Automated Specification Discovery via User-Defined Predicates

Guanhua He[1], Shengchao Qin[1]*, Wei-Ngan Chin[2] and Florin Craciun[3]

[1]Teesside University [2]National University of Singapore [3]Babes-Bolyai University

Abstract. Automated discovery of specifications 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 in the presence of user-defined predicates, which 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 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 automated 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 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. [4, 14, 24]), separation logic [22] has been applied to analyse shape properties in recent years [5, 8, 26]. These works can automatically infer method specifications in the shape domain. Some other works such as [17, 18] 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

* Corresponding author: shengchao.qin@gmail.com

information. To overcome this limitation, we propose in this paper a compositional program analysis in a combined abstract domain with *shape*, *numerical* and *bag* information. Our analysis not only handles both functional correctness and memory safety together, but can also discover relationships between shape and pure (numerical and bag) domains. Unlike traditional approaches [18] 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 [5]). Our approach can handle this kind of programs smoothly, and we will illustrate our method using the `merge` 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. [5] to a more expressive abstract domain. In summary, this paper makes the following contributions:

- We have designed a compositional analysis to discover *full program specifications* (in the form of pre-/post-conditions involving shape, numerical and bag properties) with 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 level of 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.

2 The Approach

In this section we give some preliminaries and illustrate our approach via an example.

2.1 Preliminaries

Separation Logic. Separation logic [22] 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.

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 $\mathtt{data\ Node}\ \{\ \mathtt{int\ val};\ \mathtt{Node\ next};\}$, one can define a predicate for a list with its content as

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

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 multiset (bag) union. If one wants to verify a sorting algorithm, they can specify a non-empty sorted list as follows:

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

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 [7] 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 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 $\mathtt{m{=}n{+}1{\wedge}S_1{=}S{\sqcup}\{vx\}}$.

Bi-Abduction. In an earlier work [5], a bi-abductive entailment is proposed for the *shape* 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 [5] 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 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 $\underline{\mathtt{F}} \equiv \mathtt{(mi{=}vx{\wedge}mx{=}vy{\wedge}S{=}\{vx,vy\})}$.

2.2 An Illustrative Example

We illustrate our analysis approach via the `merge` method (used in the merge-sort), which has been declared as an unverifiable example in [5], since their analysis does not keep track of data values stored in the list during their shape analysis. The method (Fig. 1) merges two sorted lists into one sorted list. Automated specification discovery for `merge` is tricky due to two facts: (1) only one input list is fully traversed; (2) both input lists are required to be sorted. For (1), if we apply the shape abduction [5], we can only discover two disjoint lists (for precondition) - one ending with null and one ending with an unknown pointer, which cannot guarantee the memory safety of the method. For (2), if an analysis cannot infer that the two input lists are sorted, it will

```
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. 1. Merging two sorted lists.

not be able to discover that the output list is sorted, which will 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. To overcome these difficulties, we propose a compositional analysis in a combined shape and pure domain, where program properties over the combined domain are processed at the same time during the analysis. Our analysis adopts a novel bi-abduction mechanism to help discover program preconditions in the combined domain.

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

$$\begin{aligned}&\mathtt{slsB}(\mathtt{root},\mathtt{mi},\mathtt{mx},\mathtt{S},\mathtt{p}) \equiv (\mathtt{root}{\mapsto}\mathtt{Node}(\mathtt{mi},\mathtt{p}){\wedge}\mathtt{mi}{=}\mathtt{mx}{\wedge}\mathtt{S}{=}\{\mathtt{mi}\}){\vee}\\ &\quad(\mathtt{root}{\mapsto}\mathtt{Node}(\mathtt{mi},\mathtt{q}){*}\mathtt{sllB}(\mathtt{q},\mathtt{m}_1,\mathtt{mx},\mathtt{S}_1,\mathtt{p}){\wedge}\mathtt{mi}{\le}\mathtt{m}_1{\wedge}\mathtt{m}_1{\le}\mathtt{mx}{\wedge}\mathtt{S}{=}\mathtt{S}_1{\sqcup}\{\mathtt{mi}\})\end{aligned}$$

Our analysis aims at finding a sound and precise specification (summary) of the method. Starting from an initial specification ($\mathsf{Pre}_0{\equiv}\mathtt{emp}, \mathsf{Post}_0{\equiv}\mathtt{false}$), our analysis iterates the method body by symbolic execution a number of times until a fixed point is reached for the pre-/post-condition pair. During the symbolic execution, we use a pair of states ($\mathtt{infP}$, Curr) to keep track of the precondition that the analysis has discovered ($\mathtt{infP}$) 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 ($\mathtt{infP}{\equiv}\mathtt{emp}$, $\mathsf{Curr}{\equiv}\mathtt{emp}$), 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

$$(\mathsf{Pre}_1{\equiv}(\mathtt{x}{=}\mathtt{null}\vee\mathtt{y}{=}\mathtt{null}), \mathsf{Post}_1{\equiv}(\mathtt{x}{=}\mathtt{null}{\wedge}\mathtt{res}{=}\mathtt{y}\vee\mathtt{y}{=}\mathtt{null}{\wedge}\mathtt{res}{=}\mathtt{x})) \tag{1}$$

where $\mathtt{res}$ 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 ($\mathtt{infP}{\equiv}\mathsf{Pre}_1$, $\mathsf{Curr}{\equiv}\mathsf{Pre}_1$) yields the summary ($\mathsf{Pre}_2$, Post_2):

$$\begin{aligned}(\mathsf{Pre}_2 \equiv\ &\mathtt{x}{=}\mathtt{null}\vee\mathtt{y}{=}\mathtt{null}\vee\mathtt{x}{\mapsto}\mathtt{Node}(\mathtt{xv}_1,\mathtt{xp}_1){*}\mathtt{y}{\mapsto}\mathtt{Node}(\mathtt{yv}_1,\mathtt{yp}_1)\\ &{\wedge}(\mathtt{xv}_1{\le}\mathtt{yv}_1{\wedge}\mathtt{xp}_1{=}\mathtt{null}\vee\mathtt{xv}_1{>}\mathtt{yv}_1{\wedge}\mathtt{yp}_1{=}\mathtt{null}),\\ \mathsf{Post}_2{\equiv}\ &\mathtt{x}{=}\mathtt{null}{\wedge}\mathtt{res}{=}\mathtt{y}\vee\mathtt{y}{=}\mathtt{null}{\wedge}\mathtt{res}{=}\mathtt{x}\vee\mathtt{x}{\mapsto}\mathtt{Node}(\mathtt{xv}_1,\mathtt{xp}_1){*}\mathtt{y}{\mapsto}\mathtt{Node}(\mathtt{yv}_1,\mathtt{yp}_1)\\ &{\wedge}(\mathtt{xv}_1{\le}\mathtt{yv}_1{\wedge}\mathtt{res}{=}\mathtt{x}{\wedge}\mathtt{xp}_1{=}\mathtt{y}\vee\mathtt{xv}_1{>}\mathtt{yv}_1{\wedge}\mathtt{res}{=}\mathtt{y}{\wedge}\mathtt{yp}_1{=}\mathtt{x}))\end{aligned} \tag{2}$$

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

$$\begin{array}{l}\mathtt{x{=}null} \vee \mathtt{y{=}null} \vee \mathtt{x{\mapsto}Node(xv_1, xp_1){*}y{\mapsto}Node(yv_1, yp_1)} \\ \quad\wedge\ (\mathtt{xv_1{\leq}yv_1{\wedge}xp_1{=}null} \vee \mathtt{xv_1{>}yv_1{\wedge}yp_1{=}null})\end{array} \tag{3}$$

$$\begin{array}{l}\vee\mathtt{x{\mapsto}Node(xv_1, xp_1){*}xp_1{\mapsto}Node(xv_2, xp_2){*}y{\mapsto}Node(yv_1, yp_1)} \\ \quad\wedge\ (\mathtt{xv_1{\leq}yv_1{\wedge}(xv_2{\leq}yv_1{\wedge}xp_2{=}null} \vee \mathtt{xv_2{>}yv_1{\wedge}yp_1{=}null}))\end{array} \tag{4}$$

$$\begin{array}{l}\vee\mathtt{x{\mapsto}Node(xv_1, xp_1){*}y{\mapsto}Node(yv_1, yp_1){*}yp_1{\mapsto}Node(yv_2, yp_2)} \\ \quad\wedge\ (\mathtt{xv_1{>}yv_1{\wedge}(xv_1{\leq}yv_2{\wedge}xp_1{=}null} \vee \mathtt{xv_1{>}yv_2{\wedge}yp_2{=}null}))\end{array} \tag{5}$$

Branch (4) says that the program only touches the second node of $\mathtt{x}$ list (the list referred to by $\mathtt{x}$) if $\mathtt{xv_1{\leq}yv_1}$. Furthermore, if $\mathtt{xv_2{\leq}yv_1}$, $\mathtt{xp_2}$ should be $\mathtt{null}$; otherwise $\mathtt{yp_1}$ must be $\mathtt{null}$ to guarantee the termination of the method and memory safety. Branch (5) states a similar condition when touching the second node of $\mathtt{y}$ 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 $\mathtt{slsB}$, we could abstract the shape of the $\mathtt{x}$ list (and that of the $\mathtt{y}$ 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 $\mathtt{x}$ list (and the $\mathtt{y}$ list) is missing. This missing information is the numerical relation between $\mathtt{xv_1}$ and $\mathtt{xv_2}$ in the $\mathtt{x}$ list (and that between $\mathtt{yv_1}$ and $\mathtt{yv_2}$ in the $\mathtt{y}$ list). In other words, we need to use abduction to discover $\mathtt{xv_1{\leq}xv_2}$ (resp. $\mathtt{yv_1{\leq}yv_2}$) during the abstraction from the shape of the $\mathtt{x}$ list (resp. the $\mathtt{y}$ list) to a sorted list segment in the branch (4) (resp. (5)), e.g. for the $\mathtt{x}$ list:

$$\mathtt{x{\mapsto}Node(xv_1, xp_1){*}xp_1{\mapsto}Node(xv_2, xp_2){*}}\underline{[\mathtt{xv_1{\leq}xv_2}]} \rhd \mathtt{slsB(x, xv_1, xv_2, xS_1, xp_2){*}R}$$

The inspiration for this *abductive abstraction* comes from the definition of the predicate $\mathtt{slsB}$. We use such predicates to help infer data structure properties that are anticipated from some program code. Note that a standard abstraction 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 $\mathtt{slsB}$ and then joining the branches with the same shape, the precondition from two iterations becomes:

$$\begin{array}{l}\mathtt{x{=}null} \vee \mathtt{y{=}null} \vee \mathtt{slsB(x, xmi_0, xmx_0, xS_0, xp_0)} * \mathtt{slsB(y, ymi_0, ymx_0, yS_0, yp_0)} \\ \quad\wedge\ (\mathtt{xmx_0{\leq}ymx_0} \wedge \mathtt{xp_0{=}null} \vee \mathtt{xmx_0{>}ymx_0} \wedge \mathtt{yp_0{=}null})\end{array}$$

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

$$\begin{array}{l}\mathsf{Pre}{\equiv}\mathtt{x{=}null} \vee \mathtt{y{=}null} \vee \mathtt{slsB(x, xmi_0, xmx_0, xS_0, xp_0)}* \\ \quad\mathtt{slsB(y, ymi_0, ymx_0, yS_0, yp_0){\wedge}(xmx_0{\leq}ymx_0{\wedge}xp_0{=}null{\vee}xmx_0{>}ymx_0{\wedge}yp_0{=}null)}, \\ \mathsf{Post}{\equiv}\mathtt{x{=}null{\wedge}res{=}y} \vee \mathtt{y{=}null{\wedge}res{=}x} \vee \mathtt{slsB(x, xmi_1, xmx_1, xS_1, xp_1)} \\ \quad\mathtt{{*}slsB(y, ymi_1, ymx_1, yS_1, yp_1){\wedge}xS_0{\sqcup}yS_0{=}xS_1{\sqcup}yS_1{\wedge}xmi_1{=}xmi_0{\wedge}ymi_1{=}ymi_0{\wedge}} \\ \quad\mathtt{(xmi_0{\leq}ymi_0{\wedge}res{=}x{\wedge}xp_1{=}y{\wedge}xmx_1{\leq}ymi_1{\vee}xmi_0{>}ymi_0{\wedge}res{=}y{\wedge}yp_1{=}x{\wedge}ymx_1{\leq}xmi_1}\end{array}$$

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-approximations and are sound for the synthesis of postconditions. However, when such steps are applied to the synthesis of preconditions, it may make the precondition too weak for the program to establish the postcondition. So after the analysis, we shall use a forward analysis process to check the discovered summary (a similar process is also carried out in [5]).

From this example, we observe that the memory safety is not only related to the shape of data structures, but may also relate to data values stored in them. For the $\mathtt{merge}$ example, our analysis can find that one input list is traversed to its end, i.e. until $\mathtt{null}$ 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 presentation, we employ a strongly-typed C-like imperative language in Fig. 2 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. 3. 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.

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

Fig. 2. A Core (C-like) Imperative Language.

Our specification language (in Fig. 3) 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 [7]. 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 heap (shape) part κ in the 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. $k^{\texttt{int}}$ 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 [7].

Using entailment [7], we define a partial order over these abstract states:

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

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

Fig. 3. The Specification Language.

where 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 [7]. In our analysis, variables include both program and logical variables.

4 Generalised Bi-Abduction for the Combined Domain

We present a new bi-abduction procedure over the combined domain (which generalises the previous bi-abduction [5] over only the shape domain).

Given σ and σ_1, the bi-abduction procedure $\sigma * [\sigma'] \rhd \sigma_1 * \sigma_2$ (shown in Fig. 4) 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 next instruction, respectively. Our abduction procedure can handle more than one predicates in the analysis, while the shape abduction [5] 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.

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

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

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

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

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

Fig. 4. Bi-Abduction rules.

The 1st 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 emp (σ), RHS is $\texttt{x}{\mapsto}\texttt{Node}(\texttt{xv}, \texttt{xp})(\sigma_1)$, the RHS can entail the LHS with residue $\texttt{x}{\mapsto}\texttt{Node}(\texttt{xv}, \texttt{xp})(\sigma')$. The abduction then checks whether σ plus the frame σ' implies $\sigma_1 * \sigma_2$ for some σ_2 ($\mathtt{emp}$ in this example), and returns $\mathtt{x}{\mapsto}\mathtt{Node}(\mathtt{xv}, \mathtt{xp})$ as the anti-frame.

The 2nd rule Unroll deals with the case where neither side entails the other, e.g. for $\mathtt{slsB}(\mathtt{x}, \mathtt{xmi}, \mathtt{xmx}, \mathtt{xS}, \mathtt{null})$ as LHS and $\exists \mathtt{p}, \mathtt{u}, \mathtt{v}{\cdot}\mathtt{x}{\mapsto}\mathtt{Node}(\mathtt{u}, \mathtt{p}) * \mathtt{p}{\mapsto}\mathtt{Node}(\mathtt{v}, \mathtt{null})$ as RHS. As the shape predicates in the antecedent σ are formed by disjunctions according to their definitions (like $\mathtt{slsB}$), 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 $\exists \mathtt{u}, \mathtt{v}{\cdot}\mathtt{xS}{=}\{\mathtt{u}, \mathtt{v}\}$ as the anti-frame σ' and discovers the nontrivial frame $\mathtt{u}{=}\mathtt{xmi} \wedge \mathtt{v}{=}\mathtt{xmx} \wedge \mathtt{u}{\leq}\mathtt{v}$ as σ_2. The function $\mathsf{data\_no}$ returns the number of data nodes in a state, e.g. it returns 1 for $\mathtt{x}{\mapsto}\mathtt{Node}(\mathtt{v}, \mathtt{p}){*}\mathtt{llB}(\mathtt{p}, \mathtt{n}, \mathtt{T})$. 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 ($\{\sigma_1, ..., \sigma_n\}$).

The 3rd rule Reverse handles the case where neither side entails the other, and the 2nd rule does not apply, e.g. $\exists \mathtt{p}, \mathtt{u}, \mathtt{v}, \mathtt{q}{\cdot}\mathtt{x}{\mapsto}\mathtt{Node}(\mathtt{u}, \mathtt{p}){*}\mathtt{p}{\mapsto}\mathtt{Node}(\mathtt{v}, \mathtt{q})$ as LHS and $\exists \mathtt{xS}{\cdot}$ $\mathtt{slsB}(\mathtt{x}, \mathtt{xmi}, \mathtt{xmx}, \mathtt{xS}, \mathtt{xp})$ 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'. Then it checks that the LHS (σ), with σ' added, does entail the RHS (σ_1) before it returns σ'. For the example above, the anti-frame is inferred as $\mathtt{u}{\leq}\mathtt{v}$.

When an abduction procedure is conducted, the first three rules should be attempted 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 $\mathtt{if}\ (\mathtt{x} == \mathtt{y})\{\mathtt{c}\}$ suggests that $\mathtt{x}$ and $\mathtt{y}$ are aliased in code $\mathtt{c}$. Note that when the third rule is applied, the abduction procedure in the premise, namely $\sigma_1 {*} [\sigma_1'] {\rhd} \sigma {*} \sigma'$, is not allowed to apply the third rule again. 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 $\mathtt{llB}(\mathtt{x}, \mathtt{n}, \mathtt{S}){\wedge}\mathtt{n}{>}2{*}[\sigma'] \rhd \mathtt{x}{\mapsto}\mathtt{Node}(\mathtt{v}_1, \mathtt{p}_1) * \mathtt{y}{\mapsto}\mathtt{Node}(\mathtt{v}_2, \mathtt{p}_2) * \sigma_2$ 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 [20] 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) one 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 Fig. 5. 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 analysiscarries 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).[1]

We first set the 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 a widening is conducted over both to ensure termination of the analysis (line 7). If the analysis cannot continue due to a program 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 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.

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

[1] The analysis uses lifted versions of these operations (indicated by †), which will be explained in more detail later.

```
Fixpoint Computation in the Combined Domain
Input: $\mathcal{T},\ t\ mn\ ((t\ x)^*;(t\ y)^*)\ \{e\},\ n$
Local: $i := 0;\ \mathsf{Pre}_i := \textbf{emp}, \mathsf{Post}_i := \textbf{false};$
 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 Post = false or $\mathsf{Post} \nvdash \mathsf{Post}_i * \mathtt{true}$ then return fail
13   else return $\mathcal{T}'$
14   end if
```

Fig. 5. Main analysis algorithm.

then it applies the pure widening for the pure part. The formal definitions of $\mathsf{join}^{\dagger}$ and $\mathsf{widen}^{\dagger}$ and other details are left in our report [13] due to page limit.

Bi-Abductive Abstract Semantics. As shown in Fig. 5, 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 semantics. Its type is defined as

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

where AllSpec contains procedure 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 semantics are the basic transition functions from a conjunctive abstract state (σ) to a conjunctive or disjunctive abstract state (σ or Δ) below:

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

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:

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

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:

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

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

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

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

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

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 $\sigma \vdash \rho\Phi_{\mathit{pr}} * \sigma'$. In this case, the execution succeeds and the post-state of the method call involves mn's postcondition as signified by $\rho_o\Phi_{\mathit{po}}$.

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

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

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

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

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):

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

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

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

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:

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

The definitions of both functions are given below:

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

$$\begin{array}{l}\mathsf{Exec}(ds)(\mathcal{T})(\sigma',\sigma) =_{\mathit{df}} \textbf{let } \Delta{=}\mathsf{exec}(ds)(\mathcal{T})\sigma \textbf{ in } (\sigma', \Delta)\\ \qquad\qquad\qquad\qquad\quad \text{where } ds \text{ is either } d[x] \text{ or } d,\ \text{except for procedure call}\end{array}$$

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

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 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 $\rho\Phi_{\mathit{pr}}$. 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:

$$\begin{array}{ll}\mathsf{Unfold}^{\dagger}(x) \bigvee(\sigma'_i, \sigma_i) & =_{\mathit{df}} \bigvee(\mathsf{Unfold}(x)(\sigma'_i, \sigma_i))\\ \mathsf{Exec}^{\dagger}(ds)(\mathcal{T}) \bigvee(\sigma'_i, \sigma_i) & =_{\mathit{df}} \bigvee(\mathsf{Exec}(ds)(\mathcal{T})(\sigma'_i, \sigma_i))\end{array}$$

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

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

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:

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

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

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

$$\frac{p_2(u_2^*) \equiv \Phi \quad \mathtt{H}(c_1)(x, v_1^*) * \sigma_1 \nvdash p_2(x, v_2^*) \wedge \pi_2 \qquad \mathtt{H}(c_1)(x, v_1^*){*}\sigma_1{*}[\sigma'] \rhd p_2(x, v_2^*){\wedge}\pi_2 \quad \mathsf{Reach}(p_2(x, v_2^*){\wedge}\pi2{*}\sigma_3){\cap}\{v_1^*\}{=}\emptyset}{\mathsf{abs_a}(\mathtt{H}(c_1)(x, v_1^*) * \sigma_1 * \sigma_3) =_{df} p_2(x, v_2^*) \wedge \pi_2 * \sigma_3}$$

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 function $\mathsf{Reach}(\sigma)$ returns all pointer variables which are reachable from free variables in the abstract state σ. 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 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 $\mathcal{P}_{\mathsf{SH}}$: $\mathsf{abs_a}^{\dagger} \bigvee \sigma_i =_{df} \bigvee \mathsf{abs_a}(\sigma_i)$, allowing it to be used in the analysis.

The soundness and termination of our analysis are given in the technical report [13].

6 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. Our experimental results were achieved with an Intel Core 2 Quad CPU 2.66GHz with 8GB RAM. We have also defined a library of predicates covering popular data structures and variety of properties. These properties can be grouped in the following categories: MS (*memory safety*): all memory accesses are safe, no dangling/null pointers dereferences; SC (*same content*): the content of the final data structure remains the same as that of the input data structure; IN (*insertion*): the input data is inserted into the final data structure; SO (*sorted*): 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; BS (*binary search*): data structures are binary search trees; DL (*double-linked list*): data structures are double-linked lists; and AL (*AVL tree*): data structures are AVL trees. The predicates required as input by our tool can be selected from the 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. Some of our results are presented in Table 1.

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

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.

In comparison to previous approaches, the first observation concerns the precision of our analysis. 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 input list length must be no less than `n`. However the previous tools based on shape domains (like Abductor [5]) 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 SO for `merge` program but in general for all sorting programs) can also not be discovered by the previous tools (like Abductor) based on a simple shape domain. There are other tools (like Xisa [6] or Thor [18]) that can work on a combined domain but require certain annotations to guide their analysis. Thor [18] requires shape information for each input parameter and Xisa [6] 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. 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`.

7 Related Work and Conclusion

Dramatic advances have been made in synthesising specifications for heap-manipulating programs. The local shape analysis [8] 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 [5, 26]. The SLAyer tool [9] implements an inter-procedural analysis for programs with shape information. A combination of shape and bag abstraction is used in [25] to verify linearizability. 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 [18] transfers a heap-processing program to a numerical one, so that size properties can be obtained by further analysis. A similar approach [10] 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 [6, 19, 20] 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. Recently, Rival and Chang [23] 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.

There are also other approaches that can synthesise shape-related program invariants. The shape analysis framework TVLA [24] is based on three-valued logic. It is capable of handling complicated data structures and properties, such as sortedness. Guo et al. [11] report a global shape analysis that discovers inductive structural shape invariants from the code. Kuncak et al. [15] develop a role system to express and track referencing relationships among objects. Hackett and Rugina [12] can deal with AVL-trees but is customised to handle only tree-like structures with height property. Bouajjani et al. [2, 3] 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.

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. [21, 16]) or a combination of appropriate abstract interpreters inside a software model checker (e.g. [1]). Compared with our work, their heap domains are mainly restricted to linked lists.

Conclusion. We have reported a program analysis which automatically discovers program specifications over a combined separation and pure(numerical and bag) domain. 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.

Acknowledgement. This work was supported in part by EPSRC project EP/G042322.

References

1. Beyer, D., Henzinger, T.A., Théoduloz, G.: Configurable software verification: Concretizing the convergence of model checking and program analysis. In: CAV (2007)
2. Bouajjani, A., Dragoi, C., Enea, C., Sighireanu, M.: On inter-procedural analysis of programs with lists and data. In: PLDI (2011)
3. Bouajjani, A., Dragoi, C., Enea, C., Sighireanu, M.: Abstract domains for automated reasoning about list-manipulating programs with infinite data. In: VMCAI (2012)
4. Bozga, M., Iosif, R., Lakhnech, Y.: Storeless semantics and alias logic. In: PEPM (2003)
5. Calcagno, C., Distefano, D., O'Hearn, P., Yang, H.: Compositional shape analysis by means of bi-abduction. J. ACM 58(6) (2011)
6. Chang, B.Y.E., Rival, X.: Relational inductive shape analysis. In: POPL (2008)
7. Chin, W.N., David, C., Nguyen, H.H., Qin, S.: Automated verification of shape, size and bag properties via user-defined predicates in separation logic. Sci. of Comp. Prog. 77 (2012)
8. Distefano, D., O'Hearn, P.W., Yang, H.: A local shape analysis based on separation logic. In: TACAS (2006)
9. Gotsman, A., Berdine, J., Cook, B.: Interprocedural shape analysis with separated heap abstractions. In: SAS (2006)
10. Gulwani, S., Lev-Ami, T., Sagiv, M.: A combination framework for tracking partition sizes. In: Shao, Z., Pierce, B.C. (eds.) POPL (2009)
11. Guo, B., Vachharajani, N., August, D.I.: Shape analysis with inductive recursion synthesis. In: PLDI (2007)
12. Hackett, B., Rugina, R.: Region-based shape analysis with tracked locations. In: POPL (2005)
13. He, G., Qin, S., Chin, W.N., Craciun, F.: Automated specification discovery in a combined abstract domain - reseach report. (2012), `http://pls.tees.ac.uk/~guan/fullspec/techreport.pdf`
14. Jonkers, H.: Abstract storage structures. In: Algorithmic Languages (1981)
15. Kuncak, V., Lam, P., Rinard, M.C.: Role analysis. In: POPL (2002)
16. Lahiri, S.K., Qadeer, S.: Back to the future: revisiting precise program verification using smt solvers. In: POPL (2008)
17. Magill, S., Tsai, M.H., Lee, P., Tsay, Y.K.: Thor: A tool for reasoning about shape and arithmetic. In: CAV (2008)
18. Magill, S., Tsai, M.H., Lee, P., Tsay, Y.K.: Automatic numeric abstractions for heap-manipulating programs. In: POPL (2010)
19. Qin, S., He, G., Luo, C., Chin, W.N., Chen, X.: Loop invariant synthesis in a combined abstract domain. Journal of Symbolic Computation 50 (2013)
20. Qin, S., Luo, C., Chin, W.N., He, G.: Automatically refining partial specifications for program verification. In: FM (2011)
21. Rakamaric, Z., Bruttomesso, R., Hu, A.J., Cimatti, A.: Verifying heap-manipulating programs in an smt framework. In: ATVA (2007)
22. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: LICS (2002)
23. Rival, X., Chang, B.Y.E.: Calling context abstraction with shapes. In: POPL (2011)
24. Sagiv, M., Reps, T.W., Wilhelm, R.: Parametric shape analysis via 3-valued logic. ACM Trans. Program. Lang. Syst. 24(3) (2002)
25. Vafeiadis, V.: Shape-value abstraction for verifying linearizability. In: VMCAI (2009)
26. Yang, H., Lee, O., Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P.W.: Scalable shape analysis for systems code. In: CAV (2008)