Automatically Refining Partial Specifications for Program Verification

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

[1] Durham University, Durham DH1 3LE, UK
[2] National University of Singapore
{chenguang.luo,shengchao.qin,guanhua.he}@durham.ac.uk
chinwn@comp.nus.edu.sg

Abstract. Automatically verifying heap-manipulating programs is a challenging task, especially when dealing with complex data structures with strong invariants, such as sorted lists and AVL trees. The verification process can greatly benefit from human assistance through specification annotations, but this process is both error-prone and requires intellectual effort from users. In this paper, we propose a new approach to program verification that allows users to provide only partial specification to methods. Our approach will then refine the given annotation into a more complete specification by discovering missing constraints. The discovered constraints may involve both numerical and multi-set properties that could be later confirmed or replaced by users. We further augment our approach by requiring only partial specification to be given for primary methods. Specifications for loops and auxiliary methods can then be systematically discovered by our augmented mechanism, with the help of information propagated from the primary methods. Our work is aimed at verifying beyond shape properties, with the eventual goal of analysing full functional properties for pointer-based data structures. Initial experiments have confirmed that we can automatically refine partial specifications with non-trivial constraints, thus making it easier for users to handle specifications with richer properties.

1 Introduction

Although research on software verification has a long and distinguished history dating back to the 1960's, it remains a challenging problem to automatically verify heap manipulating programs written in mainstream imperative languages. This is in part due to the shared mutable data structures lying in programs, and the need to track properties, such as structural numerical information (size and height), relational numerical information (balanced and sortedness properties), and content information (multi-set of values).

Human assistance is often essential in (semi-) automated program verification. The user may supply annotations at certain program point, such as loop invariants and/or method specifications. These annotations can greatly narrow down the possible program states at that point, and avoid fixed-point calculation which could be expensive and may be less precise than the user's insight.

However, an obvious disadvantage of user annotation concerns its scalability, since programs to be analysed may be complicated and their functions are also diverse. Therefore, it is not preferable to require the user to provide specification for each method and invariant for each loop when verifying a relatively large software system. Meanwhile, human is liable to make mistakes. A programmer may under-specify with too weak a precondition or over-specify with too strong a postcondition. Such mistakes could lead to failed verification, and it may be difficult for the user to discover whether the error is due to a real bug in the program, or an inappropriately supplied annotation.

To balance verification quality and human effort, we provide a novel approach to the verification of heap manipulating programs. Under our framework, the user is expected to provide the definition of shapes and properties which they want to verify, and partial specifications for primary methods that cover the desired level of correctness. Our verification will then take over the rest of the work to refine those partial specifications with derived constraints which should be satisfied by the program, or report a possible program bug if the given specifications are rejected by our verifier.

As aforesaid, our approach requires that the user design their predicates for shapes and relative properties, to capture the desired level of program correctness to be verified. For example, with a singly-linked list node `data node` $\{$ `int val`; `node next`; $\}$ as data structure, a user interested in pointer-safety may define a predicate to depict the list shape similar as in previous works [4, 12]:

$$\texttt{root}::\texttt{list}\langle\rangle \equiv (\texttt{root}=\texttt{null}) \vee (\exists \texttt{i}, \texttt{q} \cdot \texttt{root}::\texttt{node}\langle \texttt{i}, \texttt{q}\rangle * \texttt{q}::\texttt{list}\langle\rangle)$$

The sole parameter `root` for the predicate `list` is the root pointer referring to the list. A uniform notation $\texttt{p}::\texttt{c}\langle \texttt{v}^*\rangle$ is used for either a singleton heap or a predicate. If `c` is a data node, the notation represents a singleton heap, $\texttt{p}{\mapsto}\texttt{c}[\texttt{v}^*]$, e.g. the $\texttt{root}::\texttt{node}\langle \texttt{i}, \texttt{q}\rangle$ above. If `c` is a predicate name, then the data structure pointed to by `p` has the shape `c` with parameters $\texttt{v}^*$, e.g., the $\texttt{q}::\texttt{list}\langle\rangle$ above. Note also that in the inductive case, the separation conjunction $*$ ([35]) ensures that two heap portions (the head node and the tail list) are domain-disjoint.

Yet another user may be interested to track also the length of a list to analyze quantitative measures, such as heap/stack resource usage. Therefore the predicate can be defined in a similar manner as in previous work [28]:

$$\texttt{ll}\langle \texttt{n}\rangle \equiv (\texttt{root}=\texttt{null} \wedge \texttt{n}=0) \vee (\texttt{root}::\texttt{node}\langle \_, \texttt{q}\rangle * \texttt{q}::\texttt{ll}\langle \texttt{m}\rangle \wedge \texttt{n}=\texttt{m}+1)$$

where we use the following shortened notation: (i) default `root` parameter in LHS may be omitted, (ii) unbound variables, such as `q` and `m`, are implicitly existentially quantified, and (iii) _ denotes existentially quantified anonymous variable. Meanwhile, this predicate may still be extended to support a higher-level of correctness with multi-set (bag) property:

$$\texttt{llB}\langle \texttt{S}\rangle \equiv (\texttt{root}=\texttt{null} \wedge \texttt{S}=\emptyset) \vee (\texttt{root}::\texttt{node}\langle \texttt{v}, \texttt{q}\rangle * \texttt{q}::\texttt{llB}\langle \texttt{S}_1\rangle \wedge \texttt{S}=\{\texttt{v}\} \sqcup \texttt{S}_1)$$

which also implicitly suggests list segment's length with $|\texttt{S}|$. This predicate can be strengthened furthermore if necessary, so as to verify a sorting algorithm:

$$\texttt{sllB}\langle\texttt{S}\rangle\equiv(\texttt{root}{=}\texttt{null}\wedge\texttt{S}{=}\emptyset)\vee(\texttt{root}::\texttt{node}\langle\texttt{v},\texttt{q}\rangle*\texttt{q}::\texttt{sllB}\langle\texttt{S}_1\rangle\wedge\texttt{S}{=}\{\texttt{v}\}\sqcup\texttt{S}_1\wedge(\forall\texttt{x}{\in}\texttt{S}_1\cdot\texttt{v}{\leq}\texttt{x}))$$

Therefore, the user is expected to provide predicate definitions w.r.t. their required correctness level and program properties. These predicates may be non-trivial but can be reused multiple times for specifications of different methods.

Based on these predicates, the user is expected to provide partial specifications for some primary methods which are the main objects of verification. Say, for an insertion sort algorithm taking `x` as input parameter that is expected to be non-null, the user may provide $\texttt{x}::\texttt{llB}\langle\texttt{S}_1\rangle$ as precondition and $\texttt{x}::\texttt{sllB}\langle\texttt{S}_2\rangle$ as postcondition, and our approach will refine the specification as $\texttt{x}::\texttt{llB}\langle\texttt{S}_1\rangle\wedge\texttt{x}{\neq}\texttt{null}$ for pre, and $\texttt{x}::\texttt{sllB}\langle\texttt{S}_2\rangle\wedge\texttt{S}_1{=}\texttt{S}_2$ for post. Here we need the user annotations as initial specification, because we reserve the flexibility of verification w.r.t. different program properties at various correctness levels. For example, our approach can also verify the same algorithm, but for the following refined specifications:

$$\begin{array}{llllll}
\texttt{requires} & \texttt{x}::\texttt{list}\langle\rangle \;\wedge & \texttt{x}{\neq}\texttt{null} & \texttt{ensures} & \texttt{x}::\texttt{list}\langle\rangle & \\
\texttt{requires} & \texttt{x}::\texttt{ll}\langle\texttt{n}_1\rangle \;\wedge & \texttt{n}_1{>}0 & \texttt{ensures} & \texttt{x}::\texttt{ll}\langle\texttt{n}_2\rangle \;\wedge & \texttt{n}_1{=}\texttt{n}_2 \\
\texttt{requires} & \texttt{x}::\texttt{llB}\langle\texttt{S}_1\rangle \;\wedge & \texttt{x}{\neq}\texttt{null} & \texttt{ensures} & \texttt{x}::\texttt{llB}\langle\texttt{S}_2\rangle \;\wedge & \texttt{S}_1{=}\texttt{S}_2 \\
\texttt{requires} & \texttt{x}::\texttt{llB}\langle\texttt{S}\rangle \;\wedge & \texttt{x}{\neq}\texttt{null} & \texttt{ensures} & \texttt{x}::\texttt{ll}\langle\texttt{n}\rangle \;\wedge & |\texttt{S}|{=}\texttt{n}
\end{array}$$

where the discovered missing constraints are shown in shaded form. The last pre/post can be omitted in our approach if we are given a coercion lemma [31] between $\texttt{x}::\texttt{ll}\langle\texttt{n}\rangle$ and $\texttt{x}::\texttt{llB}\langle\texttt{S}\rangle$. This can help reduce the number of redundant specifications considered (or synthesised for auxiliary methods).

To summarise, our proposal for refining partial specification is aimed at harnessing the synergy between human's insights and machine's capability at automated program analysis. In particular, human's guidance can help narrow down on the most important of the numerous specifications that are possible with each program code, while automation by machine is important for minimising on the tedium faced by users. Our proposal has the following characteristics:

- *Specification Completion*: This verification refines the specification from three aspects, namely, the constraints needed in the precondition for memory and code safety, the constraints in postcondition to link the method's pre- and post-states, and the constraints that the method's post-state satisfies.
- *Flexibility*: We allow the user to define their own predicates for the program properties they want to verify, so as to provide different levels of correctness. Meanwhile we aim at, and have covered much of, full functional correctness of pointer-manipulating programs such as data structure shapes, pointer safety, structural/relational numerical constraints, and bag information.
- *Reduction of user annotations*: Our approach uses program analysis techniques effectively to reduce users' annotations. As for our experiments, the user only has to supply the partial specifications for primary methods, and the analysis will compute pre- and postconditions for loops and auxiliary methods as well as refine primary methods' specifications.

– *Semi-Automation*: We classify our approach as semi-automatic, because the user is allowed to interfere and guide the verification at any point. For instance, they may provide invariant for a loop instead of our automated invariant generation, or choose some other constraints as refinement from what the verification has discovered.

We have built a prototype implementation and carried out a number of experiments to confirm the viability of the approach as described in Section 5. In what follows, we will first depict our approach informally using a motivating example and present technical details thereafter. More related works and concluding remarks come after the experimental results.

2 The Approach

In this section, we briefly introduce the Hip/Sleek system as the base of our verification and refinement. We then use some motivating examples to informally illustrate our approach.

2.1 The Hip/Sleek System

Separation logic [20, 35] extends Hoare logic to support reasoning about shared mutable data structures. It adds two more connectives to classical logic: separation conjunction $*$ and spacial implication $-\!\!*$. The formula $\mathtt{p_1} * \mathtt{p_2}$ asserts that two heaps described by $\mathtt{p_1}$ and $\mathtt{p_2}$ are domain-disjoint, while $\mathtt{p_1} -\!\!* \mathtt{p_2}$ asserts that if the current heap is extended with a disjoint heap described by $\mathtt{p_1}$, then $\mathtt{p_2}$ holds in the extended heap. In this paper we only use separation conjunction.

For better flexibility and expressivity, Hip/Sleek allows users to define inductive shape predicates to leverage both shape and pure properties. We have illustrated several of these shape predicate definitions in the last section. For more involved examples, based on a data structure definition `data node2 { int val; node2 prev; node2 next; }`, one may define the predicate below to specify sorted doubly-linked list segments:

$$\mathtt{sdlB}\langle\mathtt{p},\mathtt{q},\mathtt{S}\rangle \equiv (\mathtt{root}{=}\mathtt{q} \wedge \mathtt{S}{=}\emptyset) \ \vee \ (\mathtt{root}{::}\mathtt{node2}\langle\mathtt{v},\mathtt{p},\mathtt{r}\rangle * \mathtt{r}{::}\mathtt{sdlB}\langle\mathtt{root},\mathtt{q},\mathtt{S_1}\rangle \wedge \mathtt{root}{\neq}\mathtt{q} \wedge \mathtt{S}{=}\{\mathtt{v}\} \sqcup \mathtt{S_1} \wedge (\forall \mathtt{x}{\in}\mathtt{S_1}\cdot\mathtt{v}{\leq}\mathtt{x}))$$

where the parameters `p` and `q` denote the `prev` field of `root` and the `next` of the list's last node, respectively. Meanwhile `S` is a bag (multi-set) parameter to represent the list's content. We can see in the base case of definition that $\mathtt{S}{=}\emptyset$, and in the recursive case that all values stored after `root` must be no less than `root`'s value.

Another example is the definition of node-balanced trees with binary search property:

$$\begin{aligned}\mathtt{nbt}\langle\mathtt{S}\rangle \equiv\ & (\mathtt{root}{=}\mathtt{null} \wedge \mathtt{S}{=}\emptyset) \ \vee \\ & (\mathtt{root}{::}\mathtt{node2}\langle\mathtt{v},\mathtt{p},\mathtt{q}\rangle * \mathtt{p}{::}\mathtt{nbt}\langle\mathtt{S_p}\rangle * \mathtt{q}{::}\mathtt{nbt}\langle\mathtt{S_q}\rangle \wedge \mathtt{S}{=}\{\mathtt{s}\} \sqcup \mathtt{S_p} \sqcup \mathtt{S_q} \wedge \\ & \quad (\forall \mathtt{x}{\in}\mathtt{S_p}\cdot\mathtt{x}{\leq}\mathtt{s}) \wedge (\forall \mathtt{x}{\in}\mathtt{S_q}\cdot\mathtt{s}{\leq}\mathtt{x}) \wedge -1{\leq}|\mathtt{S_p}|{-}|\mathtt{S_q}|{\leq}1)\end{aligned}$$

where S captures the content of the tree. We require the different in node numbers of the left and right sub-trees be within one, as the node-balanced property indicates.

User-defined predicates may then be used to specify loop invariants and method pre/post-specifications. In Hip/Sleek , the Hip verifier is used to automatically verify programs against their specifications, while the Sleek prover is invoked by the verifier to conduct entailment proofs. Given two separation formulas Δ_1 and Δ_2, Sleek attempts to prove that Δ_1 entails Δ_2; if it succeeds, it returns a frame R such that $\Delta_1 \vdash \Delta_2 * \mathtt{R}$. For instance, given the entailment

$$\mathtt{p}{::}\mathtt{ll}\langle\mathtt{n}\rangle \wedge \mathtt{n}{>}0 \vdash \exists \mathtt{q} \cdot \mathtt{p}{::}\mathtt{node}\langle\mathtt{q}\rangle$$

Sleek produces the following result after unfolding the LHS predicate:

$$\mathtt{p}{::}\mathtt{ll}\langle\mathtt{n}\rangle \wedge \mathtt{n}{>}0 \vdash \exists \mathtt{q} \cdot \mathtt{p}{::}\mathtt{node}\langle\mathtt{q}\rangle * [\mathtt{q}{::}\mathtt{ll}\langle\mathtt{n}{-}1\rangle \wedge \mathtt{n}{>}0]$$

where the inferred frame, or residue, is shown in squared brackets. The proposed analysis in this paper will use Sleek to perform deductions of separation formulas.

2.2 An Illustrative Example

We illustrate our approach using method `insert_sort` in Fig 1. We show how our analysis infers missing constraints to improve the user-supplied incomplete specification, and how it analyses auxiliary methods without user-annotations.

```
 1 data node { int val; node next; }
 2 node insert_sort(node x)
 3  requires x::llB⟨S⟩
 4  ensures  res::sllB⟨T⟩ {
 5   if (x.next == null) return x;
 6   else { node s = x.next;
 7     node r = insert_sort(s);
 8     return insert(r, x);
 9   }
10 }
11 node insert(node r, node x) {
12   if (r == null) {
13     x.next = null; return x;
14   } else if (x.val <= r.val) {
15     x.next = r; return x;
16   } else {
17     r.next = insert(r.next, x);
18     return r;
19   }
20 }
```

Fig. 1. The insertion sort program for lists.

The `insert_sort` method sorts a singly-linked list. It takes in an unsorted list starting from x with content S and returns a sorted list (lines 3 and 4 where `res` denotes the method return value). The algorithm first sorts the list referenced by `x.next` recursively (line 7), and then inserts node x into the resulted sorted list (line 8). For the node insertion, it invokes another method `insert` for which the user has not provided a specification. We call `insert` an auxiliary method and `insert_sort` a primary one.

For the primary method with a partial specification, our analysis proceeds in two steps. Firstly, starting from the partial precondition, a forward analysis is conducted to compute the postcondition of the method in the form of a *constraint abstraction.* This constraint abstraction is effectively a transfer function for the method, which may be recursively defined. During this analysis, abductive reasoning may be used whenever the current state fails to establish the precondition of the next program command. Secondly, instead of a direct fixpoint computation in the combined abstract domain (with shape, numerical and bag information), a "pure" constraint abstraction (without heap shape information) is derived from the generated constraint abstraction and the user-given partial postcondition. This pure constraint abstraction is then solved by fixpoint solvers in pure (numerical/set) domains, such as [21, 34].

The constraint abstraction of a code segment (e.g. a method) in our settings is an abstraction form of that code's postcondition, given a certain precondition. As the code may contain loops or recursive calls, its constraint abstraction can also be recursive, or in an *open form*, accordingly. To illustrate, for the following while loop and its precondition

$$\{\, \mathtt{x}{\geq}0 \wedge \mathtt{y}{=}0\} \ \mathtt{while}\ (\mathtt{x}{>}0)\ \{\, \mathtt{x} = \mathtt{x} - 1; \mathtt{y} = \mathtt{y} + 1;\, \}$$

we have its constraint abstraction as

$$\mathtt{Q}(\mathtt{x}, \mathtt{x}', \mathtt{y}, \mathtt{y}') ::= \mathtt{x}{=}0 \wedge \mathtt{x}{=}\mathtt{x}' \wedge \mathtt{y}{=}\mathtt{y}' \ \vee\ \mathtt{x}{>}0 \wedge \mathtt{Q}(\mathtt{x}{-}1, \mathtt{x}', \mathtt{y}{+}1, \mathtt{y}')$$

where we denote x and y as their values before the loop, and the primed versions as the values after the loop execution (we will explain this in more detail in Sec 3). Such constraint abstraction presents the postcondition of the while loop. Its fixpoint can be achieved with a standard fixpoint calculation process, with result $\mathtt{x}{\geq}0 \wedge \mathtt{y}{=}0 \wedge \mathtt{x}'{=}0 \wedge \mathtt{y}'{=}\mathtt{x}$. However, as will be seen later, our constraint abstraction is generally more complicated involving both shape and pure constraints, requiring us to split them for solution somehow.

As for the example, our forward analysis runs on the body of `insert_sort` to construct the constraint abstraction. For lines 5-9, it produces a disjunction as the effect of if-else (according to the if-else rule in page 28):

$$\mathtt{Q}(\mathtt{x}, \mathtt{S}, \mathtt{res}, \mathtt{T}) ::= (\text{post-state of if}) \vee (\text{post-state of else})$$

where Q represents the post-state of the if-else statement (as well as the method), and its parameters x, S, res and T are the (program and logical) variables involved in the state.

For the if branch, after the unfolding over $\mathtt{x}{::}\mathtt{llB}\langle \mathtt{S}\rangle$ (rule unfold in page 26), we know from the condition that the input sorted list x has only one node, and thus its post-state will be

$$\exists \mathtt{v} \cdot \mathtt{x}{::}\mathtt{node}\langle \mathtt{v}, \mathtt{null}\rangle \wedge \mathtt{res}{=}\mathtt{x} \wedge \mathtt{S}{=}\{\mathtt{v}\} \qquad (1)$$

Meanwhile, for the else branch, the sorted list will firstly be unrolled by one node at line 6 (rule unfold), making `x.next` point to `s` (rule assign in page 28), which references a sub-list one node shorter than the input list beginning from `x`:

$$\exists \mathtt{S_s}, \mathtt{v} \cdot \mathtt{x}{::}\mathtt{node}\langle \mathtt{v}, \mathtt{s}\rangle * \mathtt{s}{::}\mathtt{sllB}\langle \mathtt{S_s}\rangle \wedge \mathtt{S}{=}\mathtt{S_s}{\sqcup}\{\mathtt{v}\} \tag{2}$$

After that, `insert_sort` is invoked recursively with `s`. It will consume the precondition ($\mathtt{s}{::}\mathtt{sllB}\langle \mathtt{S_s}\rangle$) and ensure the postcondition (in terms of `Q`, partially according to the rule in page 27; however it will be substituted as described later). In that case, the state immediately after symbolic execution of line 7 is

$$\begin{array}{l}\mathtt{Q}(\mathtt{x}, \mathtt{S}, \mathtt{res}, \mathtt{T}) ::= \exists \mathtt{v} \cdot \mathtt{x}{::}\mathtt{node}\langle \mathtt{v}, \mathtt{null}\rangle \wedge \mathtt{res}{=}\mathtt{x} \wedge \mathtt{S}{=}\{\mathtt{v}\} \vee \\ \quad \exists \mathtt{v}, \mathtt{s}, \mathtt{S_s}, \mathtt{r}, \mathtt{S_r} \cdot \mathtt{x}{::}\mathtt{node}\langle \mathtt{v}, \mathtt{s}\rangle * \mathtt{Q}(\mathtt{s}, \mathtt{S_s}, \mathtt{r}, \mathtt{S_r}) \wedge |\mathtt{S}|{>}1 \wedge \mathtt{S}{=}\mathtt{S_s}{\sqcup}\{\mathtt{v}\}\end{array}$$

Note that existential variables (not in the parameter list of `Q`) are local variables whose quantification may be omitted for brevity. The first disjunctive branch corresponds to the base case in the method body, and the second branch captures the effect of the recursive call (with `Q`).

Then the forward analysis continues over line 8 to invoke `insert`. Because the user has provided no annotations for that method, its specifications must be synthesised. For this purpose we replace $\mathtt{Q}(\mathtt{s}, \mathtt{S_s}, \mathtt{r}, \mathtt{S_r})$ in second branch with $\mathtt{r}{::}\mathtt{sllB}\langle \mathtt{S_r}\rangle \wedge \mathtt{P}(\mathtt{s}, \mathtt{S_s}, \mathtt{r}, \mathtt{S_r})$ to make explicit the heap portion referred to by `r` before we analyse the auxiliary call $\mathtt{insert}(\mathtt{r}, \mathtt{x})$ (rule call-inf in page 27). This is safe because the following entailment relationship is added to our assumption:

$$\mathtt{Q}(\mathtt{x}, \mathtt{S}, \mathtt{res}, \mathtt{T}) \vdash \mathtt{res}{::}\mathtt{sllB}\langle \mathtt{T}\rangle \wedge \mathtt{P}(\mathtt{x}, \mathtt{S}, \mathtt{res}, \mathtt{T}) \tag{3}$$

which signifies that `Q` can be abstracted as a sorted list referenced by `res` plus some pure constraints `P` (also in constraint abstraction form, whose definition is to be derived in the next step). and hence `insert`'s precondition can be figured out from the symbolic state at call site, and its postcondition will be computed as well. The analysis for such auxiliary methods (including loops) works in the same way as that for primary methods, except that a pre-analysis is involved to figure out the raw pre/post shape information (before invoking the analysis algorithm for primary methods). More details are explained slightly later.

$$\mathtt{requires}\ \mathtt{r}{::}\mathtt{sllB}\langle \mathtt{S}\rangle * \mathtt{x}{::}\mathtt{node}\langle \mathtt{v}, \_\rangle\ \mathtt{ensures}\ \mathtt{res}{::}\mathtt{sllB}\langle \mathtt{T}\rangle \wedge \mathtt{T}{=}\mathtt{S}{\sqcup}\{\mathtt{v}\} \tag{4}$$

which indicates that the returned list has the same content as the input list (`x`) plus $\{\mathtt{v}\}$. Applying it, we obtain the following post-state for `insert_sort`:

$$\begin{array}{l}\mathtt{Q}(\mathtt{x}, \mathtt{S}, \mathtt{res}, \mathtt{T}) ::= \mathtt{x}{::}\mathtt{node}\langle \mathtt{v}, \mathtt{null}\rangle \wedge \mathtt{res}{=}\mathtt{x} \wedge \mathtt{S}{=}\{\mathtt{v}\} \vee \\ \quad \mathtt{res}{::}\mathtt{sllB}\langle \mathtt{S_{res}}\rangle \wedge \mathtt{P}(\mathtt{s}, \mathtt{S_s}, \mathtt{r}, \mathtt{S_r}) \wedge |\mathtt{S}|{>}1 \wedge \mathtt{S}{=}\mathtt{S_s}{\sqcup}\{\mathtt{v}\} \wedge \mathtt{S_{res}}{=}\mathtt{S_r}{\sqcup}\{\mathtt{v}\}\end{array}$$

The first disjunctive branch corresponds to the base case, but the second branch now captures the effect of the recursive call as well as the auxiliary call (to `insert`). In the base case, the method's return pointer (`res`) points to one node with value `v`. The recursive branch signifies that the post-state of the method concerns the recursive call and the auxiliary call (over $\mathtt{s}$ and $\mathtt{r}$), as the constraint abstraction denotes. Note that $\mathtt{T}$ will be not available (as well as its relationship with $\mathtt{S_{res}}$) until next step.

In the second step, we first derive the definition of the pure constraint abstraction $\mathtt{P}$ from the above post-state $\mathtt{Q}$. Each disjunctive branch of $\mathtt{Q}$ is used to entail the user-given post-shape (with appropriate instantiations of the parameters). The obtained frames form (via disjunction) the definition of $\mathtt{P}$. For `insert_sort`, we obtain the following pure constraint abstraction:

$$\mathtt{P(x,S,res,T)} ::= (\mathtt{T{=}S} \wedge |\mathtt{S}|{=}1) \vee (\mathtt{P(s,S_s,r,S_r)} \wedge |\mathtt{S}|{>}1 \wedge \mathtt{S{=}S_s}{\sqcup}\{\mathtt{v}\} \wedge \mathtt{T{=}S_r}{\sqcup}\{\mathtt{v}\})$$

We then use pure fixpoint solvers to obtain a closed-form formula $|\mathtt{S}|{\geq}1 \wedge \mathtt{T{=}S}$ for $\mathtt{P}$. Based on (3), we now obtain the closed-form approximation for $\mathtt{Q}$:

$$\mathtt{Q(x,S,res,T)} ::= \mathtt{res{::}sllB\langle T\rangle} \wedge |\mathtt{S}|{\geq}1 \wedge \mathtt{T{=}S}$$

The obtained pure formula is then used to refine the method's specification as

$$\mathtt{requires}\ \mathtt{x{::}llB\langle S\rangle} \wedge |\mathtt{S}|{\geq}1 \quad \mathtt{ensures}\ \mathtt{res{::}sllB\langle T\rangle} \wedge \mathtt{T{=}S}$$

which imposes more requirement in the precondition, stating that there should be at least one node in the list to be sorted for the sake of memory safety. With that obligation, the method guarantees that the result list is sorted and its content remains the same as the input list.

2.3 Analysis for the Unannotated Method in Example

The unannotated method `insert` in the example inserts a node `x` into a sorted list `r`. It judges three cases and has a non-tail-recursive call to itself in the last case (to insert `x` after list `r`'s head). As we want to minimise user's annotations, we do not require the user to supply loop invariant; instead we will calculate the loop's postcondition. Since no user-annotations are provided, our analysis synthesises its (raw) pre- and post-shapes which are then refined in the same way as for primary methods. The pre-shape is directly synthesised from the abstract program state at the call site ($\mathtt{x{::}node\langle v{,}s\rangle} * \mathtt{r{::}sllB\langle S_r\rangle}$). We unroll the recursive call once, symbolically execute the unrolled method body (starting from the pre-shape) to obtain a post-state, and then use the post-state to filter out any invalid post-shapes from the set of possible post-shapes (drawn from all available shape predicates). For this example, the possible post-shapes can be (a) $\mathtt{x{::}sllB\langle S_1\rangle} * \mathtt{res{::}sllB\langle S_2\rangle}$, and (b) $\mathtt{res{::}sllB\langle S\rangle}$, etc. The symbolic execution gives the following post-state:

$$\begin{array}{l}\mathtt{x{::}node\langle v,null\rangle} \wedge \mathtt{x{=}res} \vee \mathtt{x{::}node\langle v,r\rangle} * \mathtt{r{::}sllB\langle S_1\rangle} \wedge \mathtt{x{=}res} \wedge (\forall \mathtt{u}{\in}\mathtt{S_1}{\cdot}\mathtt{v}{\leq}\mathtt{u}) \vee \\ \quad \mathtt{r{::}node\langle u,x\rangle} * \mathtt{x{::}node\langle v,null\rangle} \wedge \mathtt{r{=}res} \wedge \mathtt{u}{\leq}\mathtt{v} \vee \\ \quad \mathtt{r{::}node\langle u,x\rangle} * \mathtt{x{::}node\langle v,r_1\rangle} * \mathtt{r_1{::}sllB\langle S_1\rangle} \wedge \mathtt{r{=}res} \wedge \mathtt{u}{\leq}\mathtt{v} \wedge (\forall \mathtt{w}{\in}\mathtt{S_1}{\cdot}\mathtt{v}{\leq}\mathtt{w})\end{array}$$

which does not entail the candidate (a), so we filter it out.Taking (b) as the post-shape, we can employ the same analysis for the primary method to obtain the specification (4) (page 7 for `insert` and continue with the analysis for the primary method.

2.4 Another Illustrative Example

We illustrate our approach with another more interesting example. We show how the user is expected to provide shape information for specifications of a primary method, and how our proposed analysis will refine such specifications with pure constraints, and derive specifications for loops without annotations.

```
 0 data node2 { int val;
     node2 prev; node2 next; }
 1 node2 sdl2nbt(node2 head,
                 node2 tail)
 2  requires  head::sdlB⟨p, q, S⟩
 3  ensures  res::nbt⟨S_res⟩
 4 {
 5  node2 root = head;
 6  node2 end = head;
 7  while(end != tail) {
 8    end = end.next;
 9    if (end != tail) {
10      end = end.next;
11      root = root.next;
12    }
13  }
14  if (head == root)
15    root.prev = null;
16  else
17    root.prev = sdl2nbt(head, root);
18  node2 tmp = root.next;
19  if (tmp == tail)
20    root.next = null;
21  else {
22    tmp.prev = null;
23    root.next = sdl2nbt(tmp, tail);
24  }
25  return root;
26 }
```

Fig. 2. The sorted doubly-linked list to node-balanced tree method.

Let us consider the method `sdl2nbt` shown in Fig 2. Taking in a sorted doubly-linked list as argument, `sdl2nbt` will convert it into a node-balanced tree together with binary search properties, as indicated in lines 2 and 3. Its algorithm proceeds as follows: first it finds the "centre" node in the list (`root`), where the difference of numbers of its left and right nodes is at most one, as Fig 3 (a) indicates (lines 5-13). Then it applies the algorithm recursively on both list segments on the centre's left and right hand sides, and regards the centre node as the tree's root, whose left and right children are the resulted subtrees' roots from the recursive calls, as in Fig 3 (b) and (c) (lines 14-25). As the data structure of doubly-linked list and binary tree are homomorphic (line 0), we reuse the nodes in the input list instead of creating a new tree, making this algorithm in-place. The parameter `head` in line 1 denotes the first node of the input list, and `tail` is where the list's last node's `next` field points to. When using this method `tail` should be set as `null` initially.

Our framework allows the user to verify and/or refine a number of properties about this code. Firstly, the transformation of shapes from initial to final states (namely, from a doubly-linked list to a binary tree) must be captured. Secondly, some structural numerical information should be inferred, so as to prove the node counts before and after the method invocation are the same and the node-

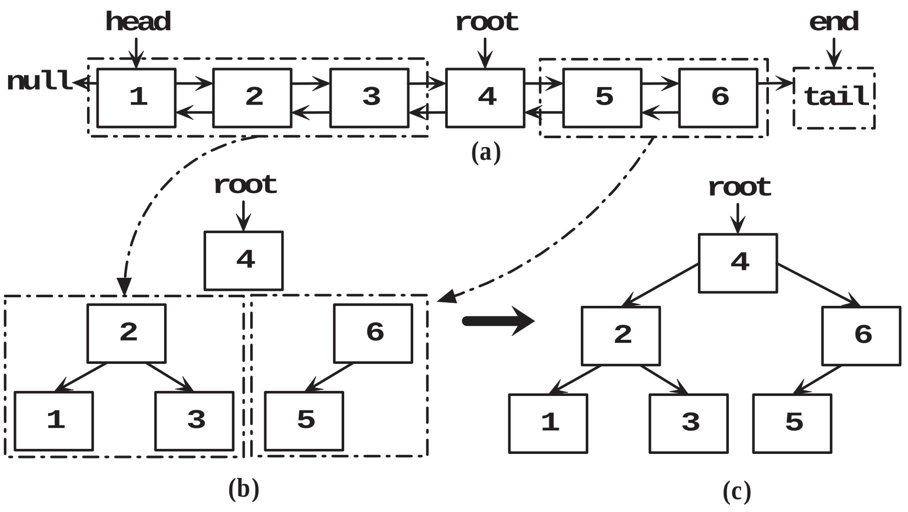

Fig. 3. Transferring from a sorted doubly-linked list to a node-balanced BST.

balanced property of the tree, etc. Meanwhile, we also want to derive relational numerical information as lists' sortedness and trees' binary search property, and finally set/bag information like the symbolic content of the list's and the tree's (in order to prove the values stored in the list and the resulted tree are the same). Finally, some obligation for memory safety should be found in the precondition, to ensure the input list is non-empty (otherwise the dereference in line 15/17 will fail). To deal with all these properties, we expect the user to provide shape information for primary methods' specifications as in Fig 2. Based on that, we try to compute the remaining constraints (including the missing parts of pure specifications for primary methods, and both pre- and post-conditions for loops and auxiliary methods).

As for the example, as the user has provided the pre- and post-shapes for method `sdl2nbt`, our analysis proceeds in two steps: generating the constraint abstraction, and solving it. The first step is mainly a symbolic execution over the program to find its postcondition, so as to generate the constraint abstraction. During this step, for any loops and/or auxiliary method calls (lines 7-13 in the example), the symbolic execution will invoke the analysis procedure again to compute their specifications for the current execution to continue (Section 2.5). Therefore we can find the while loop's postcondition as

$$\begin{array}{l} \texttt{head}::\texttt{sdlB}\langle\texttt{null},\texttt{root},\mathtt{S_h}\rangle * \texttt{root}::\texttt{sdlB}\langle\texttt{p},\texttt{tail},\mathtt{S_r}\rangle \wedge \\ \quad \texttt{end}{=}\texttt{tail} \wedge \mathtt{S}{=}\mathtt{S_h}{\sqcup}\mathtt{S_r} \wedge (\forall \mathtt{x}{\in}\mathtt{S_h}, \mathtt{y}{\in}\mathtt{S_r}\cdot\mathtt{x}{\leq}\mathtt{y}) \wedge \underline{0{\leq}|\mathtt{S_h}|{-}|\mathtt{S_r}|{\leq}1} \end{array} \tag{5}$$

which indicates that the original list segment starting from `head` is cut into two pieces with a cutpoint `root`, where both are still sorted and the content is also preserved. Meanwhile, the essential constraint (the underlined part, saying the list beginning with head is at most one node longer than that with root) to ensure the node-balanced property is derived as well.

When the symbolic execution finishes, it generates the following constraint abstraction as the postcondition of the method:

$$
\begin{array}{l}
\mathtt{Q}(\mathtt{head}, \mathtt{p}, \mathtt{q}, \mathtt{S}, \mathtt{res}, \mathtt{S_{res}}) ::= \\
\quad \mathtt{root}{::}\mathtt{node2}\langle \mathtt{v}, \mathtt{null}, \mathtt{null}\rangle \wedge \mathtt{head}{=}\mathtt{root}{=}\mathtt{res} \wedge \mathtt{tmp}{=}\mathtt{q}{=}\mathtt{tail} \wedge \mathtt{p}{=}\mathtt{null} \wedge \\
\qquad \mathtt{S}{=}\{\mathtt{v}\} \vee \\
\quad \mathtt{head}{::}\mathtt{node2}\langle \mathtt{s}, \mathtt{null}, \mathtt{root}\rangle * \mathtt{root}{::}\mathtt{node2}\langle \mathtt{v}, \mathtt{res_h}, \mathtt{null}\rangle \wedge \mathtt{res}{=}\mathtt{root} \wedge \\
\qquad \mathtt{tmp}{=}\mathtt{q}{=}\mathtt{tail} \wedge \mathtt{p}{=}\mathtt{null} \wedge \mathtt{S}{=}\{\mathtt{s}, \mathtt{v}\} \wedge \mathtt{s}{\leq}\mathtt{v} \vee \\
\quad \mathtt{res_h}{::}\mathtt{nbt}\langle \mathtt{S^h_{res}}\rangle * \mathtt{res_r}{::}\mathtt{nbt}\langle \mathtt{S^r_{res}}\rangle * \mathtt{root}{::}\mathtt{node2}\langle \mathtt{v}, \mathtt{res_h}, \mathtt{res_r}\rangle \wedge \\
\qquad \mathtt{P}(\mathtt{head}, \mathtt{p}, \mathtt{root}, \mathtt{S_h}, \mathtt{res_h}, \mathtt{S^h_{res}}) \wedge \mathtt{P}(\mathtt{tmp}, \mathtt{null}, \mathtt{tail}, \mathtt{S_r}, \mathtt{res_r}, \mathtt{S^r_{res}}) \wedge \\
\qquad \mathtt{head}{\neq}\mathtt{root} \wedge \mathtt{root}{=}\mathtt{res} \wedge \mathtt{tmp}{\neq}\mathtt{tail} \wedge \mathtt{q}{=}\mathtt{tail} \wedge \\
\qquad \mathtt{S}{=}\mathtt{S_h}{\sqcup}\{\mathtt{v}\}{\sqcup}\mathtt{S_r} \wedge (\forall \mathtt{x}{\in}\mathtt{S_h}, \mathtt{y}{\in}\mathtt{S_r}{\cdot}\mathtt{x}{\leq}\mathtt{v}{\leq}\mathtt{y}) \wedge 0{\leq}|\mathtt{S_h}|{-}|\mathtt{S_r}|{\leq}1
\end{array}
$$

where P stands for corresponding pure constraint abstraction explained below. The first two disjunctive branches are base cases of the method's invocation, and the last denotes the effect of recursive calls combined into the postcondition. The first case represents the scenario where there is only one node in the original list (with res as the method's return value). The second is for the case of two nodes, one referenced by head, pointing to the other one, root. In this case the value of head is no more than that of root. The third case is defined recursively with the constraint abstraction itself, meaning that the post-state concerns the root node and the post-states of two recursive calls over head and tmp, respectively. Note that $\mathtt{S_{res}}$ does not appear in Q's definition. Since it stands for pure properties in user-provided post-shape, it will be involved when we abstract Q against that post-shape in the next step.

The second step solves the constraint abstraction Q by finding a closed-form approximation of it. Instead of performing a fixpoint analysis directly on Q over the combined domain, we first derive a pure constraint abstraction P (with the help of SLEEK) from Q and the user-provided heap part of postcondition. Then we are able to use some existing conventional solvers [21, 34] to compute the pure fixpoint. For the sdl2nbt method, we generate the pure constraint abstraction P based on the following entailment relation:

$$
\mathtt{Q}(\mathtt{head}, \mathtt{p}, \mathtt{q}, \mathtt{S}, \mathtt{res}, \mathtt{S_{res}}) \vdash \mathtt{res}{::}\mathtt{nbt}\langle \mathtt{S_{res}}\rangle \wedge \mathtt{P}(\mathtt{head}, \mathtt{p}, \mathtt{q}, \mathtt{S}, \mathtt{res}, \mathtt{S_{res}})
$$

which produces the following pure constraint abstraction P:

$$
\begin{array}{l}
\mathtt{P}(\mathtt{head}, \mathtt{p}, \mathtt{q}, \mathtt{S}, \mathtt{res}, \mathtt{S_{res}}) ::= \\
\quad \mathtt{head}{=}\mathtt{root}{=}\mathtt{res} \wedge \mathtt{tmp}{=}\mathtt{q}{=}\mathtt{tail} \wedge \mathtt{p}{=}\mathtt{null} \wedge \mathtt{S}{=}\mathtt{S_{res}}{=}\{\mathtt{v}\} \vee \\
\quad \mathtt{head}{\neq}\mathtt{root} \wedge \mathtt{res}{=}\mathtt{root} \wedge \mathtt{tmp}{=}\mathtt{q}{=}\mathtt{tail} \wedge \mathtt{p}{=}\mathtt{null} \wedge \\
\qquad \mathtt{S}{=}\mathtt{S_{res}}{=}\{\mathtt{s}, \mathtt{v}\} \wedge \mathtt{s}{\leq}\mathtt{v} \vee \\
\quad \mathtt{P}(\mathtt{head}, \mathtt{p}, \mathtt{root}, \mathtt{S_h}, \mathtt{res_h}, \mathtt{S^h_{res}}) \wedge \mathtt{P}(\mathtt{tmp}, \mathtt{null}, \mathtt{tail}, \mathtt{S_r}, \mathtt{res_r}, \mathtt{S^r_{res}}) \wedge \\
\qquad \mathtt{head}{\neq}\mathtt{root} \wedge \mathtt{root}{=}\mathtt{res} \wedge \mathtt{tmp}{\neq}\mathtt{tail} \wedge \mathtt{q}{=}\mathtt{tail} \wedge \mathtt{S}{=}\mathtt{S_h}{\sqcup}\{\mathtt{v}\}{\sqcup}\mathtt{S_r} \wedge \\
\qquad \mathtt{S_{res}}{=}\mathtt{S^h_{res}}{\sqcup}\{\mathtt{v}\}{\sqcup}\mathtt{S^r_{res}} \wedge (\forall \mathtt{x}{\in}\mathtt{S_h}, \mathtt{y}{\in}\mathtt{S_r}{\cdot}\mathtt{x}{\leq}\mathtt{v}{\leq}\mathtt{y}) \wedge 0{\leq}|\mathtt{S_h}|{-}|\mathtt{S_r}|{\leq}1
\end{array}
$$

Note that the heap information is already eliminated from $\mathtt{P}$; instead the constraints over $\mathtt{S_{res}}$ are included during the entailment checking procedure. This allows us to solve $\mathtt{P}$ to refine the user-provided shape-only specification.

After solving $\mathtt{P}$, we achieve the following constraint:

$$\mathtt{p{=}null} \wedge \mathtt{q{=}tail} \wedge \mathtt{S{=}S_{res}} \wedge |\mathtt{S}|{\geq}1$$

with which we can refine the method's specifications as

$$\begin{array}{ll} \mathtt{requires} & \mathtt{head{::}sdlB}\langle \mathtt{p,q,S}\rangle \wedge \mathtt{p{=}null} \wedge \mathtt{q{=}tail} \wedge |\mathtt{S}|{\geq}1 \\ \mathtt{ensures} & \mathtt{res{::}nbt}\langle \mathtt{S_{res}}\rangle \wedge \mathtt{S{=}S_{res}} \end{array}$$

which proposes more requirements in the precondition, as the `head`'s `prev` field should be `null`, and the whole list's last node's `next` field must point to `tail`. Meanwhile, there should be at least one node in the list for the sake of memory safety. With those obligations, the method guarantees that the result is an node-balanced tree with binary search property, whose content is the same as the input list.

2.5 Analysis for the While Loop in Example

In the method `sdl2nbt`, there is a while loop (lines 7-13) to discover the centre node of the given list segment referenced by `head`. It traverses the list segment with two pointers `root` and `end`. The `end` pointer goes towards the list segment's tail twice as fast as `root`. When `end` arrives at the tail of the segment (`tail`), `root` will point to the list segment's centre node.

As we want to minimise user's annotations, we do not require the user to supply loop invariant; instead we will try to calculate its postcondition. As aforementioned, our analysis must first synthesise its pre- and post-states with shape information, and then proceed with its constraint abstraction. For pre it is straightforward as the program state before the loop will provide relevant shape information. For post it is done by checking the loop body (unrolled once)'s symbolic execution result against all possible abstracted shapes. For the previous example, we first generate all possible shapes according to the variables accessed by the loop, such as (a) $\mathtt{head{::}sdlB}\langle \mathtt{p_h,q_h,S_h}\rangle * \mathtt{root{::}sdlB}\langle \mathtt{p_r,q_r,S_r}\rangle$, and (b) $\mathtt{head{::}sdlB}\langle \mathtt{p_h,q_h,S_h}\rangle * \mathtt{root{::}nbt}\langle \mathtt{h_r,b_r,S_r}\rangle$, and many so forth. Then the unrolled loop body is symbolically executed several times to filter out any invalid shape as an invariant. In the example's case, executing the loop body will yield the following result:

$$\begin{array}{l} \mathtt{head{::}node2}\langle \mathtt{v,p,end}\rangle \wedge \mathtt{head{=}root} \wedge \mathtt{end{=}tail} \vee \\ \quad \mathtt{head{::}node2}\langle \mathtt{v_h,p,root}\rangle * \mathtt{root{::}node2}\langle \mathtt{v_r,head,end}\rangle \wedge \mathtt{end{=}tail} \end{array} \qquad (6)$$

where (b) is directly filtered out since (6) $\vdash (b) * \mathtt{true}$ fails. However (a) remains a candidate, as both (6) $\vdash (a) * \mathtt{true}$ holds. Therefore, regarding (a) as a possible shape post, we can employ the same approach for the whole method to generate a constraint abstraction for the while loop, and solve it to achieve formula (6) in the last section.

One more note for the while loop in this example is that the symbolic execution may actually permit more than one shapes to enter as candidates, e.g. $\texttt{head}{::}\texttt{sdlB}\langle \texttt{p}_\texttt{h}, \texttt{q}_\texttt{h}, \texttt{S}_\texttt{h}\rangle$. Generally this does not affect the analysis result, as we allow the analysis to continue with all possible postconditions computed from this while loop, and always choose the most precise final result. In the motivating example, both $\texttt{head}{::}\texttt{sdlB}\langle \texttt{p}_\texttt{h}, \texttt{q}_\texttt{h}, \texttt{S}_\texttt{h}\rangle$ and (a) are valid shape postconditions for the loop, but later the former one will cause the analysis to fail in line 15/17, because it inappropriately approximated the invariant and hence lost information about `root`. Since we synthesise all possible shapes, we can always select those shapes sufficiently strong to support further analysis to obtain a meaningful result.

3 Language and Abstract Domain

To simplify presentation, we focus on a strongly-typed C-like imperative language in Fig 4. A program *Prog* consists of type declarations *tdecl*, which can define either data type *datat* (e.g. `node`) or predicate *spred* (e.g. `llB`), and some method declarations *meth*. The definitions for *spred* and *mspec* are given later in Fig 5.

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

Fig. 4. A Core (C-like) Imperative Language.

Note that the language is expression-oriented, so the body of a method is an expression composed of standard instructions and constructors of an imperative language. e is the (recursively defined) program constructor and d and $d[v]$ are atom instructions. Note also that the language allows both call-by-value and call-by-reference method parameters (which are separated with a semicolon ;where the ones before ; are call-by-value and the ones after are call-by-reference).

Our specification language (in Fig 5) allows (user-defined) shape predicates to specify both separation and pure properties. The shape predicates *spred* and lemmas *lemma* are constructed with disjunctive constraints Φ. We require that the predicates be well-formed [32].

A conjunctive abstract program state, σ, is composed of a heap (shape) part κ and a pure part π, where π consists of γ, ϕ and φ as aliasing, numerical and bag information, respectively. We use SH to denote a set of such conjunctive states.

$$
\begin{array}{lll}
\mathit{spred} & ::= \texttt{root}{::}c\langle v^*\rangle \equiv \Phi & \quad \mathit{lemma} ::= \texttt{root}{::}c\langle v^*\rangle \wedge \pi \longleftarrow \Phi \\
\mathit{mspec} & ::= \mathit{requires}\ \Phi_{pr}\ \mathit{ensures}\ \Phi_{po} & \\
\Delta & ::= \texttt{Q}(v^*) \mid \Phi \mid \Delta_1{\vee}\Delta_2 \mid \Delta{\wedge}\pi \mid \Delta_1{*}\Delta_2 \mid \exists v{\cdot}\Delta & \\
\Phi & ::= \bigvee \sigma^* \qquad \sigma ::= \exists v^*{\cdot}\kappa{\wedge}\pi & \\
\Upsilon & ::= \texttt{P}(v^*) \mid \bigvee \omega^* \mid \Upsilon_1{\wedge}\Upsilon_2 \mid \Upsilon_1{\vee}\Upsilon_2 \mid \exists v{\cdot}\Upsilon & \\
\kappa & ::= \texttt{emp} \mid v{::}c\langle v^*\rangle \mid \kappa_1 * \kappa_2 \qquad \omega ::= \exists v^*{\cdot}\pi \qquad \pi ::= \gamma{\wedge}\phi & \\
\gamma & ::= v_1{=}v_2 \mid v{=}\texttt{null} \mid v_1{\neq}v_2 \mid v{\neq}\texttt{null} \mid \gamma_1{\wedge}\gamma_2 & \\
\phi & ::= \varphi \mid b \mid a \mid \phi_1{\wedge}\phi_2 \mid \phi_1{\vee}\phi_2 \mid \neg\phi \mid \exists v \cdot \phi \mid \forall v \cdot \phi & \\
b & ::= \texttt{true} \mid \texttt{false} \mid v \mid b_1 = b_2 \qquad a ::= s_1{=}s_2 \mid s_1{\leq}s_2 & \\
s & ::= k^{\texttt{int}} \mid v \mid k^{\texttt{int}}{\times}s \mid s_1{+}s_2 \mid -s \mid \mathit{max}(s_1{,}s_2) \mid \mathit{min}(s_1{,}s_2) \mid |\texttt{B}| & \\
\varphi & ::= v{\in}\texttt{B} \mid \texttt{B}_1{=}\texttt{B}_2 \mid \texttt{B}_1{\sqsubset}\texttt{B}_2 \mid \texttt{B}_1{\sqsubseteq}\texttt{B}_2 \mid \forall v{\in}\texttt{B}{\cdot}\phi \mid \exists v{\in}\texttt{B}{\cdot}\phi & \\
\texttt{B} & ::= \texttt{B}_1{\sqcup}\texttt{B}_2 \mid \texttt{B}_1{\sqcap}\texttt{B}_2 \mid \texttt{B}_1{-}\texttt{B}_2 \mid \{\} \mid \{v\} &
\end{array}
$$

Fig. 5. The Specification Language.

During the symbolic execution, the abstract program state at each program point will be a disjunction of σ's, denoted by Δ. Note that constraint abstractions (e.g. $\texttt{Q}(\texttt{v}^*)$) may occur in Δ during the analysis. A closed-form Δ (containing no constraint abstractions) can be normalised to the Φ form [32]. The definition of pure constraint abstraction $\texttt{P}$ is analogous to that of $\texttt{Q}$.

The memory model of our specification formulae is adapted from the model given for "early versions" of separation logic [35], except that we have extensions to handle user-defined shape predicates and related pure properties. We assume sets Loc of memory locations, Val of primitive values (with $0 \in \mathsf{Val}$ denoting $\texttt{null}$), Var of variables (program and logical variables), and ObjVal of object values stored in the heap, with $c[f_1{\mapsto}\nu_1, .., f_n{\mapsto}\nu_n]$ denoting an object value of data type c where $\nu_1, .., \nu_n$ are current values of the corresponding fields $f_1, .., f_n$. Let $s, h \models \Delta$ denote the model relation, i.e. the stack s and heap h satisfy Δ, with h, s from the following concrete domains:

$$h \in \ \mathit{Heaps} \ =_{\mathit{df}} \mathsf{Loc} \rightharpoonup_{\mathit{fin}} \mathsf{ObjVal} \qquad s \in \ \mathit{Stacks} \ =_{\mathit{df}} \mathsf{Var} \rightarrow \mathsf{Val} \cup \mathsf{Loc}$$

Note that each heap h is a finite partial mapping while each stack s is a total mapping, as in the classical separation logic [20, 35].The detailed model definitions can be found in Nguyen et al. [32].

In the analysis we use three kinds of variables in the Var set: program variables, logical variables related to program variables' shapes (such as a list's length), and logical variables to record intermediate states. For the first two groups we use variables without subscription (such as $\texttt{x}$ and $\texttt{xn}$), and denote a program variable's initial value as unprimed, and its current (and hence final) value as primed [8, 32]. For the third group, we use subscript ones like $\texttt{x}_1$ and $\texttt{xn}_1$. For instance, for a code segment $\texttt{x} := \texttt{x} + 1;\ \texttt{x} := \texttt{x} - 2$ starting with state $\{\texttt{x}{>}1\}$, we have the following reasoning procedure:

$$\{\texttt{x}'{=}\texttt{x} \wedge \texttt{x}{>}1\}\ \texttt{x:=x+1}\ \{\texttt{x}{>}1 \wedge \texttt{x}'{=}\texttt{x}{+}1\}\ \texttt{x:=x-2}\ \{\texttt{x}{>}1 \wedge \texttt{x}'{=}\texttt{x}_1{-}2 \wedge \texttt{x}_1{=}\texttt{x}{+}1\}$$

where the final value of `x` is recorded in variable $\mathtt{x}'$ and $\mathtt{x}_1$ keeps an intermediate state of `x`.

4 The Analysis

Given a method to be verified, our main analysis is performed in two steps. The first step recognises whether the method is an auxiliary one/transferred from a loop (without user annotations), or a primary method with user-supplied shape specifications. If it is auxiliary or transferred from a loop, then it will undergo a pre-processing (Fig 6) which discovers a list of candidate shape specifications for that method.

```
Algorithm Preproc($\mathcal{T}, \mathcal{S}, f, u^*, v^*, e, \sigma, x^*, y^*$)
1    $sps := [\,]$;
2    $prs :=$ SynPre($\mathcal{S}, f, u^*, v^*, \sigma, x^*, y^*$)
3    for $\Phi_{pr} \in prs$ do
4        $pos :=$ SynPost($\mathcal{T}, \mathcal{S}, f, e, \Phi_{pr}, u^*, v^*$)
5        $sps :=$ concat($sps, pos$)
6    end for
7    return ($sps, |sps|$)
end Algorithm
```

Fig. 6. Pre-processing algorithm.

The pre-processing algorithm mainly invokes the shape synthesis procedures to discover all possible pre- and post-shapes for loops and auxiliary methods, as shown in lines 1 and 4. Then the list of shape pairs (specifications) are returned and used in further analysis. The details of shape synthesis algorithms will be introduced in Section 4.3.

After the first step, the method to be verified is guaranteed with some shape specifications. Then the second step will generate constraint abstractions and solve them according to the shapes given in the specifications (described in detail in Section 4.1). The solutions are then used to refine the shape specifications with pure constraints.

For the auxiliary methods and loops, we apply a lazy scheme: as the pre-processing may yield several possible shape specifications in a list (ordered with heuristics such that the specifications with more possibility to make the whole verification succeed are closer to the list head), we try to verify each in sequence. Once a specification can be verified against the program, then it is returned and the other ones are omitted. This is reasonable as our main purpose is to verify and refine the main method, and thus we view the specifications gained for the auxiliary methods as affiliated results. In this way we try to make our verification more scalable, as will be described in later sections.

4.1 Refining Specifications for Primary Methods

The algorithm for refinement (CA_Gen_Solve) is given in Fig 7. As illustrated in Section 2.2, the analysis proceeds in two steps for a primary method with shape information given in specification, namely, the forward analysis (lines 1-3) and the pure constraint abstraction derivation and solving (lines 4-13).

```
Algorithm CA_Gen_Solve($\mathcal{T}, mn, e, \Phi_{pr}, \Phi_{po}, u^*, v^*$)
 1  Denote the CA as $\mathtt{Q}(w^*)$ where $w^*$={$u^*$,$v^*$,$v'^*$,
        pureV({$u^*, v^*, v'^*$}, $\Phi_{pr} \vee \Phi_{po}$)}
 2  $\Delta$ := Symb_Exec($\mathcal{T}, mn, e, \Phi_{pr}$)
 3  Normalize $\Delta$ to DNF, and denote as $\bigvee_{i=1}^{m} \Delta_i$
 4  for $i := 1$ to $m$ do
 5      $\mathtt{P}_i$ := Pure_Abstraction($\Phi_{po}, \Delta_i, \mathtt{Q}(w^*)$)
 6      if $\mathtt{P}_i$ = fail then return fail end if
 7  end for
 8  if $i$=$m$+1 then
 9      $\pi$ := Pure_CA_Solve($\mathtt{P}(w^*) := \bigvee_{j=1}^{m} \mathtt{P}_j$)
10      $R := t\ mn\ ((t\ u)^*; (t\ v)^*)$ requires
            ex_quan($\Phi_{pr}, \pi$) ensures ex_quan($\Phi_{po}, \pi$)
11  end if
12  if Verify($\mathcal{T}, mn, R$) then return $\mathcal{T} \cup \{R\} \setminus$
        { $t\ mn\,((t\ u)^*; (t\ v)^*)$ requires $\Phi_{pr}$ ensures $\Phi_{po}$ }
13  else return fail end if
end Algorithm
```

```
Algorithm Symb_Exec
   ($\mathcal{T}, mn, e, \Phi_{pr}$)
 1 $lastLbl := 0$
 2 do
 3   $(\Delta, l) := [\![e]\!]_{\mathcal{T}}(\Phi_{pr}, 0)$
 4   if $l > 0 \wedge lastLbl \neq l$
     then
 5     $\Phi_{pr}$:=ex_quan($\Phi_{pr}$,$\Delta$);
 6     $lastLbl := l$
 7   else if $l$>0 $\wedge$ $lastLbl$=$l$
     then return fail
 8   end if
 9 while $l > 0$
10 return $\Delta$
end Algorithm
```

Fig. 7. Refining method specifications.

Lines 1-3 analyses the method body starting from the (possibly incomplete) precondition (i.e., the pre-shape) to compute the post-state in constraint abstraction form. Along the analysis, missing pure requirements are derived and used to strengthen the precondition.

The forward analysis (line 2 on the left and line 3 on the right as invoked) is conducted using a set of symbolic execution rules to be explained in Appendix A. If the symbolic execution of the method body succeeds (suggesting that the pre-shape is sufficiently strong),the analysis moves on to the second step (lines 4-13). However, if the symbolic execution fails at some point, where the current symbolic state cannot meet the requirement of the next instruction, it can be due to the lack of pure (i.e. numerical/bag) constraints in the precondition. To deal with this, we enhance the symbolic execution with *pure abduction mechanism* (whose details are given later). For example, if we have $\mathtt{x}{::}\mathtt{ll}\langle\mathtt{n}\rangle$ as the current state and we require $\mathtt{x}{::}\mathtt{node}\langle\_, \mathtt{p}\rangle$ to update the value of `p`, then it will fail as

$\texttt{x::ll}\langle\texttt{n}\rangle$ does not necessarily guarantee $\texttt{x::node}\langle\_,\texttt{p}\rangle$. In this case we conduct the pure abduction as

$$\texttt{x::ll}\langle\texttt{n}\rangle \wedge [\texttt{n}{\geq}1] \rhd \texttt{x::node}\langle\_,\texttt{p}\rangle * \texttt{true}$$

to compute the missing pure information (in the squared bracket) such that the LHS (including the newly gained pure part) entails the RHS.

The Symb_Exec on the right is our symbolic execution algorithm. The variable *lastLbl* (initialised at line 1) is to record the program location in which the most recent pure abduction occurs. When the symbolic execution fails, it returns the current state Δ (containing pure abduction result) and the location l in which the execution fails and the abduction occurs, as shown at line 3. If the current abduction location l is different from the one recorded in *lastLbl*, it indicates that this is a new abduction. The abduction result is propagated back to strengthen the precondition of the current method and *lastLbl* is set as l (line 5). In this case, the algorithm enters the loop again starting the execution with the strengthened precondition. If the current abduction location l is the same as the one recorded in *lastLbl*, viz. the same program instruction requires abduction again, it indicates a possible bug or a specification error. For example, for a method `void foo (...) {node w := new node(0, null); goo(w); ...}` invoking a method `goo(x)` with precondition $\texttt{x::ll}\langle\texttt{n}\rangle \wedge \texttt{n}{\geq}2$, our analysis will perform an abduction to get $\texttt{n}{\geq}2$ since it is not implied by the current state. However, as `n` is for the shape of local variable `w`, it will be quantified away when propagating $\texttt{n}{\geq}2$ back, ending up with `true` being added to `foo`'s precondition. In the next round of symbolic execution, our analysis will have the same abduction at the same point.Such case is reported as fail. In this way, the symbolic execution continues until it reports an error (line 7) or reaches the end of the method body (exiting line 9).

The function $\mathsf{pureV}(V, \Delta)$ retrieves from Δ the shapes referred to by all pointer variables from V, and returns the set of logical variables used to record numerical (and bag/set) properties in these shapes, e.g. $\mathsf{pureV}(\{\texttt{x}\}, \texttt{x::ll}\langle\texttt{n}\rangle)$ returns $\{\texttt{n}\}$. This function is used in the algorithm to ensure that all free variables in Φ_{pr} and Φ_{po} are added into the parameter list of the constraint abstraction $\texttt{Q}$. The function $\mathsf{ex\_quan}(\Phi_{pr}, \pi)$ is to strengthen the precondition Φ_{pr} with the abduction result π: $\mathsf{ex\_quan}(\Delta, \pi) =_{df} \Delta \wedge \exists(\mathsf{fv}(\pi) \setminus \mathsf{fv}(\Delta)) \cdot \pi$. For example, $\mathsf{ex\_quan}(\texttt{x::ll}\langle\texttt{n}\rangle, 0{<}\texttt{m} \wedge \texttt{m}{\leq}\texttt{n})$ returns $\texttt{x::ll}\langle\texttt{n}\rangle \wedge 0{<}\texttt{n}$.

In lines 4-13, the analysis first derives a pure abstraction ($\texttt{P}(w^*)$) from the constraint abstraction obtained in the first step, and solves it using some existing provers [21, 34]. The pure abstraction is derived in lines 4-7 using the algorithm given in Fig 8. The invocation of the other provers is done in line 9. The obtained pure constraint π is then incorporated into the pre/post specifications in line 10. We carry out a post verification in line 12 using the HIP verifier [32], to ensure the strengthened precondition is sufficiently strong for memory safety.

The pure constraint abstraction derivation algorithm (Fig 8) tries to derive a branch $\texttt{P}_i$ for each branch Δ_i of $\texttt{Q}$. It proceeds in two steps. In the first step (lines 1-3), it replaces the recursive occurrence of $\texttt{Q}$ in Δ_i with $\sigma * \texttt{P}(w^*)$. In the second

```
Algorithm Pure_Abstraction($\sigma, \Delta_i, \mathtt{Q}(w^*)$)
1   Denote all appearances of $\mathtt{Q}(w^*)$ in $\Delta_i$ as $\mathtt{Q}_j(w_j^*), j = 1, ..., p$
2   Denote substitutions $\rho_j = [(([w_j^*/w^*]\sigma * \mathtt{P}(w_j^*))/\mathtt{Q}(w_j^*)]$
3   Let substitution $\rho := \rho_1 \circ \rho_2 \circ ... \circ \rho_p$ as applying all substitutions
        defined above in sequence
4   if $\rho\Delta_i \vdash \sigma * \mathtt{P}_i$ and $ispure(\mathtt{P}_i)$ then return $\mathtt{P}_i$
5   else if $\rho\Delta_i \wedge [\mathtt{P}_i'] \rhd \sigma * \mathtt{P}_i$ and $ispure(\mathtt{P}_i)$ then return $\mathtt{P}_i$
6   else return fail end if
end Algorithm
```

Fig. 8. Abstraction Algorithm.

step (lines 4-6) it tries to derive $\mathtt{P}_i$ via the entailment. If the entailment fails in line 4, pure abduction is used (line 5) to discover any missing pure constraint $\mathtt{P}_i'$ for $\rho\Delta_i$ to allow the entailment to succeed. In this case, $\mathtt{P}_i'$ is incorporated into $\mathtt{P}_i$ and is returned.

4.2 Pure abduction mechanism

We assume that the user has supplied necessary shape information in the specifications for primary methods. When an entailment fails (during symbolic execution or pure constraint abstraction derivation), we use our pure abduction mechanism (Fig. 9) to discover missing pure constraints. Note that we focus on pure abduction in this paper, though it might be possible to adapt the shape abduction technique [4] (to those with strong invariants) in case that shape information is missing from the given precondition.

$$\frac{\sigma \nvdash \sigma_1 * \mathtt{true} \quad \sigma_1 \vdash \sigma * \sigma' \quad ispure(\sigma') \quad \sigma \wedge \sigma' \vdash \sigma_1 * \sigma_2}{\sigma \wedge [\sigma'] \rhd \sigma_1 * \sigma_2}$$

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

Fig. 9. Pure abduction rules.

Our pure abduction deals with three different cases. The first rule applies when the LHS (σ) does not entail the RHS (σ_1) but the RHS entails the LHS with some pure formula (σ') as the frame; e.g. in $\mathtt{x}{::}\mathtt{ll}\langle\mathtt{n}\rangle \nvdash \mathtt{x}{::}\mathtt{node}\langle\_, \mathtt{null}\rangle$, the RHS can entail the LHS with pure frame $\texttt{n=1}$. The abduction then check to ensure $\texttt{x::ll}\langle\texttt{n}\rangle \wedge \texttt{n=1} \vdash \texttt{x::node}\langle\_,\texttt{null}\rangle{*}\sigma_2$ for some σ_2, and returns the result $\texttt{n=1}$. Note the check $\mathit{ispure}(\sigma')$ ensures that σ' contains no heap information.

In the second rule, neither side entails the other (first row), e.g. $\sigma = \texttt{x::sllB}\langle\texttt{S}\rangle$, $\sigma_1 = \exists\texttt{p},\texttt{u},\texttt{v}\cdot\texttt{x::node}\langle\texttt{u},\texttt{p}\rangle * \texttt{p::node}\langle\texttt{v},\texttt{null}\rangle$. As the shape predicates in the antecedent are formed by disjunctions according to their definitions (like the sllB), certain branches of σ may entail σ_1. As the rule suggests, to accomplish abduction $\sigma \wedge [\sigma'] \rhd \sigma_1 * \sigma_2$, we first unfold σ (2nd row) and try entailment or further abduction with the results (σ_0) against σ_1 (3rd row). If it succeeds with a pure frame σ', then we confirm the abduction by checking $\sigma \wedge \sigma' \vdash \sigma_1 * \sigma_2$ (4th row). For the example above, the abduction returns $\texttt{n=2}$ (σ') and discovers the nontrivial frame $\texttt{S}=\{\texttt{u},\texttt{v}\} \wedge \texttt{u}\leq\texttt{v}$ (σ_2). Note the function $\mathsf{data\_no}$ returns the number of data nodes in a state, e.g. it returns one for $\texttt{x::node}\langle\texttt{v},\texttt{p}\rangle * \texttt{p::ll}\langle\texttt{m}\rangle$. (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 ($\bigvee_{i=1}^{u} \sigma^i$), and returns the result as a set of formulae ($\{\sigma^1, ..., \sigma^u\}$). An instance is that it expands $\texttt{x::node}\langle\texttt{v},\texttt{p}\rangle{*}\texttt{p::ll}\langle\texttt{m}\rangle$ to be $\{\texttt{x::node}\langle\texttt{v},\texttt{p}\rangle{\wedge}\texttt{p=null}{\wedge}$ $\texttt{m=0}, \exists\texttt{u},\texttt{q},\texttt{k}\cdot\texttt{x::node}\langle\texttt{v},\texttt{p}\rangle * \texttt{p::node}\langle\texttt{u},\texttt{q}\rangle * \texttt{q::ll}\langle\texttt{q}\rangle \wedge \texttt{m=k+1}\}$.

In the third rule, neither side entails the other, and the second rule does not apply, e.g., $\sigma = \exists\texttt{p},\texttt{u},\texttt{v}\cdot\texttt{x::node}\langle\texttt{u},\texttt{p}\rangle * \texttt{p::node}\langle\texttt{v},\texttt{null}\rangle$, $\sigma_1 = \exists\texttt{S}\cdot\texttt{x::sllB}\langle\texttt{S}\rangle$. In this case the antecedent cannot be unfolded as they are data nodes. As the rule suggests, it reverses two sides of the entailment and applying the second rule to uncover the pure constraints σ_1' and σ' (2nd row). It checks that adding σ' to the LHS (σ) does entail the RHS (σ_1) (3rd row) before it returns σ'. For the example above, the abduction returns $\texttt{u}\leq\texttt{v}$ which is essential for the two nodes to form a sorted list (RHS).

4.3 Inferring Specifications for Loops and Auxiliary Methods

For auxiliary methods, as we do not expect the user to provide specification annotations, we conduct a pre-analysis (Fig 10) to synthesise the pre- and post-shapes before we conduct the refinement analysis from Fig 7. Loops are dealt with by analysing their tail-recursive versions in the same way.

The pre-shape synthesis algorithm SynPre (Fig 10 left) takes in as input the set of shape predicates ($\mathcal{S}$), the auxiliary method name (f), its formal parameters (u^*, v^*), the current symbolic state in which f is called (σ), and the corresponding actual parameters (x^*, y^*) of the invocation. The algorithm first enumerates possible shape candidates from the parameters u^*, v^* with function $\mathsf{ShpCand}$ (line 1), then tests for each shape whether it is a sound abstraction for the method's pre-shape with entailment (line 3), and rules out the ones which fail (line 4). Finally it returns a filtered list of pre-shapes.

We illustrate the function $\mathsf{ShpCand}$ with an example. If we have two parameters x and y with type node, and the user has defined two shape predicates llB and sllB with node, then the list of all possible shape candidates for the two variables (C) will be [$\texttt{x::sllB}\langle\texttt{S}\rangle * \texttt{y::sllB}\langle\texttt{T}\rangle$, $\texttt{x::llB}\langle\texttt{S}\rangle * \texttt{y::sllB}\langle\texttt{T}\rangle$, $\texttt{x::sllB}\langle\texttt{S}\rangle * \texttt{y::llB}\langle\texttt{T}\rangle$, $\texttt{x::llB}\langle\texttt{S}\rangle * \texttt{y::llB}\langle\texttt{T}\rangle$, $\texttt{x::sllB}\langle\texttt{S}\rangle$, $\texttt{y::sllB}\langle\texttt{S}\rangle$, $\texttt{x::llB}\langle\texttt{S}\rangle$,

```
Algorithm SynPre
  $(\mathcal{S}, f, u^*, v^*, \sigma, x^*, y^*)$
1  $C$ := ShpCand$(\mathcal{S}, u^*, v^*)$
2  for $\sigma_C \in C$ do
3    if $\sigma \nvdash [x^*/u^*, y^*/v^*]\sigma_C$
4    then $C$:=$C \backslash \{\sigma_C\}$ end if
5  end for
6  return $C$
end Algorithm
```

```
Algorithm SynPost $(\mathcal{T}, \mathcal{S}, f, e, \Phi_{pr}, u^*, v^*)$
1  $C$ := ShpCand$(\mathcal{S}, u^*, v^*)$
2  $\mathcal{T}'$:=$\mathcal{T} \cup \{f(u^*, v^*)$ requires $\Phi_{pr}$ ensures false $\{e\}\}$
3  $\Delta$ := Symb_Exec$(\mathcal{T}', f$, syn_unroll$(f, e), \Phi_{pr})$
4  for $\sigma_C \in C$ do
5      if $\Delta \wedge [\sigma] \not\vdash \sigma_C$ then $C := C \backslash \{\sigma_C\}$ end if
6  end for
7  return pair_spec_list$(\Phi_{pr}, C)$
end Algorithm
```

Fig. 10. Shape synthesis algorithms.

y::llB⟨S⟩, emp]. Note that ShpCand only picks up relevant shape predicates from $\mathcal{S}$. For instance, if the data structure manipulated by the method is of type `node`, it rules out shape predicates specifying other types of data structures, e.g. `dll`, `bst`, `bstB`, etc.

To synthesise post-shapes (SynPost, Fig 10 right), we also assign C as the set of candidate shapes (line 1). We unroll f's body e once (i.e. replace recursive calls to f in e with a substituted e) and symbolically execute it (line 3), assuming f has a specification *requires* Φ_{pr} *ensures* `false` (line 2). The postcondition `false` is used to ensure that the execution only considers the effect of the program branches with no recursive calls (to f itself). We then use Δ to filter out inappropriate shape candidates from C (line 5). The remaining candidates, paired with Φ_{pr}, are returned as result. The function pair_spec_list forms an ordered list of pre-/post-shape pairs, each of which has Φ_{pr} as pre-shape and a Φ_{po} in C as post-shape. Note that we prioritise shapes with same (or stronger) predicates as in precondition since it is more likely that the output will have the same or similar shape predicates as the input, e.g. `x` is expected to remain as `sllB` (or stronger) if it points to `sllB` as input.

As described before, we employ a lazy scheme when refining the synthesised pre/post-shapes (to complete specifications). We retrieve (and remove) the pre/post-shape pair from the head of the list, (1) use the refinement algorithm (Fig. 7) to obtain a specification for the auxiliary method, and (2) continue the analysis for the primary method. If the analysis for the primary method succeeds, we will ignore all other synthesised pre/post-shapes from the list. If either (1) or (2) fails, we will try the next one from the list. The initial experimental results in next section confirms that our lazy scheme and heuristics of candidate reduction keep only highly relevant candidates. For the `insert` method in Section 2.2, we filtered out 24 (of 26) post-shape candidates , and with heuristics we first chose the one which made the verification succeed and the refinement process terminated due to the lazy scheme. Note that our synthesis of shape specification could only cater to one predicate per parameter/result. In cases where more complex shape specifications are needed, we allow users to specify them directly for the respective auxiliary method.

4.4 Soundness

The soundness of our analysis is ensured by the soundness of the following: the entailment prover [32], the (pure) fixpoint calculation, the pure abstraction derivation, and the abstract semantics (w.r.t. the concrete semantics).

The underlying operational semantics of our language was given in Nguyen et al. [32]. Its concrete program state consists of stack s and heap h, as described in Section 3. Nguyen et al. [32] also defined the relation $s, h \models \Delta$ and the transitive relation $\langle s, h, e\rangle \hookrightarrow^* \langle s', h', \nu\rangle$. Based on that we have:

Lemma 1 (Sound abstract semantics). *If* $[\![e]\!]_{\mathcal{T}}(\Delta, \texttt{true}) = (\Delta', \texttt{true})$*, we have for all* s, h*, if* $s, h \models \Delta$ *and* $\langle s, h, e\rangle \hookrightarrow^* \langle s', h', -\rangle$*, then* $s', h' \models \Delta'$*.*

The proof is done by structural induction.

Lemma 2 (Sound pure abstraction). *Given a method with pre/post shape templates* `pre` *and* `post`*, if our analysis successfully computes a constraint abstraction* `Q` *in the first step, and derives a pure constraint* `P` *in the second step, then we have* $\texttt{Q} \vdash \texttt{post} \wedge \texttt{P}$*.*

The proof follows by induction and the monotonicity of program constructors.

5 Experiments and Evaluation

We have implemented a prototype system for evaluation. Our experimental results were achieved with an Intel Core 2 CPU 2.66GHz with 8Gb RAM. The four columns in Fig. 11 (in the last page) describe, resp., the analysed programs, the analysis time in seconds, and the primary methods' (given and inferred) preconditions and postconditions. All formulae with a grey background are inferred by our analysis. We have tested 25 programs with 16 shape predicates. For some programs, we have verified them with different pre/post shape templates. Programs with star $*$ have different versions for various data structures.

The results highlight the refinement of both pre- and postconditions based on user-provided shape specifications. Firstly, our approach can compute non-trivial pure constraints for postcondition, e.g. for `create` we obtain the value range in the created list, for `delete` we know the content of the result list is subsumed by that of the input list, for list-sorting algorithms we confirm the content of the output is the same as that of the input, and for tree-processing programs (`insert`, `delete` and `avl_ins`), we obtain that the height difference between the input and output trees is at most one. Meanwhile, we can calculate non-trivial requirements in precondition for memory safety or functional correctness. As an example, the `travrs` method, taking in a list with length `m` and an integer `n`, traverses towards the tail of the list for `n` steps. the analysis discovers `m`$\geq$`n` in the precondition to ensure memory safety. Another example is the `append` method concatenating two sorted lists into one. To ensure that the result list is sorted, the analysis figures out that the minimum value in the second list must be no less than the maximum value in the first list.

A second highlight is our flexibility by supporting multiple predicates. Our analysis tries to refine different specifications for the same program at various correctness levels (with different predicates), e.g. `sort_insert`, `tail_insert`, `append`. For `rand_insert`, which inserts a node into a random place (after the head) of a list, we confirm that the list's length is increased by one, but cannot verify the list is kept sorted if it was before the insertion, as the result indicates.

Another highlight is that we can reduce user annotations by synthesising specifications for auxiliary methods, given raw specifications of primary methods. For example, we have analysed a number of list-sorting algorithms with at least one auxiliary method each. We list in Fig 11 two auxiliary methods (`merge` for `merge_sort` and `flatten` for `tree_sort`) and their discovered specifications. Note that these sorting algorithms have the same specification for their primary methods (line ‡). As another example, the method `avl_ins` also has some auxiliary (recursive) methods such as calculation of tree's height, which are automatically analysed as well.

6 Conclusion

6.1 Related works

In recent years, dramatic advances have been made in automated verification of pointer safety for heap-manipulating programs. We highlight some of them here. The local shape analysis by Distefano et al. [12] was able to infer automatically loop invariants for list-processing programs, which formed the early-version SpaceInvader tool. Gotsman et al. [14] proposed an interprocedural shape analysis for the SLAyer tool. Berdine et al. [1] extended the local shape analysis [12] to handle higher-order list predicate so that more complicated real-world data structures can be analysed. Yang et al. [40] proposed a novel abstraction operation which significantly improves the scalability of the analysis. Recently, more large industrial code can be verified by the SpaceInvader tool using the compositional analysis with bi-abductive inference [4, 11].

Several shape analyses also tried to make good use of size information. In the development of the THOR tool, Magill et al. [27] proposed an adaptive shape analysis where additional numerical analysis can be used to help gain better precision. Very recently, Magill et al. [29] formulated a novel instrumentation process which inserts numerical instructions into programs, based on their shape analysis and user-provided predicates. Instrumented programs can then be used to generate pure numerical programs for further analysis. Different from their work, we take *both* shape and numerical information into consideration when performing the abstraction, and derive the numerical abstraction from the shape constraint abstraction. Our approach can be more precise as we have more information for the abstraction. Furthermore, we can directly handle data structures with stronger invariants, like sortedness and height-balanced, which have not been addressed in THOR, to the best of our knowledge. Gulwani et al. [15] combine a set domain with its cardinality domain in a general framework. Compared with these, our approach can handle data structures with stronger invariants like sortedness, height-balanced and bag-related invariants, which have not been addressed in the previous works. Another piece of work, by Chang et al. [6] and Chang and Rival [5], employs inductive checkers and checker segments to express shape and numerical information. Their work is more from a theoretical perspective as they did not verify any numerical properties on the node data.

There are also many shape analyses not using separation logic. The shape analysis framework TVLA [36] is based on three-valued logic. It is capable of handling complicated data structures and properties, such as sortedness. Guo et al. [16] reported a global shape analysis that discover inductive structural shape invariants from the code. Kuncak et al. [24] 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. Compared with these works, separation logic based approach benefits from the frame rule and hence supports local reasoning. Meanwhile, our approach heads towards full functional correctness including bag-related properties, which previous ones do not generally handle.

On the verification side, Smallfoot [2] is the first verification system based on separation logic. The Hip/Sleek verification system [31, 32] supports user-defined shape predicates over the combined shape and numerical domain. The Sleek tool has played a very important role in our analysis. The PALE system [30] transforms constraints in the pointer assertion logic (PAL) into monadic second-order logic (MSO) and discharge them with MONA [19]. Hob [39] is a modular program verification tool for shape properties. Based on Hob, Jahob [22, 39] takes Java as its target language and allows more general specification language. Havoc [7] is another verification tool for C language about heap-allocated data structures, using a novel reachability predicate. There is another recent work on refining specifications via counterexample-guided abstraction refinement [37] which is goal-driven and incrementally improves for given safety requirements. Among these works, our verification is distinguished because we free users from writing whole specifications by requiring only partial specifications, and omit user-supplied annotations for less important loops and auxiliary methods.

6.2 Conclusion

We have reported a new approach to program verification that accepts partial specifications of methods, and refines and completes them by discovering missing constraints to express numerical and bag properties, with the aim to verify full functional properties for pointer-based data structures. We further augment our approach by requiring user to provide only partial specification for primary methods. Specifications for loops and auxiliary methods can then be systematically discovered. We have built a prototype system and the initial experimental results are encouraging.

Acknowledgement. This work was supported in part by the EPSRC projects [EP/E021948/1, EP/G042322/1] and MoE ARF grant R-252-000-411-112.

References

1. J. Berdine, C. Calcagno, B. Cook, D. Distefano, P. O'Hearn, T. Wies, and H. Yang. Shape analysis for composite data structures. In *19th CAV*, July 2007.
2. J. Berdine, C. Calcagno, and P. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *FMCO*, 2005.
3. J. Berdine, B. Cook, D. Distefano, and P. O'Hearn. Automatic termination proofs for programs with shape-shifting heaps. In *18th CAV*, 2006.
4. C. Calcagno, D. Distefano, P. O'Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. In *36th POPL*, January 2009.
5. B. Chang and X. Rival. Relational inductive shape analysis. In *POPL*, 2008.
6. B. Chang, X. Rival, and G. Necula. Shape analysis with structural invariant checkers. In *SAS*, 2007.
7. S. Chatterjee, S. Lahiri, S. Qadeer, and Z. Rakamaric. A reachability predicate for analyzing low-level software. In *TACAS*, 2007.
8. W.-N. Chin, C. David, H. H. Nguyen, and S. Qin. Automated verification of shape, size and bag properties. In *12th ICECCS*, 2007.
9. W.-N. Chin, C. David, H. H. Nguyen, and S. Qin. Enhancing modular oo verification with separation logic. In POPL, January 2008.
10. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, 1977.
11. D. Distefano. Attacking large industrial code with bi-abductive inference. In *FMICS*, 2009.
12. D. Distefano, P. O'Hearn, and H. Yang. A local shape analysis based on separation logic. In *TACAS*, 2006.
13. R. Giacobazzi. Abductive analysis of modular logic programs. In *ILPS*, 1994.
14. A. Gotsman, J. Berdine, and B. Cook. Interprocedural shape analysis with separated heap abstractions. In *SAS*, 2006.
15. S. Gulwani, T. Lev-Ami, and M. Sagiv. A Combination Framework for Tracking Partition Sizes. In *POPL*, 2009.
16. B. Guo, N. Vachharajani, and D. August. Shape analysis with inductive recursion synthesis. In *PLDI*, 2007.
17. J. Gustavsson and J. Svenningsson. Constraint abstractions. In *Programs as Data Objects II*, Denmark, May 2001.
18. B. Hackett and R. Rugina. Region-based shape analysis with tracked locations. In *POPL*, 2005.
19. J. Henriksen, J. Jensen, M. Jørgensen, N. Klarlund, R. Paige, T. Rauhe, and A. Sandholm. Mona: Monadic second-order logic in practice. In *TACAS*, 1995.
20. S. Ishtiaq and P. O'Hearn. BI as an assertion language for mutable data structures. In *POPL*, 2001.
21. N. Klarlund and A. Møller. Mona version 1.4 user manual. Manual, BRICS, Department of Computer Science, Aarhus University, 2001.
22. V. Kuncak. *Modular Data Structure Verification.* PhD thesis, EECS Department, Massachusetts Institute of Technology, February 2007.
23. S. Nejati. *Refinement Relation on Partial Specifications.* MSc thesis, CS Department, University of Toronto, 2003.
24. V. Kuncak, P. Lam, and M. Rinard. Role analysis. In *POPL*, 2002.
25. V. Kuncak, P. Lam, K. Zee, and M. Rinard. Modular pluggable analyses for data structure consistency. *IEEE Transactions on Software Engineering*, 32(12), 2006.

26. S. Magill, A. Nanevski, E. Clarke, and P. Lee. Inferring Invariants in Separation Logic for Imperative List-processing Programs. In *the SPACE Workshop*, 2006.
27. S. Magill, J. Berdine, E. Clarke, and B. Cook. Arithmetic strengthening for shape analysis. In *SAS*, 2007.
28. S. Magill, M. Tsai, P. Lee, and Y. Tsay. Thor: A tool for reasoning about shape and arithmetic. In *CAV*, 2008.
29. S. Magill, M. Tsai, P. Lee, and Y. Tsay. Automatic numeric abstractions for heap-manipulating programs. In *POPL*, 2010. To appear.
30. A. Møller and M. Schwartzbach. The pointer assertion logic engine. *ACM SIGPLAN Notices*, 36(5):221–231, 2001.
31. H. H. Nguyen and W.-N. Chin. Enhancing program verification with lemmas. In *20th CAV*, 2008.
32. H. H. Nguyen, C. David, S. Qin, and W.-N. Chin. Automated verification of shape and size properties via separation logic. In *8th VMCAI*, 2007.
33. M. Parkinson and G. Bierman. Separation logic, abstraction and inheritance. In *POPL*, 2008.
34. C. Popeea and W.-N. Chin. Inferring disjunctive postconditions. In *Proceedings of 11th Asian Computing Science Conference*, 2006.
35. J. Reynolds. Separation logic: a logic for shared mutable data structures. In *17th LICS*, 2002.
36. M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems*, 24(3):217–298, 2002.
37. M. Taghdiri. *Automating Modular Verification by Refining Specifications.* PhD thesis, EECS Department, Massachusetts Institute of Technology, February 2008.
38. J. Warford. *Computer systems.* Jones & Bartlett Publishers, 2009.
39. T. Wies, V. Kuncak, K. Zee, A. Podelski, and M. Rinard. On verifying complex properties using symbolic shape analysis. *The Computing Research Repository*, abs/cs/0609104, 2006.
40. H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. O'Hearn. Scalable shape analysis for systems code. In *20th CAV*, 2008.

A Symbolic Execution Rules

This section defines the symbolic execution rules used in the first step of the constraint abstraction generation. If the program contains recursive calls to itself, the postcondition will be in a recursive (open) form.

The type of our symbolic execution is defined as

$$[\![e]\!] =_{df} \mathsf{AllSpec} \rightarrow (\mathcal{P}_{\mathsf{SH}} \times \mathsf{Int}) \rightarrow (\mathcal{P}_{\mathsf{SH}} \times \mathsf{Int})$$

where $\mathsf{AllSpec}$ contains all the specifications of all methods (extracted from the program *Prog*). The integer (label) in both input and output is used to record a program location where abduction is needed. If the integer remains zero after the symbolic execution of e, then the output state denotes the post-state of e. However, a positive number indicates that an abduction must have occurred and the resulting state (the abduction result) will be propagated back to the the method's precondition by our analysis, so that the next round of symbolic execution should succeed in the same location.

The foundation of the symbolic execution is the basic transition functions from a conjunctive abstract state to a conjunctive or disjunctive abstract state below:

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

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 $\mathsf{unfold}(x)$ unfolds 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.

For the unfolding operation $\mathsf{unfold}(x)$, there are two possible scenarios. If x refers to a data node in the current state σ, no unfolding is required and the exec operation can proceed directly. However, if x refers to a (user-defined) shape predicate, then $\mathsf{unfold}(x)$ will unfold the current state σ according to the definition of the predicate in order to expose the data node referred to by x:

$$\frac{\begin{array}{c}\mathit{isdatat}(c)\\ \sigma \vdash x{::}c\langle v^*\rangle * \sigma'\end{array}}{\mathsf{unfold}(x)\sigma \rightsquigarrow \sigma} \qquad \frac{\begin{array}{c}\mathit{isspred}(c) \qquad \sigma \vdash x{::}c\langle v^*\rangle * \sigma'\\ \mathit{fresh\ logical}\ u^*\end{array}}{\mathsf{unfold}(x)\sigma \rightsquigarrow \bigvee\{\sigma' * [x/\mathtt{root}, u^*/\mathsf{lv}(\Phi)]\Phi \mid \mathtt{root}{::}c\langle v^*\rangle{\equiv}\Phi\}}$$

where $\mathsf{lv}(\Phi)$ stands for all logical variables in Φ. 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, x.f := w, or free(x)) assumes that the unfolding $\mathsf{unfold}(x)$ has been done prior to the execution. The first three rules below are for normal symbolic execution where the current state is sufficiently strong for safe execution. The last two rules handle the cases where the symbolic execution fails and abductive reasoning can be used to discover missing pure information.

$$\frac{\mathit{isdatat}(c) \qquad \sigma \vdash x{::}c\langle v_1,..,v_n\rangle * \sigma'}{\mathsf{exec}(x.f_i)(\mathcal{T})(\sigma,0) \rightsquigarrow (\sigma' * x{::}c\langle v_1,..,v_n\rangle \wedge \mathtt{res}{=}v_i, 0)}$$

$$\frac{\mathit{isdatat}(c) \qquad \sigma \vdash x{::}c\langle v_1,..,v_n\rangle * \sigma'}{\mathsf{exec}(x.f_i := w)(\mathcal{T})(\sigma,0) \rightsquigarrow (\sigma' * x{::}c\langle v_1,..,v_{i-1},w,v_{i+1},..,v_n\rangle, 0)}$$

$$\frac{\mathit{isdatat}(c) \qquad \sigma \vdash x{::}c\langle u^*\rangle * \sigma'}{\mathsf{exec}(\mathtt{free}(x))(\mathcal{T})(\sigma,0) \rightsquigarrow (\sigma', 0)}$$

$$\frac{\mathit{isdatat}(c) \quad \sigma \nvdash x{::}c\langle u^*\rangle{*}\mathtt{true} \quad \sigma{*}[\sigma'] \rhd x{::}c\langle u^*\rangle{*}\mathtt{true}}{\mathsf{exec}(d[x])(\mathcal{T})(\sigma,0) \rightsquigarrow (\sigma', \mathit{lbl}(d[x]))}$$

$$\frac{\mathit{isdatat}(c) \quad \sigma \nvdash x{::}c\langle u^*\rangle{*}\mathtt{true} \quad \sigma{*}[\sigma'] \not\rhd x{::}c\langle u^*\rangle{*}\mathtt{true}}{\mathsf{exec}(d[x])(\mathcal{T})(\sigma,0) \rightsquigarrow (\mathtt{false}, \mathit{lbl}(d[x]))}$$

Note that the second to last rule uses an abductive reasoning (via Sleek) to discover the missing numerical information σ'. Here we use a mapping $lbl(-)$ to map any instruction in the program being analyzed to a unique positive integer label (namely the aforementioned program location). The rule changes the second element of the result to $lbl(d[x])$ which will be used by the analysis to record the instruction causing an abduction, quits the current execution, propagates the discovered information back to the precondition of the current method, and restarts the symbolic execution with the strengthened precondition. The last rule covers the scenario in which the abduction fails. Then the execution cannot continue and returns $(\texttt{false}, lbl(d[x]))$.

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

$$\mathsf{exec}(k)(\mathcal{T})(\sigma,0) \leadsto (\sigma \wedge \texttt{res}{=}k, 0) \qquad \mathsf{exec}(v)(\mathcal{T})(\sigma,0) \leadsto (\sigma \wedge \texttt{res}{=}v, 0)$$

$$\frac{isdatat(c)}{\mathsf{exec}(\texttt{new}\ c(v^*))(\mathcal{T})(\sigma,0) \leadsto (\sigma * \texttt{res}{::}c\langle v^*\rangle, 0)}$$

$$\frac{(x^*,y^*) = \mathsf{vars}(w,e) \quad (f,\mathcal{T}_1) = \mathsf{Analysis}(\mathcal{T}, \texttt{while}(w)\{e\}, \sigma, x^*, y^*) \quad \mathcal{T}' = \mathcal{T} \cup \mathcal{T}_1}{\mathsf{exec}(\texttt{while}(w)\{e\})(\mathcal{T})(\sigma,0) \leadsto \mathsf{exec}(f(x^*;y^*))(\mathcal{T}')(\sigma,0)}$$

$$\frac{t\ mn\ ((t_i\ u_i)_{i=1}^m;(t_i\ v_i)_{i=1}^n) \in \mathcal{T} \quad (f,\mathcal{T}_1) = \mathsf{Analysis}(\mathcal{T}, mn, \sigma, x^*, y^*) \quad \mathcal{T}' = \mathcal{T} \cup \mathcal{T}_1}{\mathsf{exec}(mn(x_1..x_m;y_1..y_n))(\mathcal{T})(\sigma,0) \leadsto \mathsf{exec}(mn(x_1..x_m;y_1..y_n))(\mathcal{T}')(\sigma,0)}$$

$$\frac{\begin{array}{c} t\ mn\ ((t_i\ u_i)_{i=1}^m;(t_i\ v_i)_{i=1}^n)\ requires\ \Phi_{pr}\ ensures\ \Phi_{po} \in \mathcal{T} \\ \rho = [x_i'/u_i']_{i=1}^m \circ [y_i'/v_i']_{i=1}^n \quad \sigma \vdash \rho\Phi_{pr} * \sigma' \quad \rho_o = [y_i/v_i]_{i=1}^n \circ \rho \\ \rho_l = [r_i/y_i']_{i=1}^n \quad \rho_{ol} = [r_i/y_i]_{i=1}^n \quad fresh\ logical\ r_i \end{array}}{\mathsf{exec}(mn(x_1..x_m;y_1..y_n))(\mathcal{T})(\sigma,0) \leadsto ((\rho_l\,\sigma') * (\rho_{ol} \circ \rho_o\,\Phi_{po}), 0)}$$

$$\frac{\begin{array}{c} t\ mn\ ((t_i\ u_i)_{i=1}^m;(t_i\ v_i)_{i=1}^n)\ requires\ \Phi_{pr}\ ensures\ \Phi_{po} \in \mathcal{T} \\ \rho = [x_i'/u_i']_{i=1}^m \circ [y_i'/v_i']_{i=1}^n \quad \sigma \vdash \rho\Phi_{pr} * \sigma' \quad \rho_o = [y_i/v_i]_{i=1}^n \circ \rho \\ \rho_l = [r_i/y_i']_{i=1}^n \quad \rho_{ol} = [r_i/y_i]_{i=1}^n \quad fresh\ logical\ r_i \end{array}}{\mathsf{exec}(mn(x_1..x_m;y_1..y_n))(\mathcal{T})(\sigma,0) \leadsto ((\rho_l\,\sigma') * (\rho_{ol} \circ \rho_o\,(\Phi_{pr} \wedge \mathsf{P}(u^*,v^*))), 0)}$$

$$\frac{t\ mn... \in \mathcal{T} \quad \rho = [x_i'/u_i']_{i=1}^m \circ [y_i'/v_i']_{i=1}^n \quad \sigma \nvdash \rho\Phi_{pr} * \texttt{true} \quad \sigma \wedge [\sigma'] \rhd \rho\Phi_{pr} * \texttt{true}}{\mathsf{exec}(mn(x_1..x_m;y_1..y_n))(\mathcal{T})(\sigma,0) \leadsto (\sigma', lbl(mn(...)))}$$

$$\frac{t\ mn... \in \mathcal{T} \quad \rho = [x_i'/u_i']_{i=1}^m \circ [y_i'/v_i']_{i=1}^n \quad \sigma \nvdash \rho\Phi_{pr} * \texttt{true} \quad \sigma \wedge [\sigma'] \not\rhd \rho\Phi_{pr} * \texttt{true}}{\mathsf{exec}(mn(x_1..x_m;y_1..y_n))(\mathcal{T})(\sigma,0) \leadsto (\texttt{false}, lbl(mn(...)))}$$

Note that the first three rules deal with constant (k), variable (v) and data node creation ($\texttt{new}\ c(v^*)$), respectively, while the remaining rules handle method invocation. The fourth and fifth rules are used for the invocation of a while loop or an auxiliary method which has not been analyzed, where we employ the analysis algorithm recursively to achieve its postcondition to enable application of the next rule. The sixth rule is used for the invocation of another method mn which has already been analyzed, and the call site meets the precondition of mn, as checked by the entailment $\sigma \vdash \rho\Phi_{pr} * \sigma'$. In this case, the execution succeeds and moves on. The seventh rule is for a recursive call to the current method, similar as above except that a constraint abstraction is in place as postcondition. The last two rules are for the cases where the call site cannot establish the precondition of the callee method and where abductive reasoning is employed. In both cases, the execution discontinues. The eighth rule returns the abduction result σ', which is a pure formula and will be propagated back by the analysis to strengthen the caller method's precondition. The last rule captures the scenario in which the abduction fails. Note that the operator $\circ$ is used to compose two substitutions: the substitution $\rho_2 \circ \rho_1$ works by first applying ρ_1 and then ρ_2.

To keep presentation simple, we assume there are no mutual recursions in the programs to analyze; therefore each method to be analyzed should only call itself recursively. This assumption does not lose generality, as we can always transform mutual recursion into single recursion [38] to have only one constraint abstraction `Q` in our analysis for one method.

The following rule for all commands signifies that when starting from a configuration in which the second element is positive (i.e. a faulty state), the execution will not change the state. This rule is used to skip all remaining instructions when abductive reasoning is used as a new round of symbolic execution with strengthened precondition should be started instead:

$$\frac{l > 0}{\mathsf{exec}(-)(\mathcal{T})(\sigma, l) \rightsquigarrow (\sigma, l)}$$

We can now lift unfold's domain to $\mathcal{P}_{\mathsf{SH}}$ using the following operation $\mathsf{unfold}^{\dagger}$:

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

and similarly for exec:

$$\mathsf{exec}^{\dagger}(d)(\mathcal{T})(\bigvee \sigma_i, l) =_{df} (\bigvee \sigma_i',\ \mathsf{max}\{l_i\}) \text{ where } (\sigma_i', l_i) \in \mathsf{exec}(d)(\mathcal{T})(\sigma_i, l)$$

The symbolic execution rules for program constructors e can now be defined using the lifted transition functions above. Firstly, no change will be made if starting from a faulty state, as the first rule shows. In all other cases, the symbolic execution transforms one abstract state to another w.r.t. the program instruction:

$$\begin{array}{ll}
[\![-]\!]_{\mathcal{T}}(\Delta, l) & =_{df}\ (\Delta, l), \quad \text{where } l > 0 \\
[\![d[x]]\!]_{\mathcal{T}}(\Delta, 0) & =_{df}\ \mathsf{exec}^{\dagger}(d[x])(\mathcal{T})(\mathsf{unfold}^{\dagger}(x)\Delta, 0) \\
[\![d]\!]_{\mathcal{T}}(\Delta, 0) & =_{df}\ \mathsf{exec}^{\dagger}(d)(\mathcal{T})(\Delta, 0) \\
[\![e_1; e_2]\!]_{\mathcal{T}}(\Delta, 0) & =_{df}\ [\![e_2]\!]_{\mathcal{T}} \circ [\![e_1]\!]_{\mathcal{T}}(\Delta, 0)
\end{array}$$

$$[\![v := e]\!]_{\mathcal{T}}(\Delta, 0) \ =_{df}\ [v_1/v', r_1/\mathtt{res}]([\![e]\!]_{\mathcal{T}}(\Delta, 0)) \wedge v'{=}r_1,\ \mathit{fresh}\ v_1, r_1$$

$$\frac{(\Delta_1', l_1) = [\![e_1]\!]_{\mathcal{T}}(v \wedge \Delta, 0) \qquad (\Delta_2', l_2) = [\![e_2]\!]_{\mathcal{T}}(\neg v \wedge \Delta, 0)}{[\![\mathtt{if}\ (v)\ e_1\ \mathtt{else}\ e_2]\!]_{\mathcal{T}}(\Delta, 0) =_{df} (\Delta_1' \vee \Delta_2', \mathsf{max}\{l_1, l_2\})}$$

Prog.	Time	Pre	Post
List processing programs			
create*	0.379	$\mathtt{emp} \wedge n\geq0$	$res::\mathtt{llB}\langle S\rangle \wedge n=\lvert S\rvert \wedge \forall v\in S\cdot 1\leq v\leq n$
	1.752	$\mathtt{emp} \wedge n\geq0$	$res::\mathtt{dllB}\langle rp, S\rangle \wedge n=\lvert S\rvert \wedge \forall v\in S\cdot 1\leq v\leq n$
	0.954	$\mathtt{emp} \wedge n\geq0$	$res::\mathtt{sllB2}\langle S\rangle \wedge n=\lvert S\rvert \wedge \forall v\in S\cdot 1\leq v\leq n$
sort_* insert	0.591	$x::\mathtt{ll}\langle n\rangle \wedge n\geq1$	$x::\mathtt{ll}\langle m\rangle \wedge m=n+1$
	0.789	$x::\mathtt{dll}\langle p, n\rangle \wedge n\geq1$	$x::\mathtt{dll}\langle q, m\rangle \wedge n\geq1 \wedge m=n+1 \wedge p=q$
	0.504	$x::\mathtt{sll}\langle n, xs, xl\rangle\wedge v\geq xs$	$x::\mathtt{sll}\langle m, mn, mx\rangle\wedge xs=mn\wedge mx=\max(xl,v)\wedge m=n+1$
tail_ insert	0.566	$x::\mathtt{ll}\langle n\rangle \wedge n\geq1$	$x::\mathtt{ll}\langle m\rangle \wedge m=n+1$
	0.628	$x::\mathtt{sll}\langle n, xs, xl\rangle\wedge v\geq xl$	$x::\mathtt{sll}\langle m, mn, mx\rangle \wedge v=mx \wedge mn=xs \wedge m=n+1$
rand_* insert	0.522	$x::\mathtt{ll}\langle n\rangle \wedge n\geq1$	$x::\mathtt{ll}\langle m\rangle \wedge m=n+1$
	0.830	$x::\mathtt{dll}\langle p, n\rangle \wedge n\geq1$	$x::\mathtt{dll}\langle q, m\rangle \wedge m=n+1 \wedge p=q$
	—	$x::\mathtt{sll}\langle n, xs, xl\rangle\wedge (\mathtt{fail})$	$x::\mathtt{sll}\langle m, mn, mx\rangle\wedge (\mathtt{fail})$
delete	0.630	$x::\mathtt{llB}\langle S\rangle \wedge \lvert S\rvert\geq2$	$x::\mathtt{llB}\langle T\rangle \wedge \exists a.S=T\sqcup\{a\}$
	1.024	$x::\mathtt{sllB}\langle S\rangle \wedge \lvert S\rvert\geq2$	$x::\mathtt{sllB}\langle T\rangle \wedge \exists a.S=T\sqcup\{a\}$
delete	1.252	$x::\mathtt{dllB}\langle p, S\rangle \wedge \lvert S\rvert\geq2$	$x::\mathtt{dllB}\langle q, T\rangle \wedge \exists a.S=T\sqcup\{a\} \wedge p=q$
travrs	0.296	$x::\mathtt{ll}\langle m\rangle\wedge n\geq0\wedge m\geq n$	$x::\mathtt{ls}\langle p,k\rangle * res::\mathtt{ll}\langle r\rangle\wedge p=res\wedge k=n\wedge m=n+r$
	2.205	$x::\mathtt{sllB}\langle S\rangle \wedge n\geq0\wedge\lvert S\rvert\geq n$	$x::\mathtt{slsB}\langle p, T\rangle * res::\mathtt{sllB}\langle S_2\rangle\wedge p=res\wedge\lvert T\rvert=n \wedge S=T\sqcup S_2 \wedge \forall u\in T,v\in S_2 \cdot u\leq v$
append*	0.512	$x::\mathtt{ll}\langle xn\rangle * y::\mathtt{ll}\langle yn\rangle\wedge xn\geq1$	$x::\mathtt{ll}\langle m\rangle \wedge m=xn+yn$
	0.660	$x::\mathtt{dll}\langle xp, xn\rangle * y::\mathtt{dll}\langle yp, yn\rangle \wedge xn\geq1$	$x::\mathtt{dll}\langle q, m\rangle \wedge m=xn+yn \wedge q=xp$
	0.948	$x::\mathtt{sll}\langle xn, xs, xl\rangle\wedge xl\leq ys * y::\mathtt{sll}\langle yn, ys, yl\rangle$	$x::\mathtt{sll}\langle m, rs, rl\rangle \wedge yl=rl \wedge m\geq1+yn \wedge m=xn+yn$
Sorting (main)		$x::\mathtt{llB}\langle S\rangle \wedge \lvert S\rvert\geq1$	$res::\mathtt{sllB}\langle T\rangle \wedge T=S$ (‡)
merge	4.107	$x::\mathtt{sllB}\langle S_x\rangle * y::\mathtt{sllB}\langle S_y\rangle$	$res::\mathtt{sllB}\langle T\rangle \wedge T=S_x\sqcup S_y$
flatten	2.693	$x::\mathtt{bstB}\langle S\rangle$	$res::\mathtt{sllB}\langle T\rangle \wedge T=S$
insert	0.824	$r::\mathtt{sllB}\langle S\rangle * x::\mathtt{node}\langle v, \_\rangle$	$res::\mathtt{sllB}\langle T\rangle \wedge T=S\sqcup\{v\}$
quick	2.132	$x::\mathtt{lbd}\langle S\rangle$	$x::\mathtt{lbd}\langle S_1\rangle * res::\mathtt{lbd}\langle S_2\rangle \wedge S=S_1\sqcup S_2 \wedge \forall u\in S_1 \forall v\in S_2 \cdot u\leq p\leq v$
Binary tree, binary search tree and AVL tree processing programs			
travrs	0.532	$x::\mathtt{bt}\langle S, h\rangle$	$x::\mathtt{bt}\langle T, k\rangle \wedge S=T \wedge h=k$
count	0.709	$x::\mathtt{bt}\langle S, h\rangle$	$x::\mathtt{bt}\langle T, k\rangle \wedge res=\lvert S\rvert \wedge S=T \wedge h=k$
height	0.913	$x::\mathtt{bt}\langle S, h\rangle$	$x::\mathtt{bt}\langle T, k\rangle \wedge res=h=k \wedge S=T$
insert	1.276	$x::\mathtt{bt}\langle S, h\rangle \wedge \lvert S\rvert\geq1 \wedge h\geq1$	$x::\mathtt{bt}\langle T, k\rangle \wedge T=S\sqcup\{v\} \wedge h\leq k\leq h+1$
delete	0.970	$x::\mathtt{bt}\langle S, h\rangle \wedge \lvert S\rvert\geq2 \wedge h\geq2$	$x::\mathtt{bt}\langle T, k\rangle \wedge \exists a.S=T\sqcup\{a\} \wedge h-1\leq k\leq h$
search	1.583	$x::\mathtt{bst}\langle sm, lg\rangle$	$x::\mathtt{bst}\langle mn, mx\rangle \wedge sm=mn \wedge lg=mx \wedge 0\leq res\leq1$
bst_ insert	1.720	$x::\mathtt{bst}\langle sm, lg\rangle$	$x::\mathtt{bst}\langle mn, mx\rangle \wedge (v<sm\wedge v=mn\wedge lg=mx\vee lg<v\wedge v=mx\wedge sm=mn \vee sm=mn\wedge lg=mx)$
avl_ins	11.12	$x::\mathtt{avl}\langle S, h\rangle$	$res::\mathtt{avl}\langle T, k\rangle \wedge T=S\sqcup\{v\} \wedge h\leq k\leq h+1$
sdl2nbt	5.826	$x::\mathtt{sdlB}\langle p,q,S\rangle\wedge \lvert S\rvert\geq1\wedge p=null \wedge q=tail$	$res::\mathtt{nbt}\langle T\rangle \wedge T=S$

Fig. 11. Experimental Results.