Loop Invariant Synthesis in a Combined Domain

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

[1] Teesside University, Middlesbrough, TS1 3BA, UK
[2] Durham University, Durham, DH1 3LE, UK
[3] National University of Singapore

Abstract. Automated verification of memory safety and functional correctness for heap-manipulating programs has been a challenging task, especially when dealing with complex data structures with strong invariants involving both shape and numerical properties. Existing verification systems usually rely on users to supply annotations, which can be tedious and error-prone and can significantly restrict the scalability of the verification system. In this paper, we reduce the need of user annotations by automatically inferring loop invariants over an abstract domain with both separation and numerical information. Our loop invariant synthesis is conducted automatically by a fixpoint iteration process, equipped with newly designed abstraction mechanism, and join and widening operators. Initial experiments have confirmed that we can synthesise loop invariants with non-trivial constraints.

1 Introduction

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

Since the emergence of separation logic [14, 25], dramatic advances have been made in automated software verification, e.g. the Smallfoot tool [1] for the verification on pointer safety (i.e. shape properties asserting that pointers cannot go wrong), the verification on termination [2], the verification for object-oriented programs [6, 22], and Dafny [18] and Hip/Sleek [5, 20, 21] for more general properties (both structural and numerical ones) for heap-manipulating programs.

These verification systems generally require users to provide specifications for each method as well as invariants for each loop, which is both tedious and error-prone. This also affects their scalability, as there can be many methods in a program and each method may still contain several while loops.

To conquer this problem, separation logic based shape analysis techniques are brought in, e.g., the SpaceInvader tool [3, 9, 27]. As a further step of Smallfoot, it automatically infers method specifications and loop invariants for pointer

⋆ Now with Citigroup Inc.

safety in the shape domain. Other works such as THOR [19] incorporate simple numerical information into the shape domain to allow automated synthesis of properties like list length. Their success proves the necessity and feasibility for shape analysis to help automate the verification process.

However, the prior analyses focus mainly on relatively simple properties, such as pointer safety for lists and list length information. It is difficult to apply them in the presence of more sophisticated program properties, such as:

- More flexible user-defined data structures, such as trees;
- Relational numerical properties, like sortedness and binary search property.

These properties can be part of the full functional correctness of heap-manipulating programs. The (aforementioned) Hip/Sleek tool aims to verify such properties and it allows users to define their own shape predicates to express their desired level of correctness.

In this paper, we make the first stride to improve the level of automation for Hip/Sleek-like verification systems by discovering loop invariants automatically over the combined shape and numerical domain. This proves to be a challenging problem especially since we aim towards full functional correctness that Hip/Sleek targets at. Our approach is based on the framework of abstract interpretation [7] with fixpoint computation. It makes the following contributions in summary:

- We propose a loop invariant synthesis with novel operations for abstraction, join and widening over a combined shape and numerical domain.
- We demonstrate that our analysis is sound w.r.t. concrete program semantics and always terminates.
- We have integrated our solution with Hip/Sleek and conducted some initial experiments. The experimental results confirm the viability of our solution and show that we can effectively eliminate the need for user-provision of loop invariants which were previously necessary in verification.

We shall next illustrate our approach informally via an example before presenting the formal details.

2 The Approach

Before giving an illustrative example for the analysis, we will first introduce our specification mechanism which follows the Hip/Sleek system.

2.1 Separation Logic and User-defined Predicates

Separation logic [14, 25] extends Hoare logic to support reasoning about shared mutable data structures. It adds two more connectives to classical logic: separation conjunction $*$ and spatial implication $-\!\!*$. The formula $\mathtt{p}_1 * \mathtt{p}_2$ asserts that two heaps described by the formulae $\mathtt{p}_1$ and $\mathtt{p}_2$ are domain-disjoint, while $\mathtt{p}_1 -\!\!* \mathtt{p}_2$ asserts that if the current heap is extended with a disjoint heap described by

the formula $\mathtt{p}_1$, then the formula $\mathtt{p}_2$ holds in the extended heap. In this paper we only use separation conjunction.

Similar to the HIP/SLEEK system, we allow user-defined inductive predicates to specify both separation and numerical properties. For example, with a data structure definition for a node in a list `data node { int val; node next; }`, we can define a predicate for a list as

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

The parameter `root` for the predicate `ll` is the root pointer referring to the list. Its length is denoted by `n`. A uniform notation $\mathtt{p}{::}\mathtt{c}\langle\mathtt{v}_1,..,\mathtt{v}_k\rangle$ is used for either a singleton heap or a predicate. If `c` is a data node with fields $\mathtt{f}_1,..,\mathtt{f}_k$, the notation represents a singleton heap, $\mathtt{p}{\mapsto}\mathtt{c}[\mathtt{f}_1{:}\mathtt{v}_1,..,\mathtt{f}_k{:}\mathtt{v}_k]$, e.g. the $\mathtt{root}{::}\mathtt{node}\langle\mathtt{v},\mathtt{q}\rangle$ above. If `c` is a predicate name, then the data structure pointed to by `p` has the shape `c` with parameters $\mathtt{v}_1,..,\mathtt{v}_k$, e.g., the $\mathtt{q}{::}\mathtt{ll}\langle\mathtt{m}\rangle$ above.

We can also define a list segment as follows:

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

where we use the following shortened notation: (i) default `root` parameter in LHS may be omitted, (ii) unbound variables, such as `q` and `m`, are implicitly existentially quantified, and (iii) the underscore _ denotes an existentially quantified anonymous variable.

If the user wants to verify a sorting algorithm, they can incorporate sortedness property into the above predicates as follows:

$$\begin{array}{l}\mathtt{sll}\langle\mathtt{n},\mathtt{mn},\mathtt{mx}\rangle \equiv (\mathtt{root}{::}\mathtt{node}\langle\mathtt{mn},\mathtt{null}\rangle\wedge\mathtt{n}{=}1\wedge\mathtt{mn}{=}\mathtt{mx})\vee\\ \qquad(\mathtt{root}{::}\mathtt{node}\langle\mathtt{mn},\mathtt{q}\rangle * \mathtt{q}{::}\mathtt{sll}\langle\mathtt{n}_1,\mathtt{k},\mathtt{mx}\rangle\wedge\mathtt{mn}{\le}\mathtt{k}\wedge\mathtt{n}{=}\mathtt{n}_1{+}1)\\ \mathtt{sls}\langle\mathtt{p},\mathtt{n},\mathtt{mn},\mathtt{mx}\rangle \equiv (\mathtt{root}{::}\mathtt{node}\langle\mathtt{mn},\mathtt{p}\rangle\wedge\mathtt{n}{=}1\wedge\mathtt{mn}{=}\mathtt{mx})\vee\\ \qquad(\mathtt{root}{::}\mathtt{node}\langle\mathtt{mn},\mathtt{q}\rangle * \mathtt{q}{::}\mathtt{sls}\langle\mathtt{p},\mathtt{n}_1,\mathtt{k},\mathtt{mx}\rangle\wedge\mathtt{mn}{\le}\mathtt{k}\wedge\mathtt{n}{=}\mathtt{n}_1{+}1)\end{array}$$

where `mn` and `mx` denote resp. the min and max values stored in the sorted list. Such user-supplied predicates can be used to specify loop invariants and method pre/post-specifications.

2.2 Illustrative Example

We now illustrate via an example our loop invariant synthesis process. The method `ins_sort` (Figure 1) sorts a linked list with the insertion sort algorithm. It is implemented with two nested while loops. The outer loop traverses the whole list `x`, takes out each node from it (line 7), and inserts that node into another already sorted list `r` (which is empty initially before the sorting). This insertion process makes use of the inner while loop in lines 9-11 to look for a proper position in the already sorted list for the new node to be inserted. The actual insertion takes place at lines 12-14.

To verify this program, we need to synthesise appropriate loop invariants for both while loops. Our analysis follows a standard fixpoint iteration process. It

```
0 data node { int val;                9   while (srt != null &&
              node next; }                          srt.val <= v) {
1 node ins_sort(node x)              10     prv=srt; srt=srt.next;
2  requires x::ll⟨n⟩                 11   }
3  ensures res::sll⟨n, mn, mx⟩       12   cur.next=srt;
4 {int v;                            13   if (prv != null) prv.next=cur;
5  node r,cur,srt,prv=null;          14   else r=cur;
6  while (x != null) {               15  }
7   cur=x; x=x.next; v=cur.val;      16  return r;
8   srt=r; prv=null;                 17 }
```

Fig. 1. Insertion sort for linked list.

starts with the (abstract) program state immediately before the while loop (i.e., the initial state) and symbolically executes the loop body for several iterations, until the obtained states converge to a fixpoint, which is the loop invariant.[1] At the start of each iteration, the obtained state from the previous iteration is joined with the initial state. In addition to this join operator, we have also defined an abstraction function and a widening operator both of which will help the fixpoint iteration to converge. The join and widening operators are specifically designed to handle both shape and numerical information.

As for our example, due to the presence of nested loops, each iteration of the analysis for the outer loop actually infers a loop invariant for the inner loop. We shall now illustrate how we synthesise a loop invariant for the inner loop.

Suppose that in one iteration for the outer loop, the state at line 9 becomes

$$\mathtt{r}::\mathtt{sll}\langle \mathtt{n_r}, \mathtt{a}, \mathtt{b}\rangle * \mathtt{cur}::\mathtt{node}\langle \mathtt{v}, \mathtt{x}\rangle * \mathtt{x}::\mathtt{ll}\langle \mathtt{n_x}\rangle \wedge \mathtt{srt}{=}\mathtt{r} \wedge \mathtt{prv}{=}\mathtt{null} \wedge \mathtt{n_r}{+}\mathtt{n_x}{+}1{=}\mathtt{n}$$

Note that since the inner loop does not mutate the heap part referred to by `cur` and `x` (i.e., $\mathtt{cur}::\mathtt{node}\langle \mathtt{v}, \mathtt{x}\rangle * \mathtt{x}::\mathtt{ll}\langle \mathtt{n_x}\rangle$), we can ignore it during the invariant synthesis and add it back to the program state using the frame rule of separation logic [25]. Therefore, the initial state for loop invariant synthesis becomes

$$\mathtt{r}::\mathtt{sll}\langle \mathtt{n_r}, \mathtt{a}, \mathtt{b}\rangle \wedge \mathtt{srt}{=}\mathtt{r} \wedge \mathtt{prv}{=}\mathtt{null} \wedge \mathtt{n_r}{+}\mathtt{n_x}{+}1{=}\mathtt{n} \tag{1}$$

From this state, symbolically executing the loop body once yields the state:

$$\begin{array}{c}\mathtt{r}::\mathtt{node}\langle \mathtt{a}, \mathtt{srt}\rangle * \mathtt{srt}::\mathtt{sll}\langle \mathtt{n_s}, \mathtt{c_1}, \mathtt{b}\rangle \wedge \mathtt{prv}{=}\mathtt{r} \wedge \\ \mathtt{a}{\leq}\mathtt{c_1} \wedge \mathtt{a}{\leq}\mathtt{v} \wedge \mathtt{n_r}{+}1{=}\mathtt{n}{-}\mathtt{n_x} \wedge \mathtt{n_s}{+}1{=}\mathtt{n_r}\end{array} \tag{2}$$

which says that pointer `srt` moves towards the tail of the list for one node. Then we join it with the initial state (1) to obtain

$$\begin{array}{l}(\mathtt{r}::\mathtt{sll}\langle \mathtt{n_r}, \mathtt{a}, \mathtt{b}\rangle \wedge \mathtt{srt}{=}\mathtt{r} \wedge \mathtt{prv}{=}\mathtt{null} \wedge \mathtt{n_r}{+}\mathtt{n_x}{+}1{=}\mathtt{n}) \vee \\ (\mathtt{r}::\mathtt{node}\langle \mathtt{a}, \mathtt{srt}\rangle * \mathtt{srt}::\mathtt{sll}\langle \mathtt{n_s}, \mathtt{c_1}, \mathtt{b}\rangle \wedge \\ \qquad \mathtt{prv}{=}\mathtt{r} \wedge \mathtt{a}{\leq}\mathtt{c_1} \wedge \mathtt{a}{\leq}\mathtt{v} \wedge \mathtt{n_r}{+}1{=}\mathtt{n}{-}\mathtt{n_x} \wedge \mathtt{n_s}{+}1{=}\mathtt{n_r})\end{array} \tag{3}$$

The second iteration over the loop body starts with (3) and exhibits (also) the case that `srt` runs two nodes towards tail, while `prv` goes one node. Its result is then joined with pre-state (1) to become the current state:

[1] The fixpoint iteration converges if one more iteration still yields the same result.

$$(3) \ \vee \ \mathtt{r}{::}\mathtt{node}\langle \mathtt{a}, \mathtt{prv}\rangle * \mathtt{prv}{::}\mathtt{node}\langle \mathtt{c}_1, \mathtt{srt}\rangle * \mathtt{srt}{::}\mathtt{sll}\langle \mathtt{n}_\mathtt{s}, \mathtt{c}_2, \mathtt{b}\rangle \wedge \\ \mathtt{a}{\leq}\mathtt{c}_1{\leq}\mathtt{c}_2 \wedge \mathtt{c}_1{\leq}\mathtt{v} \wedge \mathtt{n}_\mathtt{r}{+}1{=}\mathtt{n}{-}\mathtt{n}_\mathtt{x} \wedge \mathtt{n}_\mathtt{s}{+}2{=}\mathtt{n}_\mathtt{r} \quad (4)$$

Executing the loop body a third time returns a post-state where three nodes are passed by `srt`, and two by `prv`, as below:

$$(4) \ \vee \ \mathtt{r}{::}\mathtt{node}\langle \mathtt{a}, \mathtt{r}_0\rangle * \mathtt{r}_0{::}\mathtt{node}\langle \mathtt{c}_1, \mathtt{prv}\rangle * \mathtt{prv}{::}\mathtt{node}\langle \mathtt{c}_2, \mathtt{srt}\rangle * \\ \mathtt{srt}{::}\mathtt{sll}\langle \mathtt{n}_\mathtt{s}, \mathtt{c}_3, \mathtt{b}\rangle \wedge \mathtt{a}{\leq}\mathtt{c}_1{\leq}\mathtt{c}_2{\leq}\mathtt{c}_3 \wedge \mathtt{c}_2{\leq}\mathtt{v} \wedge \mathtt{n}_\mathtt{r}{+}1{=}\mathtt{n}{-}\mathtt{n}_\mathtt{x} \wedge \mathtt{n}_\mathtt{s}{+}3{=}\mathtt{n}_\mathtt{r}$$

where we have an auxiliary logical variable $\mathtt{r}_0$. Following this trend, it is predictable that every iteration hereafter will introduce an additional logical variable (referring to a list node). If we indulge in such increase in the subsequent iterations, the analysis will never terminate. Our abstraction process prevents this from happening by eliminating such logical variables as follows:

$$(4) \ \vee \ \mathtt{r}{::}\mathtt{sls}\langle \mathtt{prv}, \mathtt{n}_1, \mathtt{a}, \mathtt{c}_1\rangle * \mathtt{prv}{::}\mathtt{node}\langle \mathtt{c}_2, \mathtt{srt}\rangle * \mathtt{srt}{::}\mathtt{sll}\langle \mathtt{n}_\mathtt{s}, \mathtt{c}_3, \mathtt{b}\rangle \wedge \\ \mathtt{a}{\leq}\mathtt{c}_1{\leq}\mathtt{c}_2{\leq}\mathtt{c}_3 \wedge \mathtt{c}_2{\leq}\mathtt{v} \wedge \mathtt{n}_\mathtt{r}{+}1{=}\mathtt{n}{-}\mathtt{n}_\mathtt{x} \wedge \mathtt{n}_\mathtt{s}{+}3{=}\mathtt{n}_\mathtt{r} \wedge \mathtt{n}_1{=}2$$

Note that the heap part $\mathtt{r}{::}\mathtt{node}\langle \mathtt{a}, \mathtt{r}_0\rangle * \mathtt{r}_0{::}\mathtt{node}\langle \mathtt{c}_1, \mathtt{prv}\rangle$ is abstracted as a sorted list segment $\mathtt{r}{::}\mathtt{sls}\langle \mathtt{prv}, \mathtt{n}_1, \mathtt{a}, \mathtt{c}_1\rangle$ with $\mathtt{n}_1$ denoting the length of the segment and $\mathtt{n}_1{=}2$ added into the state. This abstraction process ensures that our analysis does not allow the shape to increase infinitely.

The fourth iteration responds with a post-state where four nodes are passed by `srt`, and three by `prv`. Therefore an abstraction is performed to remove the logical pointer variables. As a simplification of the presentation, we denote σ as $\mathtt{r}{::}\mathtt{sls}\langle \mathtt{prv},\mathtt{n}_1,\mathtt{a},\mathtt{c}_1\rangle * \mathtt{prv}{::}\mathtt{node}\langle \mathtt{c}_2,\mathtt{srt}\rangle * \mathtt{srt}{::}\mathtt{sll}\langle \mathtt{n}_\mathtt{s},\mathtt{c}_3,\mathtt{b}\rangle \wedge \mathtt{a}{\leq}\mathtt{c}_1{\leq}\mathtt{c}_2{\leq}\mathtt{c}_3 \wedge \mathtt{c}_2{\leq}\mathtt{v} \wedge \mathtt{n}_\mathtt{r}{+}1{=}\mathtt{n}{-}\mathtt{n}_\mathtt{x}$, and the abstracted result (after the fourth iteration) is

$$(4) \vee (\sigma \wedge \mathtt{n}_\mathtt{s}{+}3{=}\mathtt{n}_\mathtt{r} \wedge \mathtt{n}_1{=}2) \vee (\sigma \wedge \mathtt{n}_\mathtt{s}{+}4{=}\mathtt{n}_\mathtt{r} \wedge \mathtt{n}_1{=}3)$$

for which we have an observation that the last two disjunctions share the same shape part (as in σ). This disjunction will be transferred to the numerical domain, as follows:

$$(4) \vee (\sigma \wedge (\mathtt{n}_\mathtt{s}{+}3{=}\mathtt{n}_\mathtt{r} \wedge \mathtt{n}_1{=}2 \vee \mathtt{n}_\mathtt{s}{+}4{=}\mathtt{n}_\mathtt{r} \wedge \mathtt{n}_1{=}3))$$

This simplifies the abstraction further. After that, our widening operation compares the current state with the previous one, to look for the same (numerical) constraints that both states imply, and to replace those numerical constraints in the current state with the ones discovered by widening. This operation eventually ensures termination of our analysis. As for the example, some constraints among $\mathtt{n}_\mathtt{s}$, $\mathtt{n}_\mathtt{r}$ and $\mathtt{n}_1$ can be found to make the widened post-state become:

$$(4) \vee (\sigma \wedge \mathtt{n}_\mathtt{s}{+}\mathtt{n}_1{=}\mathtt{n}_\mathtt{r}{-}1 \wedge \mathtt{n}_1{\geq}2) \quad (5)$$

One more iteration of symbolic execution will produce the same result as (5), suggesting that it is already the fixpoint (and hence the loop invariant):

$$\begin{array}{l} \mathtt{r}{::}\mathtt{sll}\langle \mathtt{n}_\mathtt{r}, \mathtt{a}, \mathtt{b}\rangle \wedge \mathtt{srt}{=}\mathtt{r} \wedge \mathtt{prv}{=}\mathtt{null} \wedge \mathtt{n}_\mathtt{r}{+}1{=}\mathtt{n}{-}\mathtt{n}_\mathtt{x} \ \vee \\ \mathtt{r}{::}\mathtt{node}\langle \mathtt{a}, \mathtt{srt}\rangle * \mathtt{srt}{::}\mathtt{sll}\langle \mathtt{n}_\mathtt{s}, \mathtt{c}_1, \mathtt{b}\rangle \wedge \mathtt{prv}{=}\mathtt{r} \ \wedge \\ \quad \mathtt{a}{\leq}\mathtt{c}_1 \wedge \mathtt{a}{\leq}\mathtt{v} \wedge \mathtt{n}_\mathtt{r}{+}1{=}\mathtt{n}{-}\mathtt{n}_\mathtt{x} \wedge \mathtt{n}_\mathtt{s}{+}1{=}\mathtt{n}_\mathtt{r} \ \vee \\ \mathtt{r}{::}\mathtt{node}\langle \mathtt{a}, \mathtt{prv}\rangle * \mathtt{prv}{::}\mathtt{node}\langle \mathtt{c}_1, \mathtt{srt}\rangle * \mathtt{srt}{::}\mathtt{sll}\langle \mathtt{n}_\mathtt{s}, \mathtt{c}_2, \mathtt{b}\rangle \wedge \\ \quad \mathtt{a}{\leq}\mathtt{c}_1{\leq}\mathtt{c}_2 \wedge \mathtt{c}_1{\leq}\mathtt{v} \wedge \mathtt{n}_\mathtt{r}{+}1{=}\mathtt{n}{-}\mathtt{n}_\mathtt{x} \wedge \mathtt{n}_\mathtt{s}{+}2{=}\mathtt{n}_\mathtt{r} \ \vee \\ \mathtt{r}{::}\mathtt{sls}\langle \mathtt{prv}, \mathtt{n}_1, \mathtt{a}, \mathtt{c}_1\rangle * \mathtt{prv}{::}\mathtt{node}\langle \mathtt{c}_2, \mathtt{srt}\rangle * \mathtt{srt}{::}\mathtt{sll}\langle \mathtt{n}_\mathtt{s}, \mathtt{c}_3, \mathtt{b}\rangle \wedge \\ \quad \mathtt{a}{\leq}\mathtt{c}_1{\leq}\mathtt{c}_2{\leq}\mathtt{c}_3 \wedge \mathtt{c}_2{\leq}\mathtt{v} \wedge \mathtt{n}_\mathtt{r}{+}1{=}\mathtt{n}{-}\mathtt{n}_\mathtt{x} \wedge \mathtt{n}_\mathtt{s}{+}\mathtt{n}_1{=}\mathtt{n}_\mathtt{r}{-}1 \wedge \mathtt{n}_1{\geq}2 \end{array}$$

Note that although it is possible to further join the third disjunctive branch with the fourth, our analysis does not do so as it tries to keep the result as precise as possible by eliminating only auxiliary pointer variables.

With the frame part $\texttt{cur}{::}\texttt{node}\langle \texttt{v}, \texttt{x}\rangle * \texttt{x}{::}\texttt{ll}\langle \texttt{n}_\texttt{x}\rangle$ added back, the analysis for the outer loop continues. Eventually, the following loop invariant is discovered for the outer loop:

$$(\texttt{x}{::}\texttt{ll}\langle \texttt{n}_\texttt{x}\rangle \wedge \texttt{r}{=}\texttt{null} \wedge \texttt{n}_\texttt{x}{=}\texttt{n}) \vee (\texttt{r}{::}\texttt{node}\langle \texttt{a}, \texttt{null}\rangle * \texttt{x}{::}\texttt{ll}\langle \texttt{n}_\texttt{x}\rangle \wedge \texttt{n}{=}\texttt{n}_\texttt{x}{+}1) \vee (\texttt{r}{::}\texttt{sll}\langle \texttt{n}_\texttt{r}, \texttt{a}, \texttt{b}\rangle * \texttt{x}{::}\texttt{ll}\langle \texttt{n}_\texttt{x}\rangle \wedge \texttt{n}{=}\texttt{n}_\texttt{x}{+}\texttt{n}_\texttt{r} \wedge \texttt{n}_\texttt{r}{\geq}2)$$

which allows us to verify the whole method successfully using e.g. HIP/SLEEK.

3 Language and Abstract Domain

To simplify presentation, we focus on a strongly-typed C-like imperative language in Figure 2. The program *Prog* written in this language consists of declarations *tdecl*, which can either be data type declarations *datat* (e.g. `node` in Section 2), or predicate definitions *spred* (e.g. $\texttt{ll}, \texttt{ls}, \texttt{sll}, \texttt{sls}$ in Section 2.1), as well as method declarations *meth*. The definitions for *spred* and *mspec* are given later in Figure 3. Without loss of expressiveness, we use an expression-oriented language. So the body of a method (e) is an expression formed by standard commands of an imperative language. Note that d and $d[v]$ represent resp. heap-insensitive and heap sensitive commands. The language allows both call-by-value and call-by-reference method parameters (separated with a semicolon ;).

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

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

Our specification language (in Figure 3) allows (user-defined) shape predicates *spred* to specify both shape and numerical properties. Note that *spred* are constructed with disjunctive constraints Φ and numerical formulae π. We require that the predicates be well-formed [21].

A conjunctive abstract program state, σ, is composed of a heap (shape) part κ and a numerical part π, where π consists of γ and ϕ as aliasing and numerical 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 (and its set is recognised as $\mathcal{P}_{\mathsf{SH}}$). An abstract state Δ can be normalised to the Φ form.

Using entailment [21], we define a partial order over these abstract states:

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

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

Fig. 3. The Specification Language.

where R is the (computed) residue part. And we also have an induced lattice over these states as the base of fixpoint calculation for loop invariants.

The memory model of our specification formula is similar to the model given for separation logic [25], except that we have extensions to handle user-defined shape predicates and related numerical properties. In our analysis, all the variables except the program ones are logical variables. We denote a program variable's initial value as unprimed and its current value as primed [21].

4 Analysis Algorithm

Our proposed analysis algorithm is given in Figure 4.

Fixpoint Computation in Combined Domain
Input: $\mathcal{T}$, Δ_{pre}, `while` b $\{e\}$, n;
Local: $i := 0$; $\Delta_i :=$ `false`; $\Delta'_i :=$ `false`;
1 **repeat**
2 $\quad i := i + 1$;
3 $\quad \Delta_i := \mathsf{widen}^{\dagger}(\Delta_{i-1}, \mathsf{join}^{\dagger}(\Delta_{pre}, \Delta'_{i-1}))$;
4 $\quad \Delta'_i := \mathsf{abs}^{\dagger}([\![e]\!]_{\mathcal{T}}(\Delta_i \wedge b))$;
5 $\quad$ **if** $\Delta'_i =$ `false` $\vee\ \mathsf{cp\_no}(\Delta'_i) > n$ **then return** fail **end if**
6 **until** $\Delta'_i = \Delta'_{i-1}$;
7 **return** Δ'_i

Fig. 4. Main analysis algorithm.

The algorithm takes four input parameters: $\mathcal{T}$ as the program environment with all the method specifications in the program (for potential method calls in loop body), Δ_{pre} as the precondition of the loop's (symbolic) execution, the while loop itself `while` b $\{e\}$, and the number of upper bound of shared logical variables we keep during the analysis n.

Our analysis is based on abstract interpretation [7] with specifically designed operations (abs, join and widen) over this combined domain.[2] At the beginning, we initialise the iteration variable (i) and two states to begin with (Δ_i and Δ'_i). The `false`'s here as initial values denote the top element of our defined lattice

[2] Note that our analysis uses lifted versions of these operations (indicated by †), which will be explained in more details in Section 4.2.

as well as our starting point of the fixpoint iteration. Among the two states here, the unprimed version Δ_i denotes the initial state before the i^{th} execution of the loop body, and the primed one Δ'_i represents the result state after. Each iteration starts at line 1. Firstly we join together the precondition of the loop with the result state Δ'_{i-1} obtained in the previous iteration, and widen it against the initial state Δ_{i-1} of the previous iteration (line 3). Then we symbolically execute the loop body with the abstract semantics in Section 4.1 (line 4), and apply the abstraction operation to the obtained abstract state. If the symbolic execution cannot continue due to a program bug, or if we find our abstraction cannot keep the number of shared logical variables/cutpoints (counted by $\mathsf{cp\_no}$) within a specified bound (n), then a failure is reported (line 5). Otherwise we judge whether a fixpoint is already reached by comparing the current abstract state with the previous one (line 6). The fixpoint Δ'_i is returned as the loop invariant.

We will elaborate the key techniques of our analysis in what follows: the abstract semantics, the abstraction function, and the join and widening operators.

4.1 Abstract Semantics

The abstract semantics is used to execute the loop body symbolically to obtain its post-state during the loop invariant synthesis. Its type is defined as

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

where $\mathsf{AllSpec}$ contains all the specifications of all methods (extracted from the program *Prog*). For some expression e, given its precondition, the semantics will calculate the postcondition.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Based on the transition functions above, we can define the abstract semantics for a program command e as follows:

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

which form the foundation for us to analyse the loop body.

4.2 Abstraction, Join and Widening

This section describes our specifically designed abstraction, join and widening operations employed in our loop invariant synthesis process.

Abstraction function. During the symbolic execution, we may be confronted with many "concrete" shapes in postconditions of the loop body. As an example

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

$$\begin{array}{l} \mathsf{abs}(\mathtt{x}{::}\mathtt{node}\langle\_,\mathtt{z}_0\rangle * \mathtt{z}_0{::}\mathtt{node}\langle\_,\mathtt{null}\rangle) \;=\; \mathtt{x}{::}\mathtt{ll}\langle\mathtt{n}\rangle \wedge \mathtt{n}{=}2 \\ \mathsf{abs}(\mathtt{x}{::}\mathtt{node}\langle\_,\mathtt{z}_0\rangle * \mathtt{y}{::}\mathtt{node}\langle\_,\mathtt{z}_0\rangle * \mathtt{z}_0{::}\mathtt{node}\langle\_,\mathtt{null}\rangle) \;=\; \text{-} \end{array} \tag{6}$$

where both $\mathtt{x}$ and $\mathtt{y}$ are program variables, and $\mathtt{z}_0$ is an existentially quantified logical variable. In the second case $\mathtt{z}_0$ is a shared cutpoint referenced by both $\mathtt{x}$ and $\mathtt{y}$, and thus the state is not changed. As illustrated, the abstraction transition function abs eliminates unimportant cutpoints (during analysis) to ensure termination. Its type is defined as follows:

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

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

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

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

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

The first two rules eliminate logical variables (x_0) by replacing them with their equivalent expressions (e). The third rule is used to eliminate any garbage (heap part led by a logical variable x_0 unreachable from the other part of the heap) that may exist in the heap. As x_0 is already unreachable from, and not usable by, the program variables, it is safe to treat it as garbage $\mathtt{true}$, for example the $\mathtt{x}_0$ in $\mathtt{x}{::}\mathtt{node}\langle\_,\mathtt{null}\rangle * \mathtt{x}_0{::}\mathtt{node}\langle\_,\mathtt{null}\rangle$ where only $\mathtt{x}$ is a program variable.

The last rule of abs plays the most significant role which intends to eliminate shape formulae led by logical variables (all variables in v_1^*). It tries to fold data nodes up to a predicate node. It confirms that c_1 is a data node definition and c_2 is a predicate. Meanwhile it also ensures that the latter is a sound abstraction of the former by entailment checking, and the logical parameters of c_1 are not reachable from other part of the heap (so that the abstraction does not lose necessary information). The function Reach is defined as follows:

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

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

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

That is, it is composed of aliases of v and variables reachable from v. As in the previous example: $\mathsf{abs}(\mathtt{x}::\mathtt{node}\langle \_, \mathtt{z_0} \rangle * \mathtt{z_0}::\mathtt{node}\langle \_, \mathtt{null} \rangle) \rightsquigarrow \mathtt{x}::\mathtt{ll}\langle \mathtt{n_0} \rangle \wedge \mathtt{n_0}{=}2$.

During the analysis, we apply the above abstraction rules (following the given order) onto the current abstract state exhaustively until it stabilises. Such convergence is confirmed because the abstract shape domain is finite due to the bounded numbers of variables and predicates, as discussed later.

Finally the lifting function is overloaded for abs to lift both its domain and range to disjunctive abstract states $\mathcal{P}_{\mathsf{SH}}$:

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

which allows it to be used in the analysis.

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

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

where the rename function prevents naming clashes among logical variables of σ_1 and σ_2, by renaming logical variables of same name in the two states with fresh names. For example it will renew $\mathtt{x_0}$'s name in both states $\exists \mathtt{x_0} \cdot \mathtt{x_0}{=}0$ and $\exists \mathtt{x_0} \cdot \mathtt{x_0}{=}1$ to make them $\exists \mathtt{x_0} \cdot \mathtt{x_0}{=}0$ and $\exists \mathtt{x_1} \cdot \mathtt{x_1}{=}1$. After this procedure it judges whether σ_2 is an abstraction of σ_1, or the other way round. If either case holds, it regards the shape of the weaker state as the shape of the joined states, and performs joining for numerical formulae with $\mathsf{join}_{\pi}(\pi_1, \pi_2)$, the convex hull operator over numerical domain [23]. Otherwise it keeps a disjunction of the two states (as it would be unsound to join their shapes together in this case). Then we lift this operator for abstract state Δ as follows:

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

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

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

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

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

where the rename function has the same effect as above. Generally this operator is analogous to join; the only difference is that we expect the second operand σ_2 is weaker than the first σ_1, such that the widening reflects the trend of such weakening from σ_1 to σ_2. In this case it applies the widening operation $\mathsf{widen}_\pi(\pi_1, \pi_2)$ over the numerical domain [23]. Therefore, based on the widening over conjunctive abstract states, we lift the operator over (disjunctive) abstract states:

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

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

Example 1 (Abstraction, join and widening). Assume we have two abstract states, $\Delta_0 = \mathtt{x}{::}\mathtt{node}\langle\_, \mathtt{x}_0\rangle * \mathtt{x}_0{::}\mathtt{node}\langle\_, \mathtt{null}\rangle$ and $\Delta_1 = \mathtt{x}{::}\mathtt{node}\langle\_, \mathtt{x}_0\rangle * \mathtt{x}_0{::}\mathtt{node}\langle\_, \mathtt{x}_1\rangle * \mathtt{x}_1{::}\mathtt{node}\langle\_, \mathtt{null}\rangle$. We would like to discover a sound approximation for both states. Firstly we perform abstractions on both to obtain two abstract states, say, $\Delta_0' = \mathtt{x}{::}\mathtt{ll}\langle\mathtt{n}_0\rangle \wedge \mathtt{n}_0{=}2$ and $\Delta_1' = \mathtt{x}{::}\mathtt{ll}\langle\mathtt{n}_0\rangle \wedge \mathtt{n}_0{=}3$. Then these two are joined together according to shape similarity to be $\Delta_1'' = \mathtt{x}{::}\mathtt{ll}\langle\mathtt{n}_0\rangle \wedge (\mathtt{n}_0{=}2 \vee \mathtt{n}_0{=}3)$, which transfers disjunction to numerical domain. Finally the joined state is widened based on the first state Δ_0', yielding a state $\mathtt{x}{::}\mathtt{ll}\langle\mathtt{n}_0\rangle \wedge \mathtt{n}_0{\geq}2$. It is a sound abstraction of both Δ_0 and Δ_1, and finishes the analysis with one more iteration.

Soundness and termination. The soundness of our analysis is ensured by the soundness of the following: the entailment prover [21], the abstract semantics (w.r.t. concrete semantics), the abstraction operation over shapes, and the join and widening operators.

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

The proof for entailment checking is by structural induction [21]; for abstract semantics is by induction over program constructors; for abstraction follows directly the first two; and for join and widening is based on entailment checking and soundness of corresponding numerical operators.

For the termination aspect, we have the result:

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

The proof is based on two facts: the finiteness over the shape domain provided by our restriction on cutpoints, and the termination over the numerical domain guaranteed by our widening operator. The first can be proved by claiming the finiteness of all possible abstract states only with the shape information: recalling our analysis algorithm where we set an upper bound n for shared cutpoints (logical variables) we keep in track of, we know that the program and logical variables preserved in our analysis are finite. Meanwhile all possible shape predicates are limited; therefore all the shape-only abstract states are finite. The second is proved in the abstract interpretation frameworks for numerical domains [23]. These two facts guarantee the convergence of our analysis.

5 Experiments and Evaluation

We have implemented a prototype system for evaluation purpose. The prototype system was built in Objective Caml. We used Sleek [21] as the solver for entailment checking over the heap domain, and Omega constraint solver [24] and Fixcalc solver [23] for join and widening operations in the numerical domain. Our test platform was an Intel Core 2 CPU 2.66GHz system with 8Gb RAM.

Program	**Function**	**Time**
`create`	Creates a list with given length parameter	0.452
`ins_sort`	Inner loop of Fig. 1	0.824
`ins_sort`	Outer loop of Fig. 1	4.372
`delete`	Disposes a list	0.720
`traverse`	Traverses a list	0.636
`append`	Appends two lists	0.312
`partition`	Auxiliary operation used by Quick-sort	1.497
`merge`	Merges two sorted lists to be one sorted list	1.972
`split`	Divides a list into two sublists with length difference of at most one	0.354
`select`	Selects the smallest node of a list	0.692
`select_sort`	Outer loop of selection sort	4.892
`tree_insert`	Inserts a node into a binary search tree	1.364
`tree_search`	Finds a node in a binary search tree	1.294

Fig. 5. Selected Experimental Results.

Figure 5 describes the programs with which we performed experiments. The first column denotes the names of the programs. The second column states the programs' functionalities. The last column exhibits the time in second taken by our analysis. As can be seen from their functions, these programs involve recursive data structures such as (sorted) linked lists and binary (search) trees, and employ loops to manipulate these data structures (and some of them even have nested loops). Our target is to verify these programs with the help of our analysis over the loops they invoke, such that user annotations for while loops can be avoided. Our experiments have confirmed that Hip/Sleek can verify all these programs successfully when supplied with loop invariants discovered by our analysis. According to our experience, these experiments just require the bound of shared cutpoints be a reasonably small number, say no more than twice of the number of program variables.

We have two main observations from our experimental results. The first is that we can handle many data structures with rich program properties they bear. To analyse these loops, we need to deal with both the list and list segment predicates to capture the linked list data structure, as well as their sorted version for the sorting algorithms. We can also handle tree-like predicates such as binary trees and binary search trees. Meanwhile these predicates also come along with many properties such as the length of the list and size/height of the tree, and the

minimum/maximum value of a sorted list/binary search tree. Based on them, our analysis is capable of expressing the invariants of these properties in terms of the constraints over the predicates' parameters.

Beyond the number of predicates and properties we can process, another observation on our analysis is that we can process them *rather precisely*. For example the list creation program creates a list with the same length as user input, and list traverse does not change list's length. Besides these, some loops provide critical invariants for the method running them to function correctly. For example, the quicksort algorithm partitions a list into three parts, where two are lists and the third just one node, whose value is exactly in the middle of that of the two other lists (`partition` in the table). We use a list bound predicate to indicate that fact which is successfully inferred by our analysis. We can also infer that the first loop of a mergesort (`split` in the table) can divide the list into two whose length difference is at most one, which is unimportant for the algorithm's functional correctness but essential for its performance. For `tree_insert`, we have the result that the tree's height is increased at most one, and the minimum/maximum value of the new binary search tree will be exactly the inserted value, if that value is out of the value bounds of the original tree. Such invariants are sufficiently precise to prove the functional correctness of all these programs with the given predicates.

6 Related Work and Conclusion

Related works. For heap-manipulating programs with any form of recursion (be it loop or recursive method call), dramatic advances have been made in synthesising their invariants/specifications. The local shape analysis [9] infers loop invariants for list-processing programs, followed by the SpaceInvader tool to infer full method specifications over the separation domain, so as to verify pointer safety for larger industrial codes [3, 27]. The SLAyer tool [10] implements an inter-procedural analysis for programs with shape information. To deal with also size information (such as number of nodes in lists/trees), THOR [19] derives a numerical program from the original heap-processing one in a sound way, such that the size information can be obtained with a traditional loop invariant synthesis. A similar approach [11] combines a set domain (for shape) with its cardinality domain (for corresponding numerical information) in a more general framework. Compared with these works, our approach can handle data structures with stronger invariants such as sortedness and binary search property, which have not been addressed in the previous works.

One more work to be mentioned is the relational inductive shape analysis [4]. It employs inductive checkers to express both shape and numerical information. Our approach has four advantages over theirs: firstly, we try to keep as many as possible shared cutpoints (logical variables) during the analysis (within a preset bound), whereas they do not preserve such cutpoints (which is witnessed by their joining rules over the shape domain). Therefore our analysis is essentially more precise than theirs, e.g. in the second scenario of (6) described in Section 4.2.

Meanwhile, our approach can deal with data structures with loops in them (say cyclic linked-lists), whereas they do not have a mechanism to handle it. An example in point is the state $\mathtt{x}{::}\mathtt{ls}\langle \mathtt{m}, \mathtt{y}\rangle * \mathtt{y}{::}\mathtt{ls}\langle \mathtt{y}, \mathtt{n}\rangle \wedge \mathtt{n}{>}0$ involving both a shared cutpoint $\mathtt{y}$ and a circled list $\mathtt{y}{::}\mathtt{ls}\langle \mathtt{y}, \mathtt{n}\rangle \wedge \mathtt{n}{>}0$, neither of which can be handled by their work (while ours is capable of that). Another advantage of our approach over theirs is that they only demonstrate how to analyse a program with one particular shape, such as their examples analysing programs which manipulate binary search trees and red-black trees without changing the variety of shape in the heap. Comparatively, we allow different predicates to appear in the analysis of one program, like in our motivating example (thanks to our more flexible abstraction operation). Lastly, their work is mainly from a theory perspective as they do not employ numerical reasoners to solve the relational constraints in their implementation; on the contrary, we discharge all the numerical and relational constraints with automated reasoners [23, 24].

There are also many other approaches that can synthesise shape-related program invariants than those based on separation logic. The shape analysis framework TVLA [26] is based on three-valued logic. It is capable of handling complicated data structures and properties, such as sortedness. Guo et al. [12] reported a global shape analysis that discover inductive structural shape invariants from the code. Kuncak et al. [16] developed a role system to express and track referencing relationships among objects, where an object's role (type) depends on, and changes according to, the mutation of its referencing. Hackett and Rugina [13] can deal with AVL-trees but is customised to handle only tree-like structures with height property. Compared with these works, separation logic based approach benefits from the frame rule and hence supports local reasoning.

Classical abstract interpretation [7] and its applications such as automated assertion discovery [8, 15, 17] mainly focus on finding numerical program properties. Compared with their works, ours is also founded on the abstract interpretation framework but tries to discover loop invariants with *both* separation and numerical information. Meanwhile, we can also utilise their techniques of join and widening to reason about the numerical domain, as we did for the work [23].

Concluding Remarks. We have reported an analysis which allows us to synthesise sound and useful loop invariants over a combined separation and numerical domain. The key components of our analysis include novel operations for abstraction, join and widening in the combined domain. We have built a prototype system and the initial experimental results are encouraging.

Acknowledgement. This work was supported by EPSRC Projects EP/G042322/1 and EP/E021948/1 and MoE Tier-2 Project R-252-000-444-112. We thank Florin Craciun for his precious comments.

References

1. J. Berdine, C. Calcagno, and P. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *FMCO*, 2005.
2. J. Berdine, B. Cook, D. Distefano, and P. O'Hearn. Automatic termination proofs for programs with shape-shifting heaps. In *18th CAV*, 2006.

3. C. Calcagno, D. Distefano, P. O'Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. In *36th POPL*, January 2009.
4. B. Chang and X. Rival. Relational inductive shape analysis. In *POPL*, 2008.
5. W.-N. Chin, C. David, H. H. Nguyen, and S. Qin. Automated verification of shape, size and bag properties. In *12th ICECCS*, 2007.
6. W.-N. Chin, C. David, H. H. Nguyen, and S. Qin. Enhancing modular oo verification with separation logic. In POPL, January 2008.
7. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, 1977.
8. P. Cousot and R. Cousot. On abstraction in software verification. In *CAV*, 2002.
9. D. Distefano, P. O'Hearn, and H. Yang. A local shape analysis based on separation logic. In *TACAS*, 2006.
10. A. Gotsman, J. Berdine, and B. Cook. Interprocedural shape analysis with separated heap abstractions. In *SAS*, 2006.
11. S. Gulwani, T. Lev-Ami, and M. Sagiv. A Combination Framework for Tracking Partition Sizes. In *POPL*, 2009.
12. B. Guo, N. Vachharajani, and D. August. Shape analysis with inductive recursion synthesis. In *PLDI*, 2007.
13. B. Hackett and R. Rugina. Region-based shape analysis with tracked locations. In *POPL*, 2005.
14. S. Ishtiaq and P. O'Hearn. BI as an assertion language for mutable data structures. In *POPL*, 2001.
15. L. Kovács and T. Jebelean. An algorithm for automated generation of invariants for loops with conditionals. In *SYNASC (Symbolic and Numeric Algorithms for Scientific Computing)*, 2005.
16. V. Kuncak, P. Lam, and M. Rinard. Role analysis. In *POPL*, 2002.
17. K. R. M. Leino and F. Logozzo. Loop invariants on demand. In *APLAS*, 2005.
18. K. R. M. Leino. Dafny: an automatic program verifier for functional correctness. To appear at *LPAR-16*, 2010.
19. S. Magill, M. Tsai, P. Lee, and Y. Tsay. Thor: A tool for reasoning about shape and arithmetic. In *CAV*, 2008.
20. H. H. Nguyen and W.-N. Chin. Enhancing program verification with lemmas. In *20th CAV*, 2008.
21. H. H. Nguyen, C. David, S. Qin, and W.-N. Chin. Automated verification of shape and size properties via separation logic. In *8th VMCAI*, 2007.
22. M. Parkinson and G. Bierman. Separation logic, abstraction and inheritance. In *POPL*, 2008.
23. C. Popeea and W.-N. Chin. Inferring disjunctive postconditions. In *Proceedings of 11th Asian Computing Science Conference*, 2006.
24. P. Pugh. The Omega Test: a fast and practical integer programming algorithm for dependence analysis. In *Communications of the ACM*, 1992.
25. J. Reynolds. Separation logic: a logic for shared mutable data structures. In *17th LICS*, 2002.
26. M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems*, 24(3):217–298, 2002.
27. H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. O'Hearn. Scalable shape analysis for systems code. In *20th CAV*, 2008.