Software Engineering Group
Department of Computer Science
Nanjing University
http://seg.nju.edu.cn

Technical Report No. NJU-SEG-2013-IJ-007

2013-IJ-007

Loop invariant synthesis in a combined abstract domain

Shengchao Qin, Guanhua He, Chenguang Luo, Wei-Ngan Chin, Xin Chen

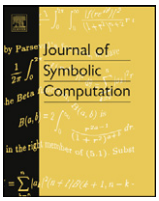
Journal of Symbolic Computation 2013

Journal of Symbolic Computation 50 (2013) 386–408

Contents lists available at SciVerse ScienceDirect

Journal of Symbolic Computation

www.elsevier.com/locate/jsc



Loop invariant synthesis in a combined abstract domain

Shengchao Qin [a,b,c], Guanhua He [a], Chenguang Luo [a], Wei-Ngan Chin [d], Xin Chen [c]

[a] School of Computing, Teesside University, Middlesbrough, TS1 3BA, UK
[b] College of Computer Science, Beijing University of Technology, China
[c] State Key Lab. for Novel Software Technology, Nanjing University, China
[d] School of Computing, National University of Singapore, Singapore

ARTICLE INFO

Article history:
Received 15 November 2011
Accepted 29 August 2012
Available online 7 September 2012

Keywords:
Loop invariant
Fixpoint analysis
Abstraction
Combining analysis
Shape analysis
Numerical analysis
Separation logic

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 to guide the verification, which can be cumbersome and error-prone by hand and can significantly restrict the usability of the verification system. In this paper, we reduce the need for some user annotations by automatically inferring loop invariants over an abstract domain with both shape and numerical information. Our loop invariant synthesis is conducted automatically by a fixed-point iteration process, equipped with newly designed abstraction mechanism, together with join and widening operators over the combined domain. We have also proven the soundness and termination of our approach. Initial experiments confirm 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 1960s, it remains a challenging problem to automatically verify heap-manipulating programs written in mainstream imperative languages. This is in part due to the use of shared mutable data structures in programs, and the need to track various program properties, such as structural numerical

E-mail addresses: s.qin@tees.ac.uk (S. Qin), g.he@tees.ac.uk (G. He).
URL: http://www.scm.tees.ac.uk/s.qin/ (S. Qin).

0747-7171/$ – see front matter
http://dx.doi.org/10.1016/j.jsc.2012.08.007

information (e.g. length and height) and relational numerical information (e.g. sortedness and binary search tree properties).

Since the emergence of separation logic (Ishtiaq and O'Hearn, 2001; Reynolds, 2002), dramatic advances have been made in automated software verification, e.g. the Smallfoot tool (Berdine et al., 2005) for the verification on pointer safety (i.e. shape properties asserting that pointers cannot go wrong), the verification on termination (Berdine et al., 2006), the verification for object-oriented programs (Chin et al., 2008; Parkinson and Bierman, 2008), and Dafny (Leino, 2010) and HIP/SLEEK (Chin et al., 2007, 2012; Nguyen and Chin, 2008; Nguyen et al., 2007) for handling more general properties (such as set-based and numerical properties for data structures) in heap-manipulating programs.

As a key to prove the correctness of loops, a suitable *loop invariant* for each loop must be provided by users in these verification systems. However, supplying invariants by hand in a sophisticated domain can be tedious and error-prone. It also affects the scalability of these tools, as each program may contain many loops.

To address this problem, various shape analysis techniques have been proposed in, e.g., the separation logic based SpaceInvader tool (Calcagno et al., 2009; Distefano et al., 2006; Yang et al., 2008), which, as a further step of Smallfoot, can automatically infer loop invariants as well as method specifications for pointer safety properties. The Hob system (Lam, 2007) offers a set-based analysis for loop invariant synthesis. Another tool THOR (Magill et al., 2008) incorporates simple numerical information into the shape domain to allow automated synthesis of properties involving length of list segments. These successes have demonstrated the feasibility to generate loop invariant automatically for shape analysis to help automate the program verification process.

However, most of the prior loop invariant analyses focus 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 complex user-defined data structures, such as height-balanced trees;
- Relational numerical properties, like sortedness and binary search property.

These properties can be part of the full functional correctness for heap-manipulating programs. The (aforementioned) HIP/SLEEK tool can handle such properties. It allows users to define their own shape predicates in conjunction with properties of interests, in order to capture a higher desired level of correctness for their programs.

In this paper, we present a technique to automatically discover loop invariants over the combined shape and numerical domain to improve the level of automation for HIP/SLEEK-like verification systems. Our approach is based on the framework of abstract interpretation (Cousot and Cousot, 1977) with fixed-point computation. We make the following technical contributions:

- We propose a loop invariant synthesis algorithm 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 prove that it always terminates.
- We have integrated our solution into the HIP/SLEEK tool 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.

The rest of the paper is structured as follows. We first illustrate our approach informally via an example (Section 2), and then give our programming and specification languages (Section 3). Formal details about loop invariant synthesis are presented in Section 4, followed by experimental results in Section 5. Related work and concluding remarks come afterwards.

2. The approach

Before presenting an illustrative example for the analysis, we first introduce our specification mechanism which follows the HIP/SLEEK system (Chin et al., 2012).

2.1. Specification mechanism

Separation logic (Ishtiaq and O'Hearn, 2001; Reynolds, 2002) 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 $p_1 * p_2$ asserts that two heaps described by the formulae p_1 and p_2 are domain-disjoint. We make use of this connective in our specifications.

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 singly linked list as

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

The parameter `root` for the predicate `ll` is the initial pointer into the first node of the linked list, or `null` if the list is empty. Its length is denoted by numerical parameter `n`. A uniform notation $\mathtt{p::c\langle v_1, \ldots, v_k\rangle}$ is used for either a singleton heap or a predicate. If `c` is a data node with fields $\mathtt{f_1, \ldots, f_k}$, the notation represents a singleton heap, $\mathtt{p} \mapsto \mathtt{c}[\mathtt{f_1} \mapsto \mathtt{v_1}, \ldots, \mathtt{f_k} \mapsto \mathtt{v_k}]$ which says that the variable `p` points to a `c` data node and the value of its fields are $\mathtt{v_1}, \ldots, \mathtt{v_k}$, e.g. $\mathtt{root::node\langle v, q\rangle}$ in the above formula. If `c` is a predicate name, then the data structure pointed to by `p` has the shape `c` with parameters $\mathtt{v_1}, \ldots, \mathtt{v_k}$, e.g. $\mathtt{q::ll\langle m\rangle}$ above.

We can also define a singly linked list segment as follows:

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

where the parameter `p` denotes the `next` field for the last node of the list segment. Note that we use the following shortened notation: (i) default `root` parameter in the left hand side (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:

$$\mathtt{sll\langle n, mn, mx\rangle} \equiv \left(\mathtt{root::node\langle mn, null\rangle} \wedge \mathtt{n} = 1 \wedge \mathtt{mn} = \mathtt{mx}\right) \vee \left(\mathtt{root::node\langle mn, q\rangle} * \mathtt{q::sll\langle n_1, k, mx\rangle} \wedge \mathtt{mn} \leqslant \mathtt{k} \wedge \mathtt{n} = \mathtt{n_1} + 1\right)$$

$$\mathtt{sls\langle p, n, mn, mx\rangle} \equiv \left(\mathtt{root::node\langle mn, p\rangle} \wedge \mathtt{n} = 1 \wedge \mathtt{mn} = \mathtt{mx}\right) \vee \left(\mathtt{root::node\langle mn, q\rangle} * \mathtt{q::sls\langle p, n_1, k, mx\rangle} \wedge \mathtt{mn} \leqslant \mathtt{k} \wedge \mathtt{n} = \mathtt{n_1} + 1\right)$$

where `mn` and `mx` denote respectively the minimum and maximum values stored in the sorted list.

As a more involved example, one may define the following predicate to specify sorted doubly-linked list segments

$$\mathtt{sds\langle p, q, n, mn, mx\rangle} \equiv \left(\mathtt{root::node2\langle mn, p, q\rangle} \wedge \mathtt{n} = 1 \wedge \mathtt{mn} = \mathtt{mx}\right) \vee \left(\mathtt{root::node2\langle mn, p, r\rangle} * \mathtt{r::sds\langle root, q, n - 1, k, mx\rangle} \wedge \mathtt{mn} \leqslant \mathtt{k}\right)$$

based on the data structure definition

```
data node2 { int val; node2 prev; node2 next; }
```

where the parameters `p` and `q` denote, respectively, the `prev` field of `root` and the `next` field of the last node of the list. Similar to the definition of `sls`, `mn` (resp. `mx`) is the minimum (resp. the maximum) value stored in the double sorted list.

Such user-supplied predicates can be employed to specify loop invariants and method specifications.

```
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.

2.2. Illustrative example

We now illustrate via an example our loop invariant synthesis process. The method `ins_sort` (Fig. 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 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 (Reynolds, 2002). 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:

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

which says that the pointer `srt` moves towards the rear of the list by one node. We then join it with the initial state (1) to obtain

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

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

$$\vee \big(\mathtt{r::node}\langle a, \mathtt{srt}\rangle * \mathtt{srt::sll}\langle n_s, c_1, b\rangle$$
$$\wedge\, \mathtt{prv} = r \wedge a \leqslant c_1 \wedge a \leqslant v \wedge n_r + 1 = n - n_x \wedge n_s + 1 = n_r\big) \tag{3}$$

The second iteration over the loop body starts with (3) and exhibits (also) the case that srt has moved by two nodes towards the rear, while prv moves by one node. Its result is then joined with pre-state (1) yielding the current state:

$$(3) \vee \mathtt{r::node}\langle a, \mathtt{prv}\rangle * \mathtt{prv::node}\langle c_1, \mathtt{srt}\rangle * \mathtt{srt::sll}\langle n_s, c_2, b\rangle$$
$$\wedge\, a \leqslant c_1 \leqslant c_2 \wedge c_1 \leqslant v \wedge n_r + 1 = n - n_x \wedge n_s + 2 = n_r \tag{4}$$

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

$$(4) \vee \mathtt{r::node}\langle a, r_0\rangle * r_0\mathtt{::node}\langle c_1, \mathtt{prv}\rangle * \mathtt{prv::node}\langle c_2, \mathtt{srt}\rangle$$
$$*\, \mathtt{srt::sll}\langle n_s, c_3, b\rangle \wedge a \leqslant c_1 \leqslant c_2 \leqslant c_3 \wedge c_2 \leqslant v \wedge n_r + 1 = n - n_x \wedge n_s + 3 = n_r$$

where we have an auxiliary logical variable 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 tolerate such increase in the subsequent iterations, the analysis will never terminate. Our abstraction process prevents this from happening by eliminating intermediate logical variables, as follows:

$$(4) \vee \mathtt{r::sls}\langle \mathtt{prv}, n_1, a, c_1\rangle * \mathtt{prv::node}\langle c_2, \mathtt{srt}\rangle * \mathtt{srt::sll}\langle n_s, c_3, b\rangle$$
$$\wedge\, a \leqslant c_1 \leqslant c_2 \leqslant c_3 \wedge c_2 \leqslant v \wedge n_r + 1 = n - n_x \wedge n_s + 3 = n_r \wedge n_1 = 2$$

Note that the heap part $\mathtt{r::node}\langle a, r_0\rangle * r_0\mathtt{::node}\langle c_1, \mathtt{prv}\rangle$ is abstracted as a sorted list segment $\mathtt{r::sls}\langle \mathtt{prv}, n_1, a, c_1\rangle$ with n_1 denoting the length of the segment and $n_1 = 2$ added into the state. This abstraction process ensures that our analysis does not allow the shape to increase infinitely. Note also that the abstraction process has made use of the fact $a \leqslant c_1$.

This fourth iteration results in a post-state where four nodes are passed by srt, and three by prv. An abstraction is performed to remove the newly created logical pointer variables. As a simplification of the presentation, let us denote σ as $\mathtt{r::sls}\langle \mathtt{prv}, n_1, a, c_1\rangle * \mathtt{prv::node}\langle c_2, \mathtt{srt}\rangle * \mathtt{srt::sll}\langle n_s, c_3, b\rangle \wedge a \leqslant c_1 \leqslant c_2 \leqslant c_3 \wedge c_2 \leqslant v \wedge n_r + 1 = n - n_x$. The result obtained after the fourth iteration is

$$(4) \vee (\sigma \wedge n_s + 3 = n_r \wedge n_1 = 2) \vee (\sigma \wedge n_s + 4 = n_r \wedge n_1 = 3)$$

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:

$$(4) \vee \big(\sigma \wedge (n_s + 3 = n_r \wedge n_1 = 2 \vee n_s + 4 = n_r \wedge n_1 = 3)\big)$$

This simplifies the abstraction further. After that, our widening operation compares the current state with the previous one, to look for a common set of (numerical) constraints that both states imply, and to replace those numerical constraints in the current state with the ones discovered by widening. This operation helps ensure the termination of our analysis. As for the example, a set of constraints among n_s, n_r and n_1 can be discovered to make the widened post-state to become:

$$(4) \vee (\sigma \wedge n_s + n_1 = n_r - 1 \wedge n_1 \geqslant 2) \tag{5}$$

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):

$$\mathtt{r::sll}\langle n_r, a, b\rangle \wedge \mathtt{srt} = r \wedge \mathtt{prv} = \mathtt{null} \wedge n_r + 1 = n - n_x$$
$$\vee\, \mathtt{r::node}\langle a, \mathtt{srt}\rangle * \mathtt{srt::sll}\langle n_s, c_1, b\rangle \wedge \mathtt{prv} = r$$
$$\wedge\, a \leqslant c_1 \wedge a \leqslant v \wedge n_r + 1 = n - n_x \wedge n_s + 1 = n_r$$

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

Fig. 2. A Core (C-like) imperative language.

$$
\begin{array}{lll}
\mathit{spred} & ::= & \texttt{root}::c\langle v^*\rangle \equiv \Phi \\
\mathit{mspec} & ::= & \mathit{requires}\ \Phi_{pr}\ \mathit{ensures}\ \Phi_{po} \\
\Phi & ::= & \bigvee \sigma^* \\
\sigma & ::= & \exists v^* \cdot \kappa \wedge \pi \\
\Delta & ::= & \Phi \mid \Delta_1 \vee \Delta_2 \mid \Delta \wedge \pi \mid \Delta_1 * \Delta_2 \mid \exists v \cdot \Delta \\
\kappa & ::= & \texttt{emp} \mid v::c\langle v^*\rangle \mid \kappa_1 * \kappa_2 \\
\pi & ::= & \gamma \wedge \phi \\
\gamma & ::= & v_1 = v_2 \mid v = \texttt{null} \mid v_1 \neq v_2 \mid v \neq \texttt{null} \mid \texttt{true} \mid \gamma_1 \wedge \gamma_2 \\
\phi & ::= & b \mid a \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \neg\phi \mid \exists v \cdot \phi \mid \forall v \cdot \phi \\
b & ::= & \texttt{true} \mid \texttt{false} \mid v \mid b_1 = b_2 \\
a & ::= & p_1 = p_2 \mid p_1 \leqslant p_2 \\
p & ::= & k^{\texttt{int}} \mid v \mid k^{\texttt{int}} \times p \mid p_1 + p_2 \mid -p \mid \mathit{max}(p_1, p_2) \mid \mathit{min}(p_1, p_2)
\end{array}
$$

Fig. 3. The specification language.

$$\vee\ \mathtt{r}::\mathtt{node}\langle \mathtt{a}, \mathtt{prv}\rangle * \mathtt{prv}::\mathtt{node}\langle \mathtt{c}_1, \mathtt{srt}\rangle * \mathtt{srt}::\mathtt{sll}\langle \mathtt{n}_s, \mathtt{c}_2, \mathtt{b}\rangle$$

$$\wedge\ \mathtt{a} \leqslant \mathtt{c}_1 \leqslant \mathtt{c}_2 \wedge \mathtt{c}_1 \leqslant \mathtt{v} \wedge \mathtt{n}_r + 1 = \mathtt{n} - \mathtt{n}_x \wedge \mathtt{n}_s + 2 = \mathtt{n}_r$$

$$\vee\ \mathtt{r}::\mathtt{sls}\langle \mathtt{prv}, \mathtt{n}_1, \mathtt{a}, \mathtt{c}_1\rangle * \mathtt{prv}::\mathtt{node}\langle \mathtt{c}_2, \mathtt{srt}\rangle * \mathtt{srt}::\mathtt{sll}\langle \mathtt{n}_s, \mathtt{c}_3, \mathtt{b}\rangle$$

$$\wedge\ \mathtt{a} \leqslant \mathtt{c}_1 \leqslant \mathtt{c}_2 \leqslant \mathtt{c}_3 \wedge \mathtt{c}_2 \leqslant \mathtt{v} \wedge \mathtt{n}_r + 1 = \mathtt{n} - \mathtt{n}_x \wedge \mathtt{n}_s + \mathtt{n}_1 = \mathtt{n}_r - 1 \wedge \mathtt{n}_1 \geqslant 2$$

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

$$\left(\mathtt{x}::\mathtt{ll}\langle \mathtt{n}_x\rangle \wedge \mathtt{r} = \mathtt{null} \wedge \mathtt{n}_x = \mathtt{n}\right) \vee \left(\mathtt{r}::\mathtt{node}\langle \mathtt{a}, \mathtt{null}\rangle * \mathtt{x}::\mathtt{ll}\langle \mathtt{n}_x\rangle \wedge \mathtt{n} = \mathtt{n}_x + 1\right)$$

$$\vee\ \left(\mathtt{r}::\mathtt{sll}\langle \mathtt{n}_r, \mathtt{a}, \mathtt{b}\rangle * \mathtt{x}::\mathtt{ll}\langle \mathtt{n}_x\rangle \wedge \mathtt{n} = \mathtt{n}_x + \mathtt{n}_r \wedge \mathtt{n}_r \geqslant 2\right)$$

which allows us to verify the entire method successfully using HIP/SLEEK verifier.

3. Language and abstract domain

To simplify presentation, we focus on a strongly-typed C-like imperative language in Fig. 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. `ll`, `ls`, `sll`, `sls` in Section 2.1), as well as method declarations *meth*. The definitions for *spred* and *mspec* are given later in Fig. 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 respectively heap-insensitive and heap-sensitive commands.

$s,h \models \Phi_1 \vee \Phi_2$	iff	$s,h \models \Phi_1$ or $s,h \models \Phi_2$
$s,h \models \exists v^* \cdot \kappa \wedge \pi$	iff	$\exists v^* \cdot s[v^* \mapsto v^*], h \models \kappa$ and $s[v^* \mapsto v^*] \models \pi$
$s,h \models \kappa_1 * \kappa_2$	iff	$\exists h_1, h_2 \cdot h_1 \perp h_2$ and $h = h_1 \cdot h_2$ and $s,h \models \kappa_1$ and $s,h \models \kappa_2$
$s,h \models \mathsf{emp}$	iff	$\mathsf{dom}(h) = \emptyset$
$s,h \models p::c\langle v_1,\ldots,v_n\rangle$	iff	$\mathit{isdatat}(c)$ and $s(p) > 0$ and $h = [s(p) \mapsto r]$ and $r = c[f_1 \mapsto v_1,\ldots,f_n \mapsto v_n]$
	or	$\mathit{isspred}(c)$ and $c\langle v_1,\ldots,v_n\rangle \equiv \Phi$ and $s,h \models [p/\mathtt{root}]\Phi$
$s \models \pi_1 \wedge \pi_2$	iff	$s \models \pi_1$ and $s \models \pi_2$
$s \models \pi$	iff	$s \models_A \pi$

Fig. 4. The semantic model.

Our specification language (in Fig. 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 (Nguyen et al., 2007).

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 ψ as aliasing and numerical information, respectively, and where p is a Presburger expression and k is a constant value. 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 a Φ form.

The memory model of our specification formula is adapted from what is given in the classical separation logic (Ishtiaq and O'Hearn, 2001; Reynolds, 2002), and our abstract domain is capable of handling user-defined shape predicates and related numerical properties described by this model. 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 $c[f_1 \mapsto v_1,\ldots,f_n \mapsto v_n]$ denoting an object value of type c where $v_1,\ldots,v_n$ are current values of the corresponding fields $f_1,\ldots,f_n$. Let $s,h \models \Delta$ denote the model relation that the stack s and heap h satisfy Δ, with s, h from the following concrete domains:

$$s \in \mathit{Stacks} =_{df} \mathsf{Var} \rightarrow \mathsf{Val} \cup \mathsf{Loc}$$

$$h \in \mathit{Heaps} =_{df} \mathsf{Loc} \rightharpoonup_{fin} \mathsf{ObjVal}$$

Note that each heap h is a finite partial mapping while each stack s is a total mapping. The detailed model definition is given in Fig. 4. We use $h_1 \perp h_2$ to denote that the heaps h_1 and h_2 have disjoint domains, and $h_1 \cdot h_2$ to indicate the union of such heaps. Note that 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 semantic model for pure formulae $s \models_A \pi$ is left in Appendix A (Fig. 8).

We use the separation logic prover Sleek (Nguyen et al., 2007) to prove whether one abstract state Δ' entails another one Δ: $\Delta' \vdash \Delta * \mathtt{R}$. Along with the proof, Sleek also computes the residue part `R` (a.k.a. the frame) which is useful for our inference framework. To prove the entailment is to check whether heap nodes in the antecedent Δ' are sufficiently precise to cover all nodes from the consequent Δ. The entailment checking procedure uses unfold/fold reasoning to deal with user-defined shape predicates with sophisticated numerical properties. During the entailment proof, the frame `R` is generated and it contains the nodes which are not consumed from the antecedent after matching up with the formula from the consequent, and numerical constraints which convey the relationship between the variables in the antecedent and consequent formulae. For instance, by entailment proof

$$\exists \mathtt{y} \cdot \mathtt{x::node\langle vx, y\rangle} * \mathtt{y::ll\langle n\rangle} \vdash \mathtt{x::ll\langle m\rangle} * \mathtt{R}$$

we can generate the residue `R` as $\mathtt{m} = \mathtt{n} + 1$, which says that `x` is a list of length $\mathtt{n} + 1$. Meanwhile, if we try to prove

$$\exists \mathtt{y} \cdot \mathtt{x::node\langle vx, y\rangle} * \mathtt{y::node\langle vy, z\rangle} \wedge \mathtt{vx} \leqslant \mathtt{vy} \vdash \mathtt{x::sls\langle n, z, mn, mx\rangle} * \mathtt{R}$$

Fixpoint Computation in Combined Domain
Input: $\mathcal{T}$, Δ_{pre}, while b $\{e\}$, n;
Local: $i := 0$; $\Delta_i := \Delta_{pre}$; $\Delta'_i := \Delta_i$;

1 **repeat**
2 $i := i + 1$;
3 $\Delta_i := \text{widen}^{\dagger}(\Delta_{i-1}, \text{join}^{\dagger}(\Delta_{i-1}, \Delta'_{i-1}))$;
4 $\Delta'_i := \text{abs}^{\dagger}([e]_{\mathcal{T}}(\Delta_i \wedge b))$;
5 **if** $\Delta'_i = \texttt{false} \vee \text{cp\_no}(\Delta'_i) > n$ **then return** fail **end if**
6 **until** $\Delta'_i = \Delta'_{i-1}$;
7 **return** Δ'_i

Fig. 5. Main analysis algorithm.

the residue R can be generated as $\texttt{n} = 2 \wedge \texttt{mn} = \texttt{vx} \wedge \texttt{mx} = \texttt{vy} \wedge \texttt{mn} \leqslant \texttt{mx}$, which shows that the length of the sorted list from x to z is 2, and the minimal value of the list is vx in node x and the maximal value is vy in node y. From the above examples, we can see that the Sleek prover can be used to eliminate quantified pointer variables, to generate more abstract shape views, and to preserve useful numerical information.

Based on the entailment relation, we define a partial order over the abstract states:

$$\Delta \preccurlyeq \Delta' =_{df} \Delta' \vdash \Delta * \texttt{R} \quad \text{for some } \texttt{R}$$

We also denote this by $\Delta' \succcurlyeq \Delta$. Based on this partial order, we also have an induced lattice over these states as the base of fixpoint calculation for loop invariants.

4. Analysis algorithm

Our proposed analysis algorithm is given in Fig. 5. The algorithm takes four input parameters: $\mathcal{T}$ as the program environment with all the method specifications in the program, Δ_{pre} as the pre-condition for the while loop (i.e. the abstract state before the loop starts), the while loop itself while b $\{e\}$, and an upper bound n on the number of shared logical variables we keep during analysis.

Our analysis is based on abstract interpretation (Cousot and Cousot, 1977) 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 starting state of the calculation is Δ_{pre}. Among the two states here, the unprimed version Δ_i denotes the initial state before the ith 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 initial state Δ_{i-1} of the previous iteration with the result state Δ'_{i-1} obtained in the previous iteration, and widen it against the state Δ_{i-1} (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 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

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

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

where AllSpec contains all the specifications of all methods (extracted from the program *Prog*). For some expression e, given its pre-condition, 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:

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

$$\frac{isdatat(c) \quad \sigma \vdash x::c\langle v^*\rangle * \sigma'}{\mathsf{rearr}(x)\sigma =_{df} \sigma}$$

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

As mentioned earlier, the test $isdatat(c)$ returns `true` only if c is a data node and $isspred(c)$ 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 previously.

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

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

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

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

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

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

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

$$\frac{t\ mn\ ((t_i\ u_i)_{i=1}^n)\ requires\ \Phi_{pr}\ ensures\ \Phi_{po} \in \mathcal{T} \quad \rho = [x_i/u_i]_{i=1}^n \quad \sigma \vdash \rho\Phi_{pr} * \sigma_{fr} \quad \sigma_{po} = \rho\Phi_{po}}{\mathsf{exec}(mn(x_1, .., x_n))(\mathcal{T})\sigma =_{df} \sigma_{po} * \sigma_{fr}}$$

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 pre-condition of mn, as signified by $\sigma \vdash \rho\Phi_{pr} * \sigma_{fr}$, where σ_{fr} is the frame part. In this case, the execution succeeds and the postcondition of mn ($\rho\Phi_{po}$) is added into the post-state.

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 $\mathcal{P}_{\mathsf{SH}}$:

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

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

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

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 shapes 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:

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

where both x and 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 x and y, and thus the state is not changed (the - denotes the same formula as input). As illustrated, the abstraction transition function abs eliminates unimportant cutpoints (during analysis) to ensure termination. Its type is defined as follows:

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

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

$$\frac{}{\mathsf{abs}(\sigma\wedge x_0 = e) =_{df} \sigma[e/x_0]}\ (\mathsf{Subst1})$$

$$\frac{}{\mathsf{abs}(\sigma\wedge e = x_0) =_{df} \sigma[e/x_0]}\ (\mathsf{Subst2})$$

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

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

The first two rules Sub1 and Sub2 eliminate logical variables (x_0) by replacing them with their equivalent expressions (e). The third rule Unreach 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 used by, the program variables, it is safe to treat it as garbage `true`, for example the $\mathtt{x}_0$ in $\mathtt{x}::\mathtt{node}\langle\_, \mathtt{null}\rangle * \mathtt{x}_0::\mathtt{node}\langle\_, \mathtt{null}\rangle$ where only `x` is a program variable.

The last rule 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 into a shape predicate. It confirms that c_1 is a data node definition and c_2 is a predicate. The predicate c_2 is selected from the user-defined predicates environments and it is the target shape to be abstracted against with. The rule ensures that the latter is a sound abstraction of the former by entailment proof, and the pointer logical parameters of c_1 are not reachable from other part of the heap (so that the abstraction does not lose necessary information). For instance, given the user-defined predicate for singly linked list $\mathtt{root}::\mathtt{ll}\langle\mathtt{n}\rangle$, the following abstraction step can take place:

$$\mathsf{abs}\big(\mathtt{x}::\mathtt{node}\langle\_, \mathtt{z}_0\rangle * \mathtt{z}_0::\mathtt{node}\langle\_, \mathtt{null}\rangle\big) = \mathtt{x}::\mathtt{ll}\langle\mathtt{n}\rangle \wedge \mathtt{n} = 2$$

where `x` is program variable and $\mathtt{z}_0$ is logical variable. The function Reach is defined as follows:

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

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

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

namely, it is composed of aliases of v and pointer variables reachable from v. The predicate $\mathtt{isptr}(x)$ checks if x is a pointer variable. For example, $\mathtt{ReachVar}(\mathtt{x}::\mathtt{node}\langle\_, \mathtt{z}_0\rangle * \mathtt{y}::\mathtt{node}\langle\_, \mathtt{z}_0\rangle * \mathtt{z}_0::\mathtt{node}\langle\_, \mathtt{p}_0\rangle, \{\mathtt{x}\}) = \{\mathtt{x}, \mathtt{z}_0, \mathtt{p}_0\}$. Note that the numerical logic parameters can be abstracted since the numerical relations are kept in pure formulae, so we do not lose numerical information here.

We apply the above abstraction rules (following the given order) onto an 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:

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

where the rename function prevents naming clashes among logical variables of σ_1 and σ_2, by assigning fresh names to logical variables with the same name in the two states. For example given a name clash on $\mathtt{x}_0$, it may change states $\exists \mathtt{x}_0 \cdot \mathtt{x}_0 = 0$ and $\exists \mathtt{x}_0 \cdot \mathtt{x}_0 = 1$ to $\exists \mathtt{x}_0 \cdot \mathtt{x}_0 = 0$ and $\exists \mathtt{x}_1 \cdot \mathtt{x}_1 = 1$ instead. 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 from the weaker state (which is more general/abstract) as the shape of the joined state, and performs joining for numerical formulae with $\mathsf{join}_\pi(\pi_1, \pi_2)$, the convex hull operator over numerical domain (Popeea and Chin, 2006). Otherwise it keeps a disjunction of the two states (as it would be unsound to join their shapes together in this case). We can lift this operator for abstract state Δ as follows:

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

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

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

where the rename function has the same effect as above. Generally this operator is analogous to the join operator; the only difference is that we expect (the shape part of) the second operand σ_2 to be weaker than (that of) the first σ_1, so 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 (Popeea and Chin, 2006).

Based on the widening over conjunctive abstract states, we lift the operator over (disjunctive) abstract states:

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

which is similar to its counterpart of the join operator.

The above three operations (abstraction, join and widening) provide termination guarantee while preserving soundness, as the following example demonstrates.

Example 4.1 *(Abstraction, join and widening).* Assume we have two abstract states,

$$\Delta_0 = \mathtt{x}::\mathtt{node}\langle \_, \mathtt{x}_0\rangle * \mathtt{x}_0::\mathtt{node}\langle \_, \mathtt{null}\rangle \quad \text{and}$$

$$\Delta_1 = \mathtt{x}::\mathtt{node}\langle \_, \mathtt{x}_0\rangle * \mathtt{x}_0::\mathtt{node}\langle \_, \mathtt{x}_1\rangle * \mathtt{x}_1::\mathtt{node}\langle \_, \mathtt{null}\rangle$$

We would like to discover a sound approximation for both states. Firstly we perform abstractions on both to obtain two abstract states, say, Δ_0' = $\mathtt{x}::\mathtt{ll}\langle \mathtt{n}_0\rangle \wedge \mathtt{n}_0 = 2$ and Δ_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 Δ_1'' = $\mathtt{x}::\mathtt{ll}\langle \mathtt{n}_0\rangle \wedge (\mathtt{n}_0 = 2 \vee \mathtt{n}_0 = 3)$, which transfers disjunction to the numerical domain. Finally the joined state is widened based on the first state Δ_0', yielding a state $\mathtt{x}::\mathtt{ll}\langle \mathtt{n}_0\rangle \wedge \mathtt{n}_0 \geqslant 2$. It is a sound abstraction of both Δ_0 and Δ_1, and finishes the analysis with one more iteration.

4.3. Soundness and termination

4.3.1. Soundness

The soundness of our analysis relies on the underlying operational semantics of our programming language, which is a small-step semantics consisting of transitions of the form:

$$\langle s, h, e\rangle \hookrightarrow \langle s_1, h_1, e_1\rangle$$

where $s(s_1)$ and $h(h_1)$ denote respectively the stacks and the heaps and where $e(e_1)$ denotes the program codes (an empty program code is denoted by -). The transitive closure of the above transition relation is denoted as $\hookrightarrow^*$. The full details of the operational semantics are given in Appendix A. Soundness of analysis is important as it allows us to directly use the loop invariant inferred without the need for another confirmation by either a user or a verifier.

We define the soundness of our analysis as follows:

Definition 4.1 *(Soundness).* Let Δ denote the loop invariant synthesised by our analysis for a while loop `while` b $\{e\}$. The analysis is sound if for all s, h, such that $s, h \models \Delta$ and $\langle s, h, e\rangle \hookrightarrow^* \langle s', h', \text{-}\rangle$ (for some s', h'), we have $s', h' \models \Delta$.

The crux to prove the soundness of our analysis is to ensure that soundness is preserved during each step of our analysis. That is, the abstract semantics, the abstraction of shapes, the join operation and the widening operation used in our analysis are all sound. From Lemma 4.2 on we will see that all these can be reduced to the soundness of entailment proof provided by Sleek, which is already proven:

Lemma 4.1 *(Soundness of entailment proof). For Δ_1 and Δ_2, if $\Delta_1 \vdash \Delta_2$ holds, then for all $s, h \models \Delta_1$, we have $s, h \models \Delta_2$.*

Proof. The soundness of the entailment proof is proven by Chin et al. (2012). □

Lemma 4.2 *(Soundness of abstract semantics). If $[e]_{\mathcal{T}}\Delta = \Delta_1$, then for all s, h, if $s, h \models \Delta$ and $\langle s, h, e\rangle \hookrightarrow \langle s_1, h_1, e_1\rangle$, then there always exists Δ_0 such that*

$$s_1, h_1 \models \Delta_0 \quad \textit{and} \quad [e_1]_{\mathcal{T}}\Delta_0 = \Delta_1$$

Proof. The proof is done by structural induction over program constructors and is left in Appendix A. □

Lemma 4.3 *(Soundness of* abs*). If $\mathsf{abs}(\sigma) = \sigma'$, then $\sigma \succcurlyeq \sigma'$.*

Proof. The soundness proof of the first two substitution rules is trivial. For the Unreach rule, we can easily prove that $\mathtt{x}_0{::}\mathtt{c}\langle \mathtt{v}^*\rangle * \sigma \vdash \mathtt{true} * \sigma$, where σ is the frame part of the entailment check. By the frame rule of separation logic, we only need to show that $\mathtt{x}_0{::}\mathtt{c}\langle \mathtt{v}^*\rangle \vdash \mathtt{true}$, which obviously is true. The result of rule Abs is obtained via a folding operation in Sleek against a user-defined predicate, so the soundness of this rule is guaranteed by that of the entailment proof in Sleek. □

Lemma 4.4 *(Soundness of* join*). If $\mathsf{join}(\sigma_1, \sigma_2) = \sigma_j$, then we have $\sigma_1 \succcurlyeq \sigma_j$ and $\sigma_2 \succcurlyeq \sigma_j$.*

Proof. Let σ_1 be $(\exists x_1^* \cdot \kappa_1 \wedge \pi_1)$, and σ_2 be $(\exists x_2^* \cdot \kappa_2 \wedge \pi_2)$. By the definition of the join operator, we have three cases:

- If $\kappa_1 \vdash \kappa_2 * \mathtt{true}$, we have $\sigma_j = \exists x_1^*, x_2^* \cdot \kappa_2 \wedge \mathsf{join}_\pi(\pi_1, \pi_2)$. Then we need to show that $\exists x_1^* \cdot \kappa_1 \wedge \pi_1 \vdash \exists x_1^*, x_2^* \cdot \kappa_2 \wedge \mathsf{join}_\pi(\pi_1, \pi_2)$ and $\exists x_2^* \cdot \kappa_2 \wedge \pi_2 \vdash \exists x_1^*, x_2^* \cdot \kappa_2 \wedge \mathsf{join}_\pi(\pi_1, \pi_2)$, which are true because $\kappa_1 \vdash \kappa_2 * \mathtt{true}$ by condition, $\kappa_2 \vdash \kappa_2$ by separation logic, and $\pi_1 \vdash \mathsf{join}_\pi(\pi_1, \pi_2)$ and $\pi_2 \vdash$

$\mathsf{join}_\pi(\pi_1, \pi_2)$ by the soundness of the convex hull operator over the numerical domains (Popeea and Chin, 2006).

- If $\kappa_2 \vdash \kappa_1 * \mathtt{true}$, the soundness proof is similar to the first case.
- Otherwise, we have $\sigma_j = \sigma_1 \vee \sigma_2$. The soundness proof of this case is trivial. □

Lemma 4.5 *(Soundness of* widen*). If* $\mathsf{widen}(\sigma_1, \sigma_2) = \sigma_w$*, then we have* $\sigma_1 \succcurlyeq \sigma_w$ *and* $\sigma_2 \succcurlyeq \sigma_w$*.*

Proof. The proof of soundness of widen is similar to the soundness proof of join. □

Based on the results above, we have

Theorem 4.1 *(Soundness). Our analysis is sound with respect to the underlying operational semantics.*

4.3.2. Termination

Next we show the termination of our analysis algorithm, which is based on two observations: the finiteness over the shape domain and the termination over the numerical domain guaranteed by our widening operator. The first can be proven by claiming the finiteness of the number 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) that we keep track of, we know that the number of program variables and logical variables preserved in our analysis are finite. Note that the number of all shape predicates are also limited; therefore all the shape-only abstract states are finite. The second is proven in the abstract interpretation frameworks for the numerical domains (Popeea and Chin, 2006). These two facts guarantee the convergence of our analysis. Such a termination proof can help assure us that our analysis never goes into an infinite loop, regardless of which loop it is made to analyse.

Lemma 4.6 *(Finiteness of the abstract shape domain). With a finite number of program variables, logical variables, data node types and shape predicates, the abstract shape domain is finite.*

Proof. Let the number of program variables, logical variables, data node types and shape predicates be m, n, l_1 and l_2, respectively, and the maximal number of fields of data nodes or arguments of the predicates be k, where $m, n, l_1, l_2, k \in \mathbb{N}$ are finite numbers. Note that the root parameter of a data structure is also counted in k. For example, the number of arguments of predicate `ll` is 3. Based on the fact that the number of variables is $(m+n)$, and k is the number of holes for variables in one single predicate, we have the possible number of atomic formulae of this predicate as $(m+n)^k$. The number of shape structure is $(l_1 + l_2)$, then the upper bound of the number of all possible different atomic shape formulae is $(l_1 + l_2) \times (m+n)^k + 1$, where 1 is for when the shape formula is `true`. So there are at most $2^{(l_1+l_2)\times(m+n)^k+1}$ shape formulae in this shape domain. Since m, n, l_1, l_2 and k are finite natural numbers, the shape domain is finite. □

Definition 4.2. A (conjunctive) state σ is *reducible* if and only if $\mathsf{abs}(\sigma) \not\vdash \sigma$. If $\mathsf{abs}(\sigma) \vdash \sigma$, then σ is *irreducible*, in which case we also say σ is *stabilised*.

Lemma 4.7 *(Termination of* abs*). For all state* σ*, the application of the four abstraction operations over* σ *exhaustively (following the given order) will terminate in finite steps within a finite shape domain.*

Proof. Let us apply abs over σ_0 exhaustively to obtain a sequence $\sigma_1, \sigma_2, \ldots, \sigma_n$, where $n \in \mathbb{N}$. By the soundness of abs, we have $\sigma_0 \vdash \sigma_1 \vdash \sigma_2 \vdash \cdots \vdash \sigma_n$, i.e. $\sigma_0 \succcurlyeq \sigma_1 \succcurlyeq \sigma_2 \succcurlyeq \cdots \succcurlyeq \sigma_n$. Since the shape parts of $\sigma_{0,\ldots,n}$ are in a finite shape domain, and the four abstraction rules do not alter the numerical parts of these states, there must exist a $\sigma_{i,0\leqslant i\leqslant n}$ which is *irreducible*/*stabilised*, i.e., $\sigma_k = \sigma_i$ for all $k \geqslant i$. □

Lemma 4.8 *(Termination of widening). Within a finite shape domain, given a sequence* σ'_n $(n \in \mathbb{N})$*, the sequence* σ_n *generated by* $\sigma_0 = \sigma'_0$ *and* $\sigma_{n+1} = \mathsf{widen}(\sigma_n, \sigma'_{n+1})$ *is ultimately stationary, i.e.* $\exists i \cdot \forall k \geqslant i \cdot \sigma_k = \sigma_i$*.*

```
1 node merge(node left, node right)
2   requires left::sll⟨n1,s1,l1⟩
             * right::sll⟨n2,s2,l2⟩
3   ensures res::sll⟨n3,s3,l3⟩∧n3=n1+n2
            ∧s3=min(s1,s2)∧l3=max(l1,l2);
4 {
5   node r = null;
6   while (left != null && right != null){
7     if (left.val <= right.val) {
8       node tmp = left;
9       left = left.next;
10      tmp.next = null;
11      r = append(r,tmp);
12    }
13    else {
14      node tmp = right;
15      right = right.next;
16      tmp.next = null;
17      r = append(r,tmp);
18    }
19  }
20  if (left == null) {
21    r = append(r, right);
22  }
23  else {
24    r = append(r, left);
25  }
26  return r;
27 }
```

Fig. 6. Merge two sorted linked list.

Proof. The proof follows the idea of the widening termination proof in cofibered domains (Venet, 1996). Similar to the proof of termination of abs, the shape part of σ_n will be stationary since the shape domain is finite. The termination of numerical part can be guaranteed by numerical join and widening (Popeea and Chin, 2006). Combining them together, σ_n will be stationary. □

Based on the above results, we conclude the termination result about our analysis algorithm as follows.

Theorem 4.2 *(Termination). The iteration of our fixpoint computation will terminate in finite steps, given a finite number of method specifications, program variables and user-defined predicates, and given an upper bound on the number of logical variables to keep during the analysis.*

5. Experiments and evaluation

We have built a prototype system using Objective Caml. In our experiments, we use Sleek (Nguyen et al., 2007) as the solver for entailment proofs, and Omega constraint solver (Pugh, 1991) and Fixcalc solver (Popeea and Chin, 2006) for join and widening operations in the numerical domain. Our test platform was an Intel Core 2 CPU 2.66GHz system with 8Gb RAM.

Example 5.1 *(Merge).* Fig. 6 shows a procedure `merge` which merges two sorted lists referred to by `left` and `right` into one sorted list, and returns the merged list as result. The pre-condition of the while loop within the `merge` procedure (starting at line 6) is calculated as

$$\texttt{left::sll}\langle n_1, s_1, l_1\rangle * \texttt{right::sll}\langle n_2, s_2, l_2\rangle \wedge \texttt{r} = \texttt{null}$$

The function `append` concatenates two sorted lists, which requires that the maximal value stored in the first input list is smaller than or equal to the minimal value of the second input list, and ensures that it returns a concatenated sorted list (referred to by `res`). The following is the specification of `append`:

$$\texttt{node append(node x, node y)}$$

$$\texttt{requires x::sll}\langle n_1, s_1, l_1\rangle * \texttt{y::sll}\langle n_2, s_2, l_2\rangle \wedge l_1 \leqslant s_2$$

$$\texttt{ensures res::sll}\langle n_1 + n_2, s_1, l_2\rangle$$

By applying our analysis algorithm to the while loop in the program, we obtain the following loop invariant, which contains seven different disjunctive branches, with each branch describing a different situation:

$$\begin{aligned}
&\texttt{left::sll}\langle n_1, s_1, l_1\rangle * \texttt{right::sll}\langle n_2, s_2, l_2\rangle \wedge \texttt{r} = \texttt{null} && (7)\\
&\quad \vee \texttt{r::node}\langle s_1, \texttt{null}\rangle * \texttt{right::sll}\langle n_2, s_2, l_2\rangle \wedge \texttt{left} = \texttt{null}\\
&\quad \wedge s_1 = l_1 \wedge s_1 < s_2 && (8)\\
&\quad \vee \texttt{r::node}\langle s_2, \texttt{null}\rangle * \texttt{left::sll}\langle n_1, s_1, l_1\rangle \wedge \texttt{right} = \texttt{null}\\
&\quad \wedge s_2 = l_2 \wedge s_2 \leqslant s_1 && (9)\\
&\quad \vee \texttt{r::node}\langle s_3, \texttt{null}\rangle * \texttt{left::sll}\langle n_1', s_1', l_1\rangle * \texttt{right::sll}\langle n_2', s_2', l_2\rangle\\
&\quad \wedge (s_3 = s_1 \wedge s_1 < s_2 \wedge n_1 = n_1' + 1 \wedge n_2' = n_2 \wedge s_1 \leqslant s_1' \wedge s_2' = s_2\\
&\quad \vee s_3 = s_2 \wedge s_2 \leqslant s_1 \wedge n_2 = n_2' + 1 \wedge n_1' = n_1 \wedge s_2 \leqslant s_2' \wedge s_1' = s_1) && (10)\\
&\quad \vee \texttt{r::sll}\langle n_3, s_3, l_3\rangle * \texttt{right::sll}\langle n_2', s_2', l_2\rangle \wedge \texttt{left} = \texttt{null} \wedge n_3 = n_1 + n_2 - n_2'\\
&\quad \wedge s_3 = \texttt{min}(s_1, s_2) \wedge l_3 = l_1 \wedge l_1 < s_2' && (11)\\
&\quad \vee \texttt{r::sll}\langle n_3, s_3, l_3\rangle * \texttt{left::sll}\langle n_1', s_1', l_1\rangle \wedge \texttt{right} = \texttt{null} \wedge n_3 = n_2 + n_1 - n_1'\\
&\quad \wedge s_3 = \texttt{min}(s_1, s_2) \wedge l_3 = l_2 \wedge l_2 \leqslant s_1' && (12)\\
&\quad \vee \texttt{r::sll}\langle n_3, s_3, l_3\rangle * \texttt{left::sll}\langle n_1', s_1', l_1\rangle * \texttt{right::sll}\langle n_2', s_2', l_2\rangle\\
&\quad \wedge n_3 = n_1 - n_1' + n_2 - n_2' \wedge s_3 = \texttt{min}(s_1, s_2) \wedge l_3 \leqslant \texttt{min}(s_1', s_2') && (13)
\end{aligned}$$

The branch (7) represents the state before any iteration. The branches (8), (9), and (10) denote three special scenarios after one iteration. The branch (8) (resp. (9)) denotes the case where initially the `left` (resp. `right`) list contains only one node which holds a value no bigger than any value stored in the `right` (resp. `left`) list, and after one iteration, `r` refers to the sole node in the initial `left` (resp. `right`) list and the `left` (resp. `right`) pointer becomes null. The branch (10) denotes the scenario where after one iteration neither the `left` list nor the `right` list is empty but `r` still refers to the node with the smallest value. The branches (11), (12) and (13) denote possible states reached after some (one or more) iterations. The branch (11) (resp. (12)) denotes the state reached after some iterations where the `left` (resp. `right`) pointer has traversed to the end of the list. The branch (13) denotes the case where neither the `left` pointer nor the `right` pointer has reached the end of their list after some iterations. In all these three branches, `r` refers to the merged list obtained so far. The shape of each branch is demonstrated in Fig. 7.

Note that branches (8), (9) and (10) are, respectively, special cases of branches (11), (12) and (13) (logically, the former formulae entail the latter ones, respectively). So we can simplify the loop invariant as

$$\begin{aligned}
&\texttt{r::sll}\langle n_3, s_3, l_3\rangle * \texttt{right::sll}\langle n_2', s_2', l_2\rangle \wedge \texttt{left} = \texttt{null} \wedge n_3 = n_1 + n_2 - n_2'\\
&\quad \wedge s_3 = \texttt{min}(s_1, s_2) \wedge l_3 = l_1 \wedge l_1 < s_2'\\
&\quad \vee \texttt{r::sll}\langle n_3, s_3, l_3\rangle * \texttt{left::sll}\langle n_1', s_1', l_1\rangle \wedge \texttt{right} = \texttt{null} \wedge n_3 = n_2 + n_1 - n_1'\\
&\quad \wedge s_3 = \texttt{min}(s_1, s_2) \wedge l_3 = l_2 \wedge l_2 \leqslant s_1'\\
&\quad \vee \texttt{r::sll}\langle n_3, s_3, l_3\rangle * \texttt{left::sll}\langle n_1', s_1', l_1\rangle * \texttt{right::sll}\langle n_2', s_2', l_2\rangle\\
&\quad \wedge n_3 = n_1 - n_1' + n_2 - n_2' \wedge s_3 = \texttt{min}(s_1, s_2) \wedge l_3 \leqslant \texttt{min}(s_1', s_2')
\end{aligned}$$

An immediate benefit to simplify the generated loop invariant is to reduce the number of disjunctions. Therefore, it can lead to the increase of the scalability of our inference system. It also simplifies the verification process which makes use of the loop invariant. Note that the soundness of our analysis ensures that we do not need to re-verify the while loop with the inferred invariant; instead, we can directly move on to verify the code fragment after the while loop, starting with the inferred loop invariant conjoined with the negation of the loop test (a sound postcondition for the while loop).

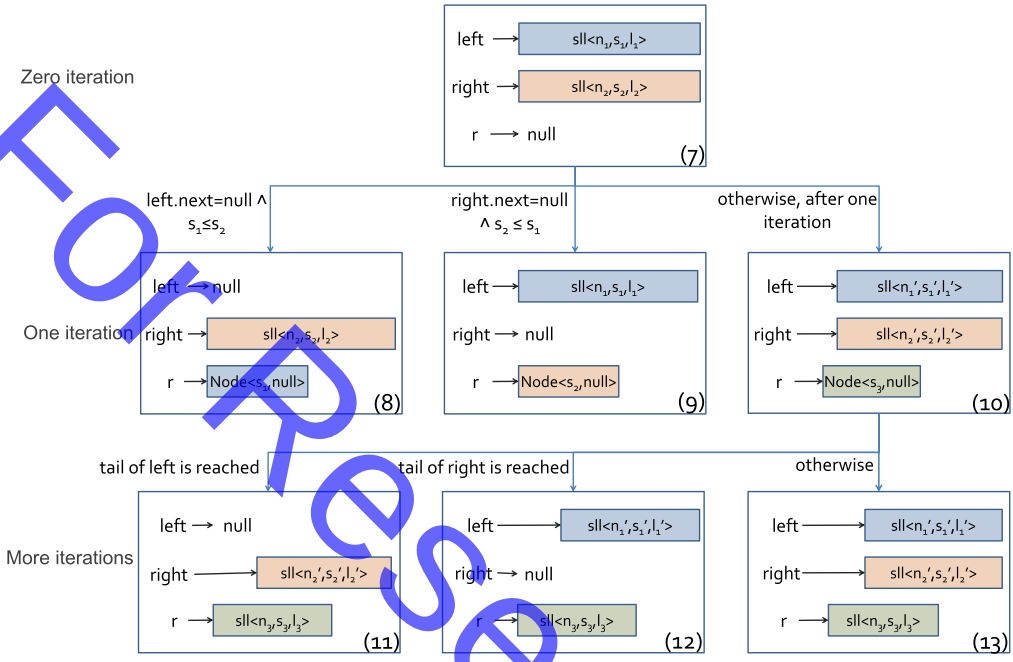


Fig. 7. Shape of each branch of the generated loop invariant in the merge example.

Table 1 depicts some of the programs we have conducted experiments with, including list processing programs, sorting algorithms, tree processing programs, and loops code from FreeRTOS (2012). Total LOC is the total number of lines of the code. 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 without the need of user annotations for loops that occurred in them. Our analysis is employed to automatically infer loop invariants for those while loops. Our experiments confirm 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 cut-points be a reasonably small number, say no more than twice of the number of program variables. Note that, it takes longer to analyse the procedures that have nested loops, such as `select_sort`, `list2tree`, and so on, because we need to analyse the inner loop multiple times. We translated the source code of the FreeRTOS kernel to our language, and successfully inferred the loop invariants of the loops which do not involve pointer arithmetic in *list.c* and *task.c*. We employ double linked sorted list predicate for the verification of FreeRTOS.

We have two main observations from our experimental results. The first is that we can handle many different kinds of data structures with rich program properties they exhibit. To analyse these loops, we need to deal with both single linked and double linked list predicates to capture the 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 capture various numerical information 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. With these predicates, our analysis is capable of capturing sophisticated numerical invariants, which are simply captured as constraints over the parameters of predicates involved.

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

Table 1
Selected Experimental Results.

Program	Function	Time
List processing programs		(Total LOC: 232)
`create`	Creates a list with given length parameter	0.452
`delete`	Disposes a list	0.720
`traverse`	Traverses a list	0.636
`length`	Counts the length of a list	0.772
`append`	Appends two sorted lists	0.312
`take`	Takes the first n elements of a list, or itself	0.852
`drop`	Returns suffix of a list after the first n elements, or empty	0.844
`reverse`	Reverse the elements of the list, in place	1.032
`filter`	Drops the elements bigger than k of a list	1.182
`lookup`	Returns the first node whose values equals to k, or `null`	0.876
`drop_even`	Drops all the elements whose indexes are even	1.332
Sorting list processing programs		(Total LOC: 178)
`ins_sort(inner)`	Inner loop of Fig. 1	0.824
`ins_sort(outer)`	Outer loop of Fig. 1	4.372
`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 processing programs		(Total LOC: 87)
`tree_search`	Finds a node in a binary search tree	1.294
`tree_insert`	Inserts a node into a binary search tree	1.364
`list2tree`	Inserts nodes of a list into a binary search tree	5.176
FreeRTOS		(Total LOC: 331)
`list.c`	One loop in FreeRTOS *list.c* with heap manipulation	4.124
`task.c`	Six loops in FreeRTOS *task.c* with heap manipulation	32.18

with the same length as a user input; the list *traverse* program does not change list's length; all elements of the return list of the *filter* program are smaller than or equal to the input value k; and the length of the return list of the *drop_even* program is between `half` and `half` $+ 1$ of the length of the original list.

Moreover, critical information may be required from some loops for their enclosing procedures to function correctly. For example, the quick-sort 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 where their length difference is at most one; such information might be unimportant for the algorithm's functional correctness but can be 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 outside the value bounds of the original tree. For code in FreeRTOS, the invariants we inferred maintain the sortedness property for the double linked list used for tasks. The invariants we discovered are sufficiently precise to prove the functional correctness of their corresponding programs with the given predicates.

6. Related work and conclusion

Related work. 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 (Distefano et al., 2006) 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 (Calcagno et al., 2009; Yang et al., 2008). The SLAyer tool (Gotsman et al., 2006) implements an inter-procedural analysis for programs with shape information. The Hob system (Lam, 2007) employs a set-based specification technique to describe heap objects and to reason about the programs, where a similar loop invariant inference algorithm is proposed. In their fixed-point iteration algorithm, the join operator is designed to find common conjuncts of pre- and post-states, and there is no need for a widening operator since it is designed for a simple heap abstract domain. Their analysis allows certain conjuncts to be dropped from the post-state to avoid too many iterations and to enforce termination. Our analysis is designed for a more complex abstract domain leveraging both shape and numerical information. It requires a widening operator to ensure convergence, and it avoids direct pruning on post-states in order to make inferred loop invariants as precise as possible. To deal with also size information (such as number of nodes in lists/trees), THOR (Magill et al., 2008) 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 (Gulwani et al., 2009) 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 (Chang and Rival, 2008). It employs inductive checkers to express both shape and numerical information. Our approach has two 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. For instance, they analyse programs which manipulate binary search trees and red-black trees without changing the variety of shapes 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).

There are also many other approaches that can synthesise shape-related program invariant, other than those based on separation logic. The shape analysis framework TVLA (Sagiv et al., 2002) is based on three-valued logic. It is capable of handling complicated data structures and properties, such as sortedness. Guo et al. (2007) report a global shape analysis that discover inductive structural shape invariants from the code. Kuncak et al. (2002) develop 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 (2005) 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 (Cousot and Cousot, 1977) and its applications such as automated assertion discovery (Cousot and Cousot, 2002; Kovacs and Jebelean, 2005; Leino and Logozzo, 2005; Furia and Meyer, 2010) 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 Popeea and Chin (2006).

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.

$s \models_A \gamma_1 \wedge \gamma_2$	iff	$s \models_A \gamma_1$ and $s \models_A \gamma_2$
$s \models_A p_1 \bowtie p_2$	iff	$s(p_1) \bowtie s(p_2)$, where $\bowtie \in \{=, \neq\}$
$s \models_A p \bowtie \texttt{null}$	iff	$s(p) \bowtie 0$, where $\bowtie \in \{=, \neq\}$
$s \models_A \texttt{true}$	always	
$s \models_A \texttt{false}$	never	
$s \models_A v$	iff	$s(v) = \texttt{true}$
$s \models_A b_1 = b_2$	iff	$s(b_1) = s(b_2)$
$s \models_A v_1 = v_2$	iff	$s(v_1) = s(v_2)$
$s \models_A v_1 \leqslant v_2$	iff	$s(v_1) \leqslant s(v_2)$
$s \models_A \phi_1 \wedge \phi_2$	iff	$s \models_A \phi_1$ and $s \models_A \phi_2$
$s \models_A \phi_1 \vee \phi_2$	iff	$s \models_A \phi_1$ or $s \models_A \phi_2$
$s \models_A \neg\phi$	iff	$s \models_A \phi$ does not hold
$s \models_A \exists v \cdot \phi$	iff	$s \models_A [k/v]\phi$ for some k
$s \models_A \forall v \cdot \phi$	iff	$s \models_A [k/v]\phi$ for all k

Fig. 8. The semantic model for numerical constraints.

Acknowledgements

This work was supported in part by EPSRC Projects EP/G042322 and MOE Project 2009-T2-1-063. We thank Florin Craciun for his invaluable comments on an earlier version of this paper and Steve Dunne for his help in proof reading.

Appendix A

A.1. Semantic for pure formulae

The semantic model for pure (numerical) formulae $s \models_A \pi$ is given in Fig. 8.

A.2. Operational semantics

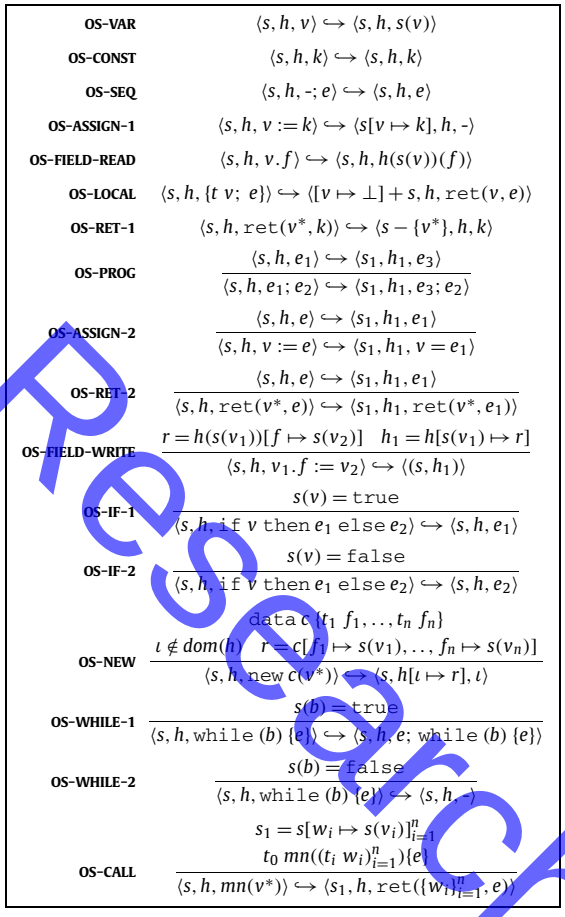
This section defines the operational semantics of our programming language. It is a small-step semantics which are essentially transitions between machine configurations. Each machine configuration is a triple consisting of:

- Heap h. As mentioned earlier we model heaps as finite partial maps from locations to object values. Object values are expected to conform to their defined class types.
- Stack s. Stacks are modelled as finite maps from variables to values. Note that a stack s is viewed as a "stackable" mapping, where a variable v may occur several times, and $s(v)$ always refers to the value of the variable v that was popped in most recently. A more formal definition for stacks would make different occurrences of the same variable with different "frame" numbers. We omit the details here.
- Current program code e. Program execution terminates when e is -, a value of type `void`.

Each reduction step can then be formalised as a small-step transition of the form:

$$\langle s, h, e\rangle \hookrightarrow \langle s_1, h_1, e_1\rangle$$

The full set of transitions is given in Fig. 9. We explain some of the notations used in them. The operation $[v \mapsto \nu] + s$ "pushes" the variable v to s with the value ν, and $([v \mapsto \nu] + s)(v) = \nu$. The operation $s - v^*$ "pops out" variables v^* from the stack s. $s[v \mapsto k]$ is a mapping which keeps all the mappings in s except that of v (which is now specified to be mapped to k). We also abuse this notation for a class type identifier c to denote a region of heap (mappings) in the form $c[f_1 \mapsto s(v_1), \ldots, f_n \mapsto s(v_n)]$, which is essentially a heap location where fields f_i are further mapped to values $s(v_i)$, $i = 1, \ldots, n$. $\perp$ represents an arbitrary value. We also introduce an intermediate construct as results returned by expressions/method calls $\texttt{ret}(v^*, e)$, where v^* will be dropped from s



$$\textbf{OS-VAR}\quad \langle s, h, v\rangle \hookrightarrow \langle s, h, s(v)\rangle$$

$$\textbf{OS-CONST}\quad \langle s, h, k\rangle \hookrightarrow \langle s, h, k\rangle$$

$$\textbf{OS-SEQ}\quad \langle s, h, \text{-}; e\rangle \hookrightarrow \langle s, h, e\rangle$$

$$\textbf{OS-ASSIGN-1}\quad \langle s, h, v := k\rangle \hookrightarrow \langle s[v \mapsto k], h, \text{-}\rangle$$

$$\textbf{OS-FIELD-READ}\quad \langle s, h, v.f\rangle \hookrightarrow \langle s, h, h(s(v))(f)\rangle$$

$$\textbf{OS-LOCAL}\quad \langle s, h, \{t\ v;\ e\}\rangle \hookrightarrow \langle [v \mapsto \bot] + s, h, \texttt{ret}(v, e)\rangle$$

$$\textbf{OS-RET-1}\quad \langle s, h, \texttt{ret}(v^*, k)\rangle \hookrightarrow \langle s - \{v^*\}, h, k\rangle$$

$$\textbf{OS-PROG}\quad \frac{\langle s, h, e_1\rangle \hookrightarrow \langle s_1, h_1, e_3\rangle}{\langle s, h, e_1; e_2\rangle \hookrightarrow \langle s_1, h_1, e_3; e_2\rangle}$$

$$\textbf{OS-ASSIGN-2}\quad \frac{\langle s, h, e\rangle \hookrightarrow \langle s_1, h_1, e_1\rangle}{\langle s, h, v := e\rangle \hookrightarrow \langle s_1, h_1, v = e_1\rangle}$$

$$\textbf{OS-RET-2}\quad \frac{\langle s, h, e\rangle \hookrightarrow \langle s_1, h_1, e_1\rangle}{\langle s, h, \texttt{ret}(v^*, e)\rangle \hookrightarrow \langle s_1, h_1, \texttt{ret}(v^*, e_1)\rangle}$$

$$\textbf{OS-FIELD-WRITE}\quad \frac{r = h(s(v_1))[f \mapsto s(v_2)] \quad h_1 = h[s(v_1) \mapsto r]}{\langle s, h, v_1.f := v_2\rangle \hookrightarrow \langle (s, h_1)\rangle}$$

$$\textbf{OS-IF-1}\quad \frac{s(v) = \texttt{true}}{\langle s, h, \texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2\rangle \hookrightarrow \langle s, h, e_1\rangle}$$

$$\textbf{OS-IF-2}\quad \frac{s(v) = \texttt{false}}{\langle s, h, \texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2\rangle \hookrightarrow \langle s, h, e_2\rangle}$$

$$\textbf{OS-NEW}\quad \frac{\texttt{data}\ c\ \{t_1\ f_1, \ldots, t_n\ f_n\} \quad \iota \notin dom(h) \quad r = c[f_1 \mapsto s(v_1), \ldots, f_n \mapsto s(v_n)]}{\langle s, h, \texttt{new}\ c(v^*)\rangle \hookrightarrow \langle s, h[\iota \mapsto r], \iota\rangle}$$

$$\textbf{OS-WHILE-1}\quad \frac{s(b) = \texttt{true}}{\langle s, h, \texttt{while}\ (b)\ \{e\}\rangle \hookrightarrow \langle s, h, e;\ \texttt{while}\ (b)\ \{e\}\rangle}$$

$$\textbf{OS-WHILE-2}\quad \frac{s(b) = \texttt{false}}{\langle s, h, \texttt{while}\ (b)\ \{e\}\rangle \hookrightarrow \langle s, h, \text{-}\rangle}$$

$$\textbf{OS-CALL}\quad \frac{s_1 = s[w_i \mapsto s(v_i)]_{i=1}^n \quad t_0\ mn((t_i\ w_i)_{i=1}^n)\{e\}}{\langle s, h, mn(v^*)\rangle \hookrightarrow \langle s_1, h, \texttt{ret}(\{w_i\}_{i=1}^n, e)\rangle}$$

Fig. 9. Operational semantics.

after the evaluation/invocation of e, to simulate the behaviour of stack. Whenever such a result is yielded, we assume it is stored in a special logical variable `res`, although `res` is never explicitly put in the stack s.

A.3. Proof of soundness of abstract semantics

In this section we prove the following lemma to complete the soundness proof for our analysis:

Lemma 4.2 *(Soundness of abstract semantics). If* $[e]_{\mathcal{T}}\Delta = \Delta_1$*, then for all* s*,* h*, if* $s, h \models \Delta$ *and* $\langle s, h, e\rangle \hookrightarrow \langle s_1, h_1, e_1\rangle$*, then there always exists* Δ_0 *such that*

$$s_1, h_1 \models \Delta_0 \quad \textit{and} \quad [e_1]_{\mathcal{T}}\Delta_0 = \Delta_1$$

Proof. The proof is done by structural induction over program constructors:

- Case `null` $\mid k \mid v \mid v.f$. Straightforward.
- Case $v := e$. There are two cases according to the operational semantics:
  - e is not a value. From operational semantics, there is e_1 s.t. $\langle s, h, e\rangle \hookrightarrow \langle s_1, h_1, e_1\rangle$, and $\langle s, h, v = e\rangle \hookrightarrow \langle s_1, h_1, v = e_1\rangle$. From abstract semantics for assignment, if $[e]_{\mathcal{T}}\Delta = \Delta_2$, and

$\Delta_1 = [v_1/v, r_1/\texttt{res}](\Delta_2) \wedge v = r_1$. By induction hypothesis, there exists Δ_0, $s_1, h_1 \models \Delta_0$ and $[e_1]_{\mathcal{T}}\Delta_0 = \Delta_2$. It concludes from the assignment rule that $[v = e_1]_{\mathcal{T}}\Delta_0 = \Delta_1$.

– e is a value. Trivial.

- Case $\texttt{new}\ c(v^*)$. From abstract semantics for new, we have $[\texttt{new}\ c(v^*)]_{\mathcal{T}}\Delta = \Delta_1$, where $\Delta_1 = \Delta * \texttt{res}::c\langle v'_1, \ldots, v'_n\rangle$. Let $\Delta_0 = \Delta_1$. From the operational semantics, we have $\langle s, h, \texttt{new}\ c(v^*)\rangle \hookrightarrow \langle s, h + [\iota \mapsto r], \iota\rangle$, where $\iota \notin dom(h)$. From $s, h \models \Delta$, we have $s, h + [\iota \mapsto r] \models \Delta_0$. Moreover, $[\iota]_{\mathcal{T}}\Delta_0 = \Delta_1$.
- Case $v_1.f := v_2$. Take $\Delta_0 = \Delta$. It concludes immediately from the exec rule for field update and the underlying operational semantics.
- Case $\texttt{free}(x)$. Denote Δ as $\bigvee_i (x::c\langle y_i^*\rangle * \sigma_i)$ and Δ_0 as $\bigvee_i \sigma_i$, then from free's operational semantics we know that if $s, h \models \Delta$ and $\langle s, h, \texttt{free}(x)\rangle \hookrightarrow \langle s_1, h_1, \text{-}\rangle$, then $s_1, h_1 \models \Delta_0$ and $\Delta_0 = \Delta_1$.
- Case $e_1; e_2$. We consider the case where e_1 is not a value (otherwise it is straightforward). From the operational semantics, we have $\langle s, h, e_1\rangle \hookrightarrow \langle s_1, h_1, e_3\rangle$. From the abstract semantics rule for sequence, we have $\vdash \{\Delta\}e_1\{\Delta_2\}$. By induction hypothesis, there exists Δ_0 s.t. $s_1, h_1 \models \Delta_0$, and $\vdash \{\Delta_0\}e_3\{\Delta_2\}$. By the sequential rule we have $[e_3; e_2]_{\mathcal{T}}\Delta_0 = \Delta_1$.
- Case $\texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2$. There are two possibilities in the operational semantics:

  – $s(v) = \texttt{true}$. We have $\langle s, h, \texttt{if}\ v\ \texttt{then}\ e_1\ \texttt{else}\ e_2\rangle \hookrightarrow \langle s, h, e_1\rangle$. Let $\Delta_0 = (\Delta \wedge v)$. It is obvious that $s, h \models \Delta_0$. From the if-conditional rule of abstract semantics, we have

  $$[e_1]_{\mathcal{T}}\Delta_0 = \Delta_2$$
  $$[e_2]_{\mathcal{T}}\Delta \wedge \neg v' = \Delta_3$$

  And we also have (due to sound weakening of postcondition)

  $$[e_1]_{\mathcal{T}}\Delta_0 = \Delta_2 \vee \Delta_3$$

  That is, $[e_1]_{\mathcal{T}}\Delta_0 = \Delta_1$.

  – $s(v) = \texttt{false}$. Analogous.
- Case $mn(v_{1\ldots n})$. For the method invocation rule, we know $\Delta \vdash \rho\Phi^i_{pr} * \Delta^i$, for $i = 1, \ldots, p$. Take $\Delta_0 = \bigvee_{i=1}^{p} \rho\Phi^i_{pr} * \Delta^i$. From the operational semantics and the above heap entailment, we have $s_1, h_1 \models \Delta_0$. Then the method invocation rule implies $\forall i \in 1 \ldots p \cdot [e_1]_{\mathcal{T}}\rho\Phi^i_{pr} * \Delta^i = \Delta^i * \Phi^i_{po}$. Therefore we have $[e_1]_{\mathcal{T}}\Delta_0 = \Delta_1$ which concludes. □

References

Berdine, J., Calcagno, C., O'Hearn, P., 2005. Smallfoot: Modular automatic assertion checking with separation logic. In: International Symposium on Formal Methods for Components and Objects. In: Lecture Notes in Computer Science, vol. 4111. Springer.

Berdine, J., Cook, B., Distefano, D., O'Hearn, P., 2006. Automatic termination proofs for programs with shape-shifting heaps. In: Computer Aided Verification. Springer, pp. 386–400.

Calcagno, C., Distefano, D., O'Hearn, P., Yang, H., 2009. Compositional shape analysis by means of bi-abduction. In: Proceedings of the ACM Symposium on Principles of Programming Languages (POPL). ACM Press, Savannah, GA, USA.

Chang, B.E., Rival, X., 2008. Relational inductive shape analysis. In: Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages.

Chin, W., David, C., Nguyen, H., Qin, S., 2007. Automated verification of shape, size and bag properties. In: Proc. 12th IEEE International Conference on Engineering Complex Computer Systems, pp. 307–320. http://dx.doi.org/10.1109/ICECCS.2007.17.

Chin, W., David, C., Nguyen, H., Qin, S., 2008. Enhancing modular OO verification with separation logic. In: Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '08. ACM, New York, NY, USA, pp. 87–99. http://doi.acm.org/10.1145/1328438.1328452.

Chin, W., David, C., Nguyen, H., Qin, S., 2012. Automated verification of shape, size and bag properties via user-defined predicates in separation logic. Science of Computer Programming 77 (9), 1006–1036. http://dx.doi.org/10.1016/j.scico.2010.07.004.

Cousot, P., Cousot, R., 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. ACM Press, New York, NY, Los Angeles, CA, pp. 238–252.

Cousot, P., Cousot, R., 2002. On abstraction in software verification. In: Proceedings of the 14th International Conference on Computer Aided Verification. Springer-Verlag, London, UK, pp. 37–56. http://portal.acm.org/citation.cfm?id=647771.734276.

Distefano, D., O'Hearn, P., Yang, H., 2006. A local shape analysis based on separation logic. In: Tools and Algorithms for the Construction and Analysis of Systems. In: Lecture Notes in Computer Science, vol. 3920. Springer.

The FreeRTOS™ project website, http://www.freertos.org, last visited: September 2012.
Furia, C.A., Meyer, B., 2010. Inferring loop invariants using postconditions. In: Fields of Logic and Computation. Springer-Verlag, Berlin, Heidelberg, pp. 277–300. http://portal.acm.org/citation.cfm?id=1983702.1983719.
Gotsman, A., Berdine, J., Cook, B., 2006. Interprocedural shape analysis with separated heap abstractions. In: Static Analysis Symposium 2006 (SAS'06).
Gulwani, S., Lev-Ami, T., Sagiv, M., 2009. A combination framework for tracking partition sizes. In: Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '09. ACM, New York, NY, USA, pp. 239–251. http://doi.acm.org/10.1145/1480881.1480912.
Guo, B., Vachharajani, N., August, D.I., 2007. Shape analysis with inductive recursion synthesis. In: Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation.
Hackett, B., Rugina, R., 2005. Region-based shape analysis with tracked locations. In: POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. ACM, New York, NY, USA, pp. 310–323. http://doi.acm.org/10.1145/1040305.1040331.
Ishtiaq, S., O'Hearn, P., 2001. Bi as an assertion language for mutable data structures. In: POPL '01: Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. ACM, New York, NY, USA, pp. 14–26. http://doi.acm.org/10.1145/360204.375719.
Kovacs, L., Jebelean, T., 2005. An algorithm for automated generation of invariants for loops with conditionals. In: Proceedings of the Seventh International Symposium on Symbolic and Numeric Algorithms for Scientific Computing. IEEE Computer Society, Washington, DC, USA, p. 245. http://dx.doi.org/10.1109/SYNASC.2005.19. http://portal.acm.org/citation.cfm?id=1114699.1115427.
Kuncak, V., Lam, P., Rinard, M., 2002. Role analysis. In: Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. ACM, pp. 17–32.
Lam, P., February 2007. The hob system for verifying software design properties. PhD thesis, Massachusetts Institute of Technology.
Leino, K., 2010. Dafny: An automatic program verifier for functional correctness. In: International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR). Citeseer.
Leino, K., Logozzo, F., 2005. Loop invariants on demand. In: Programming Languages and Systems: Third Asian Symposium, APLAS 2005, Proceedings. Tsukuba, Japan, November 2–5, 2005. Springer-Verlag, p. 119.
Magill, S., Tsai, M., Lee, P., Tsay, Y., 2008. Thor: A tool for reasoning about shape and arithmetic. In: Computer Aided Verification. Springer, pp. 428–432.
Nguyen, H.H., Chin, W., 2008. Enhancing program verification with lemmas. In: CAV '08: Proceedings of International Conference on Computer Aided Verification 2008. In: Lecture Notes in Computer Science. Springer.
Nguyen, H., David, C., Qin, S., Chin, W., 2007. Automated verification of shape and size properties via separation logic. In: VMCAI 2007: Proceedings of the 8th International Conference on Verification, Model Checking, and Abstract Interpretation. In: Lecture Notes in Computer Science, vol. 4349.
Parkinson, M., Bierman, G., 2008. Separation logic, abstraction and inheritance. In: Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '08. ACM, New York, NY, USA, pp. 75–86. http://doi.acm.org/10.1145/1328438.1328451.
Popeea, C., Chin, W., 2006. Inferring disjunctive postconditions. In: Proceedings of the 11th Asian Computing Science Conference on Advances in Computer Science: Secure Software and Related Issues. Springer-Verlag, pp. 331–345.
Pugh, W., 1991. The omega test: a fast and practical integer programming algorithm for dependence analysis. In: Proceedings of the 1991 ACM/IEEE Conference on Supercomputing, Supercomputing '91. ACM, New York, NY, USA, pp. 4–13. http://doi.acm.org/10.1145/125826.125848.
Reynolds, J.C., 2002. Separation logic: a logic for shared mutable data structures. In: Proceedings of the Seventeenth Annual IEEE Symposium on Logic in Computer Science, pp. 55–74, http://dx.doi.org/10.1109/LICS.2002.1029817.
Sagiv, M., Reps, T., Wilhelm, R., 2002. Parametric shape analysis via 3-valued logic. ACM Transactions on Programming Languages and Systems 24 (3), 217–298.
Venet, A., 1996. Abstract cofibered domains: Application to the alias analysis of untyped programs. In: Proceedings of the Third International Symposium on Static Analysis. Springer-Verlag, pp. 366–382.
Yang, H., Lee, O., Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P., 2008. Scalable shape analysis for systems code. In: Proceedings of the 20th International Conference on Computer Aided Verification, CAV '08. Springer-Verlag, Berlin, Heidelberg, pp. 385–398. http://dx.doi.org/10.1007/978-3-540-70545-1_36.