Automatically Refining Partial Specifications for Heap-Manipulating Programs

Shengchao Qin[a], Guanhua He[a], Chenguang Luo[a], Wei-Ngan Chin[c], Hongli Yang[b]

[a]*School of Computing, Teesside University*
[b]*College of Computer Science, Beijing University of Technology*
[c]*School of Computing, National University of Singapore*

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/red-black trees. The verification process can greatly benefit from human assistance through specification annotations, but this process requires intellectual effort from users and is error-prone. 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 revised by users. We further augment our approach by requiring partial specification to be given only 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.

Keywords: static program analysis, separation logic, numerical analysis, partial specification refinement, semi-automatic software verification, constraint abstraction
PACS: 07.05.Bx, 89.20.Ff

Contents

☆This is an extended version of an FM2011 paper.
☆☆This research was supported in part by EPSRC Project EP/G042322.
Email addresses: shengchao.qin@gmail.com (Shengchao Qin), g.he@tees.ac.uk (Guanhua He), chinwn@comp.nus.edu.sg (Wei-Ngan Chin), yhl.yang@gmail.com (Hongli Yang)

1. Introduction

Human assistance is often essential in (semi-) automated program verification. The user may supply annotations at certain program points, 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 with significant diversity. Therefore, it may be unreasonable to expect users to provide specifications for every method and invariants for every loop when verifying larger software systems. Furthermore, to err is human. 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 distinguish whether a failure 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, which has long been a challenging problem. To deal with such programs, which manipulate heap-allocated shared mutable data structures, one needs to keep track of not only "shape" information (for deep heap properties) but also related "pure" properties, such as structural size information (length and height), relational numerical information (balanced and sortedness properties), and content information (multi-set of symbolic values). Under our framework, the user is expected to provide partial specifications for *primary* methods with only *shape* information. Our verification will then take over the rest of the work to refine those partial specifications with derived (pure) constraints which should be satisfied by the program, or report a potential program bug if the given specifications are rejected by our verifier. This is an improvement over previous works [1, 2] where users must provide full specifications for each method and invariants for each loop. This is also significantly different from compositional shape analysis [3, 4, 5]. In spite of a higher level of automation, their analysis focuses on pointer safety only and deals primarily with a few built-in predicates over the shape domain only. Our work targets both memory safety and functional correctness and supports user-defined predicates over several abstract domains (such as shape, numerical, multi-set).

Our approach allows the user to 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 structure $\texttt{data node}\ \{\ \texttt{int val};\ \texttt{node next};\ \}$, a user interested in pointer-safety may define a list shape predicate (as in [3, 4]):

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

Note that in the inductive case, the separation conjunction $*$ ([6]) ensures that two heap portions (the head node and the tail list) are domain-disjoint. The parameter `root` for the predicate is the root pointer referring to the data structure.

Yet another user may be interested to track also the length of a list to analyse quantitative measures, using

$$\mathtt{ll(root,n)} \equiv \mathtt{(root{=}null{\wedge}n{=}0){\vee}(root{\mapsto}node(\_,q){*}ll(q,m){\wedge}n{=}m{+}1)}$$

Note that unbound variables, such as `q` and `m`, are implicitly existentially quantified, and _ is used to denote an existentially quantified anonymous variable. This predicate may be extended to capture the content information, to support a higher level of correctness with a multi-set (bag) property:

$$\mathtt{llB(root,S){\equiv}(root{=}null{\wedge}S{=}\emptyset){\vee}(root{\mapsto}node(v,q){*}llB(q,S_1){\wedge}S{=}\{v\}{\sqcup}S_1)}$$

where the length of the list is implicitly captured by the cardinality $|\mathtt{S}|$. The operator $\sqcup$ is for bag union. A further strengthening can capture also the sortedness property:

$$\mathtt{sllB(root,S){\equiv}(root{=}null{\wedge}S{=}\emptyset){\vee}(root{\mapsto}node(v,q){*}sllB(q,S_1){\wedge}S{=}\{v\}{\sqcup}S_1{\wedge}(\forall x{\in}S_1{\cdot}v{\leq}x))}$$

Therefore, users can provide predicate definitions with respect to various correctness level and program properties, which can be as simple as normal lists or as complicated as AVL trees, depending on their requirements. These predicates are non-trivial to be defined but can be reused multiple times for specifications of different methods. Hence efforts involved in such predicate design are often significantly amortised. We have built a library of predicates with respect to commonly-used data structures and useful program properties.

Based on these predicates, the user is expected to provide partial specifications for some primary methods which are the main objects of verification. For example, a sorting algorithm taking `x` as input parameter that is expected to be non-null. The user may provide $\mathtt{llB(x,S_1)}$ as precondition and $\mathtt{sllB(x,S_2)}$ as postcondition. Our approach will refine the specification as $\mathtt{llB(x,S_1){\wedge}x{\neq}null}$ for precondition, and $\mathtt{sllB(x,S_2){\wedge}S_1{=}S_2}$ for postcondition. Here we need user annotations as the initial specification, because we reserve the flexibility of verification with respect to different program properties at various correctness levels. For instance, our approach can verify the same algorithm, but for different refined specifications, such as:

$$\begin{array}{llllllll}
\mathtt{requires} & \mathtt{list(x)} & \wedge & \mathtt{x{\neq}null} & \mathtt{ensures} & \mathtt{list(x)} & & \\
\mathtt{requires} & \mathtt{ll(x,n_1)} & \wedge & \mathtt{n_1{>}0} & \mathtt{ensures} & \mathtt{ll(x,n_2)} & \wedge & \mathtt{n_1{=}n_2} \\
\mathtt{requires} & \mathtt{llB(x,S_1)} & \wedge & \mathtt{x{\neq}null} & \mathtt{ensures} & \mathtt{llB(x,S_2)} & \wedge & \mathtt{S_1{=}S_2} \\
\mathtt{requires} & \mathtt{llB(x,S)} & \wedge & \mathtt{x{\neq}null} & \mathtt{ensures} & \mathtt{ll(x,n)} & \wedge & \mathtt{|S|{=}n}
\end{array}$$

where the discovered missing constraints are shown in shaded form.

To summarise, our proposal for refining partial specification is aimed at harnessing the synergy between a human's insights and a machine's capability at automated program analysis. In particular, human guidance can help narrow down on the most important of the different 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*: We discover three types of constraints added into the user-given incomplete specification: constraints in the precondition for memory safety, (relational) constraints in postcondition to link the method's pre- and post-states, and 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 intervene 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 two motivating examples 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 [7, 6] extends Hoare logic to support reasoning about shared mutable data structures. One connective that it adds to classical logic is separation conjunction $*$. The separation formula $\mathtt{p_1{*}p_2}$ means that the heap can be split into two disjoint parts in which $\mathtt{p_1}$ and $\mathtt{p_2}$ hold respectively. Our work will make use of this connective in our specifications.

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:

$$\begin{aligned}\mathtt{sdlB(root,p,q,S)} \equiv & (\mathtt{root{=}q} \wedge \mathtt{S{=}\emptyset}) \ \vee \\ & (\mathtt{root{\mapsto}node2(v,p,t){*}sdlB(t,root,q,S_1)}\wedge\mathtt{root{\neq}q}\wedge\mathtt{S{=}\{v\}{\sqcup}S_1}\wedge(\forall\mathtt{x{\in}S_1{\cdot}v{\leq}x}))\end{aligned}$$

where the parameters `p` and `q` denote the `prev` field of `root` and the `next` field of the last node in the list 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:

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

where `S` captures the content of the tree. We require the difference 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 Δ_R such that $\Delta_1 \vdash \Delta_2{*}\Delta_R$. For instance, given the entailment query

$$\mathtt{ll(p,n)} \wedge \mathtt{n{>}0} \vdash \exists\mathtt{q}\cdot\mathtt{node(p,q)} * [\Delta_R]$$

SLEEK produces the following result after unfolding the LHS predicate:

$$\mathtt{ll(p,n)} \wedge \mathtt{n{>}0} \vdash \exists\mathtt{q}\cdot\mathtt{node(p,q)}{*}[\mathtt{ll(q,n{-}1)} \wedge \mathtt{n{>}0}]$$

where the inferred frame is shown in square brackets as the residue of the entailment check. The proposed analysis in this paper will use SLEEK to perform deductions of separation formulas.

2.2. An Illustrative Example

We firstly 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 llB(x,S)
 4  ensures  sllB(res,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 }
```

Figure 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 (as indicated by the specifications in 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 in this case.

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* [8]. 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. This pure constraint abstraction is then solved by fixpoint solvers in pure (numerical and bag) domains, such as [9, 10, 11].

The constraint abstraction of a code segment (e.g. a method) in our settings is an abstract form of the code segment'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

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

and its precondition $\{\mathtt{x}{\geq}0 \wedge \mathtt{y}{=}0\}$ 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 on Page 19):

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

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

For the if branch, after the unfolding over `llB(x,S)` (rule unfold on Page 17), we know from the condition that the input list `x` has only one node, and thus its post-state will be

$$\exists \mathtt{v} \cdot \mathtt{x}{\mapsto}\mathtt{node}(\mathtt{v},\mathtt{null}) \wedge \mathtt{res}{=}\mathtt{x} \wedge \mathtt{S}{=}\{\mathtt{v}\}$$

Meanwhile, for the else branch, the list will firstly be unrolled by one node at line 6 (rule unfold), making `x.next` point to `s` (rule assign on Page 19), which references a sub-list one node shorter than the input list beginning from `x`:

$$\exists \mathtt{S}_1,\mathtt{v} \cdot \mathtt{x}{\mapsto}\mathtt{node}(\mathtt{v},\mathtt{s}){*}\mathtt{llB}(\mathtt{s},\mathtt{S}_1) \wedge \mathtt{S}{=}\mathtt{S}_1{\sqcup}\{\mathtt{v}\}$$

After that, `insert_sort` is invoked recursively with `s`. It will consume the precondition ($\mathtt{llB}(\mathtt{s},\mathtt{S}_1)$) and ensure the postcondition in the form of `Q` but with corresponding parameters (rule call-inf in Fig. 12). In that case, the state immediately after symbolic execution of line 7 is

$$\exists \mathtt{v},\mathtt{s},\mathtt{S}_1,\mathtt{r},\mathtt{S}_\mathtt{r} \cdot \mathtt{x}{\mapsto}\mathtt{node}(\mathtt{v},\mathtt{s}){*}\mathtt{Q}(\mathtt{s},\mathtt{S}_1,\mathtt{r},\mathtt{S}_\mathtt{r}) \wedge |\mathtt{S}|{>}1 \wedge \mathtt{S}{=}\mathtt{S}_1{\sqcup}\{\mathtt{v}\}$$

Note that existential variables (not in the parameter list of `Q`) are local variables whose quantification may be omitted for brevity. The state 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 $\mathtt{Q}(\mathtt{s},\mathtt{S}_1,\mathtt{r},\mathtt{S}_\mathtt{r})$ in second branch with $\mathtt{sllB}(\mathtt{r},\mathtt{S}_\mathtt{r}) \wedge \mathtt{P}(\mathtt{s},\mathtt{S}_1,\mathtt{r},\mathtt{S}_\mathtt{r})$ to make explicit the heap portion referred to by `r` before we analyse the auxiliary call `insert(r, x)` (via rule call-unk in Fig. 12). The shape of `r` comes from the postcondition of the method, and this is safe because the following entailment relationship is added to our assumption:

$$\mathtt{Q}(\mathtt{x},\mathtt{S},\mathtt{res},\mathtt{T}) \vdash \mathtt{sllB}(\mathtt{res},\mathtt{T}) \wedge \mathtt{P}(\mathtt{x},\mathtt{S},\mathtt{res},\mathtt{T})$$

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). Our analysis then uses an augmented technique (details follow slightly later) to synthesise the following specification for `insert` based on the symbolic state at the call site:

$$\mathtt{requires}\ \mathtt{sllB}(\mathtt{r},\mathtt{S}){*}\mathtt{x}{\mapsto}\mathtt{node}(\mathtt{v},\_)\ \mathtt{ensures}\ \mathtt{sllB}(\mathtt{res},\mathtt{T}) \wedge \mathtt{T}{=}\mathtt{S}{\sqcup}\{\mathtt{v}\}$$

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`:

$$\begin{aligned}\mathtt{Q}(\mathtt{x},\mathtt{S},\mathtt{res},\mathtt{T}) ::= {} & \mathtt{x}{\mapsto}\mathtt{node}(\mathtt{v},\mathtt{null}) \wedge \mathtt{res}{=}\mathtt{x} \wedge \mathtt{S}{=}\{\mathtt{v}\} \vee \\ & \mathtt{sllB}(\mathtt{res},\mathtt{S}_\mathtt{res}) \wedge \mathtt{P}(\mathtt{s},\mathtt{S}_\mathtt{s},\mathtt{r},\mathtt{S}_\mathtt{r}) \wedge |\mathtt{S}|{>}1 \wedge \mathtt{S}{=}\mathtt{S}_\mathtt{s}{\sqcup}\{\mathtt{v}\} \wedge \mathtt{S}_\mathtt{res}{=}\mathtt{S}_\mathtt{r}{\sqcup}\{\mathtt{v}\}\end{aligned}$$

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 result (`res`) refers 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 `s` and `r`), as the constraint abstraction denotes. Note that `T` will be not available (as well as its relationship with $\mathtt{S}_\mathtt{res}$) until the next step.

In the second step, we first derive the definition of the pure constraint abstraction `P` from the above post-state `Q`. Each disjunctive branch of `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:

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

We then use pure fixpoint solvers to obtain a closed-form formula $|\mathtt{S}|{\geq}1 \wedge \mathtt{T}{=}\mathtt{S}$ for $\mathtt{P}$. Based on (2.2), we now obtain the closed-form approximation for $\mathtt{Q}$:

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

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

$$\mathtt{requires}\ \mathtt{llB}(\mathtt{x},\mathtt{S}) \wedge \boxed{|\mathtt{S}|{\geq}1}\ \ \mathtt{ensures}\ \mathtt{sllB}(\mathtt{res},\mathtt{T}) \wedge \boxed{\mathtt{T}{=}\mathtt{S}}$$

which imposes more requirements 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.

2.2.1. Analysis for the Unannotated Method

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). Since no user-annotations are provided for this auxiliary method, 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 ($\mathtt{x}{\mapsto}\mathtt{node}(\mathtt{v},\mathtt{s}){*}\mathtt{sllB}(\mathtt{r},\mathtt{S_r})$). 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) $\mathtt{sllB}(\mathtt{x},\mathtt{S_1}){*}\mathtt{sllB}(\mathtt{res},\mathtt{S_2})$, and (b) $\mathtt{sllB}(\mathtt{res},\mathtt{S})$, etc. The symbolic execution gives the following post-state:

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

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 (2.2) (page 6) for `insert` and continue with the analysis for the primary method.

2.3. 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.

The method `sdl2nbt` (Fig 2) converts a doubly-linked sorted list into a node-balanced binary search tree, as indicated by the shape-only specification in lines 2 and 3. It first finds the "centre" node in the list (`root`), where the difference between numbers of nodes to the left and to the right of the centre node is at most one (lines 5-10), as Fig 3 (a) shows. It then applies the algorithm recursively on both list segments to the left and to the right of the centre node, and regards the centre node as the tree's root, whose left and right children are the resulting subtrees' roots from the recursive calls (lines 11-17), as in Fig 3 (b) and (c). As the data structures of doubly-linked list and binary tree are homomorphic (line 0), the algorithm reuses the nodes in the input instead of creating a new tree, making itself in-place. The parameter `head` in line 1 denotes the first node of the input list, and `tail` is where the last node's `next` field points to. When using this method `tail` should be set as `null` initially. The predicates for doubly-linked sorted list segment (`sdlB`) and node-balanced binary search tree (`nbt`) are defined in Section 2.1.

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

Figure 2: The method to convert a sorted doubly-linked list to a node-balanced tree.

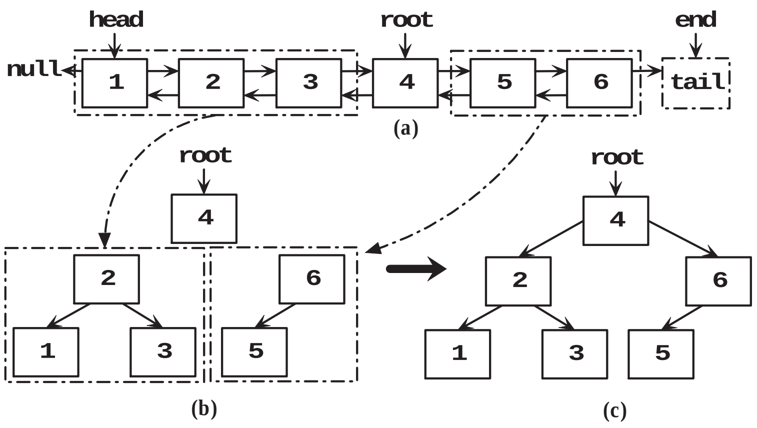

Figure 3: Transferring from a sorted doubly-linked list to a node-balanced BST.

To analyse the example, when the forward analysis reaches the while loop at line 6, it discovers that the loop has no user-supplied annotations. In that case, it uses an augmented technique (details follow slightly later) to synthesise the loop's pre- and post-shapes, and invoke the analysis procedure recursively to find additional pure constraints. In this way, we can infer the while loop's postcondition as

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

which indicates that the original list starting from `head` is cut into two sorted pieces with a cutpoint `root`. Meanwhile, the essential constraint (the underlined part, saying the list segment beginning with `head` is at most one node shorter 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:

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

where the first two disjunctive branches are base cases of the method's invocation (where there are only one and two nodes in the returned tree `res`, respectively), and the last denotes the effect of recursive calls combined into the postcondition (where `root`'s both branches are node-balanced trees). Note that the two `Q`'s in the last branch correspond to the invocations of `sdl2nbt` in lines 12 and 16. Constraints of some logical variables (like $\mathtt{S_{res}}$) will not show up until the next step.

In the second step, to derive the definition of the pure constraint abstraction `P` from the above post-state `Q`, we use each disjunctive branch of `Q` to entail the user-given post-shape (with appropriate instantiations of the parameters). During this process, all occurrences of `Q` are replaced by the post-shape conjoined with the `P` according to the entailment relation

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

The obtained frames (from the Sleek prover [2]) are used to form (via disjunction) the definition of `P`:

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

We then use pure fixpoint solvers to obtain a closed-form formula $\mathtt{p}{=}\mathtt{null} \wedge \mathtt{q}{=}\mathtt{tail} \wedge \mathtt{S}{=}\mathtt{S_{res}} \wedge |\mathtt{S}|{\geq}1$ for `P`, and refine the method's specifications as

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

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` for termination. Meanwhile, there should be at least one node in the list for memory safety. With those obligations, the method guarantees that the result is a node-balanced binary search tree, with the same content as the input list.[1]

2.3.1. Analysis for the while loop.

The while loop in `sdl2nbt` (lines 6-10) discovers 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.

Instead of requiring users to supply the loop invariant, our analysis regards the loop as a tail-recursive method and computes its specifications based on the program state in which the loop starts. Our analysis first synthesises its pre- and post-shapes, and then continues the analysis in the same way as for the main method. The pre-shape can be abstracted from the program state in which the loop starts. The post-shape synthesis is done by checking the symbolic execution result of the loop body (unrolled once) against possible abstracted

[1]We will explain how to attach the fixpoint result to both pre and post in Sec 4.

shapes. For this example, we first generate shape candidates according to the variables accessed by the loop, such as (a) $\mathtt{sdlB}(\mathtt{head}, \mathtt{p_h}, \mathtt{q_h}, \mathtt{S_h}){*}\mathtt{sdlB}(\mathtt{root}, \mathtt{p_r}, \mathtt{q_r}, \mathtt{S_r})$, and (b) $\mathtt{sdlB}(\mathtt{head}, \mathtt{p_h}, \mathtt{q_h}, \mathtt{S_h}){*}\mathtt{nbt}(\mathtt{root}, \mathtt{h_r}, \mathtt{b_r}, \mathtt{S_r})$. Then the unrolled loop body is symbolically executed to filter out those shapes that are not valid to be an abstraction of postcondition. For this example, executing the loop body yields

$$\begin{array}{l}\mathtt{head}{\mapsto}\mathtt{node2}(\mathtt{v}, \mathtt{p}, \mathtt{end}) \wedge \mathtt{head}{=}\mathtt{root} \wedge \mathtt{end}{=}\mathtt{tail} \vee \\ \quad \mathtt{head}{\mapsto}\mathtt{node2}(\mathtt{v_h}, \mathtt{p}, \mathtt{root}){*}\mathtt{root}{\mapsto}\mathtt{node2}(\mathtt{v_r}, \mathtt{head}, \mathtt{end}) \wedge \mathtt{end}{=}\mathtt{tail}\end{array} \tag{2}$$

where (b) is directly filtered out since (2) $\vdash$ $(b){*}\mathtt{true}$ fails. However (a) remains a candidate, as (2) $\vdash$ $(a){*}\mathtt{true}$ holds. Therefore, regarding (a) as a possible post-shape, we can employ the same approach to generate a constraint abstraction for the while loop, and solve it to obtain formula (1) in page 8.

One more note for the while loop in this example is that the symbolic execution may actually permit more than one shape to enter as candidates, e.g. $\mathtt{sdlB}(\mathtt{head}, \mathtt{p_h}, \mathtt{q_h}, \mathtt{S_h})$. 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 $\mathtt{sdlB}(\mathtt{head}, \mathtt{p_h}, \mathtt{q_h}, \mathtt{S_h})$ 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 approximates the invariant and hence loses 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.

$$\begin{array}{lll}
\mathit{Prog} & ::= \mathit{tdecl}^* \ \mathit{meth}^* & \mathit{tdecl} ::= \mathit{datat} \mid \mathit{spred} \\
\mathit{datat} & ::= \mathtt{data}\ c\ \{\ \mathit{field}^*\ \} & \mathit{field} ::= t\ v \qquad t ::= c \mid \tau \\
\mathit{meth} & ::= t\ \mathit{mn}\ ((t\ v)^*;(t\ v)^*)\ \mathit{mspec}^*\ \{e\} & \quad \tau ::= \mathtt{int} \mid \mathtt{bool} \mid \mathtt{void} \\
e & \multicolumn{2}{l}{::= d \mid d[v] \mid v{=}e \mid e_1;e_2 \mid t\ v;\ e \mid \mathtt{if}\ (v)\ e_1\ \mathtt{else}\ e_2 \mid \mathtt{while}\ (v)\ \{e\}} \\
d & \multicolumn{2}{l}{::= \mathtt{null} \mid k^\tau \mid v \mid \mathtt{new}\ c(v^*) \mid \mathit{mn}(u^*;v^*)} \\
d[v] & \multicolumn{2}{l}{::= v.f \mid v.f{:=}w \mid \mathtt{free}(v)}
\end{array}$$

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

The language is expression-oriented, so the body of a method is an expression e, where d (resp. $d[v]$) denotes a heap insensitive (resp. heap sensitive) atom expression. We also allow 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). We use k^τ to denote constants of type τ.

Our specification language (in Fig 5) allows (user-defined) shape predicates to specify both separation and pure properties. The shape predicates *spred* are constructed with disjunctive constraints Φ. We require that the predicates be well-formed [2]. A conjunctive abstract program state, σ, is composed of a heap (shape) part κ and a pure part π, where π consists of γ and ϕ as aliasing and numerical (size and bag) information respectively. We use SH to denote the set of such conjunctive states. 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. $\mathtt{Q}(\mathtt{v}^*)$) may occur in Δ during the analysis. A closed-form Δ (containing no constraint abstractions) can be normalised to the Φ form [2]. Pure constraint abstraction $\mathtt{P}$ is analogously defined to $\mathtt{Q}$.

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

Figure 5: The Specification Language.

The memory model of our specification formulae is adapted from the model given for "early versions" of separation logic [6], except that we consider memory cells to be structured records. We assume sets Loc of memory locations, Val of primitive values (with $0 \in \mathsf{Val}$ denoting `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, with h, s from the following concrete domains:

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

Heaps are finite partial functions mapping locations to values while stacks (stores) are total mappings from variables to values or locations, as in the classical separation logic [7, 6]. The detailed model definitions can be found in Chin et al. [12].

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 `x` and `xn`), and denote a program variable's initial value as unprimed, and its current (and hence final) value as primed [13, 2]. For the third group, we use subscript ones like $\mathtt{x}_1$ and $\mathtt{xn}_1$. For instance, for a code segment $\mathtt{x} := \mathtt{x}+1;\ \mathtt{x} := \mathtt{x}-2$ starting with state $\{\mathtt{x}{>}1\}$, we have the following reasoning procedure:

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

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

Our overall analysis algorithm is presented in Fig 6. It takes as input all available specifications and shape predicates, and the code segment to be analysed, together with an optional conjunctive program state and two variable sequences (mainly for loops and auxiliary methods). The algorithm first recognises the type of input code segment (mn in line 1). In the first two cases (while loop in line 2 and auxiliary method call in line 3), since we assume no information is given on specifications, some preprocessing work is conducted to discover possible pre- and post-shapes for the code segment with $\mathsf{Preproc}$ (Fig 7). In the case of a primary method (line 4), as we assume the shape-based specifications are given by users, no preprocessing is needed. Our constraint abstraction generation and solving algorithm is then applied to each (partial) specification

```
Algorithm Analysis($\mathcal{T}, \mathcal{S}, mn, \sigma, x^*, y^*$)
1   case $mn$ of
2     | while $(w)$ $\{e_0\}$ $\rightarrow f :=$ fresh_name(); $e :=$ if $(w)$ $\{e_0;\ f(x^*; y^*)\}$;
      $(u^*, v^*) := (x^*, y^*)$; $([(\Phi^i_{pr}, \Phi^i_{po})], n) :=$ Preproc($\mathcal{T}, \mathcal{S}, f, x^*, y^*, e_0, \sigma, x^*, y^*$);
      $prim :=$ false;
3     | $t\ mn\ ((t\ u_0)^*; (t\ v_0)^*)\ \{e_0\} \rightarrow f := mn;\ e := e_0;\ (u^*, v^*) := (u^*_0, v^*_0)$;
      $([(\Phi^i_{pr}, \Phi^i_{po})], n) :=$ Preproc($\mathcal{T}, \mathcal{S}, f, u^*, v^*, e_0, \sigma, x^*, y^*$); $prim :=$ false;
4     | $t\ mn\ ((t\ u_0)^*; (t\ v_0)^*)\ (requires\ \Phi^{pr}_i\ ensures\ \Phi^{po}_i)^m_{i=1}\ \{e_0\} \rightarrow f := mn$;
      $e := e_0;\ (u^*, v^*) := (u^*_0, v^*_0);\ n := m;\ (\Phi^i_{pr}, \Phi^i_{po})^n_{i=1} := (\Phi^{pr}_i, \Phi^{po}_i)^n_{i=1}$;
      $prim :=$ true;
5   end case
6   $rsps := \emptyset$
7   for $i := 1$ to $n$ do
8     $rsp :=$ CA_Gen_Solve($\mathcal{T}, f, e, \Phi^i_{pr}, \Phi^i_{po}, u^*, v^*$)
9     if $prim =$ false and $rsp \neq$ fail then return $(f, rsp)$
10    else if $prim =$ true then $rsps := rsps \cup rsp$
11    end if
12  end for
13  return $(f, rsps)$
end Algorithm
```

Figure 6: Main analysis algorithm.

```
Algorithm Preproc($\mathcal{T}, \mathcal{S}, f, u^*, v^*, e, \sigma, x^*, y^*$)
14  $sps := [\,]$;
15  $prs :=$ SynPre($\mathcal{S}, f, u^*, v^*, \sigma, x^*, y^*$)
16  for $\Phi_{pr} \in prs$ do
17    $pos :=$ SynPost($\mathcal{T}, \mathcal{S}, f, e, \Phi_{pr}, u^*, v^*$)
18    $sps :=$ concat($sps, pos$)
19  end for
20  return $(sps, |sps|)$
end Algorithm
```

Figure 7: Pre-processing algorithm.

to refine it (line 8). Note here we apply a lazy scheme when analysing loops and auxiliary methods: as the pre-processing may yield a list of possible shape specifications (ordered with heuristics such that the specifications with higher probability to make the whole verification succeed are more in front), we try each in sequence. Once a pre/post pair taken from the ordered list (after being refined) leads to the successful analysis of the enclosing primary method, the other ones in the list are omitted. In this way we try to make our verification more scalable, as still will be described in later sections.

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 15 and 17. 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.

4.1. Refining Specifications for Primary Methods

The algorithm for refinement (CA_Gen_Solve) is given in Fig 8. As illustrated in Section 2.3, the analysis proceeds in two steps for a primary method with shape information given in specification, namely (1) forward analysis (at lines 21-22) and (2) pure constraint abstraction generation and solving (at lines 23-30).

```
Algorithm CA_Gen_Solve($\mathcal{T}, mn, e, \Phi_{pr}, \Phi_{po}, u^*, v^*$)
21      $\Delta := $ Symb_Exec($\mathcal{T}, mn, e, \Phi_{pr}$)
22      if $\Delta =$ fail then return fail end if
23      Normalise $\Delta$ to DNF, and denote as $\bigvee_{i=1}^{m} \Delta_i$
24      $w^* := \{u^*, v^*, v'^*\} \cup$ pureV($\{u^*, v^*, v'^*\}, \Phi_{pr} \vee \Phi_{po}$)
25      $\Delta_{\mathtt{P}} :=$ Pure_CA_Gen($\Phi_{po}, \mathtt{Q}(w^*) ::= \bigvee_{i=1}^{m} \Delta_i$)
26      if $\Delta_{\mathtt{P}} =$ fail then return fail end if
27      $\pi :=$ Pure_CA_Solve($\mathtt{P}(w^*) ::= \Delta_{\mathtt{P}}$)
28      $R := t\ mn\ ((t\ u)^*; (t\ v)^*)\ requires$
            ex_quan($\Phi_{pr}, \pi$) $ensures$ ex_quan($\Phi_{po}, \pi$)
29      if Verify($\mathcal{T}, mn, R$) then return $\mathcal{T} \cup \{R\} \setminus$
          $\{\, t\ mn\ ((t\ u)^*; (t\ v)^*)\ requires\ \Phi_{pr}\ ensures\ \Phi_{po}\, \}$
30      else return fail end if
end Algorithm
```

```
Algorithm Symb_Exec
    ($\mathcal{T}, mn, e, \Phi_{pr}$)
31  $errLbls := \emptyset$
32  do
33    $(\Delta, l) := [\![e]\!]_{\mathcal{T}} mn(\Phi_{pr}, 0)$
34    if $l{>}0 \wedge l \notin errLbls$ then
35      $\Phi_{pr}$:=ex_quan($\Phi_{pr}, \Delta$);
36      $errLbls := errLbls \cup \{l\}$
37    else if $l{>}0 \wedge l \in errLbls$
      then return fail
38    end if
39  while $l > 0$
40  return $\Delta$
end Algorithm
```

Figure 8: Refining method specifications.

The forward analysis is captured as algorithm Symb_Exec to the right of Fig 8. Starting from a given pre-shape Φ_{pr}, it analyses the method body e (via symbolic execution; line 33) to compute the post-state in constraint abstraction form. The symbolic execution rules are presented in Section 4.4 and they are similar to symbolic rules used in [2], except for a novel mechanism to derive pure precondition, which we refer to as *pure abduction*.

This pure abduction mechanism is invoked whenever symbolic execution fails to prove memory safety based on the current prestate. For example, if we have $\mathtt{ll(x,n)}$ as the current state and we require $\mathtt{x{\mapsto}node(\_,p)}$ to update the value of $\mathtt{p}$, then it will fail as $\mathtt{ll(x,n)}$ does not necessarily guarantee $\mathtt{x{\mapsto}node(\_,p)}$. In this case we conduct the pure abduction as

$$\mathtt{ll(x,n)} \wedge [\mathtt{n}{\geq}1] \rhd \mathtt{x{\mapsto}node(\_,p){*}true}$$

to compute the missing pure information (in the square brackets) such that the LHS (including the newly gained pure part) entails the RHS. The variable *errLbls* (initialised at line 31) is to record the program locations in which previous pure abductions occurred. Whenever the symbolic execution fails, it returns a state Δ that contains the pure abduction result and the location l where failure was detected, as shown in line 33. If the current abduction location l is not recorded in *errLbls*, it indicates that this is a new failure. The abduction result is added to the precondition of the current method to obtain a stronger Φ_{pr}, before the algorithm enters the symbolic execution loop with variable *errLbls* updated to add in the new failure location l. This loop is repeated until symbolic execution succeeds with no memory error, or a previous failure point was re-encountered. The latter may indicate a program bug or a specification error, or may be due to the possible incompleteness of the underlying SLEEK prover we use. For example, for a method `void foo (...) {node w := new node(0,null); goo(w);...}` invoking a method `goo(x)` with precondition $\mathtt{ll(x,n)} \wedge \mathtt{n}{\geq}2$, our analysis will perform an abduction to get $\mathtt{n}{\geq}2$ 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 $\mathtt{n}{\geq}2$ 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.

Back to the main algorithm CA_Gen_Solve, the analysis next builds a heap-based constraint abstraction, named $\mathtt{Q}(w^*)$, for the post-state in line 23. This constraint abstraction is possibly recursive. (Definition † on page 8 is an example of this heap-based abstraction.) We then make use of another algorithm in Fig 9, named Pure_CA_Gen, to extract a pure constraint abstraction, named $\mathtt{P}(w^*)$, without any heap property. (Definition ‡ on page 9 is an instance of this pure abstraction.) This algorithm tries to derive a branch $\mathtt{P}_i$

for each branch Δ_i of Q. For every Δ_i it proceeds in two steps. In the first step (lines 42-44), it replaces the recursive occurrence of Q in Δ_i with $\sigma{*}\mathtt{P}(w^*)$. In the second step (lines 45-46) it tries to derive $\mathtt{P}_i$ via the entailment. If the entailment fails, then pure abduction is used to discover any missing pure constraint σ'_i for $\rho\Delta_i$ to allow the entailment to succeed. In this case, σ'_i is incorporated into σ_i (and eventually $\mathtt{P}_i$). Once this is done, we use some existing fixpoint analysis (e.g. [10]) inside Pure_CA_Solve to derive non-recursive constraint π, as a simplification of $\mathtt{P}(w^*)$. This result is then incorporated into the pre/post specifications in line 28, before we perform a post verification in line 29 using the Hip verifier [2], to ensure the strengthened precondition is strong enough for memory safety.

Two auxiliary functions used in the algorithm are described here. 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 (size and bag) properties in these shapes, e.g. $\mathsf{pureV}(\{\mathtt{x}\}, \mathtt{ll}(\mathtt{x},\mathtt{n}))$ returns $\{\mathtt{n}\}$. 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 Q. The function $\mathsf{ex\_quan}(\Delta, \pi)$ is to strengthen the state Δ with the abduction result π: $\mathsf{ex\_quan}(\Delta, \pi) =_{df} \Delta \wedge \exists(\mathsf{fv}(\pi) \setminus \mathsf{fv}(\Delta)) \cdot \pi$. It is used to incorporate the discovered missing pure constraints into the original specification. For example, $\mathsf{ex\_quan}(\mathtt{ll}(\mathtt{x},\mathtt{n}), \mathtt{0}{<}\mathtt{m} \wedge \mathtt{m}{\leq}\mathtt{n})$ returns $\mathtt{ll}(\mathtt{x},\mathtt{n}) \wedge \mathtt{0}{<}\mathtt{n}$.

```
Algorithm Pure_CA_Gen($\sigma$, $\mathtt{Q}(w^*)::=\bigvee_{i=1}^{m} \Delta_i$)
41 for $i = 1$ to $m$
42    Denote all appearances of $\mathtt{Q}(w^*)$ in $\Delta_i$ as $\mathtt{Q}_j(w_j^*), j = 1, ..., p$
43    Denote substitutions $\rho_j = [([w_j^*/w^*]\sigma{*}\mathtt{P}(w_j^*))/\mathtt{Q}_j(w_j^*)]$
44    Let substitution $\rho := \rho_1 \circ \rho_2 \circ ... \circ \rho_p$ as applying all substitutions
         defined above in sequence
45    if $(\rho\Delta_i \vdash \sigma{*}\sigma_i$ or $\rho\Delta_i \wedge [\sigma'_i] \rhd \sigma{*}\sigma_i)$ and $ispure(\sigma_i)$ then $\mathtt{P}_i := \sigma_i$
46    else return fail end if
47 end for
48 return $\bigvee_{i=1}^{m} \mathtt{P}_i$
end Algorithm
```

Figure 9: Pure constraint abstraction generation algorithm.

4.2. Pure abduction mechanism

We use the Sleek prover [2] to check Δ_1 entails Δ_2. If the entailment holds it also derives Δ_3 (a.k.a. frame) such that $\Delta_1 \vdash \Delta_2{*}\Delta_3$. However, if it fails, we assume that the shape information is sufficiently provided, and use our pure abduction mechanism ($\sigma_1 \wedge [\sigma'] \rhd \sigma_2{*}\sigma_3$ in Fig 10) to discover missing pure constraints σ' so that $\sigma_1 \wedge \sigma' \vdash \sigma_2{*}\sigma_3$.

Our pure abduction deals with three different cases. The first rule (R1) 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 $\mathtt{ll}(\mathtt{x},\mathtt{n}) \nvdash \mathtt{x}{\mapsto}\mathtt{node}(\_, \mathtt{null})$, the RHS can entail the LHS with pure frame $\mathtt{n}{=}1$. The abduction then checks to ensure$\mathtt{ll}(\mathtt{x},\mathtt{n}) \wedge \mathtt{n}{=}1 \vdash \mathtt{x}{\mapsto}\mathtt{node}(_, \mathtt{null}){*}\sigma_2$ for some σ_2, and returns the result $\mathtt{n}{=}1$. Note the check $ispure(\sigma')$ ensures that σ' contains no heap information.

In the second rule (R2), neither side entails the other but the LHS term could be unfolded. An example is $\sigma = \mathtt{sllB}(\mathtt{x},\mathtt{S})$, $\sigma_1 = \mathtt{x}{\mapsto}\mathtt{node}(\mathtt{u},\mathtt{p}){*}\mathtt{p}{\mapsto}\mathtt{node}(\mathtt{v},\mathtt{null})$. 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 and try entailment or further abduction with the results (σ_0) against σ_1. If it succeeds with a pure frame σ', then we confirm the abduction by checking $\sigma \wedge \sigma' \vdash \sigma_1{*}\sigma_2$. For the example above, the abduction returns $|\mathtt{S}|{=}2$ (σ') and discovers the nontrivial frame $\mathtt{S}{=}\{\mathtt{u},\mathtt{v}\}{\wedge}\mathtt{u}{\leq}\mathtt{v}$ (σ_2). Note that function data_no returns the number of data nodes in a state,

$$\frac{\sigma \nvdash \sigma_1{*}\texttt{true} \quad \sigma_1 \vdash \sigma{*}\sigma' \quad \mathit{ispure}(\sigma') \quad \sigma \wedge \sigma' \vdash \sigma_1{*}\sigma_2}{\sigma \wedge [\sigma'] \rhd \sigma_1{*}\sigma_2}\ (\textbf{R1})$$

$$\frac{\begin{array}{c}\sigma \nvdash \sigma_1{*}\texttt{true} \quad \sigma_1 \nvdash \sigma{*}\texttt{true} \quad \sigma_0 \in \mathsf{unroll}(\sigma) \quad \mathsf{data\_no}(\sigma_0) \leq \mathsf{data\_no}(\sigma_1) \\ (\sigma_0 \vdash \sigma_1{*}\sigma' \ \text{ or } \ \sigma_0 \wedge [\sigma_0'] \rhd \sigma_1{*}\sigma') \quad \mathit{ispure}(\sigma') \quad \sigma \wedge \sigma' \vdash \sigma_1{*}\sigma_2\end{array}}{\sigma \wedge [\sigma'] \rhd \sigma_1{*}\sigma_2}\ (\textbf{R2})$$

$$\frac{\sigma \nvdash \sigma_1{*}\texttt{true} \quad \sigma_1 \nvdash \sigma{*}\texttt{true} \quad \sigma_1 \wedge [\sigma_1'] \rhd \sigma{*}\sigma' \quad \mathit{ispure}(\sigma') \quad \sigma \wedge \sigma' \vdash \sigma_1{*}\sigma_2}{\sigma \wedge [\sigma'] \rhd \sigma_1{*}\sigma_2}\ (\textbf{R3})$$

Figure 10: Pure abduction rules.

e.g. it returns one for $\texttt{x}{\mapsto}\texttt{node}(\texttt{v},\texttt{p}){*}\texttt{ll}(\texttt{p},\texttt{m})$. (This syntactic check is important for the termination of the abduction.) The unroll operation 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 $\texttt{x}{\mapsto}\texttt{node}(\texttt{v},\texttt{p}){*}\texttt{ll}(\texttt{p},\texttt{m})$ to be $\{\texttt{x}{\mapsto}\texttt{node}(\texttt{v},\texttt{p}) \wedge \texttt{p}{=}\texttt{null} \wedge \texttt{m}{=}0, \exists \texttt{u},\texttt{q},\texttt{k} \cdot \texttt{x}{\mapsto}\texttt{node}(\texttt{v},\texttt{p}){*}\texttt{p}{\mapsto}\texttt{node}(\texttt{u},\texttt{q}){*}\texttt{ll}(\texttt{q},\texttt{q}) \wedge \texttt{m}{=}\texttt{k}{+}1\}$.

In the third rule (**R3**), neither side entails the other and the LHS term cannot be unfolded. This happens often during the abstraction stage in the analysis when we need to fold up a "concrete" state of nodes against an abstracted shape predicate, e.g., when $\sigma = \texttt{x}{\mapsto}\texttt{node}(\texttt{u},\texttt{p}){*}\texttt{p}{\mapsto}\texttt{node}(\texttt{v},\texttt{null})$ and $\sigma_1 = \exists \texttt{S} \cdot \texttt{sllB}(\texttt{x},\texttt{S})$. In this case, the rule swaps the two sides of the entailment and applies the second rule (**R2**) to uncover the pure constraints σ_1' and σ'. It checks that adding σ' to the LHS (σ) entails the RHS (σ_1) before it returns σ'. For the example, the abduction returns $\texttt{u}{\leq}\texttt{v}$ which is essential for the two nodes to form a sorted list (RHS). Note that the rule (**R3**) is only allowed to be applied once in one abduction query to avoid infinite number of swapping (between (**R2**) and (**R3**)).

Example 1. *For example, to verify the* append *procedure used in quick sort, the given precondition signifies the two inputs are sorted lists,* $\texttt{sll}(\texttt{x},\texttt{xn},\texttt{xs},\texttt{xl}){*}\texttt{sll}(\texttt{y},\texttt{yn},\texttt{ys},\texttt{yl})$*, and the postcondition is* $\texttt{sll}(\texttt{x},\texttt{m},\texttt{rs},\texttt{rl})$*. By using our abduction mechanism, our analysis can find out that* $\texttt{xl} \leq \texttt{ys}$ *which indicates that the largest element in the first list should be less than or equal to the smallest element in the second list which makes the functionality of append correct.* □

4.3. Inferring Specifications for Auxiliary Methods and Loops

For auxiliary methods[2], we conduct a pre-analysis (Fig 11) to synthesise the pre- and post-shapes before we conduct the refinement analysis from Fig 8. Loops are dealt with by analysing their tail-recursive versions in the same way. This approach alleviates the need for users to provide specification annotations for both loops and auxiliary methods.

The pre-shape synthesis algorithm SynPre (Fig 11 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 obtains possible shape candidates from the parameters u^*, v^* with $\mathsf{ShpCand}$ (line 49), then picks up a sound abstraction for the method's pre-shape with entailment, and filters out the ones which fail (line 52). Finally the pre-shape abstraction is returned. While we use an enumeration strategy here, the number of possible shape candidates per type is small as it is strictly limited by what the user provides in the primary methods, and then filtered and prioritised by our system.

[2]In practice, we treat methods without user-specified shape specifications as auxiliary.

```
Algorithm SynPre
  ($\mathcal{S}, f, u^*, v^*, \sigma, x^*, y^*$)
49  $C$ := ShpCand($\mathcal{S}, u^*, v^*$)
50  for $\sigma_C \in C$ do
51    if $\sigma \nvdash [x^*/u^*, y^*/v^*]\sigma_C$
52    then $C$:=$C \backslash \{\sigma_C\}$ end if
53  end for
54  return $C$
end Algorithm
```

```
Algorithm SynPost ($\mathcal{T}, \mathcal{S}, f, e, \Phi_{pr}, u^*, v^*$)
55  $C$ := ShpCand($\mathcal{S}, u^*, v^*$)
56  $\mathcal{T}'$:=$\mathcal{T} \cup \{f(u^*,v^*)$ requires $\Phi_{pr}$ ensures false $\{e\}\}$
57  $\Delta$ := Symb_Exec($\mathcal{T}', f$, syn_unroll($f, e$), $\Phi_{pr}$)
58  for $\sigma_C \in C$ do
59    if $\Delta \wedge [\sigma] \not\vartriangleright \sigma_C$ then $C := C \backslash \{\sigma_C\}$ end if
60  end for
61  return pair_spec_list($\Phi_{pr}, C$)
end Algorithm
```

Figure 11: Shape synthesis algorithms.

To synthesise post-shapes (SynPost, Fig 11 right), we also assign C as possible shape candidates (line 55). 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 57), assuming f has a specification *requires* Φ_{pr} *ensures* `false` (line 56). 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 find out appropriate abstraction of post-shape (line 59),which is paired with Φ_{pr} and returned as result. Here we use pure abduction to filter post-shapes to preserve as many shapes that are potentially refinable as possible. The function pair_spec_list(Φ_{pr}, C) 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.

We illustrate our procedure to generate and confirm candidate shape abstractions (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 [`sllB(x,S)`$*$`sllB(y,T)`, `llB(x,S)`$*$`sllB(y,T)`, `sllB(x,S)`$*$`llB(y,T)`, `llB(x,S)`$*$`llB(y,T)`, `sllB(x,S)`, `sllB(y,S)`, `llB(x,S)`, `llB(y,S)`, `emp`]. Elements of this list will be checked against appropriate abstract states (line 51-52 in Fig 11 left and line 59 in Fig 11 right) where unsound elements will be eliminated. For example, in the previous list, only `llB(x,S)`$*$`llB(y,T)` remains in the list and participates in further verification, given σ = `x`$\mapsto$`node(u,p)`$*$`p`$\mapsto$`node(v,null)`$*$`y`$\mapsto$`node(s,q)`$*$`q`$\mapsto$`node(t,null)`.

The initial experimental results confirm that our shape synthesis keeps only highly relevant abstractions. For the while loop in Section 2.3, we filtered out 24 (of 26) abstractions. Generally, in case that there are several abstractions as candidate specifications, we employ some other mechanisms to reduce them further. Firstly, we prioritise post-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. Secondly, 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 8) 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. 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. These mechanisms help to keep attempts over candidate specifications at a minimum level.

4.4. 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

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

where $[\![e]\!]$ takes as its first parameter the set of method specifications and as its second parameter the name of the current method. The integer (label) in both input and output is used to record a program location where abduction is needed. If the label remains zero after the symbolic execution of e, then the output state denotes the post-state of e. A positive label indicates that an abduction has occurred and the resulting state (the abduction result) will be propagated back to the method's precondition by our analysis, and the next round of symbolic execution will be required.

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

$\mathsf{unfold}(x)$	$=_{df} \mathsf{SH} \to \mathcal{P}_{\mathsf{SH}[x]}$	Unfolding
$\mathsf{exec}(d[x])$	$=_{df} (\mathsf{AllSpec}{\times}\mathsf{Names}) \to (\mathsf{SH}[x]{\times}\mathsf{Int}) \to (\mathsf{SH}{\times}\mathsf{Int})$	Heap-sensitive exec.
$\mathsf{exec}(d)$	$=_{df} (\mathsf{AllSpec}{\times}\mathsf{Names}) \to (\mathsf{SH}{\times}\mathsf{Int}) \to (\mathsf{SH}{\times}\mathsf{Int})$	Heap-insensitive exec.

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 ($\mathtt{x}{\mapsto}\mathtt{c}(\mathtt{v}^*)$), and $\mathcal{P}_{\mathsf{SH}[x]}$ contains all the (disjunctive) abstract states, each of which is composed by such conjunctive states. $\mathsf{unfold}(x)$ 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 $\mathsf{exec}(d)$ 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`:

$$\frac{}{\mathsf{unfold}(x)(\mathtt{x}{\mapsto}\mathtt{c}(\mathtt{v}^*){*}\sigma) \rightsquigarrow \mathtt{x}{\mapsto}\mathtt{c}(\mathtt{v}^*){*}\sigma}$$

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

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.

$$\frac{\sigma \vdash \mathtt{x}{\mapsto}\mathtt{c}(\mathtt{v_1}, .., \mathtt{v_n}){*}\sigma'}{\mathsf{exec}(x.f_i)(\mathcal{T},f)(\sigma, 0) \rightsquigarrow (\sigma'{*}c(x, v_1, .., v_n) \wedge \mathtt{res}{=}v_i, 0)}$$

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

$$\frac{\sigma \vdash \mathtt{x}{\mapsto}\mathtt{c}(\mathtt{u}^*){*}\sigma'}{\mathsf{exec}(\mathtt{free}(x))(\mathcal{T},f)(\sigma, 0) \rightsquigarrow (\sigma', 0)}$$

$$\frac{\sigma \nvdash \mathtt{x}{\mapsto}\mathtt{c}(\mathtt{u}^*){*}\mathtt{true} \quad \sigma{*}[\sigma'] \triangleright \mathtt{x}{\mapsto}\mathtt{c}(\mathtt{u}^*){*}\mathtt{true}}{\mathsf{exec}(d[x])(\mathcal{T},f)(\sigma, 0) \rightsquigarrow (\sigma', \mathit{lbl}(d[x]))}$$

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

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 $\mathit{lbl}(-)$ to map any instruction in the program being analysed to a unique positive integer label (namely the aforementioned program location). The rule changes the second element of the result to $\mathit{lbl}(d[x])$ 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 $(\texttt{false}, \mathit{lbl}(d[x]))$.

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

$$\mathsf{exec}(\mathsf{new}\ c(v^*))(\mathcal{T},f)(\sigma,0) \rightsquigarrow (\sigma{*}\texttt{res}{\mapsto}\texttt{c}(\texttt{v}^*), 0)$$

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

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

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

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

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

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

Figure 12: Symbolic execution rules for heap-insensitive commands.

The symbolic execution rules for heap-insensitive commands are listed in Fig. 12. The first three rules deal with constant (k), variable (v) and data node creation ($\mathsf{new}\ c(v^*)$), respectively, while the remaining rules handle method invocation. The fourth rule (**WHILE**) and fifth rule (**CALL-UNK**) are used for the invocation of a while loop or an auxiliary method which has not been analysed, where we employ the analysis algorithm recursively to achieve its postcondition to enable application of the next rule. The sixth rule (**CALL-VER**) is used for the invocation of another method mn which has already been analysed ($mn{\neq}f$), 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 (**CALL-INF**) is for a recursive call to the current method ($mn{=}f$), similar as above except that a constraint abstraction is in place as postcondition. The last two rules are for the cases where abductive reasoning is employed (**CALL-ABD**) and where the call site cannot establish the precondition of the callee method (**CALL-FAIL**). 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 analyse; therefore each method to be analysed should only call itself recursively. This assumption does not lose generality, as we can always transform mutual recursion into single recursion [14] 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:

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

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:

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

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:

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

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

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

4.5. Soundness and Termination

In this section we discuss about the soundness of our analysis and show that our analysis is also terminating.

Before proceeding to the soundness property, recalling that we have both unprimed variables (for their initial values in abstract states) and primed ones (for their current values), we realise that the concrete program states should always be linked to the primed ones. For this reason we have the following definition:

Definition 1 (Poststate)**.** *Given an abstract state Δ, Post(Δ) captures the relation between primed variables of Δ. That is,*

$$\begin{array}{l}
\mathit{Post}(\Delta) =_{df} \rho(\exists V \cdot \Delta),\ \textit{where} \\
\qquad V = \{v_1,\ldots,v_n\}\ \textit{denotes all unprimed program variables in}\ \Delta,\ \textit{and} \\
\qquad \rho = [v_1/v_1',\ldots,v_n/v_n'].
\end{array}$$

□

For example, for $\Delta = \texttt{Node}(\texttt{x}', \texttt{v}', \texttt{y}') \wedge \texttt{v}'{=}\texttt{v} \wedge \texttt{y}'{=}\texttt{null}$, we have $\mathit{Post}(\Delta) = \texttt{Node}(\texttt{x}, \texttt{v}, \texttt{y}) \wedge \texttt{y}{=}\texttt{null}$.

The soundness of our analysis is defined as follows:

Definition 2 (Soundness)**.** *For a method definition $t\ mn\ ((t\ u)^*; (t\ v)^*)\ \{e\}$, if our analysis refines its specification as $t\ mn\ ((t\ u)^*; (t\ v)^*)$ requires Φ_{pr} ensures Φ_{po} $\{e\}$, then for all $s, h \models \Phi_{pr}$, the execution from $\langle s, h, e\rangle$ never gets stuck; and if $\langle s, h, e\rangle \hookrightarrow^* \langle s', h', \nu\rangle$ for some value ν, then we have $s', h' \models \mathit{Post}(\Phi_{po})$.* □

The underlying operational semantics of our language was given in Chin et al. [12], where the small-step transition relation is of the form $\langle s, h, e\rangle \hookrightarrow \langle s', h', e'\rangle$. Its concrete program state consists of stack s and heap h, as described in Section 3. Chin et al. [12] also defines the relation $s, h \models \Delta$ and the (transitive closure) relation $\langle s, h, e\rangle \hookrightarrow^* \langle s', h', \nu\rangle$ which terminates with a value ν in some finite number of steps.

In our analysis, the synthesised missing specifications are discovered with the help of abduction and fixpoint calculators (to ensure convergence). The fixpoint calculators call join and widening operators over the numerical and bag domains. The join and widening operations usually weaken the pre-/post-conditions. Weakening a postcondition is deductively sound. However, weakening a precondition may cause the synthesised precondition unsafe for the method being analysed.

Meanwhile, if the given predicates used in pre-/post-shape templates are not good enough to describe the obligation of memory safety, our refinement of the precondition might not be sufficient for the program to execute safely without inappropriate memory access. For example, if our analysis is only supplied with a list shape predicate without any other information (such as length or content), then our analysis can never obtain a memory safety requirement like "the length of the input list should be at least `n`", even if it is needed for memory safety of the method.

So after the specification refinement, we apply the verification system HIP/SLEEK [12] to verify the program against the refined specifications so as to rule out any incorrect specifications (obtained due to the above-mentioned reasons), as shown in line 29 of our refinement algorithm in Figure 8. The soundness of our analysis is established by the soundness of the verification system HIP/SLEEK (given in [12]).

Theorem 1 (Termination)**.** *Our analysis terminates in a finite number of steps for a program with a finite number of partial specifications and a finite set of user-defined predicates.*

From the main analysis algorithm, we can see that the termination of our analysis relies on two points: the termination of fixpoint solver for any numerical and bag constraint abstraction and the termination of abduction. The termination of fixpoint solver has been proved [10, 11]. The non-termination problem of abduction may be caused by infinite number of unrolling in rule (**R2**), or infinitely recursive application of rule (**R3**). To prevent the problem, we have two restrictions when we apply the abduction rules: (i) the data_no check in (**R2**) which ensures that the unroll operation is not applied infinitely; (ii) the third rule (**R3**) is not allowed to be applied twice in one abduction query, which prevents the infinite swapping problem.

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 the tables (Fig. 13 and Fig. 14) describe, respectively, 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. For some programs, we have verified them with different pre/post shape templates. Programs with star $*$ have different versions for various data structures. The shape predicates used in the experiment but not previously given in paper are defined in Appendix A.

The results highlight the refinement of both pre- and postconditions based on user-provided shape specifications, even for complicated data structures such as AVL and red-black trees. Firstly, our approach can compute non-trivial pure constraints for postconditions, 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,

Prog.	**Time**	**Pre**	**Post**
List processing programs			
create*	0.379	$\mathtt{emp} \land n \geq 0$	$\mathtt{llB}(res, S) \land n = \lvert S \rvert \land \forall v \in S \cdot 1 \leq v \leq n$
	1.752	$\mathtt{emp} \land n \geq 0$	$\mathtt{dllB}(res, rp, S) \land n = \lvert S \rvert \land \forall v \in S \cdot 1 \leq v \leq n$
	0.954	$\mathtt{emp} \land n \geq 0$	$\mathtt{sllB2}(res, S) \land n = \lvert S \rvert \land \forall v \in S \cdot 1 \leq v \leq n$
sort_insert*	0.591	$\mathtt{ll}(x, n) \land n \geq 1$	$\mathtt{ll}(x, m) \land m = n+1$
	0.789	$\mathtt{dll}(x, p, n) \land n \geq 1$	$\mathtt{dll}(x, q, m) \land n \geq 1 \land m = n+1 \land p = q$
	0.504	$\mathtt{sll}(x, n, xs, xl) \land v \geq xs$	$\mathtt{sll}(x, m, mn, mx) \land xs = mn \land mx = \max(xl, v) \land m = n+1$
tail_insert	0.566	$\mathtt{ll}(x, n) \land n \geq 1$	$\mathtt{ll}(x, m) \land m = n+1$
	0.628	$\mathtt{sll}(x, n, xs, xl) \land v \geq xl$	$\mathtt{sll}(x, m, mn, mx) \land v = mx \land mn = xs \land m = n+1$
rand_insert*	0.522	$\mathtt{ll}(x, n) \land n \geq 1$	$\mathtt{ll}(x, m) \land m = n+1$
	0.830	$\mathtt{dll}(x, p, n) \land n \geq 1$	$\mathtt{dll}(x, q, m) \land m = n+1 \land p = q$
	—	$\mathtt{sll}(x, n, xs, xl) \land$ (fail)	$\mathtt{sll}(x, m, mn, mx) \land$ (fail)
delete	0.630	$\mathtt{llB}(x, S) \land \lvert S \rvert \geq 2$	$\mathtt{llB}(x, T) \land \exists a. S = T \sqcup \{a\}$
	1.024	$\mathtt{sllB}(x, S) \land \lvert S \rvert \geq 2$	$\mathtt{sllB}(x, T) \land \exists a. S = T \sqcup \{a\}$
travrs	0.296	$\mathtt{ll}(x, m) \land n \geq 0 \land m \geq n$	$\mathtt{ls}(x, p, k) * \mathtt{ll}(res, r) \land p = res \land k = n \land m = n+r$
	2.205	$\mathtt{sllB}(x, S) \land n \geq 0 \land \lvert S \rvert \geq n$	$\mathtt{slsB}(x, p, T) * \mathtt{sllB}(res, S_2) \land p = res \land \lvert T \rvert = n \land S = T \sqcup S_2 \land \forall u \in T, v \in S_2 \cdot u \leq v$
append*	0.512	$\mathtt{ll}(x, xn) * \mathtt{ll}(y, yn) \land xn \geq 1$	$\mathtt{ll}(x, m) \land m = xn+yn$
	0.660	$\mathtt{dll}(x, xp, xn) * \mathtt{dll}(y, yp, yn) \land xn \geq 1$	$\mathtt{dll}(x, q, m) \land m = xn+yn \land q = xp$
	0.948	$\mathtt{sll}(x, xn, xs, xl) \land xl \leq ys * \mathtt{sll}(y, yn, ys, yl)$	$\mathtt{sll}(x, m, rs, rl) \land yl = rl \land m \geq 1+yn \land m = xn+yn$

Figure 13: Experimental Results.

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 and 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 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 $*$). Further examples, avl_ins and rbt_ins also have some auxiliary (recursive) methods such as calculation of tree's height, which are automatically analysed

Prog.	**Time**	**Pre**	**Post**
Sorting programs			
Sorting (main)		$\mathtt{llB}(x,S) \wedge |S|\geq 1$	$\mathtt{sllB}(res,T) \wedge T=S$ (∗)
`merge`	4.107	$\mathtt{sllB}(x,S_x) * \mathtt{sllB}(y,S_y)$	$\mathtt{sllB}(res,T) \wedge T=S_x \sqcup S_y$
`flatten`	2.693	$\mathtt{bstB}(x,S)$	$\mathtt{sllB}(res,T) \wedge T=S$
`insert`	0.824	$\mathtt{sllB}(r,S) * x \mapsto \mathtt{node}(v,\_)$	$\mathtt{sllB}(res,T) \wedge T=S \sqcup \{v\}$
`quick`	2.132	$\mathtt{llB}(x,S)$	$\mathtt{llB}(x,S_1) * \mathtt{llB}(res,S_2) \wedge S=S_1 \sqcup S_2 \wedge \forall u \in S_1, v \in S_2 \cdot u \leq p \leq v$
Binary tree, binary search tree, AVL tree and red-black tree processing programs			
`travrs`	0.532	$\mathtt{bt}(x,S,h)$	$\mathtt{bt}(x,T,k) \wedge S=T \wedge h=k$
`count`	0.709	$\mathtt{bt}(x,S,h)$	$\mathtt{bt}(x,T,k) \wedge res=|S| \wedge S=T \wedge h=k$
`height`	0.913	$\mathtt{bt}(x,S,h)$	$\mathtt{bt}(x,T,k) \wedge res=h=k \wedge S=T$
`insert`	1.276	$\mathtt{bt}(x,S,h) \wedge |S|\geq 1 \wedge h\geq 1$	$\mathtt{bt}(x,T,k) \wedge T=S \sqcup \{v\} \wedge h\leq k\leq h+1$
`delete`	0.970	$\mathtt{bt}(x,S,h) \wedge |S|\geq 2 \wedge h\geq 2$	$\mathtt{bt}(x,T,k) \wedge \exists a. S=T \sqcup \{a\} \wedge h-1\leq k\leq h$
`search`	1.583	$\mathtt{bst}(x,sm,lg)$	$\mathtt{bst}(x,mn,mx) \wedge sm=mn \wedge lg=mx \wedge 0\leq res\leq 1$
`bst_ insert`	1.720	$\mathtt{bst}(x,sm,lg)$	$\mathtt{bst}(x,mn,mx) \wedge (v<sm \wedge v=mn \wedge lg=mx \vee lg<v \wedge v=mx \wedge sm=mn \vee sm=mn \wedge lg=mx)$
`avl_ins`	11.12	$\mathtt{avl}(x,S,h)$	$\mathtt{avl}(res,T,k) \wedge T=S \sqcup \{v\} \wedge h\leq k\leq h+1$
`rbt_ins`	8.76	$\mathtt{rbt}(x,S,0,h)$	$\mathtt{rbt}(res,T,0,k) \wedge T=S \sqcup \{v\} \wedge h\leq k\leq h+1$
`sdl2nbt`	5.826	$\mathtt{sdlB}(x,p,q,S) \wedge |S|\geq 1 \wedge p=null \wedge q=tail$	$\mathtt{nbt}(res,T) \wedge T=S$

Figure 14: Experimental Results. (Cont.)

as well. To have better efficiency, we advocate the use of smaller predicates to capture different aspects of complex data structures separately. As an example, the predicate `rbt` used for analysing `rbt_ins` captures contents and red-black colours only, leaving the binary search property to a separate predicate, e.g. `bstB`.

We have also tried our approach over part of the FreeRTOS kernel [15]. For its list processing programs `list.h` and `list.c` (472 lines with intensive manipulation over composite sorted doubly-linked lists) it took 2.85 seconds for our prototype to refine all the specifications given for the main functions, which further confirms the viability of our approach.

6. Related Work and 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. [4] was able to infer automatically loop invariants for list-processing programs, which formed the early-version SpaceInvader tool. Gotsman et al. [16] proposed an interprocedural shape analysis for the SLAyer tool. Berdine et al. [17] extended the local shape analysis [4] to handle higher-order list predicate so that more complicated real-world data structures can be analysed. Yang et al. [5] proposed a novel abstraction operation which significantly improves the scalability of the analysis. Recently, more large industrial code can be verified by the SpaceInvader/Abductor tool using the compositional analysis with bi-abductive inference [3, 18]. Due to their focus on a single shape domain, only heap information can be dealt with in their bi-abduction. However, in real code, additional information (e.g. numerical properties) may be needed even just for the verification of memory safety (e.g. red-black trees widely used in Linux Kernel). Compared with their work, our analysis focuses on a combined abstract domain in the presence of user-defined predicates, covering shape, numerical and bag information. Our analysis can deal with examples that cannot be handled by their work, as shown in the previous section, e.g. the travrs example for memory safety, the sorting examples and the avl tree, red-black tree examples for memory safety as well as functional correctness.

Several shape analyses also tried to make good use of size information. In the development of the THOR tool, Magill et al. [19] proposed an adaptive shape analysis where additional numerical analysis can be used to help gain better precision. Its abstraction mechanism is also employed in C-to-gate hardware synthesis [20]. Magill et al. [21] 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-balancedness, which have not been addressed in THOR, to the best of our knowledge. For example, the append procedure used in quick sort (mentioned in Sec. 4.2 and Fig. 13) cannot be handled in Magill et al. [21], probably due to the fact that their transformation from the original program to its heap-free program may lose certain information (in this example the contents of the lists) and that their analysis cannot transform the heap-free program back to the original program for further analysis. While in our work, the discovered pure information is propagated back, and together with the heap specification, our analysis can verify the original program directly. Gulwani et al. [22] 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. [23] and Chang and Rival [24], employs inductive checkers and checker segments to express shape and numerical information. Our previous loop invariant synthesis [25] also infers strong loop invariants with both shape and numerical information but is limited to while loop analysis. Compared with their works, ours addresses specification refinement with pure properties (including numerical and bag ones) in both pre- and postconditions by processing shape and pure information in two phases with the help of pure abduction. Our previous loop invariant synthesis [25] also infers strong loop invariants with a one-phase heavyweight abstract interpretation. Compared with this work, it is limited to loop analysis, whereas this work tackles not only loops but also methods; meanwhile this work is more lightweight as it solves the constraint abstraction in two phases where the second phase (pure constraint abstraction solving) utilises existing provers and is hence more modular and efficient.

There are also many other approaches to expressing heap-based domains than separation logic. Hackett and Rugina [26] can deal with AVL-trees but is customised to handle only tree-like structures with height property. The shape analysis framework TVLA [27] is based on three-valued logic. It is capable of handling complicated data structures and properties, such as sortedness. LRP [28] is fully decidable over multiple linked data structures and has a finite model property. Guo et al. [29] reported a global shape analysis that discover inductive structural shape invariants from the code. Kuncak et al. [30] 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. Bouajjani et al. [31] automatically synthesize invariants for list manipulating programs over infinite data domains using a graph representation of heap. Compared with these works, a 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 (except for [31] which captures various aspects of data structures, such as the size, the sum or the multiset of linked list, relations of the data at linearly ordered or successive positions).

Classical abstract interpretation [32] and its applications such as automated assertion discovery [33, 34, 35, 36] mainly focus on finding numerical program properties. Our work is complementary to these advances as our focus is more on refining specifications for heap-manipulating programs. Meanwhile, we can utilise such works as our pure solver, for example the disjunction inference [10]. Semi-automatic approaches [37, 38] are also used to infer numerical constraints for given type templates in functional programs, where data structures are mostly immutable.

On the verification side, Smallfoot [39] is the first verification system based on separation logic. The Hip/Sleek verification system [1, 2] 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 [40] transforms constraints in the pointer assertion logic (PAL) into monadic second-order logic (MSO) and discharge them with MONA.JML [41] uses model/ghost fields and model methods to specify/model Java program properties. Hob [42] is a modular program verification tool for shape properties. Based on Hob, Jahob [43, 42] takes Java as its target language and allows more general specification language. Havoc [44] is another verification tool for the C language about heap-allocated data structures, using a novel reachability predicate. Dafny [45] translates programs into Boogie 2 [46] and conducts modular verification using dynamic frames. There is another recent work on refining specifications via counterexample-guided abstraction refinement [47] which is goal-driven and incrementally improves the performance 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 them by discovering missing constraints for numerical and bag properties, aiming at full functional correctness for pointer-based data structures. We further augment our approach by requiring 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.

References

[1] H. H. Nguyen, W.-N. Chin, Enhancing program verification with lemmas, in: A. Gupta, S. Malik (Eds.), CAV, Vol. 5123 of Lecture Notes in Computer Science, Springer, 2008, pp. 355–369.

[2] H. H. Nguyen, C. David, S. Qin, W.-N. Chin, Automated verification of shape and size properties via separation logic, in: B. Cook, A. Podelski (Eds.), VMCAI, Vol. 4349 of Lecture Notes in Computer Science, Springer, 2007, pp. 251–266.

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

[4] D. Distefano, P. W. O'Hearn, H. Yang, A local shape analysis based on separation logic, in: H. Hermanns, J. Palsberg (Eds.), TACAS, Vol. 3920 of Lecture Notes in Computer Science, Springer, 2006, pp. 287–302.

[5] H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, P. O'Hearn, Scalable shape analysis for systems code, in: 20th CAV, Vol. 5123 of LNCS, Springer, 2008, pp. 385–398.

[6] J. C. Reynolds, Separation logic: A logic for shared mutable data structures, in: 17th IEEE Symposium on Logic in Computer Science (LICS 2002), IEEE Computer Society, 2002, pp. 55–74.

[7] S. S. Ishtiaq, P. W. O'Hearn, Bi as an assertion language for mutable data structures, in: Conference Record of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2001), 2001, pp. 14–26.

[8] J. Gustavsson, J. Svenningsson, Constraint abstractions, in: Programs as Data Objects II, Aarhus, Denmark, 2001, pp. 63–83.

[9] T. Nipkow, L. C. Paulson, M. Wenzel, Isabelle/HOL — A Proof Assistant for Higher-Order Logic, Vol. 2283 of LNCS, Springer, 2002.

[10] C. Popeea, W.-N. Chin, Inferring disjunctive postconditions, in: M. Okada, I. Satoh (Eds.), ASIAN, Vol. 4435 of Lecture Notes in Computer Science, Springer, 2006, pp. 331–345.

[11] T.-H. Pham, M.-T. Trinh, A.-H. Truong, W.-N. Chin, Fixbag: A fixpoint calculator for quantified bag constraints, in: G. Gopalakrishnan, S. Qadeer (Eds.), CAV, Vol. 6806 of Lecture Notes in Computer Science, Springer, 2011, pp. 656–662.

[12] W.-N. Chin, C. David, H. H. Nguyen, S. Qin, Automated verification of shape, size and bag properties via user-defined predicates in separation logic, Science of Computer Programming 77 (2012) 1006–1036. doi:10.1016/j.scico.2010.07.004.

[13] W.-N. Chin, C. David, H. H. Nguyen, S. Qin, Automated verification of shape, size and bag properties, in: 12th International Conference on Engineering of Complex Computer Systems (ICECCS 2007), IEEE Computer Society, 2007, pp. 307–320.

[14] M. Rubio-Sánchez, J. Urquiza-Fuentes, C. Pareja-Flores, A gentle introduction to mutual recursion, in: ITiCSE '08: Proceedings of the 13th annual conference on Innovation and technology in computer science education, ACM, New York, NY, USA, 2008, pp. 235–239.

[15] R. Barry, FreeRTOS reference manual: API functions and configuration options, Real Time Engineers Ltd. http://www.freertos.org/, 2009.
[16] A. Gotsman, J. Berdine, B. Cook, Interprocedural shape analysis with separated heap abstractions, in: K. Yi (Ed.), SAS, Vol. 4134 of Lecture Notes in Computer Science, Springer, 2006, pp. 240–260.
[17] J. Berdine, C. Calcagno, B. Cook, D. Distefano, P. W. O'Hearn, T. Wies, H. Yang, Shape analysis for composite data structures, in: W. Damm, H. Hermanns (Eds.), CAV, Vol. 4590 of Lecture Notes in Computer Science, Springer, 2007, pp. 178–192.
[18] D. Distefano, Attacking large industrial code with bi-abductive inference, in: FMICS, 2009, pp. 1–8.
[19] S. Magill, J. Berdine, E. M. Clarke, B. Cook, Arithmetic strengthening for shape analysis, in: H. R. Nielson, G. Filé (Eds.), SAS, Vol. 4634 of Lecture Notes in Computer Science, Springer, 2007, pp. 419–436.
[20] B. Cook, A. Gupta, S. Magill, A. Rybalchenko, J. Simsa, S. Singh, V. Vafeiadis, Finding heap-bounds for hardware synthesis, in: FMCAD, 2009, pp. 205–212.
[21] S. Magill, M.-H. Tsai, P. Lee, Y.-K. Tsay, Automatic numeric abstractions for heap-manipulating programs, in: M. V. Hermenegildo, J. Palsberg (Eds.), POPL, ACM, 2010, pp. 211–222.
[22] S. Gulwani, T. Lev-Ami, M. Sagiv, A combination framework for tracking partition sizes, in: Z. Shao, B. C. Pierce (Eds.), POPL, ACM, 2009, pp. 239–251.
[23] B.-Y. E. Chang, X. Rival, G. C. Necula, Shape analysis with structural invariant checkers, in: Static Analysis Symposium 2007 (SAS'07), Denmark, 2007, pp. 384–401.
[24] B.-Y. E. Chang, X. Rival, Relational inductive shape analysis, in: G. C. Necula, P. Wadler (Eds.), POPL, ACM, 2008, pp. 247–260.
[25] S. Qin, G. He, C. Luo, W.-N. Chin, Loop invariant synthesis in a combined domain, in: Proceedings of the 12th International Conference on Formal engineering methods and software engineering, ICFEM'10, Springer-Verlag, Berlin, Heidelberg, 2010, pp. 468–484.
[26] B. Hackett, R. Rugina, Region-based shape analysis with tracked locations, in: J. Palsberg, M. Abadi (Eds.), POPL, ACM, 2005, pp. 310–323.
[27] S. Sagiv, T. W. Reps, R. Wilhelm, Parametric shape analysis via 3-valued logic, ACM Transactions on Programming Languages and Systems 24 (3) (2002) 217–298.
[28] G. Yorsh, E. Rabinovich, M. Sagiv, A. Meyer, A. Bouajjani, A logic of reachable patterns in linked data-structures, in: FOSSACS, 2006, pp. 94–110.
[29] B. Guo, N. Vachharajani, D. I. August, Shape analysis with inductive recursion synthesis, in: J. Ferrante, K. S. McKinley (Eds.), PLDI, ACM, 2007, pp. 256–265.
[30] V. Kuncak, P. Lam, M. C. Rinard, Role analysis, in: Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2002), 2002, pp. 17–32.
[31] A. Bouajjani, C. Dragoi, C. Enea, A. Rezine, M. Sighireanu, Invariant synthesis for programs manipulating lists with unbounded data, in: 22nd International Conference on Computer Aided Verification, CAV 2010, Springer-Verlag, 2010, pp. 72–88.
[32] P. Cousot, R. Cousot, Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints, in: Conference Record of the Fourth ACM Symposium on Principles of Programming Languages (POPL 1977), ACM, 1977, pp. 238–252.
[33] P. Cousot, R. Cousot, On abstraction in software verification, in: Proceedings of the 14th International Conference on Computer Aided Verification, Springer-Verlag, London, UK, 2002, pp. 37–56.
[34] A. Gupta, R. Majumdar, A. Rybalchenko, From tests to proofs, in: TACAS '09: Proceedings of the 15th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, Springer-Verlag, Berlin, Heidelberg, 2009, pp. 262–276.
[35] A. Gupta, A. Rybalchenko, Invgen: An efficient invariant generator, in: A. Bouajjani, O. Maler (Eds.), CAV, Vol. 5643 of Lecture Notes in Computer Science, Springer, 2009, pp. 634–640.
[36] S. Srivastava, S. Gulwani, Program verification using templates over predicate abstraction, in: PLDI, 2009, pp. 223–234.
[37] P. M. Rondon, M. Kawaguchi, R. Jhala, Liquid types, in: R. Gupta, S. P. Amarasinghe (Eds.), PLDI, ACM, 2008, pp. 159–169.
[38] M. Kawaguchi, P. M. Rondon, R. Jhala, Type-based data structure verification, in: M. Hind, A. Diwan (Eds.), PLDI, ACM, 2009, pp. 304–315.
[39] J. Berdine, C. Calcagno, P. W. O'Hearn, Smallfoot: Modular automatic assertion checking with separation logic, in: F. S. de Boer, M. M. Bonsangue, S. Graf, W. P. de Roever (Eds.), FMCO, Vol. 4111 of Lecture Notes in Computer Science, Springer, 2005, pp. 115–137.
[40] A. Möller, M. I. Schwartzbach, The pointer assertion logic engine, ACM SIGPLAN Notices 36 (5) (2001) 221–231.
[41] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, E. Poll, An overview of JML tools and applications, International Journal on Software Tools for Technlogy Transfer 7 (3) (2005) 212–232.
[42] T. Wies, V. Kuncak, K. Zee, A. Podelski, M. C. Rinard, On verifying complex properties using symbolic shape analysis, CoRR abs/cs/0609104.
[43] V. Kuncak, Modular data structure verification, Ph.D. thesis, EECS Department, Massachusetts Institute of Technology (February 2007).
[44] S. Chatterjee, S. K. Lahiri, S. Qadeer, Z. Rakamaric, A reachability predicate for analyzing low-level software, in: TACAS, 2007, pp. 19–33.
[45] K. R. M. Leino, Dafny: An automatic program verifier for functional correctness, in: E. M. Clarke, A. Voronkov (Eds.), LPAR (Dakar), Vol. 6355 of Lecture Notes in Computer Science, Springer, 2010, pp. 348–370.

[46] K. R. M. Leino, P. Rümmer, A polymorphic intermediate verification language: Design and logical encoding, in: J. Esparza, R. Majumdar (Eds.), TACAS, Vol. 6015 of Lecture Notes in Computer Science, Springer, 2010, pp. 312–327.
[47] M. Taghdiri, Automating modular verification by refining specifications, Ph.D. thesis, EECS Department, Massachusetts Institute of Technology (Feburary 2008).

Appendix A. Shape Predicate Definitions

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

$$\begin{aligned}
\mathtt{ls}(\mathtt{root},\mathtt{p},\mathtt{n}) \equiv & (\mathtt{root}{=}\mathtt{p} \wedge \mathtt{n}{=}0) \vee (\mathtt{root}{\mapsto}\mathtt{node}(_,\mathtt{q}){*}\mathtt{ls}(\mathtt{q},\mathtt{p},\mathtt{m}) \wedge \mathtt{n}{=}\mathtt{m}{+}1)\\
\mathtt{dll}(\mathtt{root},\mathtt{p},\mathtt{n}) \equiv & (\mathtt{root}{=}\mathtt{p} \wedge \mathtt{n}{=}0) \vee\\
& (\mathtt{root}{\mapsto}\mathtt{node2}(\mathtt{v},\mathtt{p},\mathtt{q}){*}\mathtt{dll}(\mathtt{q},\mathtt{root},\mathtt{n_1}) \wedge \mathtt{n}{=}\mathtt{n_1}{+}1)\\
\mathtt{dllB}(\mathtt{root},\mathtt{p},\mathtt{S}) \equiv & (\mathtt{root}{=}\mathtt{p} \wedge \mathtt{S}{=}\emptyset) \vee\\
& (\mathtt{root}{\mapsto}\mathtt{node2}(\mathtt{v},\mathtt{p},\mathtt{q}){*}\mathtt{dllB}(\mathtt{q},\mathtt{root},\mathtt{S_1}) \wedge \mathtt{S}{=}\mathtt{S_1}{\sqcup}\{\mathtt{v}\})\\
\mathtt{sll}(\mathtt{root},\mathtt{n},\mathtt{mn},\mathtt{mx}) \equiv & (\mathtt{root}{=}\mathtt{null} \wedge \mathtt{n}{=}0 \wedge \mathtt{mn}{=}\mathtt{mx}) \vee\\
& (\mathtt{root}{\mapsto}\mathtt{node}(\mathtt{v},\mathtt{q}){*}\mathtt{sll}(\mathtt{q},\mathtt{n_1},\mathtt{k},\mathtt{mx}) \wedge \mathtt{n}{=}\mathtt{n_1}{+}1 \wedge \mathtt{mn}{\leq}\mathtt{k})\\
\mathtt{sllB2}(\mathtt{root},\mathtt{S}) \equiv & (\mathtt{root}{=}\mathtt{null} \wedge \mathtt{S}{=}\emptyset) \vee\\
& (\mathtt{root}{\mapsto}\mathtt{node}(\mathtt{v},\mathtt{q}){*}\mathtt{sllB2}(\mathtt{q},\mathtt{S_1}) \wedge \mathtt{S}{=}\{\mathtt{v}\}{\sqcup}\mathtt{S_1} \wedge (\forall \mathtt{x}{\in}\mathtt{S_1}{\cdot}\mathtt{v}{\geq}\mathtt{x}))\\
\mathtt{slsB}(\mathtt{root},\mathtt{p},\mathtt{S}) \equiv & (\mathtt{root}{=}\mathtt{p} \wedge \mathtt{S}{=}\emptyset) \vee\\
& (\mathtt{root}{\mapsto}\mathtt{node}(\mathtt{v},\mathtt{q}){*}\mathtt{slsB}(\mathtt{q},\mathtt{p},\mathtt{S_1}) \wedge \mathtt{S}{=}\{\mathtt{v}\}{\sqcup}\mathtt{S_1} \wedge (\forall \mathtt{x}{\in}\mathtt{S_1}{\cdot}\mathtt{v}{\leq}\mathtt{x}))\\
\mathtt{bt}(\mathtt{root},\mathtt{S},\mathtt{h}) \equiv & (\mathtt{root}{=}\mathtt{null} \wedge \mathtt{S}{=}\emptyset \wedge \mathtt{h}{=}0) \vee\\
& (\mathtt{root}{\mapsto}\mathtt{node2}(\mathtt{v},\mathtt{p},\mathtt{q}){*}\mathtt{bt}(\mathtt{p},\mathtt{S_p},\mathtt{h_p}){*}\mathtt{bt}(\mathtt{q},\mathtt{S_q},\mathtt{h_q}) \wedge \mathtt{S}{=}\mathtt{S_p}{\sqcup}\mathtt{S_q} \wedge \mathtt{h}{=}1{+}\mathsf{max}(\mathtt{h_p},\mathtt{h_q}))\\
\mathtt{bst}(\mathtt{root},\mathtt{mn},\mathtt{mx}) \equiv & (\mathtt{root}{=}\mathtt{null} \wedge \mathtt{mn}{=}\mathtt{mx}) \vee\\
& (\mathtt{root}{\mapsto}\mathtt{node2}(\mathtt{v},\mathtt{p},\mathtt{q}){*}\mathtt{bst}(\mathtt{p},\mathtt{mn},\mathtt{mx_l}){*}\mathtt{bst}(\mathtt{q},\mathtt{mn_r},\mathtt{mx}) \wedge \mathtt{mx_l}{\leq}\mathtt{v}{\leq}\mathtt{mn_r})\\
\mathtt{bstB}(\mathtt{root},\mathtt{S}) \equiv & (\mathtt{root}{=}\mathtt{null} \wedge \mathtt{S}{=}\emptyset) \vee\\
& (\mathtt{root}{\mapsto}\mathtt{node2}(\mathtt{v},\mathtt{p},\mathtt{q}){*}\mathtt{bstB}(\mathtt{p},\mathtt{S_p}){*}\mathtt{bstB}(\mathtt{q},\mathtt{S_q}) \wedge\\
& \quad \mathtt{S}{=}\{\mathtt{v}\}{\sqcup}\mathtt{S_p}{\sqcup}\mathtt{S_q} \wedge \forall \mathtt{v_p}{\in}\mathtt{S_p}, \mathtt{v_q}{\in}\mathtt{S_q} \cdot \mathtt{v_p}{\leq}\mathtt{v}{\leq}\mathtt{v_q})\\
\mathtt{avl}(\mathtt{root},\mathtt{S},\mathtt{h}) \equiv & (\mathtt{root}{=}\mathtt{null} \wedge \mathtt{S}{=}\emptyset \wedge \mathtt{h}{=}0) \vee\\
& (\mathtt{root}{\mapsto}\mathtt{node2}(\mathtt{v},\mathtt{p},\mathtt{q}){*}\mathtt{avl}(\mathtt{p},\mathtt{S_p},\mathtt{h_p}){*}\mathtt{avl}(\mathtt{q},\mathtt{S_q},\mathtt{h_q}) \wedge\\
& \quad \mathtt{S}{=}\mathtt{S_p}{\sqcup}\mathtt{S_q} \wedge \mathtt{h}{=}1{+}\mathsf{max}(\mathtt{h_p},\mathtt{h_q}) \wedge -1{\leq}\mathtt{h_p}{-}\mathtt{h_q}{\leq}1)\\
\mathtt{rbt}(\mathtt{root},\mathtt{S},\mathtt{c},\mathtt{h}) \equiv & (\mathtt{root}{=}\mathtt{null} \wedge \mathtt{S}{=}\emptyset \wedge \mathtt{c}{=}0 \wedge \mathtt{h}{=}0) \vee\\
& (\mathtt{root}{\mapsto}\mathtt{node2}(\mathtt{v},\mathtt{p},\mathtt{q}){*}\mathtt{rbt}(\mathtt{p},\mathtt{S_p},\mathtt{c_p},\mathtt{h_p}){*}\mathtt{rbt}(\mathtt{q},\mathtt{S_q},\mathtt{c_q},\mathtt{h_q}) \wedge\\
& \quad \mathtt{S}{=}\mathtt{S_p}{\sqcup}\mathtt{S_q} \wedge ((\mathtt{c}{=}0 \wedge \mathtt{h}{=}\mathtt{h_p}{+}1 \wedge \mathtt{h_p}{=}\mathtt{h_q}) \vee (\mathtt{c}{=}1 \wedge \mathtt{c_p}{=}\mathtt{c_q}{=}0 \wedge \mathtt{h}{=}\mathtt{h_p}{=}\mathtt{h_q}))
\end{aligned}$$

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