operations (indicated by †), which will be explained in more details later.

Fixpoint Computation for Pre/Post in the Combined Domain
Input: $\mathcal{T},\ t\ mn\ ((t\ x)^*;(t\ y)^*)\ \{e\},\ n$
Local: $i := 0;\ \mathsf{Pre}_i := \mathtt{emp}, \mathsf{Post}_i := \mathtt{false};$

```
1   $\mathcal{T}' := \mathcal{T} \cup \{t\ mn\ ((t\ x)^*;(t\ y)^*)\ \textit{requires}\ \mathsf{Pre}_0\ \textit{ensures}\ \mathsf{Post}_0\ \{e\}\};$
2   repeat
3     $i := i + 1;$
4     $(\mathsf{Pre}_i, \mathsf{Post}_i) := [\![e]\!]^{\mathsf{A}}_{\mathcal{T}'}(\mathsf{Pre}_{i-1}, \mathsf{Pre}_{i-1});$
5     $(\mathsf{Pre}_i, \mathsf{Post}_i) := (\mathsf{abs_a}^{\dagger}(\mathsf{Pre}_i), \mathsf{abs}^{\dagger}(\mathsf{Post}_i));$
6     $(\mathsf{Pre}_i, \mathsf{Post}_i) := (\mathsf{join}^{\dagger}(\mathsf{Pre}_{i-1}, \mathsf{Pre}_i), \mathsf{join}^{\dagger}(\mathsf{Post}_{i-1}, \mathsf{Post}_i));$
7     $(\mathsf{Pre}_i, \mathsf{Post}_i) := (\mathsf{widen}^{\dagger}(\mathsf{Pre}_{i-1}, \mathsf{Pre}_i), \mathsf{widen}^{\dagger}(\mathsf{Post}_{i-1}, \mathsf{Post}_i));$
8     if $\mathsf{Pre}_i$=false or $\mathsf{Post}_i$=false or $\mathsf{cp\_no}(\mathsf{Pre}_i)>n$ or $\mathsf{cp\_no}(\mathsf{Post}_i)>n$
      then return fail end if
9     $\mathcal{T}' := \mathcal{T} \cup \{t\ mn\ ((t\ x)^*;(t\ y)^*)\ \textit{requires}\ \mathsf{Pre}_i\ \textit{ensures}\ \mathsf{Post}_i\ \{e\}\};$
10  until $\mathcal{T}'$ does not change
11  $\mathsf{Post} = [\![e]\!]_{\mathcal{T}'}\mathsf{Pre}_i;$
12  if $\mathsf{Post} = \mathtt{false}$ or $\mathsf{Post} \nvdash \mathsf{Post}_i * \mathtt{true}$ then return fail
13  else return $\mathcal{T}'$
14  end if
```

Figure 7: Main analysis algorithm.

bug, or cannot keep the number of shared logical variables/cutpoints (counted by $\mathsf{cp\_no}$) within a specified bound (n), then a failure is reported (line 8). At the end of each iteration, the inferred summary is used to update the specification of mn (line 9), which will be used for recursive calls (if any) of mn in the next iteration. Finally we judge whether a fixed-point is already reached (line 10). The last few lines (from line 11) ensure that inferred specifications are indeed sound using a standard abstract semantics (without abduction). Any unsound specifications will be ruled out.

In case of the mutually recursive procedures our main algorithm from Fig. 7 takes as input the mutually recursive procedures $mn_1..mn_k$ (instead of only one procedure mn) and at each i-iteration (lines 2-10) it computes a specification $(\mathsf{Pre}^k_i, \mathsf{Post}^k_i)$ for each mn_k procedure. The fixed-point iteration stops when the specification of each procedure mn_k reaches a fixed-point $(\mathsf{Pre}^k, \mathsf{Post}^k)$.

Intuitively, the $\mathsf{join}^\dagger$ operator is applied over two abstract states, and tries to find a common shape as an abstraction for the separation part of both states. If such common shape is found, it performs convex hull and bag join for the pure parts. Otherwise it keeps a disjunction of the two states. The $\mathsf{widen}^\dagger$ is analogous to $\mathsf{join}^\dagger$. The difference is that we expect the heap portion of the first state is subsumed by the second one, and then it applies the pure widening for the pure part.

In the rest of this section, we shall first present the core technicals about bi-abductive abstract semantics and the abstraction mechanism (including abductive abstraction). The formal definitions of $\mathsf{join}^\dagger$ and $\mathsf{widen}^\dagger$ and other details will be given later in Sec 4.3, before we discuss about the soundness and termination of our analysis in Sec 4.4.

4.1. Bi-Abductive Abstract Semantics

As shown in Fig. 7, our analysis employs two abstract semantics: a bi-abductive abstract semantics (i.e. the one equipped with abduction) (line 4), and an underlying abstract semantics (i.e. the one without abduction) (line 11). We first give the definition of the underlying abstract semantics. Its type is defined as

$$[\![e]\!] \;:\; \mathsf{AllSpec} \to \mathcal{P}_{\mathsf{SH}} \to \mathcal{P}_{\mathsf{SH}}$$

where $\mathsf{AllSpec}$ contains procedure/method specifications. For some program $\mathtt{e}$ and its given precondition Δ, the semantics calculates the postcondition $[\![\mathtt{e}]\!]_{\mathcal{T}}\Delta$, for a given set of method specifications $\mathcal{T}$.

The essential constituents of the underlying abstract semantics are the basic transition functions from a conjunctive abstract state (σ) to a conjunctive or disjunctive abstract state (σ or Δ) below:

$$\begin{array}{lcll}
\mathsf{unfold}(x) & : & \mathsf{SH} \to \mathcal{P}_{\mathsf{SH}[x]} & \text{Unfolding} \\
\mathsf{exec}(d[x]) & : & \mathsf{AllSpec} \to \mathsf{SH}[x] \to \mathcal{P}_{\mathsf{SH}} & \text{Heap-sensitive execution} \\
\mathsf{exec}(d) & : & \mathsf{AllSpec} \to \mathsf{SH} \to \mathcal{P}_{\mathsf{SH}} & \text{Heap-insensitive execution}
\end{array}$$

where $\mathsf{SH}[x]$ denotes the set of conjunctive abstract states in which each element has x exposed as the head of a data node ($x{\mapsto}c(v^*)$), and $\mathcal{P}_{\mathsf{SH}[x]}$ contains all the (disjunctive) abstract states, each of which is composed by such conjunctive states. Here $\mathsf{unfold}(x)$ rearranges the symbolic heap so that the cell referred to by x is exposed for access by heap sensitive commands $d[x]$ via the second transition function $\mathsf{exec}(d[x])$. The third function defined for other (heap insensitive) commands d does not require such exposure of x.

The unfolding function is defined by the following two rules:

$$\frac{\sigma \vdash x{\mapsto}c(v^*) * \sigma'}{\mathsf{unfold}(x)\sigma \rightsquigarrow \sigma} \qquad\qquad \frac{\sigma \vdash p(x, u^*) * \sigma' \quad p(\mathtt{root}, v^*){\equiv}\Phi}{\mathsf{unfold}(x)\sigma \rightsquigarrow \sigma' * [x/\mathtt{root}, u^*/v^*]\Phi}$$

The symbolic execution of heap-sensitive commands $d[x]$ (i.e. $x.f_i$, $x.f_i := w$, or $\mathtt{free}(x)$) assumes that the rearrangement $\mathsf{unfold}(x)$ has been done prior to execution:

$$\frac{\sigma \vdash x{\mapsto}c(v_1, .., v_n) * \sigma'}{\mathsf{exec}(x.f_i)(\mathcal{T})\sigma \rightsquigarrow \sigma \wedge \mathtt{res}{=}v_i} \qquad\qquad \frac{\sigma \vdash x{\mapsto}c(u^*) * \sigma'}{\mathsf{exec}(\mathtt{free}(x))(\mathcal{T})\sigma \rightsquigarrow \sigma'}$$

$$\frac{\sigma \vdash x{\mapsto}c(v_1, .., v_n) * \sigma'}{\mathsf{exec}(x.f_i := w)(\mathcal{T})\sigma \rightsquigarrow \sigma' * x{\mapsto}c(v_1, .., v_{i-1}, w, v_{i+1}, .., v_n)}$$

The symbolic execution rules for heap-insensitive commands are as follows:

$$\mathsf{exec}(k)(\mathcal{T})\sigma =_{df} \sigma \wedge \mathtt{res}{=}k \qquad \frac{isdatat(c)}{\mathsf{exec}(\mathtt{new}\ c(v^*))(\mathcal{T})\sigma =_{df} \sigma * c(\mathtt{res}, v^*)}$$

$$\mathsf{exec}(x)(\mathcal{T})\sigma =_{df} \sigma \wedge \mathtt{res}{=}x$$

$$\frac{\begin{array}{c} t\ mn\ ((t_i\ u_i)_{i=1}^m; (t'_i\ v_i)_{i=1}^n)\ \mathit{requires}\ \Phi_{pr}\ \mathit{ensures}\ \Phi_{po} \in \mathcal{T} \\ \rho = [x'_i/u_i]_{i=1}^m \circ [y'_i/v_i]_{i=1}^n \quad \sigma \vdash \rho\Phi_{pr} * \sigma' \\ \rho_o = [r_i/v_i]_{i=1}^n \circ [x'_i/u'_i]_{i=1}^m \circ [y'_i/v'_i]_{i=1}^n \quad \rho_l = [r_i/y'_i]_{i=1}^n \quad \mathit{fresh\ logical}\ r_i \end{array}}{\mathsf{exec}(mn(x_1,..,x_m;y_1,..,y_n))(\mathcal{T})\sigma \rightsquigarrow (\rho_l\sigma') * (\rho_o\Phi_{po})}$$

The first three rules deal with constant (k), variable (x) and data node creation ($\mathtt{new}\ c(v^*)$), respectively, while the last rule handles method invocation. The test $isdatat$(c) returns true iff c is a data node. In the last rule, the call site is ensured to meet the precondition of mn, as signified by $\sigma \vdash \rho\Phi_{pr} * \sigma'$. In this case, the execution succeeds and the post-state of the method call involves mn's postcondition as signified by $\rho_o\Phi_{po}$.

A lifting function † is defined to lift unfold's domain to $\mathcal{P}_{\mathsf{SH}}$:

$$\mathsf{unfold}^\dagger(x) \bigvee \sigma_i =_{df} \bigvee(\mathsf{unfold}(x)\sigma_i)$$

and this function is overloaded for exec to lift both its domain and range to $\mathcal{P}_{\mathsf{SH}}$:

$$\mathsf{exec}^\dagger(d)(\mathcal{T}) \bigvee \sigma_i =_{df} \bigvee(\mathsf{exec}(d)(\mathcal{T})\sigma_i)$$

Based on the transition functions above, we can define the underlying abstract semantics for a program e as follows (where loops are already translated into tail-recursions):

$$\begin{array}{ll}
[\![d[x]]\!]_{\mathcal{T}}\Delta & =_{df}\ \mathsf{exec}^\dagger(d[x])(\mathcal{T}) \circ \mathsf{unfold}^\dagger(x)\Delta \\
[\![d]\!]_{\mathcal{T}}\Delta & =_{df}\ \mathsf{exec}^\dagger(d)(\mathcal{T})\Delta \\
[\![e_1;e_2]\!]_{\mathcal{T}}\Delta & =_{df}\ [\![e_2]\!]_{\mathcal{T}} \circ [\![e_1]\!]_{\mathcal{T}}\Delta \\
[\![x:=e]\!]_{\mathcal{T}}\Delta & =_{df}\ [x'/x, r'/\mathtt{res}]([\![e]\!]_{\mathcal{T}}\Delta) \wedge x{=}r' \quad \mathit{fresh\ logical}\ x', r' \\
[\![\mathtt{if}\ (v)\ e_1\ \mathtt{else}\ e_2]\!]_{\mathcal{T}}\Delta & =_{df}\ ([\![e_1]\!]_{\mathcal{T}}(v{\wedge}\Delta)) \vee ([\![e_2]\!]_{\mathcal{T}}(\neg v{\wedge}\Delta))
\end{array}$$

Bi-Abductive Semantics. We shall now present the definitions of our bi-abductive abstract semantics. Its type is

$$[\![e]\!]^{\mathsf{A}}\ :\ \mathsf{AllSpec} \to (\mathcal{P}_{\mathsf{SH}} \times \mathcal{P}_{\mathsf{SH}}) \to (\mathcal{P}_{\mathsf{SH}} \times \mathcal{P}_{\mathsf{SH}})$$

It takes a piece of program code and a specification table, and maps a pair of (disjunctive) set of symbolic heaps to another such pair (where the first in the pair is the accumulated precondition and the second is the current state). It relies on the following two basic functions:

$$\begin{array}{ll}
\mathsf{Unfold}(x) & :\ (\mathsf{SH} \times \mathsf{SH}) \to (\mathcal{P}_{\mathsf{SH}} \times \mathcal{P}_{\mathsf{SH}}) \\
\mathsf{Exec}(ds) & :\ \mathsf{AllSpec} \to (\mathsf{SH} \times \mathsf{SH}) \to (\mathcal{P}_{\mathsf{SH}} \times \mathcal{P}_{\mathsf{SH}})
\end{array}$$

The definitions of both functions are given below:

```
$\mathsf{Unfold}(x)(\sigma', \sigma) =_{df}$
    **if** $(\sigma{*}[\sigma_m] \rhd x{\mapsto}c(y^*){*}\mathtt{true}$ *for fresh logical vars* $y^*) \wedge (\sigma'{*}\sigma_m \nvdash \mathtt{false})$
    **then let** $\Delta{=}\mathsf{unfold}(x)(\sigma{*}\sigma_m)$ **in** $(\sigma'{*}\sigma_m, \Delta)$
    **else** $(\sigma', \mathtt{false})$

$\mathsf{Exec}(ds)(\mathcal{T})(\sigma', \sigma) =_{df}$ **let** $\Delta{=}\mathsf{exec}(ds)(\mathcal{T})\sigma$ **in** $(\sigma', \Delta)$
                                   where $ds$ is either $d[x]$ or $d$, except for procedure call
```

$$\frac{\begin{array}{c} t\ mn\ ((t_i\ u_i)_{i=1}^{m}; (t'_i\ v_i)_{i=1}^{n})\ \mathit{requires}\ \Phi_{pr}\ \mathit{ensures}\ \Phi_{po} \in \mathcal{T} \\ \rho = [x'_i/u_i]_{i=1}^{m} \circ [y'_i/v_i]_{i=1}^{n} \quad \sigma * [\sigma'_1] \rhd \rho\Phi_{pr} * \sigma_1 \quad \sigma' {*} \sigma'_1 \nvdash \texttt{false} \\ \rho_o = [r_i/v_i]_{i=1}^{n} \circ [x'_i/u'_i]_{i=1}^{m} \circ [y'_i/v'_i]_{i=1}^{n} \quad \rho_l = [r_i/y'_i]_{i=1}^{n} \quad \mathit{fresh\ logical\ vars}\ r_i \end{array}}{\mathsf{Exec}(mn(x_{1..m}; y_{1..n}))(\mathcal{T})(\sigma', \sigma) =_{df} (\sigma' * \sigma'_1, (\rho_o\Phi_{po}) {*} (\rho_l\sigma_1))}$$

The Unfold function firstly tests (using bi-abduction) whether the node $x {\mapsto} c(y^*)$ is in σ, if not, abduction is applied to find the missing σ_m. If σ' and σ_m do not contradict, it unfolds $\sigma * \sigma_m$ to expose x (via the unfold function defined earlier in this section), and adds σ_m to the precondition. Otherwise, it returns false for the current state.

The Exec function symbolically executes the command ds (via the exec function defined earlier in this section) and translates the current state σ to a disjunction of new states Δ. The special case is the method invocation, which may require bi-abduction to be applied for the current state. When the method mn is invoked, we take its current specification (Φ_{pr}, Φ_{po}) from $\mathcal{T}$, and substitute the formal parameters u_i and v_i by the current arguments x'_i and y'_i respectively. Note that prime notations x'_i and y'_i denote the current values of x_i and y_i in the current state σ. Then we apply bi-abduction from the current state σ to the precondition $\rho\Phi_{pr}$. If it succeeds, the discovered missing state σ'_1 will be propagated back to the precondition σ' to help make the symbolic execution to succeed. The postcondition of mn, Φ_{po} is substituted by ρ_o in order to be added to the current state. Since the variables y_i are call-by-reference, we let r_i to be the intermediate variables, while the variables y'_i denote the latest values.

A lifting function † is defined to lift Unfold's and Exec's domains:

$$\begin{array}{ll} \mathsf{Unfold}^{\dagger}(x)(\bigvee_i \sigma'_i, \bigvee_j \sigma_j) & =_{df} \bigvee_{i,j}(\mathsf{Unfold}(x)(\sigma'_i, \sigma_j)) \\ \mathsf{Exec}^{\dagger}(ds)(\mathcal{T})(\bigvee_i \sigma'_i, \bigvee_j \sigma_j) & =_{df} \bigvee_{i,j}(\mathsf{Exec}(ds)(\mathcal{T})(\sigma'_i, \sigma_j)) \end{array}$$

Based on the above functions, the bi-abductive abstract semantics is defined as follows:

$$\begin{array}{ll} [\![d[x]]\!]^{\mathsf{A}}_{\mathcal{T}}(\Delta', \Delta) & =_{df} \mathsf{Exec}^{\dagger}(d[x])(\mathcal{T}) \circ \mathsf{Unfold}^{\dagger}(x)(\Delta', \Delta) \\ [\![d]\!]^{\mathsf{A}}_{\mathcal{T}}(\Delta', \Delta) & =_{df} \mathsf{Exec}^{\dagger}(d)(\mathcal{T})(\Delta', \Delta) \\ [\![e_1; e_2]\!]^{\mathsf{A}}_{\mathcal{T}}(\Delta', \Delta) & =_{df} [\![e_2]\!]^{\mathsf{A}}_{\mathcal{T}} \circ [\![e_1]\!]^{\mathsf{A}}_{\mathcal{T}}(\Delta', \Delta) \\ [\![x := e]\!]^{\mathsf{A}}_{\mathcal{T}}(\Delta', \Delta) & =_{df} [x'/x, r'/\texttt{res}]([\![e]\!]^{\mathsf{A}}_{\mathcal{T}}(\Delta', \Delta \wedge x{=}r')) \quad \text{fresh logical } x', r' \\ [\![\texttt{if}\ (v)\ e_1\ \texttt{else}\ e_2]\!]^{\mathsf{A}}_{\mathcal{T}}(\Delta', \Delta) & =_{df} ([\![e_1]\!]^{\mathsf{A}}_{\mathcal{T}}(\Delta', v{\wedge}\Delta)) \vee ([\![e_2]\!]^{\mathsf{A}}_{\mathcal{T}}(\Delta', \neg v{\wedge}\Delta)) \end{array}$$

Example. Considering the abductive abstract execution

$$[\![\texttt{y:=x.next}]\!]^{\mathsf{A}}_{\mathcal{T}}(\texttt{llB(x,n,B)}, \texttt{llB(x,n,B)})$$

which unfolds the current state by $\mathsf{Unfold}(\texttt{x})(\texttt{llB(x,n,B)}, \texttt{llB(x,n,B)})$ and finds that x should refer to a non-empty list with property $\exists \texttt{v}_0, \texttt{B}_1 \cdot \texttt{n}{\geq}1 {\wedge} \texttt{B}{=}\{\texttt{v}_0\}{\sqcup}\texttt{B}_1$ in the precondition. With this information, we can unfold $\texttt{llB(x,n,B)}$ to $\exists \texttt{v}_0, \texttt{B}_1, \texttt{p}_1 \cdot\ \texttt{x}{\mapsto}\texttt{Node}(\texttt{v}_0, \texttt{p}_1) {*} \texttt{llB}(\texttt{p}_1, \texttt{n}{-}1, \texttt{B}_1)$. Executing y:=x.next gets $\texttt{y:=p}_1$ in the current state. The result of the execution is the pair

$$(\exists \texttt{v}_0, \texttt{B}_1 {\cdot} \texttt{llB(x,n,B)} {\wedge} \texttt{n}{\geq}1 {\wedge} \texttt{B}{=}\{\texttt{v}_0\}{\sqcup}\texttt{B}_1,\ \exists \texttt{v}_0, \texttt{B}_1 {\cdot} \texttt{x}{\mapsto}\texttt{Node}(\texttt{v}_0, \texttt{y}) {*} \texttt{llB}(\texttt{y}, \texttt{n}{-}1, \texttt{B}_1) {\wedge} \texttt{B}{=}\{\texttt{v}_0\}{\sqcup}\texttt{B}_1)$$

4.2. The Abstraction Mechanism

We have specifically-designed abstraction, join and widening operations employed in our analysis process. We focus on abstraction in this subsection and leave join and widening operators to the next subsection.

Abstraction function. During the symbolic execution, we may be confronted with many "concrete" shapes in program states. As an example of list traversal, the list may contain one node, two nodes, or even more nodes in the list, which the analysis cannot enumerate infinitely. The abstraction function deals with those situations by abstracting the (potentially infinite) concrete situations into more abstract shapes, to ensure finiteness over the shape domain. Our rationale is to keep only program variables and shared cutpoints; all other logical variables will be abstracted away. As an instance, the first state below can be further abstracted (as shown), while the second one cannot:

$$\begin{array}{l}\mathsf{abs}(\mathtt{x}{\mapsto}\mathtt{Node}(\mathtt{v_1},\mathtt{z_0}) * \mathtt{z_0}{\mapsto}\mathtt{Node}(\mathtt{v_2},\mathtt{null})) \ = \ \mathtt{llB}(\mathtt{x},\mathtt{n},\mathtt{S}) \wedge \mathtt{n}{=}2 \wedge \mathtt{S} = \{\mathtt{v_1},\mathtt{v_2}\} \\ \mathsf{abs}(\mathtt{x}{\mapsto}\mathtt{Node}(\mathtt{v_1},\mathtt{z_0}) * \mathtt{y}{\mapsto}\mathtt{Node}(\mathtt{v_2},\mathtt{z_0}) * \mathtt{z_0}{\mapsto}\mathtt{Node}(\mathtt{v_3},\mathtt{null})) \ = \ \text{-}\end{array}$$

where both x and y are program variables, and $\mathtt{z_0}$ is an existentially quantified logical variable. - denotes the result is unchanged (i.e. same as the input). In the second case $\mathtt{z_0}$ is a shared cutpoint referenced by both x and y, and is therefore preserved. As illustrated, the abstraction transition function abs eliminates unimportant cutpoints (during analysis) to ensure termination. Its type is defined as follows:

$$\mathsf{abs} \ : \ \mathsf{SH} \rightarrow \mathsf{SH} \qquad\qquad \text{Abstraction}$$

which indicates that it takes in a conjunctive abstract state σ and abstracts it as another conjunctive state σ'. Below are the abstraction rules.

$$\mathsf{abs}(\sigma \wedge x_0{=}e) =_{df} \sigma[e/x_0] \qquad \mathsf{abs}(\sigma \wedge e{=}x_0) =_{df} \sigma[e/x_0]$$

$$\frac{x_0 \notin \mathsf{Reach}(\sigma)}{\mathsf{abs}(\mathtt{H}(c)(x_0,v^*) * \sigma) =_{df} \sigma * \mathtt{true}}$$

$$\frac{\begin{array}{c}p_2(u_2^*) \equiv \Phi \qquad \mathtt{H}(c_1)(x,v_1^*) * \sigma_1 \vdash p_2(x,v_2^*) * \sigma_2 \\ \mathsf{Reach}(p_2(x,v_2^*) * \sigma_2 * \sigma_3) \cap \{v_1^*\} = \emptyset\end{array}}{\mathsf{abs}(\mathtt{H}(c_1)(x,v_1^*) * \sigma_1 * \sigma_3) =_{df} p_2(x,v_2^*) * \sigma_2 * \sigma_3}$$

where $\mathtt{H}(c)(x,v^*)$ denotes $x{\mapsto}c(v^*)$ if c is a data node or $c(x,v^*)$ if c is a predicate.

The first two rules eliminate logical variables (x_0) by replacing them with their equivalent expressions (e). The third rule is used to eliminate any garbage (i.e. heap part led by a logical variable x_0 unreachable from the other part of the heap) that may exist in the heap. As x_0 is already unreachable from, and not usable by, the program variables, it is safe to treat it as garbage true, for example the heap part referred to by $\mathtt{x_0}$ in $\mathtt{node}(\mathtt{x},\_,\mathtt{null}) * \mathtt{node}(\mathtt{x_0},\_,\mathtt{null})$ where only x is a program variable.

The last rule of abs plays the most significant role, which intends to eliminate shape formulae led by logical variables (all variables in v_1^*). It tries to fold data nodes up to a shape predicate. It confirms that c_1 is a data node declaration and p_2 is a predicate definition. The predicate p_2 is selected from the user-defined predicates environments and it is the target shape to be abstracted against with. The rule ensures that the latter is a sound abstraction of the former by entailment proof, and the pointer logical parameters of c_1 are not reachable from the other part of the heap (so that the abstraction does not lose necessary information). The function Reach is defined as follows:

$$\mathsf{Reach}(\sigma) =_{df} \bigcup_{v\in\mathtt{fv}(\sigma)} \mathtt{ReachVar}(\kappa{\wedge}\pi, v) \ \mathtt{where} \ \sigma ::= \exists \mathtt{u}^*{\cdot}\kappa{\wedge}\pi$$

that returns all pointer variables which are reachable from free variables in the abstract state σ. The function $\mathtt{ReachVar}(\kappa{\wedge}\pi, v)$ returns the minimal set of pointer variables satisfying the relationship below:

$$\begin{array}{l}\mathtt{ReachVar}(\kappa{\wedge}\pi,v){\supseteq}\{v\}{\cup}\{z_2|\exists z_1,\pi_1{\cdot}z_1{\in}\mathtt{ReachVar}(\kappa{\wedge}\pi,v){\wedge}\pi{=}(z_1{=}z_2{\wedge}\pi_1){\wedge} \\ \mathtt{isptr}(z_2)\} \cup \{z_2|\exists z_1,\kappa_1{\cdot}z_1{\in}\mathtt{ReachVar}(\kappa{\wedge}\pi,v){\wedge}\kappa{=}(c(z_1,..,z_2,..){*}\kappa_1){\wedge}\mathtt{isptr}(z_2)\}\end{array}$$

Viz. it is composed of aliases of v and pointer variables reachable from v. The predicate $\mathtt{isptr}(x)$ checks if x is a pointer variable. The pure logic parameters can be abstracted since the pure relations are kept in pure formulae, so we do not lose numerical information here. Then the lifting function is applied for abs to lift both its domain and range to disjunctive abstract states $\mathcal{P}_{\mathsf{SH}}$:

$$\mathsf{abs}^{\dagger} \bigvee \sigma_i =_{df} \bigvee \mathsf{abs}(\sigma_i)$$

which allows it to be used in the analysis.

Abductive Abstraction. As we mentioned earlier in the `merge` example, to verify such programs may require very precise preconditions that a standard abstraction mechanism may fail to achieve. To cater for such a need, we design a novel *abductive abstraction* function $\mathsf{abs_a}$, which equips abstraction with an abductive reasoning capacity where necessary. In such scenarios, user-specified predicates can offer some guidance in the abstraction in order to discover extra data structure properties for precondition. The new abductive abstraction function is given as follows:

$$\mathsf{abs_a}(\sigma \wedge x_0{=}e) =_{df} \sigma[e/x_0] \qquad \mathsf{abs_a}(\sigma \wedge e{=}x_0) =_{df} \sigma[e/x_0] \qquad \frac{x_0 \notin \mathsf{Reach}(\sigma)}{\mathsf{abs_a}(\mathtt{H}(c)(x_0, v^*) * \sigma) =_{df} \sigma * \mathtt{true}}$$

$$\frac{p_2(u_2^*) \equiv \Phi \qquad \mathtt{H}(c_1)(x, v_1^*) * \sigma_1 \vdash p_2(x, v_2^*) \wedge \pi_2 \qquad \mathsf{Reach}(p_2(x, v_2^*) \wedge \pi_2 * \sigma_3) \cap \{v_1^*\} = \emptyset}{\mathsf{abs_a}(\mathtt{H}(c_1)(x, v_1^*) * \sigma_1 * \sigma_3) =_{df} p_2(x, v_2^*) \wedge \pi_2 * \sigma_3}$$

$$\frac{p_2(u_2^*) \equiv \Phi \quad \mathtt{H}(c_1)(x, v_1^*) * \sigma_1 \nvdash p_2(x, v_2^*) \wedge \pi_2 \quad \mathtt{H}(c_1)(x, v_1^*){*}\sigma_1{*}[\sigma'] \rhd p_2(x, v_2^*){\wedge}\pi_2 \quad \mathsf{Reach}(p_2(x, v_2^*){\wedge}\pi_2{*}\sigma_3){\cap}\{v_1^*\}{=}\emptyset}{\mathsf{abs_a}(\mathtt{H}(c_1)(x, v_1^*) * \sigma_1 * \sigma_3) =_{df} p_2(x, v_2^*) \wedge \pi_2 * \sigma_3}$$

As mentioned earlier, $\mathtt{H}(c)(x, v^*)$ denotes $x{\mapsto}c(v^*)$ if c is a data node or $c(x, v^*)$ if c is a predicate, and the function $\mathsf{Reach}(\sigma)$ returns all pointer variables which are reachable from free variables in the abstract state σ. Similar to the standard abstraction, the first two rules eliminate logical variables, and the third rule drops heap garbage that is unreachable from program variables. The fourth rule combines shape formulae and eliminate logical pointer variables which are not reachable from other program variables. The predicate p_2 is selected from the user-defined predicates environments and it is the target shape to be abstracted to.

The last rule applies when the state $\mathtt{H}(c_1)(x, v_1^*) * \sigma_1$ cannot be abstracted to the predicate p_2 using the standard abstraction but can be abstracted to predicate p_2 with the help of abductive reasoning. When applying such an abstraction function during the precondition discovery, the extra information σ' discovered by abduction will be propagated back to the precondition to improve the precision.

The lifting function is applied for $\mathsf{abs_a}$ to lift both its domain and range to disjunctive abstract states $\mathcal{P}_{\mathsf{SH}}$:

$$\mathsf{abs_a}^{\dagger} \bigvee \sigma_i =_{df} \bigvee \mathsf{abs_a}(\sigma_i)$$

which allows it to be used in the analysis.

Example. Consider the abductive abstraction $\mathsf{abs_a}(\mathtt{x}{\mapsto}\mathtt{Node}(\mathtt{v_0}, \mathtt{p_0}) * \mathtt{p_0}{\mapsto}\mathtt{Node}(\mathtt{v_1}, \mathtt{p_1}))$ against the user-defined predicate `sls` for sorted singly linked list segments (see Appendix A for the predicate definition). The standard abstraction process would fail because the numerical relation between $\mathtt{v_0}$ and $\mathtt{v_1}$ is not available in the above formula. Such missing information can be found by abduction so that the above abductive abstraction would succeed and have the result as $\mathtt{sls}(\mathtt{x}, 2, \mathtt{v_0}, \mathtt{v_1}, \mathtt{p_1})$. A similar example can be found in the `merge` example illustrated in Sec 2.2.2 (the abduction (†)).

4.3. Join and Widening

Join operator. The operator join is applied over two conjunctive abstract states, trying to find a common shape as a sound abstraction for both:

```
$\mathsf{join}(\sigma_1, \sigma_2) =_{df}$
    let $\sigma_1', \sigma_2'$=rename($\sigma_1, \sigma_2$) in
    match $\sigma_1', \sigma_2'$ with $(\exists x_1^* \cdot \kappa_1 \wedge \pi_1), (\exists x_2^* \cdot \kappa_2 \wedge \pi_2)$ in
        if $\sigma_1 \vdash \sigma_2 * \mathtt{true}$ then $\exists x_1^*, x_2^* \cdot \kappa_2 \wedge (\mathsf{join}_\pi(\pi_1, \pi_2))$
        else if $\sigma_2 \vdash \sigma_1 * \mathtt{true}$ then $\exists x_1^*, x_2^* \cdot \kappa_1 \wedge (\mathsf{join}_\pi(\pi_1, \pi_2))$
        else $\sigma_1 \vee \sigma_2$
```

where the rename function avoids naming clashes among logical variables of σ_1 and σ_2, by renaming logical variables of the same name in the two states with fresh names. For example, it will renew $\mathtt{x_0}$'s name in both states $\exists \mathtt{x}_0 \cdot \mathtt{x}_0{=}0$ and $\exists \mathtt{x}_0 \cdot \mathtt{x}_0{=}1$ to make them $\exists \mathtt{x}_1 \cdot \mathtt{x}_1{=}0$ and $\exists \mathtt{x}_2 \cdot \mathtt{x}_2{=}1$. After this procedure it judges whether σ_2 is an abstraction of σ_1, or the other way round. If either case holds, it regards the shape of the weaker state as the shape of the joined states, and performs joining for pure formulae with $\mathsf{join}_\pi(\pi_1, \pi_2)$, the join operator over pure domains proposed in [31, 32]. Otherwise it keeps a disjunction of the two states (as it would be unsound to join their shapes together in this case).

For example, if we try $\mathsf{join}(\mathtt{x{=}null}, \mathtt{slsB}(\mathtt{x}, \mathtt{xmi}_0, \mathtt{xmx}_0, \mathtt{xS}_0, \mathtt{xp}_0))$, as the predicate `slsB` in second argument indicates the list contains at least one nodes, and the first argument does not have any shape, so we cannot join the two formulae. This join will keep a disjunction of the two arguments, and give $\mathtt{x{=}null} \vee \mathtt{slsB}(\mathtt{x}, \mathtt{xmi}_0, \mathtt{xmx}_0, \mathtt{xS}_0, \mathtt{xp}_0)$. If we try

$$\mathsf{join}(\mathtt{x}{\mapsto}\mathtt{Node}(\mathtt{xmi}_0, \mathtt{xp}_0), \mathtt{slsB}(\mathtt{x}, \mathtt{xmi}_0, \mathtt{xmx}_0, \mathtt{xS}_0, \mathtt{xp}_0)),$$

since $\mathtt{x}{\mapsto}\mathtt{Node}(\mathtt{xmi}_0, \mathtt{xp}_0)$ can be viewed as $\mathtt{slsB}(\mathtt{x}, \mathtt{xmi}_0, \mathtt{xmi}_0, \{\mathtt{xmi}_0\}, \mathtt{xp}_0)$, the two arguments then can be joined as

$$\mathtt{slsB}(\mathtt{x}, \mathtt{xmi}_0, \mathtt{xmx}_0, \mathtt{xS}_0, \mathtt{xp}_0) \wedge \mathtt{xmi}_0 {\leq} \mathtt{xmx}_0 \wedge \{\mathtt{xmi}_0\} {\sqsubseteq} \mathtt{xS}_0.$$

We lift this operator for the abstract state Δ as follows:

$$\mathsf{join}^\dagger(\Delta_1, \Delta_2) =_{df} \textbf{match } \Delta_1, \Delta_2 \textbf{ with } (\textstyle\bigvee_i \sigma_i^1),\ (\bigvee_j \sigma_j^2) \textbf{ in } \bigvee_{i,j} \mathsf{join}(\sigma_i^1, \sigma_j^2)$$

which essentially joins all pairs of disjunctions from the two abstract states, and makes a disjunction of them.

Widening operator. The finiteness of the shape domain is confirmed by the abstraction function. To ensure the termination of the whole analysis, we still need to guarantee the convergence over the pure domain. This task is accomplished by the widening operator, which is defined as:

```
$\mathsf{widen}(\sigma_1, \sigma_2) =_{df}$
  $\textbf{let } \sigma_1', \sigma_2' = \mathsf{rename}(\sigma_1, \sigma_2) \textbf{ in}$
  $\textbf{match } \sigma_1',\ \sigma_2' \textbf{ with } (\exists x_1^* \cdot \kappa_1 \wedge \pi_1),\ (\exists x_2^* \cdot \kappa_2 \wedge \pi_2) \textbf{ in}$
    $\textbf{if } \sigma_1 \vdash \sigma_2 * \mathtt{true} \textbf{ then } \exists x_1^*, x_2^* \cdot \kappa_2 \wedge (\mathsf{widen}_\pi(\pi_1, \pi_2))$
    $\textbf{else } \sigma_1 \vee \sigma_2$
```

In the widen operator, we expect that the second operand σ_2 to be weaker than the first σ_1, such that the widening reflects the trend of such weakening from σ_1 to σ_2. In this case, it returns the shape of σ_2 with the pure part by applying the widening operation $\mathsf{widen}_\pi(\pi_1, \pi_2)$ to the pure domain [31, 32]. For example,

$$\begin{array}{l}\mathsf{widen}(\exists \mathtt{xmx}_1, \mathtt{xS}_1 {\cdot} \mathtt{slsB}(\mathtt{x}, \mathtt{xmi}_0, \mathtt{xmx}_1, \mathtt{xS}_1, \mathtt{xp}_0) \wedge \mathtt{xv}{=}\mathtt{xmx}_0 \wedge \mathtt{xS}{=}\mathtt{xS}_1, \\ \quad \exists \mathtt{xmx}_2, \mathtt{xS}_2 {\cdot} \mathtt{slsB}(\mathtt{x}, \mathtt{xmi}_0, \mathtt{xmx}_2, \mathtt{xS}_2, \mathtt{xp}_0) \wedge \mathtt{xv}{\leq}\mathtt{xmx}_2 \wedge \mathtt{xS}{\sqsubseteq}\mathtt{xS}_2) := \\ \exists \mathtt{xmx}_0, \mathtt{xS}_0 {\cdot} \mathtt{slsB}(\mathtt{x}, \mathtt{xmi}_0, \mathtt{xmx}_0, \mathtt{xS}_0, \mathtt{xp}_0) \wedge \mathtt{xv}{\leq}\mathtt{xmx}_0 \wedge \mathtt{xS}{\sqsubseteq}\mathtt{xS}_0.\end{array}$$

We also lift the operator over (disjunctive) abstract states:

$$\mathsf{widen}^\dagger(\Delta_1, \Delta_2) =_{df} \textbf{match } \Delta_1, \Delta_2 \textbf{ with } (\textstyle\bigvee_i \sigma_i^1),\ (\bigvee_j \sigma_j^2) \textbf{ in} \bigvee_{i,j} \mathsf{widen}(\sigma_i^1, \sigma_j^2)$$

4.4. Soundness and Termination

The underlying operational semantics of our language follows from that given in [11, 30]. Its concrete program state consists of stack s and heap h, as described in Section 3. Chin et al. [11, 30] also defined the relation $s, h \models \Phi$ and the transitive relation $\langle s, h, e\rangle \hookrightarrow^* \langle s', h', \nu\rangle$. We first explain what we mean by the soundness of our analysis.

Definition 4.1 (Soundness). *Our analysis is sound if it returns sound pre- and post-conditions for the program under analysis. That is, for any pair of pre- and post-conditions (*Pre*,*Post*) returned by the analysis for a program* e*, the following holds: for all* s, h *such that* $s, h \models \mathit{Post}(\mathsf{Pre})$ *and* $\langle s, h, e\rangle \hookrightarrow^* \langle s', h', -\rangle$ *(for some* s', h'*), we have* $s', h' \models \mathit{Post}(\mathsf{Post})$*.*

Here the function $Post(\Delta)$ (defined in [11]) captures the relation between primed variables of Δ. For example, given $\Delta = \mathtt{x}'{\mapsto}\mathtt{node(3,null)} \wedge \mathtt{y{=}5} \wedge \mathtt{y'{>}y{+}1}$, $Post(\Delta) = \mathtt{x}{\mapsto}\mathtt{node(3,null)} \wedge \mathtt{y{>}6}$.

Theorem 4.1 (Soundness). *The analysis given in Fig 7 is sound, i.e., it returns sound pre- and post-conditions for the program under analysis.*

It follows the soundness of the entailment checking ([11]), the soundness of the abstract semantics, the soundness of the operations of abstraction, join and widening. The only place that the analysis may generate unsound results is the use of abstraction (over-approximation) in precondition inference, but any unsound results would have been filtered out by the post-check process via the sound HIP verification system ([11, 30]).

Proposition 4.1 (Sound entailment checking). *If $\Delta \vdash \Delta'$, we have for all s, h, if $s, h \models Post(\Delta)$, then $s, h \models Post(\Delta')$.*

The proof is done by a structural induction over formulae constructors of the abstract domain (details can be found in [11]).

Proposition 4.2 (Sound abstract semantics). *If $[\![e]\!]_{\mathcal{T}}\Delta = \Delta_1$, we have for all s, h, if $s, h \models Post(\Delta)$ and $\langle s, h, e\rangle \hookrightarrow \langle s', h', e'\rangle$, then there exists Δ', such that $s', h' \models Post(\Delta')$ and $[\![e']\!]_{\mathcal{T}}\Delta' = \Delta_1$.*

The proof is done by structural induction over program constructors for e. The detailed proof is given in the Appendix of the PhD thesis [18].

Proposition 4.3 (Sound abstraction). *Given an abstract state Δ, if $\mathsf{abs}(\Delta) = \Delta'$, then $\Delta \vdash \Delta'$.*

The proof for the first two substitution rules is trivial. For the third rule, what we need to prove is that $\mathtt{H(c)(x_0, v^*)} * \sigma \vdash \sigma * \mathtt{true}$, which follows immediately from that $\mathtt{H(c)(x_0, v^*)} \vdash \mathtt{true}$ and the frame rule of separation logic. For the fourth rule, the conclusion $\mathtt{H(c_1)(x, v_1^*)} * \sigma_1 * \sigma_3 \vdash \mathtt{p_2(x, v_2^*)} * \sigma_2 * \sigma_3$ follows from the entailment in the premise $\mathtt{H(c_1)(x, v_1^*)} * \sigma_1 \vdash \mathtt{p_2(x, v_2^*)} * \sigma_2$ and the frame rule of separation logic.

Proposition 4.4 (Sound join and widening). *The join and widening operators are sound. That is,*

- *For any σ_1 and σ_2, if $\mathsf{join}(\sigma_1, \sigma_2) = \sigma_j$, then we have both $\sigma_1 \vdash \sigma_j$ and $\sigma_2 \vdash \sigma_j$.*
- *For any σ_1 and σ_2, if $\mathsf{widen}(\sigma_1, \sigma_2) = \sigma_w$, then we have both $\sigma_1 \vdash \sigma_w$ and $\sigma_2 \vdash \sigma_w$.*

The proof is left in [18].

Theorem 4.2 (Termination). *Our analysis terminates in a finite number of steps for a finite number of input parameters, a finite number of shared cutpoints and a finite set of user-defined predicates.*

The convergence of our analysis is guaranteed by two facts: the finiteness of shape-only abstract states and the termination over the numerical and bag domain. The first fact is guaranteed by the finite number of shared cutpoints, program and logical variables and shape predicates. The second fact is ensured by the termination of the abstract interpretation frameworks for numerical domains [31, 32].

5. Experiments and Evaluation

We have implemented a prototype system and evaluated it over a number of heap-manipulating programs to test the viability and precision of our approach. We have so far focused our experiments on common textbook data structures, such as singly linked lists, doubly linked lists, binary search trees, nearly balanced trees (e.g. AVL), and typical operations manipulating them, such as creation, insertion, deletion, append, traverse, sorting etc. While these textbook programs are very easy to understand, they pose non-trivial difficulties for automated specification inference, especially when properties of interest require us to keep track of not only structural (shape) information, but also content (numerical and bag) information of those data structures, e.g. the Merge example illustrated earlier.

To facilitate the experimental study, we have defined a library of predicates covering common data structures and a variety of properties. These properties can be grouped in the following categories:

- MemSafe (*memory safety*): that all memory accesses are safe, and there are no dangling/null pointer dereferences;
- SameCont (*same content*): that the content of the final data structure (after manipulation) remains the same as that of the input data structure;
- Insert (*insertion*): that a given data node is inserted into the input data structure to form the final data structure;
- Sorted (*being sorted*): that data structures are sorted according to a criterion, eg. in case of a list each node's content is less than or equal to its successor's;
- BinarySearch (*binary search*): that data structures are binary search trees;
- DoubleLL (*double-linked list*): that data structures are double-linked lists; and
- AVLT (*AVL tree*): that data structures are AVL trees.

The predicates required as input by our prototype tool can be selected from our pre-defined library or can be supplied by users, according to the input program data structures and the properties of interest. Usually, the upper bound of cutpoints is set to be twice the number of input program variables to improve the precision. We present some of our results in Table 1 (the timing was achieved with an Intel Core 2 Quad CPU 2.66GHz with 8GB RAM). Note that instead of presenting directly the inferred pre/post- specifications, which contain too much detail thus will not fit into the table, we list the categories of properties involved.

We shall highlight two interesting aspects of our evaluation. The first observation concerns the precision of our analysis in comparison to previous approaches. Since our tool uses a combined domain it can discover more expressive specifications to guarantee memory safety and functional correctness. For example in case of the `take` program which traverses the list down for a user-specified number `n` of nodes, we can find that the length of the input list must be no less than `n`. However the previous tools based on shape domains (like Abductor [7]) can only discover a precondition that requires the input list to be non-empty which would not be sufficient to guarantee memory safety. Moreover more complex functional properties regarding the data structures content (like Sorted for `merge` program but in general for all sorting programs) also can not be discovered by the previous tools (like Abductor) based on a simple shape domain. There are other tools (like Xisa [9] or Thor [29]) that can work on a combined domain but require certain annotations to guide their analysis. Thor [29] requires shape information for each input parameter and Xisa [9] requires shape information for program variables used in loops. Since our shape domain includes tree data structures, our tool is able to discover complex functional specifications for binary search trees and AVL trees in contrast to the previous approaches. For example in case of the `flatten` program our tool is able to discover that the input data structure is a binary search tree while the output data structure is a sorted doubly linked list having the same data content (values stored inside the nodes) as that of the input.

The second observation regarding our experimental results is that the analysis may discover more than one correct specification for some programs. For example, given two predicates, ordinary linked list and sorted list, we can obtain two specifications for most of the sorting algorithms (thanks to our novel abductive abstraction mechanism). When there are more than one user-supplied predicate definitions, the analysis can have multiple choices during the abstraction. Multiple specifications can be useful in program verification, e.g. the sorted version for the `append` method, where the two input lists and the output list are all sorted, is useful in the verification of `quick_sort`, while the sorted list version for the `insert` method is also useful to help verify the functional correctness of `insert_sort`. This aspect of the evaluation also shows the important role that user-defined predicates play in our proposed analysis and demonstrates that the level of precision of the inferred results can be directly linked to the level of precision of the user-defined predicates that helped guide the specification inference.

Let us elaborate a bit more about this using the `append` method shown in Fig. 8. If the list segment predicate $\mathtt{ls}(\mathtt{x},\mathtt{p},\mathtt{n})$ (see Appendix A) is used to guide the abstraction, our analysis is able to discover the

Prog.	LOC	Time	Prop
Singly Linked List			
create	10	1.12	MemSafe
delete	9	1.20	MemSafe/Sorted
insert	9	1.16	MemSafe/Sorted/Insert
traverse	9	1.35	MemSafe/Sorted/SameCont
length	11	1.28	MemSafe/Sorted/SameCont
append	11	1.47	MemSafe/Sorted/SameCont
take	12	1.28	MemSafe/Sorted/SameCont
reverse	13	1.72	MemSafe/SameCont
filter	15	2.37	MemSafe/Sorted
Sorting algorithm			
insert_sort	32	2.72	MemSafe/SameCont/Sorted
merge_sort	78	4.18	MemSafe/SameCont/Sorted
quick_sort	70	5.72	MemSafe/SameCont/Sorted
select_sort	45	3.16	MemSafe/SameCont/Sorted
Doubly Linked List			
create	15	1.47	MemSafe/DoubleLL
append	24	2.53	MemSafe/DoubleLL/SameCont/Sorted
insert	22	2.32	MemSafe/DoubleLL/Insert/Sorted
Binary Search Tree			
create	18	2.58	MemSafe/BinarySearch
delete	48	4.76	MemSafe/BinarySearch
insert	22	3.57	MemSafe/BinarySearch/Insert
search	22	2.78	MemSafe/BinarySearch/SameCont
height	15	1.56	MemSafe/BinarySearch/SameCont
count	17	1.63	MemSafe/BinarySearch/SameCont
flatten	32	2.74	MemSafe/BinarySearch/DoubleLL/SameCont/Sorted
AVL Tree			
insert	114	27.57	MemSafe/BinarySearch/AVLT/Insert
delete	239	34.42	MemSafe/BinarySearch/AVLT

Table 1: Experimental Results. The column **LOC** is for the number of program lines; **Time** expresses our tool running time (in seconds); **Prop** denotes the inferred specification properties.

```
1 void append(Node x, Node y)
2 {
3   Node w;
4   w = x.next;
5   if (w == null) { x.next = y;}
6   else { append (w, y);}
7 }
```

Figure 8: appending two singly linked lists.

following specification:

$$\begin{array}{lcl} \mathsf{Pre}_1 & \equiv & \mathtt{ls}(\mathtt{x},\mathtt{null},\mathtt{n_x}) * \mathtt{ls}(\mathtt{y},\mathtt{null},\mathtt{n_y}) \wedge \mathtt{n_x}{>}0 \\ \mathsf{Post}_1 & \equiv & \mathtt{ls}(\mathtt{x},\mathtt{y},\mathtt{n_x}) * \mathtt{ls}(\mathtt{y},\mathtt{null},\mathtt{n_y}) \wedge \mathtt{n_x}{>}0 \end{array}$$

Note our analysis infers also the precondition $\mathtt{n_x}{>}0$ to ensure memory safety (or the field access `x.next` would dereference a null pointer if `x` is an empty list).

If the non-empty sorted list predicate $\mathtt{sllB}(\mathtt{x},\mathtt{mi},\mathtt{mx},\mathtt{S})$ (Sec 2.1) and the sorted list segment predicate $\mathtt{slsB}(\mathtt{x},\mathtt{mi},\mathtt{mx},\mathtt{S},\mathtt{p})$ (Sec 2.2.2) are available to help guide the abstraction process, our analysis discovers the following specification instead:

$$\begin{array}{lcl} \mathsf{Pre}_2 & \equiv & \mathtt{sllB}(\mathtt{x},\mathtt{mi_x},\mathtt{mx_x},\mathtt{S_x}) * \mathtt{sllB}(\mathtt{y},\mathtt{mi_y},\mathtt{mx_y},\mathtt{S_y}) \wedge \mathtt{mx_x}{\le}\mathtt{mi_y} \\ \mathsf{Post}_2 & \equiv & \mathtt{slsB}(\mathtt{x},\mathtt{mi_x},\mathtt{mx_x},\mathtt{S_x},\mathtt{y}) * \mathtt{sllB}(\mathtt{y},\mathtt{mi_y},\mathtt{mx_y},\mathtt{S_y}) \wedge \mathtt{mx_x}{\le}\mathtt{mi_y} \end{array}$$

It states that the two input lists are sorted and also that the maximum value stored in the first list is no bigger than the minimum value stored in the second list ($\mathtt{mx_x}{\le}\mathtt{mi_y}$). Under such an inferred precondition, the final data structure (including a list segment from `x` to `y` and a tail list started at `y`) remains sorted.

In summary, we have successfully evaluated the viability and precision of the proposed analysis over the combined domain via a number of common heap-allocated data structures, where structural, numerical and bag properties are taken into account in order to automatically infer program specifications involving a variety of properties of interest. Our evaluation has also confirmed that the novel abductive abstraction mechanism is useful to help discover program specifications at the desired level of abstraction, guided by user-defined predicates with an appropriate level of precision. We envisage more experimental studies to be conducted in future work on sizeable real-world code (e.g. Linux kernels) to further evaluate the modularity and scalability of our approach, once we have integrated the proposed analysis with more advanced shape analysis (e.g. [25]) that helps infer inductive predicates automatically.

6. Related Work and Conclusion

Dramatic advances have been made in synthesising specifications for heap-manipulating programs. The local shape analysis [13] infers loop invariants for list-processing programs, followed by the SpaceInvader/Abductor tool to infer full method specifications over the separation domain, so as to verify pointer safety for larger industrial codes [7, 42]. The SLAyer tool [14] implements an inter-procedural analysis for programs with shape information. A combination of shape and bag abstraction is used in [41] to verify linearizability. Calcagno et al. [8] describes an analysis for determining lock invariants with only linked lists. Lee et al. [27] presents a shape analysis specifically tailored to overlaid data structures. Compared with them, our abstraction is more general since it is driven by predicates and is not restricted to linked lists. To deal with size information (such as number of nodes in lists/trees), Thor [29] transfers a heap-processing program to a numerical one, so that size properties can be obtained by further analysis. A similar approach [15] combines a set domain (for shape) with its cardinality domain (for corresponding numerical information) in a more general framework. Compared with these works, our approach can discover specifications with stronger invariants such as sortedness and bag-related properties, which have not been addressed in the previous works. The analyses [9, 33, 34] can all handle shape and numerical information over a combined domain, but require user given preconditions for the program whereas here we compute the whole specification at once. Rival and Chang [37] propose an inductive predicate to summarise call stacks along with heap structures in a context of a whole-program analysis. In contrast our analysis is modular. Trinh et al. [40] proposes a bi-abductive analysis to infer pure information for pre/post specifications, while our bi-abductive reasoning is for the combined (separation and pure) domain. Recently, Le et al. [25] propose a modular specification inference via second-order bi-abductive entailment, which is able to infer method specifications (by resolving unknown predicates) from scratch or guided by user-supplied assertions. Their work is more powerful and general as it does not rely on user-defined predicates but it is currently for the shape domain only. Brotherston and Gorogiannis [6] propose a novel cyclic abduction to infer the safety and termination preconditions. However, their proposal is currently limited to a simple imperative language without procedures. More recently, Le et. al. [26] applies abductive reasoning to obtain a powerful modular analysis for termination and non-termination specifications, which is orthogonal to our analysis focused on memory safety and functional correctness. Our work uses the same programming language and specification language as the HIP/SLEEK system [11, 10], therefore shares the same underlying operational semantics and memory model. Different from the program verifier HIP which verifies program code against its given specifications, our focus here is a compositional analysis that automatically infers program specifications, which (in particular our bi-abductive reasoning) is built on top of the SLEEK entailment prover.

There are also other approaches that can synthesise shape-related program invariants. The shape analysis framework TVLA [39] is based on three-valued logic. It is capable of handling complicated data structures and properties, such as sortedness. Guo et al. [16] report a global shape analysis that discovers inductive structural shape invariants from the code. Kuncak et al. [23] develop a role system to express and track referencing relationships among objects. Hackett and Rugina [17] can deal with AVL-trees but is customised to handle only tree-like structures with height property. Bouajjani et al. [3, 4] propose a program analysis in an abstract domain with SL3 (Singly-Linked List Logic) and size, sortedness and multi-set properties. However, their heap domain is restricted to singly-linked list only, and their shape analysis is separated from numerical and mutli-set analyses. Compared with these works, separation logic based approaches benefit from the frame rule with support for local reasoning. The Forrester system [20] proposes a fully automated shape synthesis in terms of graph transformations over forest automata. Dillig et al. [12] present an analysis for constructing precise and compact method summaries. Unfortunately, both these works lack the ability to handle recursive data structures. In the matching logic framework, a set of predicates is typically assumed for program verification [38]. The work [1] extends this with specification inference. However, it currently does not deal with the inference of inductive data structure abstractions.

There are also approaches which unify reasoning over shape and data using either a combination of appropriate decision procedures inside Satisfiability-Modulo-Theories (SMT) solvers (e.g. [35, 24]) or a combination of appropriate abstract interpreters inside a software model checker (e.g. [2]). Compared with our work, their heap domains are mainly restricted to linked lists.

Concluding Remarks. We have reported a program analysis which automatically discovers program specifications over a combined separation and pure(numerical and bag) domain, guided by user-defined inductive predicates. The novel components of our analysis include an abductive abstract semantics and an abductive abstraction mechanism (for precondition discovery) in the combined domain. We have built a prototype system and the initial experimental results are encouraging.

In our analysis, the (abductive) abstraction process is guided by user-defined predicates. It works well when precise user-defined (inductive) predicates are available since such predicates can help guide the analysis to produce desirable results. In case such predicates are not available or weaker (than expected) predicates are given (e.g. `llB` instead of `sllB`), our analysis would be either unable to produce the specifications needed to ensure the memory safety and functional correctness of the program (e.g. in the `merge` example) or produce less precise specifications (e.g. in the append example). It remains an interesting future work to explore the possibility of extending a more powerful shape analysis, e.g. [25], to help infer precise inductive predicates automatically. Another future direction is to enhance the analysis procedure (e.g. with static debugging functionalities) so that it can offer users more information should the analysis fail. It would also be interesting to see if any machine learning techniques can be brought in to offer some help in such cases.

Acknowledgement. This work was supported in part by the EPSRC project EP/G042322/2 and the NSFC project 61373033.

References

[1] M. Alpuente, M. A. Feliú, and A. Villanueva. Automatic inference of specifications using matching logic. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 127–136, 2013.

[2] D. Beyer, T. A. Henzinger, and G. Théoduloz. Configurable software verification: Concretizing the convergence of model checking and program analysis. In *International Conference on Computer Aided Verification*, pages 504–518, 2007.

[3] A. Bouajjani, C. Dragoi, C. Enea, and M. Sighireanu. On inter-procedural analysis of programs with lists and data. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 578–589, 2011.

[4] A. Bouajjani, C. Dragoi, C. Enea, and M. Sighireanu. Abstract domains for automated reasoning about list-manipulating programs with infinite data. In *International Conf. on Verification, Model Checking, and Abstract Interpretation*, pages 1–22, 2012.

[5] M. Bozga, R. Iosif, and Y. Lakhnech. Storeless semantics and alias logic. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 55–65. ACM, 2003.

[6] J. Brotherston and N. Gorogiannis. Cyclic abduction of inductively defined safety and termination preconditions. In *Static Analysis Symposium*, pages 68–84. Springer, 2014.

[7] C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. *Journal of the ACM*, 58(6):26, 2011.

[8] C. Calcagno, D. Distefano, and V. Vafeiadis. Bi-abductive resource invariant synthesis. In *Asian Symposium on Programming Languages and Systems*, pages 259–274, 2009.

[9] B.-Y. E. Chang and X. Rival. Relational inductive shape analysis. In *ACM Symposium on Principles of Programming Languages*, pages 247–260. ACM, 2008.

[10] W.-N. Chin, C. David, and C. Gherghina. A HIP and SLEEK verification system. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pages 9–10, 2011.

[11] W.-N. Chin, C. David, H. H. Nguyen, and S. Qin. Automated verification of shape, size and bag properties via user-defined predicates in separation logic. *Sci. of Comp. Prog.*, 77:1006–1036, 2012.

[12] I. Dillig, T. Dillig, A. Aiken, and M. Sagiv. Precise and compact modular procedure summaries for heap manipulating programs. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 567–577, 2011.

[13] D. Distefano, P. W. O'Hearn, and H. Yang. A local shape analysis based on separation logic. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 287–302. Springer, 2006.

[14] A. Gotsman, J. Berdine, and B. Cook. Interprocedural shape analysis with separated heap abstractions. In *Static Analysis Symposium*, pages 240–260. Springer, 2006.

[15] S. Gulwani, T. Lev-Ami, and M. Sagiv. A combination framework for tracking partition sizes. In Zhong Shao and Benjamin C. Pierce, editors, *ACM Symposium on Principles of Programming Languages*, pages 239–251. ACM, 2009.

[16] B. Guo, N. Vachharajani, and D. I. August. Shape analysis with inductive recursion synthesis. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 256–265. ACM, 2007.

[17] B. Hackett and R. Rugina. Region-based shape analysis with tracked locations. In *ACM Symposium on Principles of Programming Languages*, pages 310–323, New York, NY, USA, 2005. ACM.

[18] Guanhua He. *Program Analysis in a Combined Abstract Domain.* PhD thesis, School of Engineering and Computing Sciences, Durham University, 2011.

[19] Guanhua He, Shengchao Qin, Wei-Ngan Chin, and Florin Craciun. Automated specification discovery via user-defined predicates. In *International Conference on Formal Engineering Methods*, pages 398–415. Springer, 2013.

[20] L. Holik, O. Lengál, A. Rogalewicz, J. Simácek, and T. Vojnar. Fully automated shape analysis based on forest automata. In *International Conference on Computer Aided Verification*, pages 740–755, 2013.

[21] S. S. Ishtiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. In *ACM Symposium on Principles of Programming Languages*, pages 14–26. ACM, 2001.

[22] H.B.M. Jonkers. Abstract storage structures. In *Algorithmic Languages.* North Holland, 1981.

[23] V. Kuncak, P. Lam, and M. C. Rinard. Role analysis. In *ACM Symposium on Principles of Programming Languages*, pages 17–32, 2002.

[24] S. K. Lahiri and S. Qadeer. Back to the future: revisiting precise program verification using smt solvers. In *ACM Symposium on Principles of Programming Languages*, pages 171–182, 2008.

[25] Q. L. Le, C. Gherghina, S. Qin, and W.-N. Chin. Shape analysis via second-order bi-abduction. In *International Conference on Computer Aided Verification*, pages 18–22, July 2014.

[26] T. C. Le, S. Qin, and W.-N. Chin. Termination and Non-Termination Specification Inference. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation.* ACM, 2015.

[27] O. Lee, H. Yang, and R. Petersen. Program analysis for overlaid data structures. In *International Conference on Computer Aided Verification*, 2011.

[28] S. Magill, M.-H. Tsai, P. Lee, and Y.-K. Tsay. Thor: A tool for reasoning about shape and arithmetic. In *International Conference on Computer Aided Verification*, pages 428–432. Springer, 2008.

[29] S. Magill, M.-H. Tsai, P. Lee, and Y.-K. Tsay. Automatic numeric abstractions for heap-manipulating programs. In *ACM Symposium on Principles of Programming Languages*, pages 211–222. ACM, 2010.

[30] H. H. Nguyen, C. David, S. Qin, and W.-N. Chin. Automated verification of shape and size properties via separation logic. In *International Conf. on Verification, Model Checking, and Abstract Interpretation*, pages 251–266. Springer, 2007.

[31] T.-H. Pham, M.-T. Trinh, A.-H. Truong, and W.-N. Chin. Fixbag: A fixpoint calculator for quantified bag constraints. In *International Conference on Computer Aided Verification*, pages 656–662. Springer, 2011.

[32] C. Popeea and W.-N. Chin. Inferring disjunctive postconditions. In *Asian Computing Science Conference*, pages 331–345. Springer, 2006.

[33] S. Qin, G. He, C. Luo, W.-N. Chin, and X. Chen. Loop invariant synthesis in a combined abstract domain. *Journal of Symbolic Computation*, 50:386–408, 2013.

[34] S. Qin, C. Luo, W.-N. Chin, and G. He. Automatically refining partial specifications for program verification. In *International Symposium on Formal Methods*, pages 369–385. Springer, 2011.

[35] Z. Rakamaric, R. Bruttomesso, A. J. Hu, and A. Cimatti. Verifying heap-manipulating programs in an smt framework. In *International Symposium on Automated Technology for Verification and Analysis*, pages 237–252, 2007.

[36] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *ACM/IEEE Symposium on Logic in*

Computer Science, pages 55–74. IEEE Computer Society, 2002.

[37] X. Rival and B.-Y. E. Chang. Calling context abstraction with shapes. In *ACM Symposium on Principles of Programming Languages*, pages 173–186, 2011.

[38] G. Rosu and A. Stefanescu. Checking reachability using matching logic. In *OOPSLA*, pages 555–574. ACM, 2012.

[39] M. Sagiv, T. W. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.*, 24(3):217–298, 2002.

[40] M.-T. Trinh, Q. L. Le, C. David, and W.-N. Chin. Bi-abduction with pure properties for specification inference. In *11th Asian Symposium on Programming Languages and Systems*, pages 107–123, 2013.

[41] V. Vafeiadis. Shape-value abstraction for verifying linearizability. In *International Conf. on Verification, Model Checking, and Abstract Interpretation*, pages 335–348, 2009.

[42] H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. W. O'Hearn. Scalable shape analysis for systems code. In *International Conference on Computer Aided Verification*, pages 385–398. Springer, 2008.

Appendix A. Shape Predicate Definitions

Below are the definitions of those shape predicates used in the experiments but not given in paper:

$$\mathtt{ls(root,p,n)} \equiv (\mathtt{root{=}p} \wedge \mathtt{n{=}0}) \vee (\mathtt{root{\mapsto}Node(\_,q)} * \mathtt{ls(q,p,m)} \wedge \mathtt{n{=}m{+}1})$$

(singly linked list segment with size information)

$$\mathtt{lsB(root,S,p)} \equiv (\mathtt{root{=}p} \wedge \mathtt{S{=}\{\}}) \vee (\mathtt{root{\mapsto}Node(v,q)} * \mathtt{ls(q,S_1,p)} \wedge \mathtt{S{=}S_1{\sqcup}\{v\}})$$

(singly linked list segment with multi-set of contents)

$$\begin{aligned}\mathtt{dll(root,p,n)} \equiv\ & (\mathtt{root{=}p} \wedge \mathtt{n{=}0}) \vee \\ & (\mathtt{root{\mapsto}Node2(v,p,q)} * \mathtt{dll(q,root,n_1)} \wedge \mathtt{n{=}n_1{+}1})\end{aligned}$$

(doubly linked list with size information, where
data Node2 {int val; Node2 prev; Node2 next; })

$$\begin{aligned}\mathtt{dllB(root,p,S)} \equiv\ & (\mathtt{root{=}p} \wedge \mathtt{S{=}\emptyset}) \vee \\ & (\mathtt{root{\mapsto}Node2(v,p,q)} * \mathtt{dllB(q,root,S_1)} \wedge \mathtt{S{=}S_1{\sqcup}\{v\}})\end{aligned}$$

(doubly linked list with bag/multi-set information)

$$\begin{aligned}\mathtt{dlsB(root,pr,pi,po,S)} \equiv\ & (\mathtt{root{=}po} \wedge \mathtt{pr{=}pi} \wedge \mathtt{S{=}\emptyset}) \vee \\ & (\mathtt{root{\mapsto}Node2(v,pr,q)} * \mathtt{dlsB(q,root,pi,po,S_1)} \wedge \mathtt{S{=}S_1{\sqcup}\{v\}})\end{aligned}$$

(doubly linked list segment with bag/multi-set information)

$$\begin{aligned}\mathtt{sll(root,n,mn,mx)} \equiv\ & (\mathtt{root{=}null} \wedge \mathtt{n{=}0} \wedge \mathtt{mn{=}mx}) \vee \\ & (\mathtt{root{\mapsto}Node(mn,q)} * \mathtt{sll(q,n_1,k,mx)} \wedge \mathtt{n{=}n_1{+}1} \wedge \mathtt{mn{\leq}k})\end{aligned}$$

(sorted singly linked list with minimum and maximum values)

$$\begin{aligned}\mathtt{sls(root,n,mn,mx,p)} \equiv\ & (\mathtt{root{=}p} \wedge \mathtt{n{=}0} \wedge \mathtt{mn{=}mx}) \vee \\ & (\mathtt{root{\mapsto}Node(mn,q)} * \mathtt{sls(q,n_1,k,mx,p)} \wedge \mathtt{n{=}n_1{+}1} \wedge \mathtt{mn{\leq}k})\end{aligned}$$

(sorted singly linked list segment with minimum and maximum values)

$$\begin{aligned}\mathtt{sllB2(root,S)} \equiv\ & (\mathtt{root{=}null} \wedge \mathtt{S{=}\emptyset}) \vee \\ & (\mathtt{root{\mapsto}Node(v,q)} * \mathtt{sllB2(q,S_1)} \wedge \mathtt{S{=}\{v\}{\sqcup}S_1} \wedge (\forall \mathrm{x}{\in}\mathrm{S}_1{\cdot}\mathrm{v}{\geq}\mathrm{x}))\end{aligned}$$

(sorted singly linked list with bag/multi-set information)

$$\begin{aligned}\mathtt{slsB2(root,p,S)} \equiv\ & (\mathtt{root{=}p} \wedge \mathtt{S{=}\emptyset}) \vee \\ & (\mathtt{root{\mapsto}Node(v,q)} * \mathtt{slsB2(q,p,S_1)} \wedge \mathtt{S{=}\{v\}{\sqcup}S_1} \wedge (\forall \mathrm{x}{\in}\mathrm{S}_1{\cdot}\mathrm{v}{\leq}\mathrm{x}))\end{aligned}$$

(sorted singly linked list segment with bag/multi-set information)

$$\begin{aligned}\mathtt{sdllB(root,p,S)} \equiv\ & (\mathtt{root{=}p} \wedge \mathtt{S{=}\emptyset}) \vee \\ & (\mathtt{root{\mapsto}Node2(v,p,q)} * \mathtt{sdllB(q,root,S_1)} \wedge \mathtt{S{=}S_1{\sqcup}\{v\}}) \\ & \wedge \mathrm{S}{=}\{\mathtt{v}\}{\sqcup}\mathrm{S}_1 \wedge (\forall \mathrm{x}{\in}\mathrm{S}_1{\cdot}\mathrm{v}{\leq}\mathrm{x}))\end{aligned}$$

(sorted doubly linked list with bag/multi-set information)

$$\begin{aligned}\mathtt{sdlsB(root,pr,pi,po,S)} \equiv\ & (\mathtt{root{=}po} \wedge \mathtt{pr{=}pi} \wedge \mathtt{S{=}\emptyset}) \vee \\ & (\mathtt{root{\mapsto}Node2(v,pr,q)} * \mathtt{sdlsB(q,root,pi,po,S_1)} \wedge \mathtt{S{=}S_1{\sqcup}\{v\}}) \\ & \wedge \mathrm{S}{=}\{\mathtt{v}\}{\sqcup}\mathrm{S}_1 \wedge (\forall \mathrm{x}{\in}\mathrm{S}_1{\cdot}\mathrm{v}{\leq}\mathrm{x}))\end{aligned}$$

(sorted doubly linked list segment with bag/multi-set information)

$$\begin{aligned}
\mathtt{bt(root,S,h)} \equiv & (\mathtt{root{=}null} \wedge \mathtt{S}{=}\emptyset \wedge \mathtt{h{=}0}) \vee \\
& (\mathtt{root}{\mapsto}\mathtt{Node3(v,p,q)} * \mathtt{bt(p,S_p,h_p)} * \mathtt{bt(q,S_q,h_q)} \\
& \wedge \mathtt{S{=}S_p{\sqcup}S_q} \wedge \mathtt{h{=}1{+}max(h_p,h_q)}) \\
& \text{(binary tree with bag/multi-set and height information, where} \\
& \quad \texttt{data Node3 \{int val; Node3 left; Node3 right;\}})
\end{aligned}$$

$$\begin{aligned}
\mathtt{bst(root,mn,mx)} \equiv & (\mathtt{root{=}null} \wedge \mathtt{mn{=}mx}) \vee \\
& (\mathtt{root}{\mapsto}\mathtt{Node3(v,p,q)} * \mathtt{bst(p,mn,mx_l)} * \mathtt{bst(q,mn_r,mx)} \wedge \mathtt{mx_l{\le}v{\le}mn_r}) \\
& \text{(binary search tree with min/max information)}
\end{aligned}$$

$$\begin{aligned}
\mathtt{bstB(root,S)} \equiv & (\mathtt{root{=}null} \wedge \mathtt{S}{=}\emptyset) \vee \\
& (\mathtt{root}{\mapsto}\mathtt{Node3(v,p,q)} * \mathtt{bstB(p,S_p)} * \mathtt{bstB(q,S_q)} \wedge \\
& \quad \mathtt{S{=}\{v\}{\sqcup}S_p{\sqcup}S_q} \wedge \forall \mathtt{v_p{\in}S_p, v_q{\in}S_q} \cdot \mathtt{v_p{\le}v{\le}v_q}) \\
& \text{(binary search tree with bag/multi-set information)}
\end{aligned}$$

$$\begin{aligned}
\mathtt{avl(root,S,h)} \equiv & (\mathtt{root{=}null} \wedge \mathtt{S}{=}\emptyset \wedge \mathtt{h{=}0}) \vee \\
& (\mathtt{root}{\mapsto}\mathtt{Node3(v,p,q)} * \mathtt{avl(p,S_p,h_p)} * \mathtt{avl(q,S_q,h_q)} \wedge \\
& \quad \mathtt{S{=}S_p{\sqcup}S_q} \wedge \mathtt{h{=}1{+}max(h_p,h_q)} \wedge -1{\le}\mathtt{h_p{-}h_q}{\le}1) \\
& \text{(AVL tree with min/max and height information)}
\end{aligned}$$

$$\begin{aligned}
\mathtt{rbt(root,S,c,h)} \equiv & (\mathtt{root{=}null} \wedge \mathtt{S}{=}\emptyset \wedge \mathtt{c{=}0} \wedge \mathtt{h{=}0}) \vee \\
& (\mathtt{root}{\mapsto}\mathtt{Node3(v,p,q)} * \mathtt{rbt(p,S_p,c_p,h_p)} * \mathtt{rbt(q,S_q,c_q,h_q)} \wedge \\
& \quad \mathtt{S{=}S_p{\sqcup}S_q} \wedge ((\mathtt{c{=}0} \wedge \mathtt{h{=}h_p{+}1} \wedge \mathtt{h_p{=}h_q}) \vee (\mathtt{c{=}1} \wedge \mathtt{c_p{=}c_q{=}0} \wedge \mathtt{h{=}h_p{=}h_q})) \\
& \text{(red-black tree with bag/multi-set, colour, height information)}
\end{aligned}$$

Note: c=0 denotes the color of node is black and c=1 denotes red node.

Appendix B. Selected Discovered Specifications

We list below some of the discovered specifications during our experiments.

Singly Linked List

`Node create(int n)`
Pre ::= $0 \le \mathtt{n}$
Post ::= $\mathtt{ls(res,null,n)}$

`Node delete(Node l, int v)`
Pre ::= $\mathtt{sllB(l,mi,mx,S)}$
Post ::= $\mathtt{sllB(res,mi_1,mx_1,S_1)} \wedge (\mathtt{res} = \mathtt{l} \wedge \mathtt{mi_1} = \mathtt{mi} \wedge \mathtt{mx_1} \le \mathtt{mx} \vee \mathtt{res} \ne \mathtt{l} \wedge \mathtt{mi} \le \mathtt{mi_1} \wedge \mathtt{mx_1} = \mathtt{mx})$
$\wedge (\mathtt{S} = \mathtt{S_1} \sqcup \{\mathtt{v}\} \vee \mathtt{S} = \mathtt{S_1})$

`Node insert(Node l, Node x)`

Pre ::= $\mathtt{sllB}(\mathtt{l},\mathtt{mi},\mathtt{mx},\mathtt{S}) * \mathtt{x}{\mapsto}\mathtt{Node}(\mathtt{v},\mathtt{p})$

Post ::= $\mathtt{res}{\mapsto}\mathtt{Node}(\mathtt{v},\mathtt{l}) * \mathtt{sllB}(\mathtt{l},\mathtt{mi},\mathtt{mx},\mathtt{S}) \wedge \mathtt{res} = \mathtt{x} \wedge \mathtt{v} \le \mathtt{mi}$

$\quad\vee\ \mathtt{slsB}(\mathtt{res},\mathtt{mi},\mathtt{mx}_1,\mathtt{S}_1,\mathtt{x}) * \mathtt{sllB}(\mathtt{x},\mathtt{v},\mathtt{mx}_2,\mathtt{S}_2) \wedge \mathtt{res} = \mathtt{l} \wedge \mathtt{mx}_1 < \mathtt{v} \wedge \mathtt{v} \le \mathtt{mx}_2 \wedge \mathtt{S}_1 \sqcup \mathtt{S}_2 = \mathtt{S} \sqcup \{\mathtt{v}\}$

`void traverse(Node l)`

Pre ::= $\mathtt{sllB}(\mathtt{l},\mathtt{mi},\mathtt{mx},\mathtt{S})$

Post ::= $\mathtt{sllB}(\mathtt{l},\mathtt{mi},\mathtt{mx},\mathtt{S})$

`int length(Node l)`

Pre ::= $\mathtt{sllB}(\mathtt{l},\mathtt{mi},\mathtt{mx},\mathtt{S})$

Post ::= $\mathtt{sllB}(\mathtt{l},\mathtt{mi},\mathtt{mx},\mathtt{S}) \wedge \mathtt{res} = |\mathtt{S}|$

`void append(Node x, Node y)`

Pre_1 ::= $\mathtt{ls}(\mathtt{x},\mathtt{null},\mathtt{n}_\mathtt{x}) * \mathtt{ls}(\mathtt{y},\mathtt{null},\mathtt{n}_\mathtt{y}) \wedge \mathtt{n}_\mathtt{x}{>}0$

Post_1 ::= $\mathtt{ls}(\mathtt{x},\mathtt{y},\mathtt{n}_\mathtt{x}) * \mathtt{ls}(\mathtt{y},\mathtt{null},\mathtt{n}_\mathtt{y})$

Pre_2 ::= $\mathtt{sllB}(\mathtt{x},\mathtt{mi}_\mathtt{x},\mathtt{mx}_\mathtt{x},\mathtt{S}_\mathtt{x}) * \mathtt{sllB}(\mathtt{y},\mathtt{mi}_\mathtt{y},\mathtt{mx}_\mathtt{y},\mathtt{S}_\mathtt{y}) \wedge \mathtt{mx}_\mathtt{x} \le \mathtt{mi}_\mathtt{y}$

Post_2 ::= $\mathtt{slsB}(\mathtt{x},\mathtt{mi}_\mathtt{x},\mathtt{mx}_\mathtt{x},\mathtt{S}_\mathtt{x},\mathtt{y}) * \mathtt{sllB}(\mathtt{y},\mathtt{mi}_\mathtt{y},\mathtt{mx}_\mathtt{y},\mathtt{S}_\mathtt{y}) \wedge \mathtt{mx}_\mathtt{x} \le \mathtt{mi}_\mathtt{y}$

`Node take(Node l, int n)`

Pre ::= $\mathtt{sllB}(\mathtt{l},\mathtt{mi},\mathtt{mx},\mathtt{S}) \wedge |\mathtt{S}| \le \mathtt{n}$

Post ::= $\mathtt{slsB}(\mathtt{l},\mathtt{mi},\mathtt{mx}_1,\mathtt{S}_1,\mathtt{res}) * \mathtt{sllB}(\mathtt{res},\mathtt{mi}_2,\mathtt{mx},\mathtt{S}_2) \wedge \mathtt{S} = \mathtt{S}_1 \sqcup \mathtt{S}_2 \wedge \mathtt{mx}_1 \le \mathtt{mi}_2 \wedge |\mathtt{S}_1| = \mathtt{n}$

`Node reverse(Node l)`

Pre ::= $\mathtt{lsB}(\mathtt{l},\mathtt{S},\mathtt{null}) \wedge |\mathtt{S}| > 0$

Post ::= $\mathtt{lsB}(\mathtt{res},\mathtt{S}_1,\mathtt{l}) * \mathtt{l}{\mapsto}\mathtt{Node}(\mathtt{v}_1,\mathtt{null}) \wedge \mathtt{S} = \mathtt{S}_1 \sqcup \{\mathtt{v}_1\}$

`Node filter(Node x, int k)`

Pre ::= $\mathtt{sllB}(\mathtt{x},\mathtt{mi},\mathtt{mx},\mathtt{S})$

Post ::= $\mathtt{sllB}(\mathtt{res},\mathtt{mi}_1,\mathtt{mx}_1,\mathtt{S}_1) \wedge (\forall \mathtt{v}{\in}\mathtt{S}_1{\cdot}\mathtt{v} \le \mathtt{k}){\wedge}(\forall \mathtt{v}{\in}(\mathtt{S}{-}\mathtt{S}_1){\cdot}\mathtt{v}{>}\mathtt{k}){\wedge}\mathtt{S}_1{\sqsubseteq}\mathtt{S}$

Sorting algorithm

`void insert_sort(Node l)`

Pre ::= $\mathtt{lsB}(\mathtt{l},\mathtt{null},\mathtt{S})$

Post ::= $\mathtt{slsB2}(\mathtt{l},\mathtt{null},\mathtt{S})$

void merge_sort(Node l)
Pre ::= $\mathtt{lsB}(\mathtt{l},\mathtt{null},\mathsf{S})$
Post ::= $\mathtt{slsB2}(\mathtt{l},\mathtt{null},\mathsf{S})$

void quick_sort(Node l)
Pre ::= $\mathtt{lsB}(\mathtt{l},\mathtt{null},\mathsf{S})$
Post ::= $\mathtt{slsB2}(\mathtt{l},\mathtt{null},\mathsf{S})$

void select_sort(Node l)
Pre ::= $\mathtt{lsB}(\mathtt{l},\mathtt{null},\mathsf{S})$
Post ::= $\mathtt{slsB2}(\mathtt{l},\mathtt{null},\mathsf{S})$

Double Linked List

Node2 create(int n)
Pre ::= $0 \le \mathtt{n}$
Post ::= $\mathtt{dll}(\mathtt{res},\mathtt{null},\mathtt{n})$

void append(Node2 x, Node2 y)
Pre ::= $\mathtt{sdllB}(\mathtt{x},\mathtt{p_x},\mathtt{mi_x},\mathtt{mx_x},\mathsf{S_x}) * \mathtt{sdllB}(\mathtt{y},\mathtt{p_y},\mathtt{mi_y},\mathtt{mx_y},\mathsf{S_y}) \wedge \mathtt{mx_x} \le \mathtt{mi_y}$
Post ::= $\mathtt{sdlsB}(\mathtt{x},\mathtt{p_x},\mathtt{mi_x},\mathtt{mx_x},\mathsf{S_x},\mathtt{o_x},\mathtt{y}) * \mathtt{sdllB}(\mathtt{y},\mathtt{o_x},\mathtt{mi_y},\mathtt{mx_y},\mathsf{S_y}) \wedge \mathtt{mx_x}$

void insert(Node2 l, int v)
Pre ::= $\mathtt{sdllB}(\mathtt{l},\mathtt{p},\mathtt{mi},\mathtt{mx},\mathsf{S})$
Post ::= $\mathtt{sdllB}(\mathtt{l},\mathtt{p},\mathtt{mi_1},\mathtt{mx_1},\mathsf{S_1}) \wedge \mathtt{mi_1} \le \mathtt{mi} \wedge \mathtt{mx} \le \mathtt{mx_1} \wedge \mathsf{S_1} = \mathsf{S} \sqcup \{\mathtt{v}\}$

Binary Search Tree

Node3 create(int n)
Pre ::= $0 \le \mathtt{n}$
Post ::= $\mathtt{bstB}(\mathtt{res},\mathsf{S})$

void delete(Node3 t, int v)
Pre ::= $\mathtt{bstB}(\mathtt{t},\mathsf{S})$
Post ::= $\mathtt{bstB}(\mathtt{t},\mathsf{S_1}) \wedge \mathsf{S} = \mathsf{S_1} \vee \mathsf{S} = \mathsf{S_1} \sqcup \{\mathtt{v}\}$

`void insert(Node3 t, int v)`
Pre ::= $\mathtt{bstB}(\mathtt{t}, \mathtt{S})$
Post ::= $\mathtt{bstB}(\mathtt{t}, \mathtt{S}_1) \wedge \mathtt{S}_1 = \mathtt{S} \sqcup \{\mathtt{v}\}$

`int height(Node3 t)`
Pre ::= $\mathtt{bt}(\mathtt{t}, \mathtt{S}, \mathtt{h})$
Post ::= $\mathtt{bt}(\mathtt{t}, \mathtt{S}, \mathtt{h}) \wedge \mathtt{res} = \mathtt{h}$

`int count(Node3 t)`
Pre ::= $\mathtt{bstB}(\mathtt{t}, \mathtt{S})$
Post ::= $\mathtt{bstB}(\mathtt{t}, \mathtt{S}) \wedge \mathtt{res} = |\mathtt{S}|$

`Node3 flatten(Node3 t)`
Pre ::= $\mathtt{bstB}(\mathtt{t}, \mathtt{S})$
Post ::= $\mathtt{sdlsB}(\mathtt{res}, \mathtt{null}, \mathtt{t}, \mathtt{o}_\mathtt{r}, \mathtt{S}_\mathtt{r}) * \mathtt{sdllB}(\mathtt{t}, \mathtt{o}_\mathtt{r}, \mathtt{S}_\mathtt{t}) \wedge \mathtt{S} = \mathtt{S}_\mathtt{r} \sqcup \mathtt{S}_\mathtt{t}$

AVL Tree

`void insert(Node3 avl, int v)`
Pre ::= $\mathtt{avl}(\mathtt{t}, \mathtt{S}, \mathtt{h})$
Post ::= $\mathtt{avl}(\mathtt{t}, \mathtt{S}_1, \mathtt{h}_1) \wedge \mathtt{S}_1 = \mathtt{S} \sqcup \{\mathtt{v}\} \wedge \mathtt{h} \leq \mathtt{h}_1 \leq \mathtt{h} + 1$

`void delete(Node3 avl, int v)`
Pre ::= $\mathtt{avl}(\mathtt{t}, \mathtt{S}, \mathtt{h})$
Post ::= $\mathtt{avl}(\mathtt{t}, \mathtt{S}_1, \mathtt{h}_1) \wedge (\mathtt{S}_1 = \mathtt{S} \vee \mathtt{S} = \mathtt{S}_1 \sqcup \{\mathtt{v}\}) \wedge \mathtt{h} - 1 \leq \mathtt{h}_1 \leq \mathtt{h}$