Automated Specification Discovery in a Combined Abstract Domain

Guanhua He[1], Shengchao Qin[1], Wei-Ngan Chin[2] and Chenguang Luo[3]

[1] Teesside University, Middlesbrough TS1 3BA, UK
[2] National University of Singapore
[3] Citigroup Inc.
S.Qin@tees.ac.uk, G.He@tees.ac.uk, chinwn@comp.nus.edu.sg

Abstract. Discovering program specifications automatically for heap-manipulating programs is a challenging task due to the complexity of aliasing and mutability of data structures used. This paper describes a compositional analysis framework for discovering program specifications in a combined abstract domain with shape, numerical and bag (multi-set) information. The framework analyses each method and derives its summary independently from its callers. We propose a novel abstraction method with a bi-abduction technique in the combined domain to discover pre/post-conditions which cannot be automatically inferred before. The analysis does not only prove the 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 automatic program analysis, certain kinds of program properties have been well explored over the last decades, such as numerical properties in linear abstraction domain, and shape properties for list-manipulating programs in separation domain. However, previous works are not sufficient to analyse automatically program properties in complex mixed domains, especially for programs with sophisticated data structures and strong invariants involving both structural and pure information. Examples of such properties include, that a list remains sorted during the execution of a program, that a binary search tree is balanced before and after the execution of a program procedure, and that the elements of a list remain unchanged after reversing the list. The difficulty is not only due to sharing and mutability of data structures under manipulation, but also the need to track often closely intertwined program properties, such as structural numerical information (length and height), symbolic contents of data structures (bag of values stored in a tree), and relational numerical information (sortedness and balancedness).

In addition to classical shape analyses (e.g. [1, 5, 13, 27]), separation logic [12, 24] has been applied to analyse shape properties of heap-manipulating programs

in recent years [2, 6, 28]. These works can automatically infer method specifications for shape properties of programs. A more recent work THOR [15, 16] also incorporated simple numerical information into the shape domain to allow automated synthesis of properties like data structure size information.

However, these previous analyses mainly focus on relatively simple properties, such as pointer safety for lists and list length information. To analyse more complex properties of heap-manipulating programs, such as sortedness and balancedness properties, our recent work [22] offers a template-based approach, whereby users supply shape templates in pre-/post-conditions of procedures and the analysis infers the missing pure information to complete the given templates. While that approach does not require an analysis in the combined domain, one of its limitations is it relies on users to supply the pre-/post-shape templates. If the supplied templates do not cover all the required heap portion, or are not precise enough, or even are essentially unsound for the program (an extreme example being {`true`} Prog {`false`}), it will fail to discover the full specifications.

To overcome these limitations, in this paper we propose a direct one-pass analysis in a combined abstraction with *separation*, *numerical* and *bag* information. To the best of our knowledge, this is the first time where such a combination of domains have been used together for inferring pre/post specifications. One advantage of doing so is that we do not only analyse functional correctness and memory safety separately, but also find close relationships between shape and pure (numerical and bag) domains. What we propose is a compositional analysis by abstract interpretation in such a combined domain. That is, we analyse a program fragment without any given contextual information, and we analyse each method in a modular way independent of its callers. To generate pre-/postconditions, our analysis adopts a new bi-abduction mechanism over the combined domain, which extends the bi-abduction technique proposed by Calcagno et al. [2]. As in our previous work, our analysis allows users to supply creatively defined shape predicates, while it does not require users to supply partial specifications or annotations for program code to be analysed.

In summary, this paper makes the following contributions:

- We have designed a program analysis framework which can discover program pre/post-conditions (involving heap, numerical and bag properties) automatically without being given any contextual information about the program.
- For such framework, we have described a compositional analysis for abstract interpretation in a combined pure and shape domain.
- We have defined novel operations for abstraction, join and widening with an extended bi-abduction technique over the combined domain.
- We have conducted some initial experiments. These experimental results help us confirm the viability and precision of our solution in finding non-trivial program specifications.

Outline. The rest of the paper is structured as follows. We first illustrate our approach informally via two examples (Section 2), and then give our programming and specification languages (Section 3) as well as our abduction mechanism

(Section 4). Formal details about specification discovery are presented in Section 5, followed by initial experimental results in Section 6. Lastly, we present related work and some concluding remarks.

2 The Approach

In the section we will first introduce our specification mechanism, followed by illustrative examples of our analysis.

2.1 Separation Logic and User-defined Predicates

Separation logic [24] extends Hoare logic to support reasoning about shared mutable data structures. It provides separation conjunction ($*$) to form formulae like $\mathtt{p_1}*\mathtt{p_2}$ to assert that two heaps described by $\mathtt{p_1}$ and $\mathtt{p_2}$ are domain-disjoint. In our analysis, we use this approach to allow user-defined inductive predicates to specify both separation and pure properties. For example, with a data structure definition for a node in a list `data node { int val; node next; }`, we can define a predicate for a list with its content as

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

The parameter `root` for the predicate `llB` is the root pointer referring to the list. Its length is denoted by `n`, and content is `S`. A uniform notation $\mathtt{p}::\mathtt{c}\langle \mathtt{v}^*\rangle$ is used for either a singleton heap or a predicate. If `c` is a data node, the notation represents a singleton heap, $\mathtt{p}{\mapsto}\mathtt{c}[\mathtt{v}^*]$, e.g. the $\mathtt{root}::\mathtt{node}\langle \mathtt{v}, \mathtt{q}\rangle$ above. If `c` is a predicate name, then the data structure pointed to by `p` has the shape `c` with parameters $\mathtt{v}^*$, e.g., the $\mathtt{q}::\mathtt{llB}\langle \mathtt{n_1}, \mathtt{S_1}\rangle$ above.

If users want to verify a sorting algorithm, they can incorporate sortedness property as follows:

$$\begin{array}{l}\mathtt{sllB}\langle \mathtt{S}\rangle \equiv (\mathtt{root}{=}\mathtt{null}\wedge \mathtt{S}{=}\emptyset)\vee \\ \quad (\mathtt{root}::\mathtt{node}\langle \mathtt{v}, \mathtt{q}\rangle * \mathtt{q}::\mathtt{sllB}\langle \mathtt{S_1}\rangle \wedge \mathtt{S}{=}\mathtt{S_1}\sqcup\{\mathtt{v}\}\wedge \forall \mathtt{u}{\in}\mathtt{S_1}\cdot \mathtt{v}{\leq}\mathtt{u})\end{array}$$

where we use the following shortened notation: (i) default `root` parameter on the left hand side may be omitted, (ii) unbound variables, such as `q`, `v` and $\mathtt{S_1}$, are implicitly existentially quantified. Later we may also use the underscore _ to denote an existentially quantified anonymous variable.

2.2 Illustrative Examples

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

```
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 } }
```

Fig. 1. Filtering the elements of a list.

Our analysis aims at finding sound and precise specification (summary) (Pre, Post) of the method. Before the analysis, we use a pair (emp, 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 of states (Pre, Curr) to keep trace of the precondition we discovered and the current state we reached, respectively. If the current precondition is not sufficient to operate the program command, we use an abductive inference mechanism to synthesise a candidate precondition M as the missing precondition. At the beginning of the analysis, this pair is (emp, 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 ($\mathtt{emp}\wedge\mathtt{x}=\mathtt{x}'\wedge\mathtt{k}=\mathtt{k}'$, emp) before line 2 in the first iteration, where we use primed variables to keep track of the current value of program variables, and unprimed variables to keep the initial value in the precondition. (Since the value of k is not changed during the program execution, we omit $\mathtt{k}=\mathtt{k}'$ in the presentation.) The branch from line 5 to line 13 is "short-circuited" in the first iteration, as the current Post of the method (false) is applied as the effect of the recursive call. To enter line 4, the condition x == null needs to be satisfied with precondition. We apply abduction mechanism and discover x=null to add to precondition. After executing return x, we have an initial summary of the method:

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

where res denotes the value returned by the program.

In the second iteration, the specification (1) is updated as a new summary of the method. The starting point of the analysis is reset to (emp, emp). By executing the branch in line 4, we have the same result as summary (1). The expression x.val tries to access a field of x, by abduction, $\mathtt{x}{::}\mathtt{Node}\langle\mathtt{fv}_0, \mathtt{fp}_0\rangle$ is added to Pre, where $\mathtt{fv}_0$ and $\mathtt{fp}_0$ are fresh logical variables. After line 8, the paired state is ($\mathtt{x}{::}\mathtt{Node}\langle\mathtt{fv}_0, \mathtt{fp}_0\rangle\wedge\mathtt{fv}_0\leq\mathtt{k}\wedge\mathtt{x}=\mathtt{x}'$, $\mathtt{x}{::}\mathtt{Node}\langle\mathtt{fv}_0, \mathtt{fp}_0\rangle\wedge\mathtt{t}=\mathtt{fp}_0\wedge\mathtt{fv}_0\leq\mathtt{k}\wedge\mathtt{x}=\mathtt{x}'$). Now we can use the summary (1) for the method call, which requires t=null, i.e. $\mathtt{fp}_0$=null to be added to Pre and returns $\mathtt{t}=\mathtt{fp}_0$ to be added to Curr. Note that $\mathtt{x}{::}\mathtt{Node}\langle\mathtt{fv}_0, \mathtt{fp}_0\rangle$ 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 $\mathtt{fp}_0$ is a reachable variable from program variable x, we add $\mathtt{fp}_0$=null to

Pre, instead of $\mathtt{t}{=}\mathtt{null}$. After line 8, the summary of the branch from lines 5 to 8 is found:

$$(\mathtt{x}{::}\mathtt{Node}\langle \mathtt{fv}_0, \mathtt{null}\rangle{\wedge}\mathtt{fv}_0{\leq}\mathtt{k}{\wedge}\mathtt{x}{=}\mathtt{x}', \mathtt{res}{::}\mathtt{Node}\langle \mathtt{fv}_0, \mathtt{null}\rangle{\wedge}\mathtt{fv}_0{\leq}\mathtt{k}{\wedge}\mathtt{res}{=}\mathtt{x}'{\wedge}\mathtt{x}{=}\mathtt{x}') \tag{2}$$

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

$$(\mathtt{x}{::}\mathtt{Node}\langle \mathtt{fv}_0, \mathtt{null}\rangle{\wedge}\mathtt{fv}_0{>}\mathtt{k}{\wedge}\mathtt{x}{=}\mathtt{x}',\ \mathtt{emp}{\wedge}\mathtt{fv}_0{>}\mathtt{k}{\wedge}\mathtt{res}{=}\mathtt{null}{\wedge}\mathtt{x}'{=}\mathtt{null}) \tag{3}$$

By joining the formulae (1), (2) and (3), and eliminating intermediate logical variables, we update a new summary for the method

$$\begin{array}{l}(\mathsf{Pre}_2, \mathsf{Post}_2) := (\mathsf{Pre}_1 \vee \mathtt{x}{::}\mathtt{Node}\langle \mathtt{fv}_0, \mathtt{null}\rangle{\wedge}\mathtt{x}{=}\mathtt{x}', \\ \mathsf{Post}_1{\vee}\mathtt{emp}{\wedge}\mathtt{fv}_0{>}\mathtt{k}{\wedge}\mathtt{res}{=}\mathtt{null}{\wedge}\mathtt{x}'{=}\mathtt{null}{\vee}\mathtt{res}{::}\mathtt{Node}\langle \mathtt{fv}_0, \mathtt{null}\rangle{\wedge}\mathtt{fv}_0{\leq}\mathtt{k}{\wedge}\mathtt{res}{=}\mathtt{x}')\end{array} \tag{4}$$

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

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

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

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

Following a similar symbolic execution process with the method summary (6), we compute a result for the fourth iteration to be the same as the last one, namely (6), which is a fixed point desired for our method summary. Note that we eliminate $\mathtt{x}'$ as an existentially quantified variable in the postcondition, since $\mathtt{x}$ 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, we shall use a forward analysis process to check the discovered summary.

Example of Merge. Another motivating example we shall present is `merge`, which has been declared as an unverified example in [3], since their method do not keep track of values stored in the list. This example (Figure 2) merges two sorted lists into one sorted list. If either of the input list is empty, then the other list is returned; if not, we select the smallest element of both sorted lists, and make the next field of the smallest node points to the result of merging the tail list of this node with the other list. The shape predicate selected for this example is `sls` which keeps track on both the minimal (`sm`) and maximal (`lg`) values of a sorted list.

$$\begin{aligned}&\texttt{sls}\langle \texttt{n}, \texttt{sm}, \texttt{lg}, \texttt{p}\rangle \equiv \texttt{root::Node}\langle \texttt{sm}, \texttt{p}\rangle \wedge \texttt{n=1} \wedge \texttt{lg=sm} \ \vee \\ &\quad \texttt{root::Node}\langle \texttt{sm}, \texttt{q}\rangle * \texttt{q::sls}\langle \texttt{n}_1, \texttt{s}_1, \texttt{lg}, \texttt{p}\rangle \wedge \texttt{n=n}_1\texttt{+1} \wedge \texttt{sm} \leq \texttt{s}_1 \wedge \texttt{s}_1 \leq \texttt{lg}\end{aligned}$$

```
1  Node merge(Node x, Node y)
2  {
3    if (x == null) {
4      return y;
5    } else if (y == null) {
6      return x;
7    } else
8    if (x.val <= y.val) {
9      Node t = x.next;
10     x.next = merge(t, y);
11     return x;
12   } else {
13     Node t = y.next;
14     y.next = merge(x, t);
15     return y;
16 } }
```

Fig. 2. Merging two sorted lists.

Suppose after the third iteration of symbolically executing the code, we have generated a precondition, as follows:

$$\begin{aligned}&\texttt{x=null} \vee \texttt{y=null} \vee \texttt{x::Node}\langle \texttt{xv}_1, \texttt{xp}_1\rangle * \texttt{y::Node}\langle \texttt{yv}_1, \texttt{yp}_1\rangle \\ &\quad \wedge (\texttt{xv}_1 \leq \texttt{yv}_1 \wedge \texttt{xp}_1\texttt{=null} \vee \texttt{xv}_1 > \texttt{yv}_1 \wedge \texttt{yp}_1\texttt{=null}) \qquad (7) \\ &\vee \texttt{x::Node}\langle \texttt{xv}_1, \texttt{xp}_1\rangle * \texttt{xp}_1\texttt{::Node}\langle \texttt{xv}_2, \texttt{xp}_2\rangle * \texttt{y::Node}\langle \texttt{yv}_1, \texttt{yp}_1\rangle \\ &\quad \wedge (\texttt{xv}_1 \leq \texttt{yv}_1 \wedge (\texttt{xv}_2 \leq \texttt{yv}_1 \wedge \texttt{xp}_2\texttt{=null} \vee \texttt{xv}_2 > \texttt{yv}_1 \wedge \texttt{yp}_1\texttt{=null})) \qquad (8) \\ &\vee \texttt{x::Node}\langle \texttt{xv}_1, \texttt{xp}_1\rangle * \texttt{y::Node}\langle \texttt{yv}_1, \texttt{yp}_1\rangle * \texttt{yp}_1\texttt{::Node}\langle \texttt{yv}_2, \texttt{yp}_2\rangle \\ &\quad \wedge (\texttt{xv}_1 > \texttt{yv}_1 \wedge (\texttt{xv}_1 \leq \texttt{yv}_2 \wedge \texttt{xp}_1\texttt{=null} \vee \texttt{xv}_1 > \texttt{yv}_2 \wedge \texttt{yp}_2\texttt{=null})) \qquad (9)\end{aligned}$$

Branch (8) says that the program only touches the second node of `x` if $\texttt{xv}_1 \leq \texttt{yv}_1$. If $\texttt{xv}_2 \leq \texttt{yv}_1$, $\texttt{xp}_2$ should be `null`; otherwise $\texttt{yp}_1$ must be `null` to guarantee the termination of the method and memory safety. Branch (9) states a similar condition when touching the second node of `y`. This formula is very precise, but not scalable if the analysis is continued. According to the given user-defined predicate `sls`, we could abstract the shapes of `x` and `y` to be a sorted list. However, the formula is not sufficient to do that, i.e. the sortedness information about `x` and `y` is missing. This missing information is the numerical relation between $\texttt{xv}_1$ and $\texttt{xv}_2$ in `x`, and $\texttt{yv}_1$ and $\texttt{yv}_2$ in `y`. The guidance for this abstraction comes from the predicate `sls`. We use such user-defined predicates to infer data structure properties that are anticipated from some program codes. By applying abstraction (equipped with abduction mechanism) against the predicate `sls` and then

joining the branches with the same shape, the precondition from two iterations becomes:

$$\mathtt{x{=}null} \vee \mathtt{y{=}null} \vee \mathtt{x{::}sls}\langle \mathtt{xn}_0, \mathtt{xsm}_0, \mathtt{xlg}_0, \mathtt{xp}_0\rangle * \mathtt{y{::}sls}\langle \mathtt{yn}_0, \mathtt{ysm}_0, \mathtt{ylg}_0, \mathtt{yp}_0\rangle \\ \wedge\, 1{\leq}\mathtt{xn}_0{\leq}2 \wedge 1{\leq}\mathtt{yn}_0{\leq}2 \wedge (\mathtt{xlg}_0{\leq}\mathtt{ylg}_0 \wedge \mathtt{xp}_0{=}\mathtt{null} \vee \mathtt{xlg}_0{>}\mathtt{ylg}_0 \wedge \mathtt{yp}_0{=}\mathtt{null})$$

Continuing the analysis, the fixed point of the program summary is

$$(\mathsf{Pre}, \mathsf{Post}) := (\mathtt{x{=}null} \vee \mathtt{y{=}null} \vee \mathtt{x{::}sls}\langle \mathtt{xn}_0, \mathtt{xsm}_0, \mathtt{xlg}_0, \mathtt{xp}_0\rangle * \\ \mathtt{y{::}sls}\langle \mathtt{yn}_0, \mathtt{ysm}_0, \mathtt{ylg}_0, \mathtt{yp}_0\rangle \wedge (\mathtt{xlg}_0{\leq}\mathtt{ylg}_0 \wedge \mathtt{xp}_0{=}\mathtt{null} \vee \mathtt{xlg}_0{>}\mathtt{ylg}_0 \wedge \mathtt{yp}_0{=}\mathtt{null}), \\ \mathtt{x{=}null} \wedge \mathtt{res{=}y} \vee \mathtt{y{=}null} \wedge \mathtt{res{=}x} \vee \mathtt{x{::}sls}\langle \mathtt{xn}_1, \mathtt{xsm}_1, \mathtt{xlg}_1, \mathtt{xp}_1\rangle \\ * \mathtt{y{::}sls}\langle \mathtt{yn}_1, \mathtt{ysm}_1, \mathtt{ylg}_1, \mathtt{yp}_1\rangle \wedge \mathtt{xn}_1{+}\mathtt{yn}_1{=}\mathtt{xn}_0{+}\mathtt{yn}_0 \wedge \mathtt{xsm}_1{=}\mathtt{xsm}_0 \wedge \mathtt{ysm}_1{=}\mathtt{ysm}_0 \wedge \\ (\mathtt{xsm}_0{\leq}\mathtt{ysm}_0 \wedge \mathtt{res{=}x} \wedge \mathtt{xp}_1{=}\mathtt{y} \wedge \mathtt{xlg}_1{\leq}\mathtt{ysm}_1 \vee \mathtt{xsm}_0{>}\mathtt{ysm}_0 \wedge \mathtt{res{=}y} \wedge \mathtt{yp}_1{=}\mathtt{x} \wedge \mathtt{ylg}_1{\leq}\mathtt{xsm}_1)$$

From this example, we can observe that the memory safety of shape analysis is related to the values stored in the list. Our analysis can find that only one list is traversed to its end, i.e. until `null` is reached, and the other 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 smaller element.

3 Language and Abstract Domain

To simplify presentation, we employ a strongly-typed C-like imperative language in Figure 3 to demonstrate our approach. The 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 `sls`), as well as method declarations *meth*. Note that we allow methods to have no specifications, which are discovered by our analysis then. The definitions for *spred* and *mspec* are given later in Figure 4. 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. The language allows both call-by-value and call-by-reference method parameters, separated with a semicolon (;).

Prog	::= *tdecl** *meth**	*tdecl* ::= *datat* | *spred*	
datat	::= `data` c { *field** }	*field* ::= $t\ v$	t ::= $c \mid \tau$
meth	::= $t\ mn\ ((t\ v)^*; (t\ v)^*)$	*mspec** $\{e\}$	τ ::= `int` | `bool` | `void`
e	::= $d \mid d[v] \mid v{:=}e \mid e_1; e_2 \mid t\ v;\ e \mid$ `if` $(v)\ e_1$ `else` e_2		
d	::= `null` $\mid k^\tau \mid v \mid$ `new` $c(v^*) \mid mn(u^*; v^*)$		
$d[v]$	::= $v.f \mid v.f{:=}w \mid$ `free`(v)		

Fig. 3. A Core (C-like) Imperative Language.

Our specification language (in Figure 4) allows (user-defined) shape predicates *spred* to specify program properties in our combined domain. Note that *spred* are constructed with disjunctive constraints Φ. We require that the predicates be well-formed [17]. A conjunctive abstract program state σ has mainly two parts: the heap (shape) part κ in separation domain and the pure part π in convex polyhedra domain and bag (multi-set) domain, where π consists of γ, ϕ and φ as aliasing, numerical and multi-set information, respectively. The set of all σ's 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 [17].

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

Fig. 4. The Specification Language.

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

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

where $\mathtt{R}$ 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 is similar to the model given for separation logic [24], except that we have extensions to handle user-defined shape predicates and related pure properties [17]. In our analysis, all the variables except the program ones are logical variables. We denote a program variable's initial value as unprimed and its current value as primed [17].

4 Bi-Abduction

In this section we introduce our bi-abduction algorithm for discovering missing information in precondition. The bi-abduction extends previous works [2, 7, 23] with more power to work over our combined domain.

Given σ and σ_1, bi-abduction aims to find the anti-frame σ' and frame part σ_2 such that

$$\sigma * [\sigma'] \rhd \sigma_1 * \sigma_2$$

where σ and σ_1 can be considered as the current program state and the requirement of next instruction, respectively, σ' is the missing part which will be

$$\frac{\sigma \not\vdash \sigma_1 * \mathtt{true} \quad \sigma_1 \vdash \sigma * \sigma' \quad \sigma * \sigma' \vdash \sigma_1 * \sigma_2}{\sigma * [\sigma'] \rhd \sigma_1 * \sigma_2}\ \textbf{Residue}$$

$$\frac{\sigma \not\vdash \sigma_1 * \mathtt{true} \quad \sigma_1 \not\vdash \sigma * \mathtt{true} \quad \sigma_0 \in \mathsf{unroll}(\sigma) \quad \mathsf{data\_no}(\sigma_0) \leq \mathsf{data\_no}(\sigma_1) \quad \sigma_0 \vdash \sigma_1 * \sigma' \text{ or } \sigma_0 * [\sigma_0'] \rhd \sigma_1 * \sigma' \quad \sigma * \sigma' \vdash \sigma_1 * \sigma_2}{\sigma * [\sigma'] \rhd \sigma_1 * \sigma_2}\ \textbf{Unroll}$$

$$\frac{\sigma \not\vdash \sigma_1 * \mathtt{true} \quad \sigma_1 \not\vdash \sigma * \mathtt{true} \quad \sigma_1 * [\sigma_1'] \rhd \sigma * \sigma' \quad \sigma * \sigma' \vdash \sigma_1 * \sigma_2}{\sigma * [\sigma'] \rhd \sigma_1 * \sigma_2}\ \textbf{Reverse}$$

$$\frac{\sigma \not\vdash \sigma_1 * \mathtt{true} \quad \sigma_1 \not\vdash \sigma * \mathtt{true} \quad \sigma * \sigma_1 \vdash \sigma_1 * \sigma_2}{\sigma * [\sigma_1] \rhd \sigma_1 * \sigma_2}\ \textbf{Missing}$$

Fig. 5. Bi-Abduction rules.

propagated back to the precondition and make the analysis continue, and frame part σ_2 is the residue from σ. The bi-abduction rules are exhibited in Figure 5.

The first rule `Residue` triggers when the LHS (σ) does not imply the RHS (σ_1) but the RHS implies the LHS with some formula (σ') as the residue. This rule is quite general and applies in many cases. For the example $\mathtt{emp} \not\vdash \mathtt{x}{::}\mathtt{Node}\langle \mathtt{xv}, \mathtt{xp}\rangle$, the RHS can entail the LHS with residue $\mathtt{x}{::}\mathtt{Node}\langle \mathtt{xv}, \mathtt{xp}\rangle$. The abduction then checks whether σ plus the frame information σ' implies $\sigma_1 * \sigma_2$ for some σ_2 (`emp` in this example), and returns $\mathtt{x}{::}\mathtt{Node}\langle \mathtt{xv}, \mathtt{xp}\rangle$ as the anti-frame.

The second rule `Unroll` deals with neither side implies the other, e.g. for $\mathtt{x}{::}\mathtt{sls}\langle \mathtt{xn}, \mathtt{xsm}, \mathtt{xlg}, \mathtt{null}\rangle$ as LHS and $\exists \mathtt{p}, \mathtt{u}, \mathtt{v} \cdot \mathtt{x}{::}\mathtt{node}\langle \mathtt{u}, \mathtt{p}\rangle * \mathtt{p}{::}\mathtt{node}\langle \mathtt{v}, \mathtt{null}\rangle$ as RHS. As the shape predicates in the antecedent σ are formed by disjunctions according to their definitions (like `sls`), 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 $\sigma \wedge \sigma' \vdash \sigma_1 * \sigma_2$. For the example above, the abduction returns $\mathtt{xn}{=}2$ as the anti-frame σ' and discovers the nontrivial frame $\mathtt{u}{=}\mathtt{xsm} \wedge \mathtt{v}{=}\mathtt{xlg} \wedge \mathtt{u}{\leq}\mathtt{v}$ (σ_2). Note the function `data_no` returns the number of data nodes in a state, e.g. it returns one for $\mathtt{x}{::}\mathtt{node}\langle \mathtt{v}, \mathtt{p}\rangle * \mathtt{p}{::}\mathtt{llB}\langle \mathtt{n}, \mathtt{T}\rangle$. (This syntactic check is important for the termination of the abduction.) 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 ($\{\sigma_1, ..., \sigma_n\}$).

In the third rule `Reverse`, neither side entails the other, and the second rule does not apply, for example $\exists \mathtt{p}, \mathtt{u}, \mathtt{v}, \mathtt{q} \cdot \mathtt{x}{::}\mathtt{node}\langle \mathtt{u}, \mathtt{p}\rangle * \mathtt{p}{::}\mathtt{node}\langle \mathtt{v}, \mathtt{q}\rangle$ as LHS and $\exists \mathtt{S} \cdot \mathtt{x}{::}\mathtt{sls}\langle 3, \mathtt{xsm}, \mathtt{xlg}, \mathtt{xp}\rangle$ as RHS. In this case the antecedent cannot be unfolded as they are already 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'. Then it checks that the LHS (σ), with σ' added, does imply the RHS (σ_1) before it returns σ'. For the example above, the anti-frame is $\mathtt{q}{::}\mathtt{Node}\langle \mathtt{w}, \mathtt{xp}\rangle \wedge \mathtt{u}{\leq}\mathtt{v} \wedge \mathtt{v}{\leq}\mathtt{w}$ to establish the three node sorted list.

When an abduction is conducted, the first three rules should be tried first; if they do not succeed in finding a solution, then the last rule `Missing` is invoked to add the consequence to the antecedent, provided that they are consistent. It is effective for situations like $\mathtt{x}{::}\mathtt{node}\langle \_, \_\rangle \not\vdash \mathtt{y}{::}\mathtt{node}\langle \_, \_\rangle$, where we should add $\mathtt{y}{::}\mathtt{node}\langle \_, \_\rangle$ to the LHS directly (as the other three rules do not apply here).

The abduction procedures presented in earlier work [23, 22] have mainly focused on discovering pure information with the assumption that either complete or partial shape information is available. Our bi-abduction algorithm presented in this paper generalises them 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; 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 too many solutions of the anti-frame σ' for the entailment $\sigma_1 * \sigma' \vdash \sigma_2 * \texttt{true}$ to succeed. Therefore, we define "quality" of anti-frame solutions with the partial order $\preceq$ defined in the last section, i.e. the smaller (weaker) one in two abduction solutions is regarded as better. 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.

5 Analysis Algorithm

Our proposed analysis algorithm is given in Figure 6. The algorithm takes three input parameters: $\mathcal{T}$ as the program environment possibly with some of the method specifications in the program and ready for newly inferred specifications to be added in, the procedure to be analysed $t\ mn\ ((t\ x)^*; (t\ y)^*)\ \{e\}$, and the upper bound of shared logical variables that we keep during the analysis n.

Our analysis is based on the abstract interpretation framework. It has two distinct features: the abduction exploited in the abstract semantics, and the specifically designed operations ($\mathsf{abs_a}$, join and widen) over this combined domain.[4] 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 $\texttt{emp}$ as the initial precondition because we know nothing about the footprint of the code. The initial postcondition is set to $\texttt{false}$ denotes the top element of the lattice of our abstraction domain.

Each iteration starts at line 1. Firstly we calculate the pre- and post-conditions for the program based on the last iteration's result, with a forward analysis using the abduction-based semantics (line 3). We perform abstraction on both pre- and postconditions obtained to preserve shape domain's finiteness. If the symbolic execution cannot continue due to a program bug, or if we find our abstraction 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 5). Otherwise the obtained results are joined with the results from last iteration (line 6), and a widening is conducted over both to ensure termination of the analysis (line 8).

[4] Note that our analysis uses lifted versions of these operations (indicated by †), which will be explained in more details later.

Fixpoint Computation in 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   repeat
2     i := i + 1;
3     (Pre_i, Post_i) := [[e]]^A_T(emp, emp);
4     (Pre_i, Post_i) := (abs_a^†(Pre_i), abs^† Post_i);
5     if Pre_i=false or Post_i=false or cp_no(Pre_i)>n or cp_no(Post_i)>n
      then return fail end if
6     Pre_i := join^†(Pre_{i-1}, Pre_i);
7     Post_i := join^†(Post_{i-1}, Post_i);
8     Pre_i := widen^†(Pre_{i-1}, Pre_i);
9     Post_i := widen^†(Post_{i-1}, Post_i);
10  until (Pre_i, Post_i) = (Pre_{i-1}, Post_{i-1})
11  T' := T ∪ {t mn ((t x)*;(t y)*) requires Pre_i ensures Post_i {e}};
12  Post = [[e]]_{T'} Pre_i;
13  if Post = false or Post ⊬ Post_i * true then return fail
14  else return T'
15  end if
```

Fig. 6. Main analysis algorithm.

Finally we judge whether a fixed-point is already reached by comparing the current abstract state with the one from previous iteration (line 10). The last few lines (from line 11) are for soundness purposes. We will run a forward analysis over the method body with the discovered specifications to see whether they are sound. If so then the analysis succeeds; otherwise fail is returned.

As aforementioned, the kernel of our analysis include the abstract semantics with abduction, and the abstraction, join and widening operators, which are elaborated respectively in what follows.

5.1 Abstract Semantics

As shown in the algorithm, we use two kinds of abstract semantics to analyse the program: an abstract semantics with abduction to derive the specification for the program (line 3), and another underlying semantics to ensure soundness for the analysis result (line 11). As the first semantics is in fact based on the second one, we will first define the second semantics followed by the first one.

The type of our underlying semantics is defined as

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

where AllSpec contains procedure specifications (extracted from the program *Prog*). For some expression e, given its precondition, the semantics will calculate the postcondition.

The foundation of the semantics is the basic transition functions from a conjunctive abstract state (σ) to a conjunctive or disjunctive abstract state (σ or Δ) below:

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

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 ($x{::}c\langle v^*\rangle$), 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:

$$
\frac{\mathit{isdatat}(c) \quad \sigma \vdash x{::}c\langle v^*\rangle * \sigma'}{\mathsf{unfold}(x)\sigma \rightsquigarrow \sigma}
\qquad
\frac{\mathit{isspred}(c) \quad \sigma \vdash x{::}c\langle v^*\rangle * \sigma' \quad \mathtt{root}{::}c\langle v^*\rangle{\equiv}\Phi}{\mathsf{unfold}(x)\sigma \rightsquigarrow \sigma' * [x/\mathtt{root}, u^*/v^*]\Phi}
$$

The test $\mathit{isdatat}(c)$ returns `true` only if c is a data node and $\mathit{isspred}(c)$ returns `true` only if c is a shape predicate.

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 the execution:

$$
\frac{\mathit{isdatat}(c) \qquad \sigma \vdash x{::}c\langle v_1, .., v_n\rangle * \sigma'}{\mathsf{exec}(x.f_i)(\mathcal{T})\sigma \rightsquigarrow \sigma' * x{::}c\langle v_1, .., v_n\rangle \wedge \mathtt{res}{=}v_i}
$$

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

$$
\frac{\mathit{isdatat}(c) \qquad \sigma \vdash x{::}c\langle u^*\rangle * \sigma'}{\mathsf{exec}(\mathtt{free}(x))(\mathcal{T})\sigma \rightsquigarrow \sigma'}
$$

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

$$
\mathsf{exec}(k)(\mathcal{T})\sigma \rightsquigarrow \sigma \wedge \mathtt{res}{=}k
\qquad\qquad
\mathsf{exec}(x)(\mathcal{T})\sigma \rightsquigarrow \sigma \wedge \mathtt{res}{=}x
$$

$$
\frac{\mathit{isdatat}(c)}{\mathsf{exec}(\mathtt{new}\ c(v^*))(\mathcal{T})\sigma \rightsquigarrow \sigma * \mathtt{res}{::}c\langle v^*\rangle}
$$

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

Note that 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. 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 $\rho_{ol} \circ \rho_o\Phi_{po}$.

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 abstract semantics for a program e as follows (where loops are already translated into tail-recursions):

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

Next we define the abstract semantics with abduction used in our analysis, whose type is

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

It takes a piece of program and a specification table, to map a (disjunctive) set of pair of symbolic heaps to another such set (where the first in the pair is the accumulated precondition and the second is the current state).

This semantics also consists of the basic transition functions which compose the atomic instructions' semantics and then the program constructors' semantics. Here the basic transition functions are lifted as

$$\begin{array}{l}
\mathsf{Unfold}(x)(\sigma', \sigma) =_{df} \\
\quad \textbf{let } \Delta{=}\mathsf{unfold}(x)\sigma \text{ and } \mathsf{S}{=}\{(\sigma', \sigma_1) \mid \sigma_1 \in \Delta\} \\
\quad \textbf{in if } (\mathtt{false} \notin \Delta) \textbf{ then } \mathsf{S} \\
\quad\quad \textbf{else if } (\Delta \vdash x{=}a \text{ for some } a{\in}\mathsf{SVar}) \text{ and} \\
\quad\quad\quad\quad (\sigma' \not\vdash a{::}c\langle y^*\rangle * \mathtt{true} \text{ for fresh } \{y^*\}{\subseteq}\mathsf{LVar}) \\
\quad\quad \textbf{then } \mathsf{S} \cup \{(\sigma' * x{::}c\langle y^*\rangle, \Delta * x{::}c\langle y^*\rangle)\} \\
\quad\quad \textbf{else } \ \mathsf{S} \cup \{(\sigma', \mathtt{false})\}
\end{array}$$

$$\begin{array}{l}
\mathsf{Exec}(ds)(\sigma', \sigma) =_{df} \textbf{let } \sigma_1{=}\mathsf{exec}(ds)\sigma \textbf{ in } \{(\sigma', \sigma_1) \mid \sigma_1 \in \Delta\} \\
\quad \text{where } ds \text{ is either } d[x] \text{ or } d \text{ when } d \text{ is not method call}
\end{array}$$

$$\frac{\begin{array}{c}
t\ mn\ ((t_i\ u_i)_{i=1}^{m}; (t_i\ v_i)_{i=1}^{n})\ \textit{requires}\ \Phi_{pr}\ \textit{ensures}\ \Phi_{po} \in \mathcal{T} \\
\rho = [x'_i/u_i]_{i=1}^{m} \circ [y'_i/v_i]_{i=1}^{n} \qquad \sigma * [\sigma'_1] \rhd \rho\Phi_{pr} * \sigma_1 \\
\rho_o = [r_i/v_i]_{i=1}^{n} \circ [x'_i/u'_i]_{i=1}^{m} \circ [y'_i/v'_i]_{i=1}^{n} \quad \rho_l = [r_i/y'_i]_{i=1}^{n} \quad \textit{fresh logical } r_i
\end{array}}{\begin{array}{l}
\mathsf{Exec}(mn(x_1, .., x_m; y_1, .., y_n))(\mathcal{T})(\sigma, \sigma') =_{df} \\
\qquad \{(\sigma_2, \rho_o(\sigma' * \sigma'_1)) \mid \sigma_2 \in (\rho_l\sigma_1) * (\rho_o\Phi_{po})\}
\end{array}}$$

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

$$\mathsf{Unfold}^{\dagger}(x)\bigvee(\sigma'_i,\sigma_i) \quad =_{df} \bigvee(\mathsf{Unfold}(x)(\sigma'_i,\sigma_i))$$
$$\mathsf{Exec}^{\dagger}(ds)(\mathcal{T})\bigvee(\sigma'_i,\sigma_i) =_{df} \bigvee(\mathsf{Exec}(ds)(\mathcal{T})(\sigma'_i,\sigma_i))$$

Based on the above transition functions, the abstract semantics with abduction is as follows:

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

where for assignment we apply the substitution on both abstract states in the pair.

5.2 Abstraction, Join and Widening

We have specifically designed the abstraction, join and widening operations employed in our analysis process.

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, or 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:

$$\begin{array}{l}
\mathsf{abs}(\mathtt{x}{::}\mathtt{node}\langle \mathtt{v}_1,\mathtt{z}_0\rangle * \mathtt{z}_0{::}\mathtt{node}\langle \mathtt{v}_2,\mathtt{null}\rangle) \ =\ \mathtt{x}{::}\mathtt{llB}\langle \mathtt{n},\mathtt{S}\rangle \wedge \mathtt{n}{=}2 \wedge \mathtt{S}=\{\mathtt{v}_1,\mathtt{v}_2\}\\
\mathsf{abs}(\mathtt{x}{::}\mathtt{node}\langle \mathtt{v}_1,\mathtt{z}_0\rangle * \mathtt{y}{::}\mathtt{node}\langle \mathtt{v}_2,\mathtt{z}_0\rangle * \mathtt{z}_0{::}\mathtt{node}\langle \mathtt{v}_3,\mathtt{null}\rangle) \ =\ \text{-}
\end{array}$$

where both $\mathtt{x}$ and $\mathtt{y}$ are program variables, and $\mathtt{z}_0$ is an existentially quantified logical variable. In the second case $\mathtt{z}_0$ is a shared cutpoint referenced by both $\mathtt{x}$ and $\mathtt{y}$, 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:

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

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

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

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

$$\frac{\begin{array}{cc} isdatat(c_1) & c_2\langle u_2^*\rangle \equiv \Phi \\ p{::}c_1\langle v_1^*\rangle * \sigma_1 \vdash p{::}c_2\langle v_2^*\rangle * \sigma_2 & \mathsf{Reach}(p{::}c_2\langle v_2^*\rangle * \sigma_2) \cap \{v_1^*\} = \emptyset \end{array}}{\mathsf{abs}(p{::}c_1\langle v_1^*\rangle * \sigma_1) =_{df} p{::}c_2\langle v_2^*\rangle * \sigma_2}$$

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 (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 $\mathtt{x}_0$ in $\mathtt{x}{::}\mathtt{node}\langle \_, \mathtt{null}\rangle * \mathtt{x}_0{::}\mathtt{node}\langle \_, \mathtt{null}\rangle$ 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 c_2 is a predicate definition. The predicate c_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 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:

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

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. Note that the pure logic parameters can be abstracted since the pure relations are kept in pure formulae, so we do not loss 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.

It might have already been noticed that we use a specifically designed abstraction with abduction for precondition ($\mathsf{abs}_{\mathsf{a}}$) which is used to discover extra obligations for the abstraction of precondition. It consists of two rules:

$$\mathsf{abs}_{\mathsf{a}}(\sigma) =_{df} \mathsf{abs}(\sigma)$$

$$\frac{\begin{array}{cc} isdatat(c_1) & c_2\langle u_2^*\rangle \equiv \Phi \\ p{::}c_1\langle v_1^*\rangle * \sigma_1 * [\sigma'] \rhd p{::}c_2\langle v_2^*\rangle * \sigma_2 & \mathsf{Reach}(p{::}c_2\langle v_2^*\rangle * \sigma_2) \cap \{v_1^*\} = \emptyset \end{array}}{\mathsf{abs}_{\mathsf{a}}(p{::}c_1\langle v_1^*\rangle * \sigma_1) =_{df} p{::}c_2\langle v_2^*\rangle * \sigma_2}$$

The first rule makes use of abs and does not find new constraints for precondition. The second rule tries to abstract the state $p{::}c_1\langle v_1^*\rangle * \sigma_1$ with a stronger predicate c_2 against which abs failed to abstract, and discovers extra σ' to be propagated back to the precondition for the abstraction to succeed. The lifting function for $\mathsf{abs_a}$ is analogously defined as that for abs.

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

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

where the rename function avoids naming clashes among logical variables of σ_1 and σ_2, by renaming logical variables of same name in the two states with fresh names. For example it will renew $\mathtt{x_0}$'s name in both states $\exists \mathtt{x_0} \cdot \mathtt{x_0}{=}\mathtt{0}$ and $\exists \mathtt{x_0} \cdot \mathtt{x_0}{=}\mathtt{1}$ to make them $\exists \mathtt{x_0} \cdot \mathtt{x_0}{=}\mathtt{0}$ and $\exists \mathtt{x_1} \cdot \mathtt{x_1}{=}\mathtt{1}$. 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 convex hull operator over pure domain [18, 20]. Otherwise it keeps a disjunction of the two states (as it would be unsound to join their shapes together in this case). Then we lift this operator for abstract state Δ as follows:

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

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

Widening operator. The finiteness of 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.

The widening operator $\mathsf{widen}(\sigma_1, \sigma_2)$ is defined as:

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

where the rename function has the same effect as above. Generally this operator is analogous to join; the only difference is that we expect the second operand σ_2 is 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 applies widening operation $\mathsf{widen}_\pi(\pi_1, \pi_2)$

over the pure domain [18, 20]. Therefore, based on the widening over conjunctive abstract states, we lift the operator over (disjunctive) abstract states:

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

which is similar as its counterpart of the join operator. These three operations provides termination guarantee while preserving soundness, as the following example demonstrates.

5.3 Soundness and Termination

The soundness of our analysis is ensured by the soundness of the following: the entailment prover [17], the abstract semantics (w.r.t. concrete semantics), the abstraction operation over shapes, and the join and widening operators.

Theorem 1 (Soundness). *Our analysis is sound following the soundness of entailment checking, abstract semantics, and the operations of abstraction, join and widening.*

The proof for entailment checking is by structural induction over abstract domain [17]; for abstract semantics is by induction over program constructors; for abstraction follows directly the first two; and for join and widening is based on entailment checking and soundness of corresponding pure operators. Finally, as abstraction and widening for precondition are essentially unsound, we perform in the analysis algorithm a final check to ensure soundness, which is guaranteed by the soundness of abstract semantics.

For the termination aspect, we have the result:

Theorem 2 (Termination). *The iteration of our fixpoint computation will terminate in finite steps for finite input of program and specification.*

The proof is based on two facts: the finiteness over shape domain provided by our restriction on cutpoints, and the termination over numerical domain guaranteed by our widening operator. The first can be proved by claiming the finiteness of all possible abstract states only with shape information: recalling our analysis algorithm where we set an upper bound n for shared cutpoints (logical variables) we keep in track of, we know that the program and logical variables preserved in our analysis are finite. Meanwhile all possible shape predicates are limited; therefore all the shape-only abstract states are finite. The second is proved in the abstract interpretation frameworks for numerical domains [18, 20]. These two facts guarantee the convergence of our analysis.

6 Experiments and Evaluation

We have implemented a prototype system and evaluated it over a number of heap-manipulating programs to test its viability and precision. We used Sleek [17] as the solver for entailment checking over the heap domain, and Fixcalc [20] and Fixbag [19] for numerical and bag domain. Our experimental results were

achieved with an Intel Core 2 Quad CPU 2.66GHz with 8Gb RAM. The experiment results are presented in Table 1 which shows respectively the analysed methods, the number of code lines, and the analysis time in seconds. We have analysed all of the method successfully, including programs processing AVL tree with its binary-search and height-balanced properties.

Prog.	**LOC**	**Time**	**Prog.**	**LOC**	**Time**
Single Linked List			Double Linked List		
create	10	1.12	create	15	1.47
delete	9	1.20	append	24	2.53
traverse	9	1.35	insert	22	2.32
length	11	1.28	Binary Search Tree		
append	11	1.47	create	18	2.58
take	12	1.28	delete	48	4.76
reverse	13	1.72	insert	22	3.57
filter	15	2.37	search	22	2.78
drop_2nd	12	1.42	height	15	1.56
Sorting algorithm			count	17	1.63
insert_sort	32	2.72	flatten	32	2.74
merge_sort	78	4.18	AVL Tree		
quick_sort	70	5.72	insert	114	27.57
select_sort	45	3.16	delete	239	34.42

Table 1. Experimental Results.

We note down two observations on the experimental results. The first is that the analysis may discover more than one specifications for some programs. For example, if we give two predicates, ordinary linked list and sorted list, for sorting algorithm, we can obtain two specifications for most of them. The reason is that a sorted list is also a linked list. When there are more than one predicate definitions supplied, the analysis can have multiple choices during the abstraction.

The other observation concerns the precision of the analysis, which is witnessed by the rich information inferred in the specifications. For precondition, our analysis discovers sufficient information to guarantee memory safety. For example, the preconditions of some programs requires their input data structure are non-empty in order that memory safety is preserved. For the take program which traverses the list down a user-specified number `n` of nodes, we can find that the list length must be no less than `n`.

One restriction of our analysis we observed is that it requires a proper predicate to depict the requirement of a program. For example, if we only give an ordinary linked list for verifying merge sort, it will not succeed, because the stored value information in the list will not be preserved during the analysis.

7 Related Work and Conclusion

Related works. Dramatic advances have been made in synthesising heap-manipulating programs' specifications. The local shape analysis [6] infers loop invariants for list-processing programs, followed by the SpaceInvader tool to infer full method specifications over the separation domain, so as to verify pointer safety for larger industrial codes [3, 28]. The SLAyer tool [8] implements an inter-procedural analysis for programs with shape information. To deal with size information (such as number of nodes in lists/trees), THOR [16] derives a numerical program from the original heap-processing one in a sound way, such that the size properties can be obtained by numerical analysis. A similar approach [9] 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 specification with stronger invariants such as sortedness and bag-related properties, which have not been addressed in the previous works. Two more works to be mentioned are relational inductive shape analysis [4] and our previous inference works [21, 22]. These works can handle shape and numerical information over a combined domain. However they still require user annotation for the program code whereas we compute the whole specification at once.

There are also other approaches that can synthesise shape-related program invariants other than those based on separation logic. The shape analysis framework TVLA [27] is based on three-valued logic. It is capable of handling complicated data structures and properties, such as sortedness. Guo et al. [10] reported a global shape analysis that discover inductive structural shape invariants from the code. Kuncak et al. [14] developed a role system to express and track referencing relationships among objects, where an object's role (type) depends on, and changes according to, the mutation of its referencing. Hackett and Rugina [11] can deal with AVL-trees but is customised to handle only tree-like structures with height property. Type-based approaches [25, 26] are also used to infer numerical constraints for given type templates, but limited to capture flow sensitive constraints. Compared with these works, separation logic based approach benefits from the frame rule with support for local reasoning.

Concluding Remarks. We have reported in this paper a program analysis which automatically discovers program specifications over a combined separation and pure domain. The key components of our analysis include an abduction for precondition discovery, and novel operations for abstraction, join and widening in the combined domain. We have built a prototype system and the initial experimental results are encouraging.

Acknowledgement. This work was supported by EPSRC project EP/G042322 and also MoE Tier-2 project R-252-000-411-112.

References

1. Bozga, M., Iosif, R., Lakhnech, Y.: Storeless semantics and alias logic. In: PEPM. pp. 55–65. ACM (2003)

2. Calcagno, C., Distefano, D., O'Hearn, P., Yang, H.: Compositional shape analysis by means of bi-abduction. In: POPL. ACM Press (2009)
3. Calcagno, C., Distefano, D., O'Hearn, P., Yang, H.: Compositional shape analysis by means of bi-abduction. Accepted for publication in Journal of the ACM (2011)
4. Chang, B.Y.E., Rival, X.: Relational inductive shape analysis. In: POPL. pp. 247–260. ACM (2008)
5. Deutsch, A.: Interprocedural may-alias analysis for pointers: Beyond -limiting. In: PLDI. pp. 230–241 (1994)
6. Distefano, D., O'Hearn, P.W., Yang, H.: A local shape analysis based on separation logic. In: TACAS. pp. 287–302. Springer (2006)
7. Giacobazzi, R.: Abductive analysis of modular logic programs. In: Bruynooghe, M. (ed.) ILPS'94. pp. 377–391. The MIT Press (1994)
8. Gotsman, A., Berdine, J., Cook, B.: Interprocedural shape analysis with separated heap abstractions. In: SAS. pp. 240–260. Springer (2006)
9. Gulwani, S., Lev-Ami, T., Sagiv, M.: A combination framework for tracking partition sizes. In: Shao, Z., Pierce, B.C. (eds.) POPL. pp. 239–251. ACM (2009)
10. Guo, B., Vachharajani, N., August, D.I.: Shape analysis with inductive recursion synthesis. In: PLDI. pp. 256–265. ACM (2007)
11. Hackett, B., Rugina, R.: Region-based shape analysis with tracked locations. In: POPL. pp. 310–323. ACM, New York, NY, USA (2005)
12. Ishtiaq, S.S., O'Hearn, P.W.: Bi as an assertion language for mutable data structures. In: POPL. pp. 14–26. ACM (2001)
13. Jonkers, H.: Abstract storage strucutres. In: Algorithmic Languages. North Holland (1981)
14. Kuncak, V., Lam, P., Rinard, M.C.: Role analysis. In: POPL. pp. 17–32 (2002)
15. Magill, S., Tsai, M.H., Lee, P., Tsay, Y.K.: Thor: A tool for reasoning about shape and arithmetic. In: CAV. pp. 428–432. Springer (2008)
16. Magill, S., Tsai, M.H., Lee, P., Tsay, Y.K.: Automatic numeric abstractions for heap-manipulating programs. In: POPL. pp. 211–222. ACM (2010)
17. Nguyen, H.H., David, C., Qin, S., Chin, W.N.: Automated verification of shape and size properties via separation logic. In: VMCAI. pp. 251–266. Springer (2007)
18. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic, LNCS, vol. 2283. Springer (2002)
19. Pham, T.H., Trinh, M.T., Truong, A.H., Chin, W.N.: Fixbag: A fixpoint calculator for quantified bag constraints. In: CAV. pp. 656–662. Springer (2011)
20. Popeea, C., Chin, W.N.: Inferring disjunctive postconditions. In: ASIAN. pp. 331–345. Springer (2006)
21. Qin, S., He, G., Luo, C., Chin, W.N.: Loop invariant synthesis in a combined domain. In: ICFEM. pp. 468–484. Springer (2010)
22. Qin, S., Luo, C., Chin, W.N., He, G.: Automatically refining partial specifications for program verification. In: FM. pp. 369–385. Springer (2011)
23. Qin, S., Luo, C., He, G., Craciun, F., Chin, W.N.: Verifying heap-manipulating programs with unknown procedure calls. In: ICFEM. pp. 171–187. Springer (2010)
24. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: LICS. pp. 55–74. IEEE Computer Society (2002)
25. Rondon, P.M., Kawaguchi, M., Jhala, R.: Liquid types. In: PLDI. pp. 159–169. ACM (2008)
26. Rondon, P.M., Kawaguchi, M., Jhala, R.: Low-level liquid types. In: POPL. pp. 131–144. ACM (2010)
27. Sagiv, M., Reps, T.W., Wilhelm, R.: Parametric shape analysis via 3-valued logic. ACM Trans. Program. Lang. Syst. 24(3), 217–298 (2002)

28. Yang, H., Lee, O., Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P.W.: Scalable shape analysis for systems code. In: CAV. pp. 385–398. Springer (2008)