This full version, available on TeesRep, is the authors post-print of:


For the published version see:
10.1016/j.jsc.2012.08.007

When citing this source, please use the final published version as above.
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.

Keywords: loop invariant, fixpoint analysis, abstraction, combining analysis, shape analysis, numerical analysis, separation logic
PACS: 07.05.Bx, 89.20.Ff

1. Introduction

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

Since the emergence of separation logic [1, 2], dramatic advances have been made in automated software verification, e.g. the Smallfoot tool [3] for the verification on pointer safety (i.e. shape properties asserting that pointers cannot go wrong), the verification on termination [4], the verification for object-oriented programs [5, 6], and Dafny [7] and Hi/Sleek [8, 9, 10, 11] 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 [12, 13, 14], which, as a further step of Smallfoot, can automatically infer loop invariants as well as method specifications for pointer safety properties. The Hob system [15] offers a set-based analysis for loop invariant synthesis. Another tool THOR [16] 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 [17] 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 [9].

2.1. Specification mechanism

Separation logic [1, 2] extends Hoare logic to support reasoning about shared mutable data structures. One connective that it adds to classical logic is separation conjunction \( \ast \). The separation formula \( p_1 \ast p_2 \) 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

\[
\text{data node} = \{ \text{int val; node next} \}
\]

we can define a predicate for a singly linked list as

\[
\text{root}::\text{ll}(n) \equiv (\text{root}=\text{null} \land n=0) \lor (\exists v, q, m \cdot \text{root}::\text{node}(v, q) \ast \text{q}::\text{ll}(m) \land n=m+1)
\]

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 \( p::c(v_1, \ldots, v_k) \) is used for either a singleton heap or a predicate. If \( c \) is a data node with fields \( f_1, \ldots, f_k \), the notation represents a singleton heap, \( p::c[f_1 \mapsto v_1, \ldots, f_k \mapsto v_k] \) which says that the variable \( p \) points to a \( c \) data node and the value of its fields are \( v_1, \ldots, v_k \), e.g. root::node(v, q) in the above formula. If \( c \) is a predicate name, then the data structure pointed to by \( p \) has the shape \( c \) with parameters \( v_1, \ldots, v_k \), e.g. q::ll(m) above.
We can also define a singly linked list segment as follows:

\[ \text{ls}(p, n) \equiv (\text{root}=p \land n=0) \lor (\text{root}::\text{node}(_, q) \land q::\text{ls}(p, m) \land n=m+1) \]

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:

\[ \text{sll}(n, m, \text{mn}, \text{mx}) \equiv (\text{root}::\text{node}(\text{mn}, \text{null}) \land n=1 \land \text{mn}=\text{mx}) \lor \\
(\text{root}::\text{node}(\text{mn}, q) \land q::\text{sll}(n_1, k, \text{mx}) \land \text{mn}\leq k \land n=n_1+1) \]

\[ \text{sls}(p, n, m, \text{mn}, \text{mx}) \equiv (\text{root}::\text{node}(\text{mn}, p) \land n=1 \land \text{mn}=\text{mx}) \lor \\
(\text{root}::\text{node}(\text{mn}, q) \land q::\text{sls}(p, n_1, k, \text{mx}) \land \text{mn}\leq k \land n=n_1+1) \]

where \( \text{mn} \) and \( \text{mx} \) 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

\[ \text{sds}(p, q, n, \text{mn}, \text{mx}) \equiv (\text{root}::\text{node2}(\text{mn}, p, q) \land n=1 \land \text{mn}=\text{mx}) \lor \\
(\text{root}::\text{node2}(\text{mn}, p, r) \land r::\text{sds}(\text{root}, q, n-1, k, \text{mx}) \land \text{mn}\leq k) \]

based on the data structure definition

\[ \text{data} \text{ node2} \{ \text{int} \text{ val}; \text{node2} \text{ prev}; \text{node2} \text{ next}; \} \]

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 \( \text{sls} \), \( \text{mn} \) (respectively \( \text{mx} \)) is the minimum (respectively the maximum) value stored in the double sorted list.

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

2.2. Illustrative Example

```
0 data node { int val; node next; } 9 while (srt != null &&
     srt.val <= v) { 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 }
```

Figure 1: Insertion sort for linked list.

We now illustrate via an example our loop invariant synthesis process. The method \texttt{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.\footnote{The fixpoint iteration converges if one more iteration still yields the same result.} 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

$$r::sll(n_r, a, b) \land \text{cur::node}(\nu, x) \land x::sll(n_x) \land \text{srt}=r \land \text{prv}=null \land n_r+n_x+1=n$$

Note that since the inner loop does not mutate the heap part referred to by cur and x (i.e., cur::node(\nu, x) + x::sll(n_x)), we can ignore it during the invariant synthesis and add it back to the program state using the frame rule of separation logic [2]. Therefore, the initial state for loop invariant synthesis becomes

$$r::sll(n_r, a, b) \land \text{srt}=r \land \text{prv}=null \land n_r+n_x+1=n \quad (1)$$

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

$$r::\text{node}(a, \text{srt}) \land \text{srt::sll}(n_c, c_1, b) \land \text{prv}=r \land a \leq c_1 \land a \leq v \land n_r+1=n-n_x \land n_r+1=n_r \quad (2)$$

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

$$(r::sll(n_r, a, b) \land \text{srt}=r \land \text{prv}=null \land n_r+n_x+1=n) \lor$$

$$r::\text{node}(a, \text{srt}) \land \text{srt::sll}(n_c, c_1, b) \land$$

$$\text{prv}=r \land a \leq c_1 \land a \leq v \land n_r+1=n-n_x \land n_r+1=n_r) \quad (3)$$

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:

$$(3) \lor r::\text{node}(a, \text{prv}) \land \text{prv::node}(c_1, \text{srt}) \land \text{srt::sll}(n_s, c_2, b) \land$$

$$a \leq c_1 \land c_2 \land c_1 \leq v \land n_s+1=n-n_x \land n_s+2=n_r \quad (4)$$

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

$$(4) \lor r::\text{node}(a, r_b) \land r_b::\text{node}(c_1, \text{prv}) \land \text{prv::node}(c_2, \text{srt}) \land$$

$$\text{srt::sll}(n_s, c_3, b) \land a \leq c_1 \land c_2 \land c_3 \land c_2 \leq v \land n_r+1=n-n_x \land n_s+3=n_r$$

where we have an auxiliary logical variable r_b. 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:

$$(4) \lor r::sll(\text{prv}, n_t, a, c_1) \land \text{prv::node}(c_2, \text{srt}) \land \text{srt::sll}(n_s, c_3, b) \land$$

$$a \leq c_1 \land c_2 \land c_3 \land c_2 \leq v \land n_r+1=n-n_x \land n_s+3=n_r \land n_t=2$$
Note that the heap part \( r::\text{node}(a, r_a) \ast r_b::\text{node}(c_1, \text{prv}) \) is abstracted as a sorted list segment \( r::\text{sls}(\text{prv}, n_1, a, c_1) \) 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 \( a \leq c_1 \).

This fourth iteration results in a post-state where four nodes are passed by \( \text{srt} \), and three by \( \text{prv} \). An abstraction is performed to remove the newly created logical pointer variables. As a simplification of the presentation, let us denote with \( n \) a

Note also that the abstraction process has made use of the fact \( \text{shape} \). So the body of a method (e.g. \( \text{node} \) in Section 2), or predicate definitions \( \text{spred} \) (e.g. \( \text{ll}, \text{sls}, \text{sls} \) in Section 2.1), as well as method declarations \( \text{meth} \). The definitions for \( \text{spred} \) and \( \text{mspec} \) 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 \( d[y] \) represent resp. heap-insensitive and heap sensitive commands.

Our specification language (in Fig. 3) allows (user-defined) shape predicates \( \text{spred} \) to specify both shape and numerical properties. Note that \( \text{spred} \) are constructed with disjunctive constraints \( \Phi \) and numerical formulae \( \pi \). We require that the predicates be well-formed [11].

3. Language and Abstract Domain

To simplify presentation, we focus on a strongly-typed C-like imperative language in Fig. 2. The program \( \text{Prog} \) written in this language consists of declarations \( \text{decl} \), which can either be data type declarations \( \text{data} \) (e.g. \( \text{node} \) in Section 2), or predicate definitions \( \text{spred} \) (e.g. \( \text{ll}, \text{sls}, \text{sls} \) in Section 2.1), as well as method declarations \( \text{meth} \). The definitions for \( \text{spred} \) and \( \text{mspec} \) 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 \( d[y] \) represent resp. heap-insensitive and heap sensitive commands.

Our specification language (in Fig. 3) allows (user-defined) shape predicates \( \text{spred} \) to specify both shape and numerical properties. Note that \( \text{spred} \) are constructed with disjunctive constraints \( \Phi \) and numerical formulae \( \pi \). We require that the predicates be well-formed [11].

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

\[
\begin{align*}
(4) & \lor (\sigma \land n_2+3=n_r \land n_1=2) \lor (\sigma \land n_2+4=n_r \land n_1=3) \\
(5) & \lor (\sigma \land n_2+n_2=n_r-1 \land n_1\geq 2)
\end{align*}
\]

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 \( \text{cur}::\text{node}(v, x) \ast x::\text{ll}(n_2) \) added back, the analysis for the outer loop continues. Eventually, the following loop invariant is discovered for the outer loop:

\[
(x::\text{ll}(n_2) \land r=\text{null} \land n_2=n) \lor (r::\text{node}(a, \text{null}) \ast x::\text{ll}(n_2) \land n=n_2+1) \lor (r::\text{ll}(n_r, a, b) \ast x::\text{ll}(n_r) \land n=n_r+n_r \land n_r\geq 2)
\]

which allows us to verify the entire method successfully using \( \text{Hip/Sleek} \) verifier.
A conjunctive abstract program state, $\sigma$, is composed of a heap (shape) part $\kappa$ and a numerical part $\pi$, where $\pi$ consists of $\gamma$ and $\phi$ as aliasing and numerical information, respectively, and where $\rho$ 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 $\sigma$’s, denoted by $\Delta$ (and its set is recognised as $\mathcal{P}_{\text{SH}}$). An abstract state $\Delta$ can be normalised to a $\Phi$ form.

The memory model of our specification formula is adapted from what is given in the classical separation logic [1, 2], and our abstract domain is capable of handling user-defined shape predicates and related numerical properties described by this model. We assume sets $\text{Loc}$ of memory locations, $\text{Val}$ of primitive values (with 0 ∈ $\text{Val}$ denoting $\text{null}$), $\text{Var}$ of variables (program and logical variables), and $\text{ObjVal}$ 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 $\Delta$, with $s, h$ from the following concrete domains:

\[
s \in \text{Stacks} =_{df} \text{Var} \to \text{Val} \cup \text{Loc}
\]

\[
h \in \text{Heaps} =_{df} \text{Loc} \to_{fin} \text{ObjVal}
\]

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 $h_1 \sqcup h_2$ 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 isdatat(c) returns true only if c is a data node and issspred(c) returns true only if c is a shape predicate. The semantic model for pure formulae \( s \models_A \pi \) is left in the Appendix (Fig. A.8).

We use the separation logic prover S\textsc{leek} [11] to prove whether one abstract state \( \Delta' \) entails another one \( \Delta : \Delta' \vdash A \). Along with the proof, S\textsc{leek} 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 \( \Delta' \) are sufficiently precise to cover all nodes from the consequent \( \Delta \). 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

\[
\exists y \cdot x::\text{node}(vx, y) \land y::ll(n) \land x::ll(m) \vdash R
\]

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

\[
\exists y \cdot x::\text{node}(vx, y) \land y::\text{node}(vy, z) \land vx\leq vy \land x::\text{lls}(n, z, mn, mx) \vdash R
\]

the residue \( R \) can be generated as \( n=2 \land \text{mn}=vx \land \text{mx}=vy \land mn\leq mx \), 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 S\textsc{leek} 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:

\[
\Delta \preceq \Delta' \equiv_{df} \Delta' \vdash A \quad \text{for some } R
\]

We also denote this by \( \Delta' \succeq \Delta \). 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: \( T \) as the program environment with all the method specifications in the program, \( \Delta_{pre} \) as the precondition for the while loop (i.e. the abstract state before the loop starts), the while loop itself \( \text{while } b \{ e \} \), and an upper bound \( n \) on the number of shared logical variables we keep during analysis.
Fixpoint Computation in Combined Domain

**Input:** $T$, $\Delta_{\text{pre}}$, \textbf{while} $b \mid e \mid n$;

**Local:** $i := 0$; $\Delta_i := \Delta_{\text{pre}}$; $\Delta'_i := \Delta_i$;

1. \textbf{repeat}
2. $i := i + 1$;
3. $\Delta_i := \text{widen}^\dagger(\Delta_{i-1}, \text{join}^\dagger(\Delta_{i-1}, \Delta'_{i-1}))$;
4. $\Delta'_i := \text{abs}^\dagger([e]_T(\Delta_i \land b))$;
5. \textbf{if} $\Delta'_i = \text{false} \lor \text{cp\_no}(\Delta'_i) > n$
   \textbf{then} \textbf{return} \text{fail} \textbf{end if} \\
6. \textbf{until} $\Delta'_i = \Delta'_{i-1}$;
7. \textbf{return} $\Delta'_i$

Figure 5: Main analysis algorithm.

Our analysis is based on abstract interpretation [17] 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 ($\Delta_i$ and $\Delta'_i$). The starting state of the calculation is $\Delta_{\text{pre}}$. Among the two states here, the unprimed version $\Delta_i$ denotes the initial state before the $i^{th}$ execution of the loop body, and the primed one $\Delta'_i$ represents the result state after. Each iteration starts at line 1. Firstly we join together the initial state $\Delta_{i-1}$ of the previous iteration with the result state $\Delta'_{i-1}$ obtained in the previous iteration, and widen it against the state $\Delta_i$ (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 $\text{cp\_no}$) 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 $\Delta'_i$ is returned as the loop invariant.

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

4.1. Abstract Semantics

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

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

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

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

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

\(^2\)Note that our analysis uses lifted versions of these operations (indicated by $\dagger$), which will be explained in more details in Section 4.2.
where $\text{SH}[x]$ denotes the set of conjunctive abstract states in which each element has $x$ exposed as the head of a data node $(x:c(v'))$, and $\mathcal{P}_{\text{SH}[x]}$ contains all the (disjunctive) abstract states, each of which is composed by such conjunctive states. Here $\text{rearr}(x)$ 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 $\text{exec}(d[x])$. The third function defined for other (heap insensitive) commands $d$ does not require such exposure of $x$.

$$\text{isdatat}(c) \quad \sigma \vdash x:c(v') * \sigma'$$

$$\text{rearr}(x)\sigma =_{df} \sigma$$

$$\text{isspred}(c) \quad \sigma \vdash x:c(u') * \sigma' \quad \text{root}::c(v') \equiv \Phi$$

$$\text{rearr}(x)\sigma =_{df} \sigma' * [x/\text{root}, u'/v'] \Phi$$

As mentioned earlier, the test $\text{isdatat}(c)$ returns $\text{true}$ only if $c$ is a data node and $\text{isspred}(c)$ returns $\text{true}$ only if $c$ is a shape predicate.

The symbolic execution rules for heap-sensitive commands $d[x]$ (i.e. $x.f_i, x.f_i := w$, or $\text{free}(x)$) assumes that the rearrangement $\text{rearr}(x)$ has been done previously:

$$\text{isdatat}(c) \quad \sigma \vdash x:c(v_1,..,v_n) * \sigma'$$

$$\text{exec}(x.f_i)(T)\sigma =_{df} \sigma' * x:c(v_1,..,v_n) \land \text{res}=v_i$$

$$\text{isdatat}(c) \quad \sigma \vdash x:c(v_1,..,v_n) * \sigma'$$

$$\text{exec}(x.f_i := w)(T)\sigma =_{df} \sigma' * x:c(v_1,..,v_{i-1},w,v_{i+1},..,v_n)$$

$$\text{isdatat}(c) \quad \sigma \vdash x:c(u') * \sigma'$$

$$\text{exec}(\text{free}(x))(T)\sigma =_{df} \sigma'$$

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

$$\text{exec}(k)(T)\sigma =_{df} \sigma \land \text{res}=k$$

$$\text{exec}(x)(T)\sigma =_{df} \sigma \land \text{res}=x$$

$$\text{isdatat}(c)$$

$$\text{exec}(\text{new } c(v'))(T)\sigma =_{df} \sigma * \text{res}::c(v')$$

Note that the first three rules deal with constant ($k$), variable ($x$) and data node creation ($\text{new } c(v')$), respectively, while the last rule handles method invocation. In the last rule, the call site is ensured to meet the precondition of $\text{mn}$, as signified by $\sigma \vdash \rho \Phi_{pr} * \sigma_{fr}$, where $\sigma_{fr}$ is the frame part. In this case, the execution succeeds and the postcondition of $\text{mn} (\rho \Phi_{po})$ is added into the post-state.

A lifting function $\dagger$ is defined to lift $\text{rearr}$’s domain to $\mathcal{P}_{\text{SH}}$:

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

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

$$\text{exec}^\dagger(d)(T) \bigvee \sigma_i =_{df} \bigvee (\text{exec}(d)(T)\sigma_i)$$
Based on the transition functions above, we can define the abstract semantics for a program command $e$ as follows:

\[ [d[x]]_\Delta = \text{exec}^d([d[x]](\mathcal{T})) \circ \text{rearr}^d(x)_\Delta \]

\[ [d]_\Delta = \text{exec}^d([d](\mathcal{T})) \Delta \]

\[ [e_1; e_2]_\Delta = \text{exec}^d([e_1]_\Delta \circ [e_2]_\Delta \]

\[ [x := e]_\Delta = \text{exec}^d([x/x', r'/\mathcal{R}][e]_\Delta) \land x = x' \land \text{fresh logical } x', r' \]

\[ [\text{if } v \text{ then } e_1 \text{ else } e_2]_\Delta = \text{exec}^d([e_1]_{(v \land \Delta)} \lor [e_2]_{(-v \land \Delta)}) \]

which form the foundation for us to analyze 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 post-conditions 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:

\[
\text{abs}(x::\text{node}(\_), z_b) + z_b::\text{node}(\_, \text{null})) = x::\text{ll}(n) \land n = 2 \\
\text{abs}(x::\text{node}(\_), z_b) + y::\text{node}(z_b, z_b::\text{node}(\_, \text{null})) = -
\]

where both $x$ and $y$ are program variables, and $z_b$ is an existentially quantified logical variable. In the second case $z_b$ 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 $\text{abs}$ eliminates unimportant cutpoints (during analysis) to ensure termination. Its type is defined as follows:

\[
\text{abs} : \text{SH} \rightarrow \text{SH} \quad \text{Abstraction}
\]

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

\[
\begin{align*}
\text{abs} (\sigma \land x_0 = e) & \equiv_{df} \sigma[e/x_0] \quad \text{(Subst1)} \\
\text{abs} (\sigma \land e = x_0) & \equiv_{df} \sigma[e/x_0] \quad \text{(Subst2)} \\
x_0 \notin \text{Reach}(\sigma) & \Rightarrow \text{abs}: x_0::\text{c}(v^*) \land \sigma = \sigma \land \text{true} \quad \text{(Unreach)} \\
\text{isdata} (c_1) & \Rightarrow \text{abs}: p::c_1(v^*) \land \sigma_1 = \sigma_1 \land p::c_2(v^*_{\sigma_2}) \land \sigma_2 = \Phi \quad \text{(Abs)}
\end{align*}
\]

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 $x_0$ in $x::\text{node}(\_, \text{null}) + x_0::\text{node}(\_, \text{null})$ 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 $v^*$). 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 $\text{root}::\text{ll}(n)$, the following abstraction step can take place:

$$\text{abs}(x::\text{node}(\cdot, z_9) \ast z_9::\text{node}(\cdot, \text{null})) = x::\text{ll}(n) \land n=2$$

where $x$ is program variable and $z_9$ is logical variable. The function $\text{Reach}$ is defined as follows:

$$\text{Reach}(\sigma) =_{df} \bigcup_{v \in \text{Ev}(\sigma)} \text{ReachVar}(k \land \pi, v) \text{ where } \sigma := \exists u^* \cdot k \land \pi$$

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

$$\{} \{ v \} \cup \{ z_2 \mid \exists z_1, \pi_1 \cdot z_1 \in \text{ReachVar}(k \land \pi, v) \land \pi_1=(z_1=z_2 \land \pi_1) \land \text{isptr}(z_2) \} \cup \{ z_2 \mid \exists y_1, k_1 \cdot z_1 \in \text{ReachVar}(k \land \pi, v) \land k_1=(z_1::(\cdot, ..., \cdot) \ast k_1) \land \text{isptr}(z_2) \} \subseteq \text{ReachVar}(k \land \pi, v)$$

namely, it is composed of aliases of $v$ and pointer variables reachable from $v$. The predicate $\text{isptr}(x)$ checks if $x$ is a pointer variable. For example, $\text{ReachVar}(x::\text{node}(\cdot, z_9) \ast y::\text{node}(\cdot, z_0) \ast z_9::\text{node}(\cdot, p_0), (x)) = \{ x, z_8, p_0 \}$. 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 $\text{abs}$ to lift both its domain and range to disjunctive abstract states $\mathcal{P}_{\Sigma}$:

$$\text{abs}^+ \bigvee \sigma_i =_{df} \bigvee \text{abs}(\sigma_i)$$

which allows it to be used in the analysis.

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

$$\text{join}(\sigma_1, \sigma_2) =_{df} \text{let } \sigma'_1, \sigma'_2 = \text{rename}(\sigma_1, \sigma_2) \text{ in } \text{match } \sigma'_1, \sigma'_2 \text{ with } (\exists x'_1 \cdot k_1 \land \pi_1), (\exists x'_2 \cdot k_2 \land \pi_2) \text{ in } \text{if } k_1 \supset k_2 \ast \text{true then } \exists x'_1, x'_2 \cdot k_2 \land (\text{join}_k(\pi_1, \pi_2)) \text{ else if } k_2 \supset k_1 \ast \text{true then } \exists x'_1, x'_2 \cdot k_1 \land (\text{join}_k(\pi_1, \pi_2)) \text{ else } \sigma_1 \lor \sigma_2$$

where the $\text{rename}$ function prevents naming clashes among logical variables of $\sigma_1$ and $\sigma_2$, by assigning fresh names to logical variables with the same name in the two states. For example given a name clash on $x_9$, it may change states $\exists x_9 \cdot x_9=\theta$ and $\exists x_9 \cdot x_9=1$ to $\exists x_9 \cdot x_9=\theta$ and $\exists x_9 \cdot x_1=1$ instead. After this procedure it judges whether $\sigma_2$ is an abstraction of $\sigma_1$, 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 $\text{join}_n(\pi_1, \pi_2)$, the convex hull operator over numerical domain [18]. 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 $\Delta$ as follows:

$$\text{join}^+(\Delta_1, \Delta_2) =_{df} \text{match } \Delta_1, \Delta_2 \text{ with } (\lor, \sigma_1^+), (\lor, \sigma_2^+) \text{ in } \lor_{i,j} \text{join}(\sigma_1^i, \sigma_2^j)$$

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 $\text{widen}(\sigma_1, \sigma_2)$ is defined as

$$\text{widen}(\sigma_1, \sigma_2) \triangleq \text{df}
\begin{align*}
\text{let} \; \sigma'_1, \sigma'_2 & = \text{rename}(\sigma_1, \sigma_2) \text{ in} \\
\text{match} \; \sigma'_1, \sigma'_2 \text{ with } (\exists x^n \cdot \kappa_1 \land \pi_1), (\exists x^n \cdot \kappa_2 \land \pi_2) \text{ in} \\
\text{if } \kappa_1 \vdash \kappa_2 \land \text{true} \text{ then } \exists x^n, x^n \cdot \kappa_2 \land (\text{widen}(\pi_1, \pi_2)) \\
\text{else } \sigma_1 \lor \sigma_2
\end{align*}$$

where the $\text{rename}$ 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 $\sigma_2$ to be weaker than (that of) the first $\sigma_1$, so that the widening reflects the trend of such weakening from $\sigma_1$ to $\sigma_2$. In this case it applies the widening operation $\text{widen}(\pi_1, \pi_2)$ over the numerical domain [18].

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

$$\text{widen}^\lor(\Delta_1, \Delta_2) \triangleq \text{df} \; \text{match} \; \Delta_1, \Delta_2 \text{ with } (\lor_j, \sigma_j^1), (\lor_j, \sigma_j^2) \text{ in } \lor_j, \text{widen}(\sigma_j^1, \sigma_j^2)$$

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, $\Delta_0 = x::\text{node}(\_ x_9)\ast x_9::\text{node}(\_ \text{null})$ and $\Delta_1 = x::\text{node}(\_ x_9) \ast x_9::\text{node}(\_ x_{10}) \ast x_{10}::\text{node}(\_ \text{null})$. We would like to discover a sound approximation for both states. Firstly we perform abstractions on both to obtain two abstract states, say, $\Delta'_0 = x::\text{ll}(n_9) \land n_9=2$ and $\Delta'_1 = x::\text{ll}(n_9) \land n_9=3$. Then these two are joined together according to shape similarity to be $\Delta''_0 = x::\text{ll}(n_9) \land (n_9=2 \lor n_9=3)$, which transfers disjunction to the numerical domain. Finally the joined state is widened based on the first state $\Delta''_0$, yielding a state $x::\text{ll}(n_9) \land n_9\geq2$. It is a sound abstraction of both $\Delta_0$ and $\Delta_1$, 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:

$$s, h, e \leadsto s', h_1, e_1$$

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 $\cdot$). The transitive closure of the above transition relation is denoted as $\leadsto^*$. The full details of the operational semantics are given in the Appendix. 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 $\Delta$ denote the loop invariant synthesised by our analysis for a while loop $\text{while } b [e]$. The analysis is sound if for all $s, h$, such that $s, h \models \Delta$ and $s, h, e \leadsto^*(s', h', \cdot)$ (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 $\Delta_1$ and $\Delta_2$, 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 [9].
Lemma 4.2 (Soundness of abstract semantics). If $[e]_T \Delta = \Delta_1$, then for all $s, h$, if $s, h \models \Delta$ and $(s, h, e) \rightarrow (s_1, h_1, e_1)$, then there always exists $\Delta_0$ such that $s_1, h_1 \models \Delta_0$ and $[e_1]_T \Delta_0 = \Delta_1$

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

Lemma 4.3 (Soundness of abs). If $\text{abs}(\sigma) = \sigma'$, then $\sigma \geq \sigma'$.

Proof. The soundness proof of the second part of the two abstraction rules is trivial. For the Unreach rule, we can easily prove that $x_0 :: c(\nu') + \sigma + \text{true} + \sigma$, where $\sigma$ is the frame part of the entailment check. By the Frame Rule of separation logic, we only need to show that $x_0 :: c(\nu') + \text{true}$, 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 $\text{join}(\sigma_1, \sigma_2) = \sigma_j$, then we have $\sigma_1 \geq \sigma_j$ and $\sigma_2 \geq \sigma_j$.

Proof. Let $\sigma_1$ be $(\exists x_1 : \nu_1 \land \pi_1)$, and $\sigma_2$ be $(\exists x_2 : \nu_2 \land \pi_2)$. By the definition of the join operator, we have three cases:

- If $\sigma_1 \not\vdash k_1 \ast \text{true}$, we have $\sigma_j = (\exists x_1 : \nu_1 \land \pi_1) \land (\exists x_2 : \nu_2 \land \pi_2) \land \text{join}(\pi_1, \pi_2)$, then we need to show that $\exists x_1 : \nu_1 \land \pi_1 \land \exists x_2 : \nu_2 \land \pi_2 \land \text{join}(\pi_1, \pi_2)$, which are true because $\sigma_1 \not\vdash k_1 \ast \text{true}$ by condition, $k_1 \vdash k_2$ by separation logic, and $\pi_1 \vdash \text{join}(\pi_1, \pi_2)$ and $\pi_2 \vdash \text{join}(\pi_1, \pi_2)$ by the soundness of the convex hull operator over the numerical domains [18].

- If $\sigma_1 \vdash k_1 \ast \text{true}$, the soundness proof is similar to the first case.

- Otherwise, we have $\sigma_j = \sigma_1 \lor \sigma_2$. The soundness proof of this case is trivial.

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

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 [18]. 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 11 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 $\sigma$ is reducible if and only if $\text{abs}(\sigma) \not\vdash \sigma$. If $\text{abs}(\sigma) \vdash \sigma$, then $\sigma$ is irreducible, in which case we also say $\sigma$ is stabilised.

Lemma 4.7 (Termination of $\text{abs}$). For all state $\sigma$, the application of the four abstraction operations over $\sigma$ exhaustively (following the given order) will terminate in finite steps within a finite shape domain.

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

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

Proof. The proof follows the idea of the widening termination proof in cofibered domains [19]. Similar to the proof of termination of $\text{abs}$, the shape part of $\sigma_n$ will be stationary since the shape domain is finite. The termination of numerical part can be guaranteed by numerical join and widening [18]. Combining them together, $\sigma_n$ 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 [11] as the solver for entailment proofs, and Omega constraint solver [20] and Fixcalc solver [18] 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:

$\text{left}::\text{sll}(n_1, s_1, l_1) * \text{right}::\text{sll}(n_2, s_2, l_2) \land r = \text{null}$

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:

node append(node x, node y)
requires $x::\text{sll}(n_1, s_1, l_1) + y::\text{sll}(n_2, s_2, l_2) \land l_1 \leq s_2$
ensures $\text{res}::\text{sll}(n_1 + n_2, s_1, l_2)$;

By applying our analysis algorithm to the while loop in the program, we obtain the following loop invariant, which
1 node merge(node left, node right) 
2 requires \(\text{sll}(n_1, s_1, l_1)\) * right::\(\text{sll}(n_2, s_2, l_2)\)
3 ensures res::\(\text{sll}(n_3, s_3, l_3)\) \& \(n_3 = n_1 + n_2\) 
\& \(s_3 = \min(s_1, s_2) \& l_3 = \max(l_1, l_2)\);
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 tmp.next = null;
25 r = append(r, tmp);
26 return r;
27 }

Figure 6: Merge two sorted linked list.

contains seven different disjunctive branches, with each branch describing a different situation:

\[
\begin{align*}
\text{left::}\text{sll}(n_1, s_1, l_1) & \ast \text{right::}\text{sll}(n_2, s_2, l_2) \ast r = \text{null} & (7) \\
\lor \text{r::node}(s_1, \text{null}) & \ast \text{right::}\text{sll}(n_2, s_2, l_2) \ast \text{left} = \text{null} & \ast s_1 = l_1 & \ast s_1 < s_2 & (8) \\
\lor \text{r::node}(s_2, \text{null}) & \ast \text{left::}\text{sll}(n_1, s_1, l_1) \ast \text{right} = \text{null} & \ast s_2 = l_2 & \ast s_2 \leq s_1 & (9) \\
\lor r::\text{sll}(n_3, s_3, l_3) & \ast \text{right::}\text{sll}(n_2', s_2', l_2') \ast \text{left} = \text{null} & \ast n_3 = n_1 + n_2 - n_2' & \\
& \ast s_3 = \min(s_1, s_2) \ast l_3 = l_1 \leq l_2 & s_3' \ast (10) \\
\lor \text{r::}\text{sll}(n_3, s_3, l_3) & \ast \text{left::}\text{sll}(n_1', s_1', l_1') \ast \text{right} = \text{null} & \ast n_3 = n_2 + n_1 - n_1' & \\
& \ast s_3 = \min(s_1, s_2) \ast l_3 = l_2 \leq s_3' \ast (12) \\
\lor \text{r::}\text{sll}(n_3, s_3, l_3) & \ast \text{left::}\text{sll}(n_1', s_1', l_1') \ast \text{right::}\text{sll}(n_2', s_2', l_2') \ast \text{left} = \text{null} & \ast n_3 = n_1 - n_1' + n_2 - n_2' & \\
& \ast s_3 = \min(s_1, s_2) \ast l_3 \leq \min(s_1', s_2') \ast (13)
\end{align*}
\]

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

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

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).

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 [21]. 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
<table>
<thead>
<tr>
<th>Program</th>
<th>Function</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>List processing programs</td>
<td>(Total LOC: 232)</td>
<td></td>
</tr>
<tr>
<td>create</td>
<td>Creates a list with given length parameter</td>
<td>0.452</td>
</tr>
<tr>
<td>delete</td>
<td>Disposes a list</td>
<td>0.720</td>
</tr>
<tr>
<td>traverse</td>
<td>Traverses a list</td>
<td>0.636</td>
</tr>
<tr>
<td>length</td>
<td>Counts the length of a list</td>
<td>0.772</td>
</tr>
<tr>
<td>append</td>
<td>Appends two sorted lists</td>
<td>0.312</td>
</tr>
<tr>
<td>take</td>
<td>Takes the first n elements of a list, or itself</td>
<td>0.852</td>
</tr>
<tr>
<td>drop</td>
<td>Returns suffix of a list after the first n elements, or empty</td>
<td>0.844</td>
</tr>
<tr>
<td>reverse</td>
<td>Reverse the elements of the list, in place</td>
<td>1.032</td>
</tr>
<tr>
<td>filter</td>
<td>Drops the elements bigger than k of a list</td>
<td>1.182</td>
</tr>
<tr>
<td>lookup</td>
<td>Returns the first node whose values equals to k, or null</td>
<td>0.876</td>
</tr>
<tr>
<td>drop_even</td>
<td>Drops all the elements whose indexes are even</td>
<td>1.332</td>
</tr>
<tr>
<td>Sorting list processing programs (Total LOC: 178)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ins_sort(inner)</td>
<td>Inner loop of Fig. 1</td>
<td>0.824</td>
</tr>
<tr>
<td>ins_sort(outer)</td>
<td>Outer loop of Fig. 1</td>
<td>4.372</td>
</tr>
<tr>
<td>partition</td>
<td>Auxiliary operation used by Quick-sort</td>
<td>1.497</td>
</tr>
<tr>
<td>merge</td>
<td>Merges two sorted lists to be one sorted list</td>
<td>1.972</td>
</tr>
<tr>
<td>split</td>
<td>Divides a list into two sublists with length difference of at most one</td>
<td>0.354</td>
</tr>
<tr>
<td>select</td>
<td>Selects the smallest node of a list</td>
<td>0.692</td>
</tr>
<tr>
<td>select_sort</td>
<td>Outer loop of selection sort</td>
<td>4.892</td>
</tr>
<tr>
<td>Tree processing programs (Total LOC: 87)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>tree_search</td>
<td>Finds a node in a binary search tree</td>
<td>1.294</td>
</tr>
<tr>
<td>tree_insert</td>
<td>Inserts a node into a binary search tree</td>
<td>1.364</td>
</tr>
<tr>
<td>list2tree</td>
<td>Inserts nodes of a list into a binary search tree</td>
<td>5.176</td>
</tr>
<tr>
<td>FreeRTOS</td>
<td>(Total LOC: 331)</td>
<td></td>
</tr>
<tr>
<td>list.c</td>
<td>One loop in FreeRTOS list.c with heap manipulation</td>
<td>4.124</td>
</tr>
<tr>
<td>task.c</td>
<td>Six loops in FreeRTOS task.c with heap manipulation</td>
<td>32.18</td>
</tr>
</tbody>
</table>

Table 1: Selected Experimental Results.

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 confirms that Hn/Sleek can verify all these programs successfully when supplied with loop invariants discovered by our analysis. According to our experience, these experiments just require the bound of shared cutpoints be a reasonably small number, say no more than twice of the number of program variables. 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 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 \(\text{half}\) and \(\text{half}+1\) 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 quicksort algorithm partitions a list into three parts, where two are lists and the third just one node, whose value is exactly in the middle of that of the two other lists (*partition* in the table). We use a list bound predicate to indicate that fact which is successfully inferred by our analysis. We can also infer that the first loop of a mergesort (*split* in the table) can divide the list into two 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 [13] 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 [12, 14]. The SLayer tool [22] implements an inter-procedural analysis for programs with shape information. The Hob system [15] 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 [16] 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 [23] 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 [24]. 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 \(x::\text{ls}(m,y)+y::\text{ls}(y,n)\land n<\emptyset\) involving both a shared cutpoint \(y\) and a circled list \(y::\text{ls}(y,n)\land n<\emptyset\), 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 [25] is based on three-valued logic. It is capable of handling complicated data structures and properties, such as sortedness. Guo et al. [26] report a global shape analysis
that discover inductive structural shape invariants from the code. Kuncak et al. [27] 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 [28] 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 [17] and its applications such as automated assertion discovery [29, 30, 31, 32] 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 [18].

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

Acknowledgement. This work was supported 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 to Steve Dunne for his help in proof reading.

Reference

Appendix A. Appendix

Appendix A.1. Semantic for Pure Formulae

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

Appendix 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.

20
Each reduction step can then be formalised as a small-step transition of the form:

\[(s, h, e) \xrightarrow{\text{step}} (s_1, h_1, e_1)\]

The full set of transitions is given in Figure A.9. We explain some of the notations used in them. The operation \([v \mapsto v'] + s\) "pushes" the variable \(v\) to \(s\) with the value \(v\), and \(([v \mapsto v'] + s)(v) = v\). The operation \(s \mapsto v\) "pops out" variables \(v\) from the stack \(s\). \(s[v := k]\) 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 \(s(v_i), i = 1, \ldots, n\). \(\bot\) represents an arbitrary value. We also introduce an intermediate construct as results returned by expressions/method calls \(\text{ret}(v', e)\), where \(v'\) will be dropped from \(s\) 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 \(\text{res}\), although \(\text{res}\) is never explicitly put in the stack \(s\).

**Appendix 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 \([e]_T \Delta = \Delta_1\), then for all \(s, h, \text{if } s, h \models \Delta \text{ and } (s, h, e) \xrightarrow{\text{step}} (s_1, h_1, e_1)\), then there always exists \(\Delta_0\) such that

\[s_1, h_1 \models \Delta_0 \quad \text{and} \quad [e_1]_T \Delta_0 = \Delta_1\]

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

- Case \(\text{null} \mid k \mid v \mid v.f\). 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. \((s, h, e) \xrightarrow{\text{step}} (s_1, h_1, e_1)\), and \((s, h, v:=e) \xrightarrow{\text{step}} (s_1, h_1, v:=e_1)\). From abstract semantics for assignment, if \([e_1]_T \Delta = \Delta_2\), and \(\Delta_1 = [v_1/v, r_1/\text{res}]\Delta_2\). By induction hypothesis, there exists \(\Delta_0\), \(s_1, h_1 \models \Delta_0\) and \([e_1]_T \Delta_0 = \Delta_2\). It concludes from the assignment rule that \([v = e_1]_T \Delta_0 = \Delta_1\).
  - \(e\) is a value. Trivial.
OS-VAR  \( \langle s, h, v \rangle \mapsto \langle s, h, s(v) \rangle \)

OS-CONST  \( \langle s, h, k \rangle \mapsto \langle s, h, k \rangle \)

OS-SEQ  \( \langle s, h, e_1 \rangle \mapsto \langle s, h, e \rangle \)

OS-ASSIGN-1  \( \langle s, h, v := k \rangle \mapsto \langle s[v := k], h, - \rangle \)

OS-FIELD-READ  \( \langle s, h, v, f \rangle \mapsto \langle s, h, h(s(v))(f) \rangle \)

OS-LOCAL  \( \langle s, h, [t; e] \rangle \mapsto \langle [v \mapsto \bot] + s, h, \text{ret}(v, e) \rangle \)

OS-RET-1  \( \langle s, h, \text{ret}(v', k) \rangle \mapsto \langle s[v'], h, k \rangle \)

OS-PROG  \( \langle s, h, e_1; e_2 \rangle \mapsto \langle s_1, h_1, e_3; e_2 \rangle \)

OS-ASSIGN-2  \( \langle s, h, v := e \rangle \mapsto \langle s_1, h_1, v := e_1 \rangle \)

OS-RET-2  \( \langle s, h, \text{ret}(v', e) \rangle \mapsto \langle s_1, h_1, \text{ret}(v', e_1) \rangle \)

OS-FIELD-WRITE  \( \langle s, h, v_1.f := v_2 \rangle \mapsto \langle \langle s, h_1 \rangle \rangle \)

OS-IF-1  \( \langle s, h, \text{if } v \text{ then } e_1 \text{ else } e_2 \rangle \mapsto \langle s, h, e_1 \rangle \)

OS-IF-2  \( \langle s, h, \text{if } v \text{ then } e_1 \text{ else } e_2 \rangle \mapsto \langle s, h, e_2 \rangle \)

OS-NEW  \( \langle s, h, \text{new } c(v') \rangle \mapsto \langle s, h[l_i \mapsto r_i], i \rangle \)

OS-WHILE-1 \( \langle s, h, \text{while } (b) [e] \rangle \mapsto \langle s, h, e \rangle \)

OS-WHILE-2 \( \langle s, h, \text{while } (b) [e] \rangle \mapsto \langle s, h, \bot \rangle \)

OS-CALL \( \langle s, h, \text{call } (v') \rangle \mapsto \langle s, h, \text{ret}(w_1, e) \rangle \)

Figure A.9: Operational semantics.
• Case **new** \(c(v')\). From abstract semantics for **new**, we have \(\lbrack\text{new } c(v')\rbrack_T \Delta = \Delta_1\), where \(\Delta_1 = \Delta \ast \texttt{res}::c(v'_1, \ldots, v'_n)\). Let \(\Delta_0 = \Delta_1\). From the operational semantics, we have \(\langle s, h, \text{new } c(v') \rangle \leadsto (s, h+[i \mapsto v], i)\), where \(i \notin \text{dom}(h)\).

From \(s, h \models \Delta\), we have \(s, h+[i \mapsto r] \models \Delta_0\). Moreover, \(\lbrack i \rbrack_T \Delta_0 = \Delta_1\).

• 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 **free**\(\bar{x}\). Denote \(\Delta\) as \(\bigvee_i \langle x_i::c(v'_i) \ast \sigma_i \rangle\) and \(\Delta_0\) as \(\bigvee_i \sigma_i\), then from **free**'s operational semantics we know that if \(s, h \models \Delta\) and \(\langle s, h, \text{free}(\bar{x}) \rangle \leadsto (s_1, h_1, -)\), 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 \(\langle s, h, e_1 \rangle \leadsto (s_1, h_1, e_3)\). From the abstract semantics rule for sequence, we have \(\triangledown \{\Delta\} e_1 \Delta_2 \). By induction hypothesis, there exists \(\Delta_0\) s.t. \(s_1, h_1 \models \Delta_0\), and \(\triangledown \{\Delta_0\} e_3 \Delta_2\). By the sequential rule we have \(\lbrack e_1; e_2 \rbrack_T \Delta_0 = \Delta_1\).

• Case **if** \(v\) **then** \(e_1\) **else** \(e_2\). There are two possibilities in the operational semantics:

  - \(s(v) = \text{true}\). We have \(\langle s, h, \text{if } v \text{ then } e_1 \text{ else } e_2 \rangle \leadsto (s, h, e_1)\). Let \(\Delta_0 = \langle \Delta \wedge v \rangle\). It is obvious that \(s, h \models \Delta_0\).

    From the if-conditional rule of abstract semantics, we have:

    \[
    \lbrack e_1 \rbrack_T \Delta_0 = \Delta_2
    \]
    \[
    \lbrack e_2 \rbrack_T \Delta \wedge \neg v' = \Delta_3
    \]

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

    \[
    \lbrack e_1 \rbrack_T \Delta_0 = \Delta_2 \vee \Delta_3
    \]

    That is, \(\lbrack e_1 \rbrack_T \Delta_0 = \Delta_1\).

  - \(s(v) = \text{false}\). Analogous.

• Case \(\text{inv}(v_1, \bar{a})\). For the method invocation rule, we know \(\Delta \ast \rho \Phi'_{p_1} \ast \Delta'\), for \(i = 1, \ldots, p\). Take \(\Delta_0 = \bigvee_{i=1}^p \rho \Phi'_{p_i} \ast \Delta'\).

From the operational semantics and the above heap entailment, we have \(s_1, h_1 \models \Delta_0\). Then the method invocation rule implies \(\forall i \in 1 \ldots p \cdot \lbrack e_1 \rbrack_T \rho \Phi'_{p_i} \ast \Delta' = \Delta' \ast \Phi'_{p_i}\). Therefore we have \(\lbrack e_1 \rbrack_T \Delta_0 = \Delta_1\) which concludes.