Refining Existential Properties in Separation Logic Analyses

Matko Botinčan¹, Mike Dodds², and Stephen Magill³

¹ matko.botincan@gmail.com
² University of York, mike.dodds@york.ac.uk
³ stephen.magill@gmail.com

Abstract. In separation logic program analyses, tractability is generally achieved by restricting invariants to a finite abstract domain. As this domain cannot vary, loss of information can cause failure even when verification is possible in the underlying logic. In this paper, we propose a CEGAR-like method for detecting spurious failures and avoiding them by refining the abstract domain. Our approach is geared towards discovering existential properties, e.g. “list contains value x”. To diagnose failures, we use abduction, a technique for inferring command preconditions. Our method works backwards from an error, identifying necessary information lost by abstraction, and refining the forward analysis to avoid the error. We define domains for several classes of existential properties, and show their effectiveness on case studies adapted from Redis, Azureus and FreeRTOS.

1 Introduction

Abstraction is often needed to automatically prove safety properties of programs, but finding the right abstraction can be difficult. Techniques based on CEGAR (Counter-Example-Guided Abstraction Refinement) [11,20] can automatically synthesise an abstraction that is sufficient for proving a given property. Particularly successful has been the application of CEGAR to predicate abstraction [15], enabling automated verification of a wide range of (primarily control-flow driven) safety properties [1,8,19].

Meanwhile, separation logic has emerged as a useful domain for verifying shape-based safety properties [2,3,7,14,26]. Its success stems from its ability to compositionally represent heap operations. The domain of separation logic formulae is infinite, so to ensure termination, program analyses abstract them by applying a function with a finite codomain [13]. Although this approach has proved effective in practice, it does not provide a means to recover from spurious errors caused by over-abstraction.

This paper proposes a method for automated tuning of abstractions in separation logic analyses. Instead of a single abstraction, our method works with families of abstractions parameterised by multisets, searching for a parameter that makes the analysis succeed. Expanding the multiset refines the abstraction, i.e. makes it more precise.

Our method uses a forward analysis that computes a fixpoint using the current parameterised abstraction, and a backward analysis that refines this abstraction by expanding the multiset parameter. To identify the cause of the error and propagate that information backwards along the counter-examples, we use abduction, a technique for calculating sufficient preconditions of program commands [7]. We use the difference in
symbolic states generated during forward and backward analysis to select new elements to add to the multiset.

Our approach focuses on existential properties, where we need to track some elements of a data structure more precisely than the others. For example, we define the domain of “lists containing at least particular values” (where the multiset parameter specifies the values). In general, our approach works well for similar existential properties, e.g. “lists containing a particular subsequence”. Existential properties arise e.g. when verifying that key-value stores preserve values, or in structures that depend on sentinel nodes.

Our approach is complementary to standard separation logic shape analyses. It adds a new tool to the analysis toolbox, but it is not a general abstraction-refinement solution. In particular, universal properties such as “all list nodes contain a particular value” are not handled. This is a result of our analysis structure: when forward analysis fails, we look for portions of the symbolic state sufficient to avoid the fault, and seek to protect them from abstraction. This is intrinsically an existential process.

1.1 Related Work

As in Berdine et al. [4] we wish to automate the process of ‘tweaking’ shape abstractions. In [4], abstract counter-examples are passed to a SMT solver, which produces concrete counter-example traces. These traces determine so-called doomed states, conceptually the same as those singled out for refinement by our procedure. An advantage of our approach is that we use information from the failed proof to inform the abstraction refinement step, rather than exhaustively trying possible refinements as in [4]. This aside, the two approaches are largely complementary: [4] focuses on discovering shape refinements, while our work focuses on data properties.

Our approach operates lazily, but in contrast to lazy abstraction [19], we do not recompute the abstract post operator each time we refine the abstraction. The intermediate formulae we compute during backward analysis can be seen as interpolants [22], but rather than taking these directly for refining the abstraction, we use them to select new parameters which have the effect of refining the abstraction. Such automatic discovery of parameters for parameterised domains is similar to Naik et al. [23], however, instead of analyzing concrete tests, we analyze abstract counter-examples.

Our notion of an abstraction function is similar to widening [12]. However, refining the abstraction with the least upper bound (as Gulavani and Rajamani [17]) would not converge due to the presence of recursive data structures. [9] gives a widening for shape domains, but this widening does not account for data, nor explicitly track existential properties. Refinement with an interpolated widen [16], while similar to ours, is also not applicable as we do not work in a complete lattice that is closed under Craig interpolation. Shape analyses such as TVLA [24] have been adapted for abstraction refinement [5,21], however, we believe these approaches could not automatically handle verification of existential properties such as those in [5].

The refinement process in our approach assumes a parameterised domain of symbolic heaps which can be refined by augmenting the multiset of parameters. Compared to predicate abstraction, where the abstract domain is constructed and refined automatically, in our approach we first have to hand-craft a parameterised domain. In part this
reflects the intrinsic complexity of shape properties compared to properties verifiable by standard predicate abstraction.

Several authors have experimented with separation logic domains recording existential information about stored data, e.g. [25,10]. Some of these domains could be formulated in our multiset-parametric approach, and vice versa. However, our work differs in that we focus on automating the process of refining the abstraction.

2 Intuitive Description of Our Approach

We now illustrate how over-abstraction can cause traditional separation-logic analyses to fail, and how our approach recovers from such failures. Our running example, given in Figure 1, is a simple instance of the pattern where a value is inserted into an pre-existing data-structure, the data-structure is further modified, and the program then assumes the continued presence of the inserted value. Our code first constructs a linked list of arbitrary length (we use \( * \) for non-deterministic choice). It picks an arbitrary value for \( x \), and creates a node storing this value. It extends the list with arbitrarily more nodes. Finally, it searches for the node storing \( x \) and faults if it is absent.

Suppose our abstract domain consists of the predicates \( \text{emp} \), representing the empty heap, \( \text{node}(x, y, d) \), representing a linked list node at address \( x \) with next pointer \( y \) and data contents \( d \), and \( \text{list}(x, y) \), representing a non-empty list segment of unrestricted length starting at address \( x \) and ending with a pointer to \( y \). Nodes and list segments are related by the following recursive definition:

\[
\text{list}(x, y) \triangleq \text{node}(x, y, d') \lor (\text{node}(x, n', d') \ast \text{list}(n', y))
\]
(Primed variables—\(x', y', \) etc.—indicate logical variables that are existentially quantified). A traditional analysis, e.g. [13], starts with the pre-condition \(\text{emp}\) and propagates symbolic states over the control-flow graph (right of Fig. 1). Consider the execution of the program that adds a single node in the first while loop (node \(l_1\)) then adds \(x\) to the list, skips the second loop, and then searches for \(x\) (node \(l_8\)). Following the two list insertions (node \(l_5\)) we obtain symbolic state
\[
\text{node}(r, r', x) \ast \text{node}(r', \text{nil}, d')
\]
As is typical, assume the analysis applies the following abstraction step:
\[
\text{node}(r, r', x) \ast \text{node}(r', \text{nil}, d') \rightarrow \text{list}(r, \text{nil})
\]
That is, it forgets list length and data values once there are two nodes in the list. At the head of the third while-loop (node \(l_8\)) it unfolds \(\text{list}(r, \text{nil})\) back to the single-node case, yielding \(\text{node}(r, \text{nil}, x')\). Since this state is too weak to show that \(x = x'\), the path where \(\text{res}\) is not set to 1 appears feasible, and the analysis cannot prove \(\text{assert(\text{res}==1)}\).

Our solution. The analysis has failed spuriously because it has abstracted away the existence of the node containing \(x\). We cannot remove abstraction entirely, and we also cannot pick a tailored abstraction \textit{a priori}, because the appropriate abstraction is sensitive to the target program and the required safety property. Instead, we work with a parameterised family of abstractions. Starting with the coarsest abstraction, we modify its parameters based on spurious failures, automatically tailoring the abstraction to the property we want to prove.

For our example, we augment the domain with a family of predicates \(\text{list}(\cdot, \cdot, \{d\})\), representing a list where at least one node holds the value \(d\) (this domain is defined in §4). Upon failing to prove the program, our backwards analysis looks for extensions of symbolic states that \textit{would} satisfy \(\text{assert(\text{res}==1)}\), and so avoid failure. Technically, this is achieved by posing successive abduction queries along the counter-example path. If an extension is found, then the difference between the formulae from forward and backward analysis identifies the cause of the spurious failure. In our example, the analysis infers that the failure was due to the abstraction of the node storing \(x\). We refine the abstraction so nodes containing \(x\) are rewritten to \(\text{list}(\cdot, \cdot, \{x\})\), “remembering” the existence of \(x\). This suffices to prove the program correct.

3 Analysis Structure

Symbolic heaps. A symbolic heap \(\Delta\) is a formula of the form \(\Pi \land \Sigma\) where \(\Pi\) (the pure part) and \(\Sigma\) (the spatial part) are defined by:
\[
\Pi ::= \true \mid \false \mid e = e \mid e \neq e \mid p(\bar{e}) \mid \Pi \land \Pi
\]
\[
\Sigma ::= \text{emp} \mid \text{s(\bar{e})} \mid \Sigma \ast \Sigma
\]
Here \(e\) ranges over (heap-independent) expressions (built over program and logical variables), \(p(\bar{e})\) over pure predicates and \(s(\bar{e})\) over spatial predicates. Logical variables are (implicitly) existentially quantified; the set of all such variables in \(\Delta\) is denoted by \(\text{EVars}(\Delta)\). \(\Sigma_1 \ast \Sigma_2\) holds if the state can be split into two parts with disjoint domains, one satisfying \(\Sigma_1\) and the other \(\Sigma_2\). A disjunctive symbolic heap is obtained by combining symbolic heaps (both the pure and spatial part) with disjunction. We identify a disjunctive heap with the set of its disjuncts, and also denote such heaps with \(\Delta\). The set of all consistent symbolic (resp. disjunctive) heaps is denoted by \(\text{SH}\) (resp. \(\mathcal{P}(\text{SH})\)).
Abstract domain. Our abstract domain is the join-semilattice \( (P(SH)^T, \sqsubseteq, \sqcup, \top) \), where \( P(SH)^T \triangleq P(SH) \cup \{\top\} \), the partial order is given by the entailment relation \( \vdash \), the join \( \sqcup \) is disjunction, and the top element, \( \top \), represents error.

We assume a sound theorem prover that can deal with entailments between symbolic heaps, frame inference, and abduction queries (square brackets denote the computed portion of the entailment):

- \( \Delta_1 \vdash \Delta_2 * [\Delta_F] \) (frame inference): given \( \Delta_1 \) and \( \Delta_2 \), find the frame \( \Delta_F \) such that \( \Delta_1 \vdash \Delta_2 \sqcup \Delta_F \) holds;
- \( \Delta_1 * [\Delta_A] \vdash \Delta_2 \) (abduction): given \( \Delta_1 \) and \( \Delta_2 \), find the ‘missing’ assumption \( \Delta_A \) such that \( \Delta_1 * \Delta_A \vdash \Delta_2 \) holds.

Specifications and programs. We assume that each atomic command \( c \in \text{Cmd} \) is associated with a specification \( \{P\} c \{Q\} \), consisting of a precondition \( P \) and a postcondition \( Q \) in \( SH \) (in fact, our case studies use specifications expressed by using points-to and (dis)equalities only). We define \( \text{assume}(e) \triangleq \{\text{true}\} \{e\} \) and \( \text{assert}(e) \triangleq \{e\} \{e\} \).

Specifications are interpreted using standard partial correctness: \( \{P\} c \{Q\} \) holds iﬀ when executing \( c \) from a state satisfying \( P \), \( c \) does not fault, and if it terminates then the resulting state satisfies \( Q \). As is standard in separation logic, we also assume specifications are tight: \( c \) will not access any resources outside of the ones described in \( P \).

We represent programs using a variant of intra-procedural control-flow graphs [19] over the set of atomic commands \( \text{Cmd} \). A CFG consists of a set of nodes \( N \) containing distinguished starting and ending nodes \( \text{start}, \text{end} \in N \), and functions, \( \text{succ}: N \rightarrow P(N) \) and \( \text{cmd}: N \times N \rightarrow \text{Cmd} \), representing node successors and edge labels. All nodes either have a single successor, or all outgoing edges are labelled with command \( \text{assume}(e) \) for the condition \( e \) that must hold for that edge to be taken.

Forward and backward transfer. We define the abstract forward semantics of each atomic command \( c \) by a function \([c]: SH \rightarrow P(SH)^T\). The function \([c]\), fusing together rearrangement (materialisation) and symbolic execution [24][2][3][7], is defined using the frame rule, which allows any triple \( \{P\} c \{Q\} \) to be extended by an arbitrary frame \( \Delta_F \) that is not modified by \( c \):

\[
[c](\Delta) \triangleq \begin{cases} \top & \text{if } \Delta \vdash \Delta_F \text{ and } \Delta \vdash P * \Delta_F \\ \{Q * \Delta_F | \Delta \vdash P * \Delta_F\} & \text{otherwise.} \end{cases}
\]

When there is no \( \Delta_F \) such that \( \Delta \vdash P * \Delta_F \), the current heap \( \Delta \) does not satisfy the precondition \( P \) of the command, and so execution may result in an error. We assume that the prover filters out inconsistent heaps. Lifting disjunctions to sets on the left-hand side is justified by the disjunction rule of Hoare logic. We lift \([c]\) to a forward transfer function \( P(SH)^T \rightarrow P(SH)^T \) by mapping \( \top \) to \( \top \) and a set of symbolic heaps to the join of their \([c]\)-images.

We use abduction to transfer symbolic heaps backwards: given a specification \( \{P\} c \{Q\} \) and disjunctive symbolic heap \( \Delta \), if \( \Delta_A \) is such that \( Q * \Delta_A \vdash \Delta \) then \( \{P * \Delta_A\} c \{\Delta\} \), i.e., we can “push” \( \Delta \) backwards over \( c \) to obtain \( P * \Delta_A \) as a pre-state. This gives rise to a backward transfer function \( [c]^{-1}: P(SH) \rightarrow P(SH) \) defined by:

\[
[c]^{-1}(\Delta) \triangleq \text{choose}(\{P * \Delta_A | Q * \Delta_A \vdash \Delta\})
\]
The heuristic function \(\text{choose}(-)\) selects a ‘good’ abduction solution (there can be many, e.g. a trivial one, false). For some fragments best solutions are possible: e.g. the disjunctive points-to fragment with (dis)equalities \([7]\), a variation of which we use in our backward analysis. Along assume-edges we have \(\text{wp}([\text{assume}(e)], \Delta) = -e \lor \Delta\).

### 3.1 Forward Analysis, Abstraction Function, and Parametricity

Forward analysis attempts to compute an inductive invariant \(N \rightarrow \mathcal{P}(\text{SH})^\top\). It gradually weakens the strongest property by propagating symbolic heaps along CFG edges using the forward transfer, and joining the obtained \([e]\)-images at each CFG node. Since our abstract domain is infinite, and transfer functions are not necessarily monotone, forward propagation alone may not reach a fixpoint, or even converge towards one.

We call a pair \((\text{SH}, \text{abs})\) an analysis. To ensure termination, propagated symbolic heaps are abstracted into a finite set, and the propagation process is made inflationary. Abstraction is realised by a function \(\text{abs}: \text{SH} \rightarrow \text{CSH}\) whose codomain is a finite subset \(\text{CSH}\) of \(\text{SH}\). At each step, \(\text{abs}\) replaces the propagated symbolic heap with a logically weaker one in \(\text{CSH} \cup \{\top\}\). We require \(\text{abs}\) to be inflationary, i.e., that it soundly over-approximates symbolic heaps with respect to \(\vdash\). Making the propagation inflationary means that instead of computing the (least) fixed-point of the functional \(\Phi: (N \rightarrow \mathcal{P}(\text{SH})^\top) \rightarrow (N \rightarrow \mathcal{P}(\text{SH})^\top)\), we compute the inflationary fixed-point of the functional \(X \mapsto X \cup \Phi(X)\).

**Definition 1 (analysis comparison).** Let \(\text{abs}: \text{SH} \rightarrow \text{CSH}\) and \(\text{abs}': \text{SH} \rightarrow \text{CSH}'\) be abstraction functions. We say that \(\text{abs}'\) refines \(\text{abs}\), written \(\text{abs} \preceq \text{abs}'\), if \(\text{CSH} \subseteq \text{CSH}'\) and for every \(\Delta \in \text{SH}\), \(\text{abs}'(\Delta) \vdash \text{abs}(\Delta)\). We say that \((\text{SH}, \text{abs}')\) is more precise than \((\text{SH}, \text{abs})\) if \(\text{abs} \preceq \text{abs}'\).

In §3.2 and §4 we introduce families of analyses whose abstraction functions are parameterised by a multiset (such analyses are parametric in the sense of \([23]\)). For any such family \((\text{SH}, \text{abs}_S)_{S \in \mathcal{M}}\), where \(\mathcal{M}\) is some family of multisets and \(\text{abs}_S: \text{SH} \rightarrow \text{CSH}_S\), we require that if \(S \subseteq S'\) then \(\text{abs}_S \preceq \text{abs}_{S'}\).

### 3.2 Forward-Backward Abstraction Refinement Algorithm

We now define an intra-procedural version of our analysis formally (we believe it could be made inter-procedural without difficulty – see \([6]\)). Let \((\text{SH}, \text{abs}_S)_{S \in \mathcal{M}}\) be a family of analyses parameterised by a multiset. Our method for abstraction refinement starts with the analysis \((\text{SH}, \text{abs}_0)\), and iteratively refines the abstraction by adding terms to the multiset \(S\). The goal is to eventually obtain \(S\) such that using the analysis \((\text{SH}, \text{abs}_S)\) we can compute a sufficient inductive invariant.

**Forward analysis.** Algorithm\([1]\) shows a forward analysis from §3.1 extended with abstraction refinement. The algorithm computes a fixpoint by constructing an abstract reachability tree (ART). An ART is a tree \(T = (T, E, t_0) \in \text{ART}\) where \(T\) is the set of nodes, \(E\) the set of edges and \(t_0\) the root node. We write \(E_T\) to refer to the set of edges

\[\forall A, \Box \rightarrow (A, \Box)\text{ is inflationary if for every } a, \text{ we have } a \subseteq f(a).\]
associated with a particular ART $T$. Nodes in $T$ are of the form $(n, \Delta) \in \mathbb{N} \times \text{SH}$ and represent the abstract states visited during the fixpoint computation. We use the following functions to deal with the ART: $\text{parent}_T : T \setminus \{t_0\} \rightarrow T$ returning the unique parent of a node, $\text{depth}_T : T \rightarrow \mathbb{N}_0$ returning the length of the path from $t_0$ to $t$, and $\text{nodes}_\text{at}_T : \mathbb{N}_0 \rightarrow \mathcal{P}(T)$ returning the set of all nodes at the given depth. For $T = (T, E, t_0)$ and $T' = (T', E', t_0')$, we write $T \subseteq T'$ to indicate that $T$ is a subtree of $T'$, i.e., that $T \subseteq T'$, and $E \subseteq E'$, and $t_0 = t_0'$. We write $\text{cmd}(n, n')$ to represent the command labelling the edge between nodes $n$ and $n'$ and $\text{spec}(n, n')$ for the corresponding specification.

The algorithm iteratively propagates $\llbracket \cdot \rrbracket$:images of previously-computed abstract states along CFG edges, applies abstraction if the result is consistent, and joins each newly computed state with the previously-computed states at the same node. We store the invariant computed at each step using a map $\text{inv} : \text{ART} \rightarrow (\mathbb{N} \rightarrow \mathcal{P}(\text{SH})^\uparrow)$, reflecting the fact that the invariant at a control point can be recovered from the node labels of the ART. If we have $\text{nodes}_\text{at}_T(k) = \emptyset$ for the current depth $k$, then we have successfully computed an inductive invariant without reaching an error.

Suppose at some point the transfer function returns $\top$, i.e., the forward analysis fails to prove a property (e.g., a pure assertion or a memory safety pre-condition of a heap-manipulating command). This can happen due to either a true violation of the property, or a spurious error caused by losing too much information somewhere along the analysis. The algorithm then invokes Algorithm 2 [Refine] to check for feasibility of the error and, if it is spurious, to refine the abstraction.

**Backward analysis.** Algorithm [Refine] operates by backward analysis of abstract counter-examples. Rather than using weakest preconditions as in CEGAR, [Refine] uses abduction to propagate formulae backwards along an abstract counter-example and
Definition 2. An abstract counter-example is a sequence $(n_0, \Delta_0) \ldots (n_k, \Delta_k)$ with:
- $n_0 = \text{start}$ and for all $0 < i \leq k$, $n_i \in \text{succ}(n_{i-1})$;
- $\Delta_0 = \text{emp}$, for all $0 < i \leq k$, $\Delta_i \in \text{abs}_S(\llbracket \text{cmd}(n_{i-1}, n_i) \rrbracket(\Delta_{i-1}))$ and $\Delta_k = \top$.

Figure 2 shows the abstract counter-example for the error discussed in [2]. This is the sequence of symbolic heaps computed by the analysis on its way to the error. This counter-example covers the case where the list contains just one node. The error results from over-abstraction, which has erased the information that this node contains the value 0 (this can be seen in the last non-error state, $t_{13}$).

Refine begins by finding a resource or pure assumption sufficient to avoid the terminal error in the counter-example. Let $(n_0, \Delta_0) \ldots (n_k, \top)$ be an abstract counter-example with $\llbracket \text{cmd}(n_{k-1}, n_k) \rrbracket = \{P_k\} \cup_k \{Q_k\}$. Since $\llbracket c_k \rrbracket(\Delta_{k-1}) = \top$, $\Delta_{k-1}$ misses

\begin{verbatim}
Refine($T$, $S$; $M$) : $N_0 \times M$
begin
  $k := \text{depth}(T) - 1$;
  $t_{\text{curr}} := t$; $t_{\text{prev}} := \text{parent}(t)$;
  $\{P\}, \{\} := \text{spec}(t_{\text{prev}}, n, t_{\text{curr}}, n)$;
  Solve $t_{\text{prev}}.\Delta \ast [\Delta_A] \vdash P \ast \text{true}$;
  $\Delta_{\text{prev}} := t_{\text{prev}}.\Delta \ast \Delta_A$;
  path$_{\text{fwd}} := t_{\text{prev}}.\Delta$; path$_{\text{bwd}} := \Delta_{\text{prev}}$;
  while $k > 0$ do
    $t_{\text{curr}} := t_{\text{prev}}$; $t_{\text{prev}} := \text{parent}(t_{\text{curr}})$; $\Delta_{\text{curr}} := \Delta'_{\text{prev}}$;
    $\Delta_{\text{prev}} := \llbracket \text{cmd}(t_{\text{prev}}, n, t_{\text{curr}}, n) \rrbracket^{-}
    \Delta_{\text{curr}}$;
    path$_{\text{fwd}} := t_{\text{curr}}.\Delta$; path$_{\text{bwd}} := \Delta_{\text{curr}}$.
    if $t_{\text{prev}}.\Delta \vdash \Delta'_{\text{prev}}$ then
      $S := S \cup \text{SelectSymbols}(\text{path}_{\text{fwd}}, \text{path}_{\text{bwd}})$;
      Delete $t_{\text{curr}}$-subtree of $T$;
    end
    $k := k - 1$;
  end
throw "error";

Algorithm 2: Backward analysis of counter-example by abduction.
\end{verbatim}
some assumption required to satisfy $P_k$. To find this, **Refine** solves the following abduction query (line 6)—here $\Delta_{k-1}$ is a rearrangement of $\Delta_{k-1}$, for example to expose particular memory cells:

$$\Delta_{k-1} \ast [\Delta_A] \vdash P_k \ast \text{true}.$$  

The resulting symbolic heap $\Delta_A$ expresses resources or assumptions that, in combination with $\Delta_{k-1}$, suffice to guarantee successful execution of $c_k$. If $\Delta_A$ is false then $\Delta_{k-1} \ast [\Delta_A]$ is inconsistent; if this happens then the analysis will have to find a refinement under which $(n_{k-1}, \Delta_{k-1})$ can be proved to be unreachable.

Letting $\Delta'_{k-1} := \Delta_{k-1} \ast \Delta_A$, **Refine** computes a sufficient resource for the preceding state (line 11):

$$\Delta'_{k-2} := \text{cmd}(n_{k-2}, n_{k-1})\uparrow (\Delta'_{k-1}).$$

If $\Delta_{k-2} \vdash \Delta'_{k-2}$ then in the step from $\Delta_{k-2}$ to $\Delta_{k-1}$ a loss of precision has happened, and we use the additional information in $\Delta'_{k-1}$ to refine the abstraction (line 13). Otherwise, we continue pushing backwards, and generate $\Delta'_{k-3}, \Delta'_{k-4}$, etc.

Eventually, **Refine** either halts with $\Delta_i \vdash \Delta'_i$ for some $i \geq 0$, or in the last step obtains $\Delta_0 \not\vdash \Delta'_0$. In the former case, **Refine** invokes the procedure **SelectSymbols**, passing it the forward and the backward sequence of symbolic heaps leading to the error (line 14). The symbols it generates are added to the multiset $S$, refining the abstraction. In the latter case, we did not find a point for refining the abstraction, so **Refine** reports a (still possibly spurious) error (line 18). Note that in this case the computed heap $\Delta'_0$ is a sufficient pre-condition to avoid this particular abstract counter-example.

If **Refine** calls **SelectSymbols** to update the abstraction, it discards the current node and all its descendants from the ART (line 15). The ART below the refinement point will be recomputed in subsequent iterations using (possibly) stronger invariants.

**Theorem 1 (Soundness).** If the algorithm terminates without throwing an error, the computed map $invT$ is an inductive invariant not containing $\top$.

**Proof.** Refinement in Alg. 1 is achieved by augmenting $S$ with new elements selected by **SelectSymbols**. Since $\text{abs}_S \preceq \text{abs}_S'$ for $S \subseteq S'$, this is immediately sound. □

**Refinement heuristic.** **SelectSymbols** stands for some heuristic function which refines the abstraction. It takes two sequences, $\text{path}_{\text{fwd}}$ and $\text{path}_{\text{bwd}}$: the former is a path taken by the forward analysis from the $i$-th node of the counter-example to the error node (such that $\Delta_i \vdash \Delta'_i$ in Alg. 2), while the latter is a path sufficient to avoid the error. **SelectSymbols** tries to identify symbols present in the error-avoiding path that have been lost in the forward, overly-abstracted path. Conceptually, **SelectSymbols** can be seen as a simpler analogue of the predicate discovery heuristics [1,19] (it synthesizes only symbolic constants rather than predicates).

The heuristic in our implementation works by examining the syntactic structure of formulae $\Delta$ and $\Delta'$ for which $\Delta \vdash \Delta'$ has been established. The heuristic starts by identifying congruence classes of terms occurring in both formulæ and building a tree of equalities between program variables in each congruence class. Our separation logic prover preserves the common syntactic parts of $\Delta$ and $\Delta'$ by explicitly recording
substitutions, ensuring we can recover a mapping between common variables occurring in both formulae. The heuristic then exhaustively traverses equalities in the congruence classes for each term of $\Delta'$, and checks whether equalities can be used to strengthen $\Delta$ without making it inconsistent. Intuitively, because these equalities are mentioned in the calculated sufficient resource, they will likely be significant for program correctness. The variables in identified equalities are then used to strengthen the abstraction. We found this heuristic worked well in our case studies (see §).

Running example revisited. In § we saw a spurious error caused by over-abstracting values in the list. To fix this, we augmented the domain with predicates $\text{list}(\neg, \neg, \{d\})$, representing a list that has at least one node with value $d$. We now show the refinement step in this domain. The backward analysis begins by solving the abduction query

$$(\text{node}(x, \text{nil}, d') \land x = d \land t = \text{nil} \land \text{res} = 0) \land [\Delta'_{13}] \vdash \text{res} = 1 \land [\_]$$

This yields $\Delta'_{13} = \text{false}$ as the only solution. The analysis then generates the following sequence of symbolic heaps (we omit some for brevity). Compare with the abstract counter example in Fig. 2 here $\Delta'_5$ corresponds to node $t_5$:

$\Delta'_{12} = (t \neq \text{nil} \land \text{res} = 0 \land \text{true})$

$\Delta'_{10} = ((d = x \land \text{true}) \lor (t' \neq \text{nil} \land \text{res} = 0 \land \text{node}(t, t', d') \land \text{true}))$

$\Delta'_9 = ((\text{node}(t, t', x) \land \text{true}) \lor (t' \neq \text{nil} \land \text{res} = 0 \land \text{node}(t, t', d') \land \text{true}))$

$\Delta'_8 = ((\text{res} = 0 \lor t = \text{nil} \land \text{true}) \lor (\text{node}(t, t', x) \land \text{true}) \lor$

$\quad (t' \neq \text{nil} \land \text{res} = 0 \land \text{node}(t, t', d') \land \text{true}))$

$\Delta'_7 = (x = \text{nil} \land \text{true}) \lor (\text{node}(x, t', x) \land \text{true}) \lor (t' \neq \text{nil} \land \text{node}(x, t', d') \land \text{true}))$

$\Delta'_6 = (x = t' \land \text{true})$

The algorithm stops at $\Delta'_6$, since $\Delta_5 \vdash \Delta'_6$, and calls $\text{SelectSymbols}$ to augment the abstraction. Our implementation looks for equalities in each $\Delta'$ that can be used to strengthen $\Delta$. In this case, in $\Delta'_{10}$ the heuristic identifies $d = x$ to strengthen the corresponding $\Delta_{10} = \text{node}(x, \text{nil}, d') \land t = x \land \text{res} = 0 \land d = d'$. Thus the heuristic selects the variable $x$ to augment the abstraction’s multiset.

In the unrefined analysis, any predicate $\text{list}(\neg, \neg, S)$ will be abstracted to $\text{list}(\neg, \neg, \emptyset)$ (equivalent to $\text{list}(\neg, \neg)$). Adding $x$ to the multiset means that predicates of the form $\text{list}(\neg, \neg, \{x\})$ will be protected from abstraction. We restart the forward analysis from $t_5$. This time the error is avoided, and we obtain the following abstract states:

$t'_6 = (t_6, \text{list}(x, \text{nil}, \{x\})) \ldots$

$t'_9 = (t_9, \text{list}(x, \text{nil}, \{x\}) \land t = r \land \text{res} = 0)$

Executing from $t_9$ to $t_{10}$ gives two possible post-states: $\text{node}(r, r', d') \land \text{list}(r', \text{nil}, \{x\}) \land t = r \land \text{res} = 0 \land d = d'$ and $\text{node}(x, r', x) \land \text{list}(r', \text{nil}, \emptyset) \land t = r \land \text{res} = 0 \land d = x$. In fact, this refined abstraction suffices to prove the absence of errors on all paths, which completes the analysis. (Other examples may need multiple refinements)
We describe in detail linked lists with value refinement and sketch two other multiset families: linked lists with address refinement, and sorted linked lists with value refinement. Details for the latter two can be found in Appendix B. All three families are experimentally evaluated in Section 5.3.3.

Fig. 3. Recursive definition of the list predicate in domain $\Sigma^{\text{mls}}$.

\[
\Delta \land x' = e \xrightarrow{\Sigma^{\text{mls}}} \Delta[e/x']
\]
\[
\Delta \ast \sigma(x', e, \_ ) \xrightarrow{\Sigma^{\text{mls}}} \Delta \ast \text{true} \quad \text{if } x' \notin \text{EVars}(\Delta)
\]
\[
\Delta \ast \sigma_1(x', y', \_ ) \ast \sigma_2(y', x', \_ ) \xrightarrow{\Sigma^{\text{mls}}} \Delta \ast \text{true} \quad \text{if } x', y' \notin \text{EVars}(\Delta)
\]
\[
\Delta \ast \sigma_1(e_1, x', S_1) \ast \sigma_2(x', e_2, S_2) \xrightarrow{\Sigma^{\text{mls}}} \Delta \ast \text{list}(e_1, \text{nil}, \text{pr}_T(S_1 \cup S_2, IT))
\]
\[
\Delta \ast \text{true} \quad \text{if } x' \notin \text{EVars}(\Delta, e_1, e_2) \land \Delta \vdash e_2 = \text{nil}
\]
\[
\left(\Delta \ast \sigma_1(e_1, x', S_1) \ast \sigma_2(x', e_2, S_2) \ast \sigma_3(e_3, f, S_3)\right) \xrightarrow{\Sigma^{\text{mls}}} \left(\text{list}(e_1, e_2, \text{pr}_T(S_1 \cup S_2, IT)) \ast \Delta \ast \sigma_3(e_3, f, S_3)\right)
\]
\[
\Delta \ast \text{true} \quad \text{if } x' \notin \text{EVars}(\Delta, e_1, e_2, e_3) \land \Delta \vdash e_2 = e_3
\]

Fig. 4. Abstract reduction system $\xrightarrow{\Sigma^{\text{mls}}}$ defining the abstraction function $\text{.abs}^{\text{mls}}$. (In the rules, $\sigma, \sigma_i$ range over $\{\text{node}, \text{list}\}$. The pure assumption $IT$ is supplied by the analysis.)

4 Example Multiset-Parametric Analyses

We describe in detail linked lists with value refinement and sketch two other multiset families: linked lists with address refinement, and sorted linked lists with value refinement. Details for the latter two can be found in Appendix B. All three families are experimentally evaluated in Section 5.3.3.

Linked lists with value refinement is domain used in our running example (Fig. 2). List segments are instrumented with a multiset representing the lower bound on the frequency of each variable or constant. The abstraction function is parameterised by a multiset controlling which symbols are abstracted. By expanding the multiset, the preserved frequency bounds are increased, and so the abstraction is refined.

The domain $\text{SH}^{\text{mls}}$ contains spatial predicates node($\cdot$, $\cdot$, $\{d\}$) and list($\cdot$, $\cdot$, $S$) for all $S$ and $d \in S$. Here $x, y$ are locations, $d$ is a data value, $S$ is a multiset:

- node($x, y, \{d\}$) holds if $x$ points to a node whose next field contains $y$ and data field contains $d$, i.e., node($x, y, \{d\}$) $\triangleq$ $x \mapsto \{\text{next}: y, \text{data}: d\}$.
- list($x, y, S$) holds if $x$ points to the first node of a non-empty list segment that ends with $y$, and for each value $d \in \text{dom}(S)$, there are at least $S(d)$ nodes that store $d$.

The recursive definition of list in $\text{SH}^{\text{mls}}$ is shown in Fig. 3. We use these equivalences as folding and unfolding rules when solving entailment queries in $\text{SH}^{\text{mls}}$.

**Abstraction.** Let $T$ be a finite multiset of program variables and constants. In Fig. 4, we define a parametric reduction system $\xrightarrow{T}$, which rewrites symbolic heaps from...
lists developed by Distefano et al. [13, table 2].

family, we refine on the existence of particular values in a sorted list interval with examples where different data-structure invariants are needed. In our third analysis, we can apply the idea of value refinement to different basic domains, allowing us to deal

4.2 Sorted Linked Lists with Value Refinement

The final reduction rule replaces every predicate list(e, f, S) with the bounded predicate list(e, f, prT(S, Π)). The operator prT extracts the maximal subset of S such that no element appears more frequently than it does in T (modulo given pure assumptions Π). Let ∼Π be the equivalence relation x ∼Π y ≜ Π ⊢ x = y. Fix a representative for each equivalence class of ∼Π, and for a multiset S, denote by S/Π the multiset of ∼Π-representatives where the multiplicity of a representative x is ∑y∼Π y S(y). Writing x · n for a multiset element x occurring with multiplicity n, we define prT by

prT(S, Π) ≜ { x·n | x·k′ ∈ S/Π ∧ 3d′·m′ ∈ T/Π. Π ⊢ x = d′ ∧ n = min(k′, m′)}. As ∼Π has no infinite reduction sequences, it gives rise to an abstraction function absΠ by exhaustively applying the rules until none apply.

Lemma 1 (Finiteness). If T is finite and there are only finitely many program variables then the domain CSHΠT ≜ { Δ | Δ ∭ false ∧ Δ ∭ absΠ } is finite.

Lemma 2 (Soundness). As Δ ∼ΠT Δ′ implies Δ ⊢ Δ′, absΠT : SH → CSHΠT is a sound abstraction function.

Lemma 3 (Monotonicity). If T1 ⊆ T2 then absΠT1 ⊆ absΠT2.

4.1 Linked Lists with Address Refinement

Rather than preserving certain values in the list, we might need to preserve nodes at particular addresses.

For example, we might use the procedure shown on the right. Given pre-condition list(x, x) * node(x, n, _) * list(n, nil), the procedure will return 1. However, the standard list abstraction will forget the existence of the node pointed to by x, making this impossible to prove.

To preserve information of this kind, we combine the domain of linked lists, SHΠ, with a multiset refinement that preserves particular addresses. Because node addresses are unique, the domain contains just list and node predicates, rather than predicates instrumented with multisets. The reduction system ∼ΠT protects addresses in the multiset T from abstraction. As before, refinement consists of adding new addresses to the multiset.

4.2 Sorted Linked Lists with Value Refinement

We can apply the idea of value refinement to different basic domains, allowing us to deal with examples where different data-structure invariants are needed. In our third analysis family, we refine on the existence of particular values in a sorted list interval, rather than a simple segment. The domain SHΠs contains the predicate list_s, parameterised by an interval of the form [α, β], which stores the bounds of the values in the list, and a multiset S, which bounds on the frequency of particular values in the interval. The abstraction function ∼ΠT works in a similar way to SHΠs: the operator prT caps the frequency set S, limiting the number of values that are preserved by abstraction.

int remove(Node x) {
    ... // (border cases)
    p = hd; c = p->next;
    while (c!=nil) {
        if (c==x) {
            p->next = c->next;
            return 1;
        }
        p = c; c = p->next;
    }
    return 0;
}
equal sorted to \( x \) and \( \text{rem} \) put the operations del except used for the analysis, \( \bot \) where it threw an error with Experimental Results. Benchmarks verified by the analysis are marked with Table 1.

for example, by removing duplicate bindings to optimize for space usage. specifications across dynamic updates which may modify the data structures involved—following deletion a key is no longer bound in the store. Furthermore, we check these Redis [18]. They check various aspects of functional correctness—for example, that the order indicated by the benchmark name. The symbols \( \ast \), \( \ast \text{del} \) and \( \ast \text{r} \) respectively denote applying all operations any number of times with any argument, all operations except del, and all operations but excluding \( x \) as an argument. For Map benchmarks the operations put (insert a key-value pair), get (retrieve a value for the given key), rem (remove a key with the associated value) and bound (check if the key is bound) the no. symbolic states in the final ART, and \#Quer the no. queries sent to the prover.

5 Experimental Evaluation

We implemented Algorithm 1 and abstract domains \( \text{SH}^\text{mls} \), \( \text{SH}^\text{rls} \) and \( \text{SH}^\text{sls} \) in the separation logic tool coreStar [6]. Aside from superficial tweaks, we used an identical algorithm and SelectSymbols heuristic for all of our case studies. We used client-oriented specifications [13] describing data structures from Redis (a key-value store), Azureus (a BitTorrent client) and FreeRTOS (real time operating system). Table 1 shows results.

| No | Benchmark                                                                 | Result | Dom  | \(|T|\) | \(#\text{Refn}\) | \(#\text{ART}\) | \(#\text{Quer}\) |
|----|---------------------------------------------------------------------------|--------|------|--------|-----------------|----------------|----------------|
| Set | \( \text{add}(x) \rightarrow \ast \text{mem}(x) \)                      | \checkmark | \( \text{SH}^\text{mls} \) | 1          | 1          | 83            | 162            |
| 2  | \( \ast \text{add}(x) \rightarrow \ast \text{del} \rightarrow \ast \text{mem}(x) \) | \checkmark | \( \text{SH}^\text{mls} \) | 1          | 1          | 104           | 193            |
| 3  | \( \ast \text{del} \rightarrow \text{mem}(x) \rightarrow \ast \text{r} \rightarrow \ast \text{mem}(x) \) | \checkmark | \( \text{SH}^\text{mls} \) | 1          | 1          | 165           | 280            |
| 4  | \( \ast \text{add}(x) \rightarrow \text{all_eq} \rightarrow \text{to} \rightarrow x \) | \( \infty \) | \( \text{SH}^\text{mls} \) |             |             |                |                |
| 5  | \( \ast \text{add}(x) \rightarrow \text{all} \rightarrow \text{sorted} \) | \( \top \) | \( \text{SH}^\text{mls} \) |             |             |                |                |
| Multiset | \( \text{add}(x) \rightarrow \ast \text{add}(x) \rightarrow \text{del}(x) \rightarrow \text{mem}(x) \) | \checkmark | \( \text{SH}^\text{mls} \) | 2          | 1          | 67            | 91             |
| 7  | \( \ast \text{add}(x) \rightarrow \ast \text{del} \rightarrow \text{mem}(x) \) | \checkmark | \( \text{SH}^\text{mls} \) | 1          | 1          | 112           | 205            |
| 8  | \( \ast \text{del} \rightarrow \text{mem}(x) \rightarrow \ast \text{r} \rightarrow \text{mem}(x) \) | \checkmark | \( \text{SH}^\text{mls} \) | 1          | 1          | 171           | 312            |
| 9  | \( \ast \text{add}(x) \rightarrow \ast \text{del} \rightarrow \ast \text{add}(x) \rightarrow \ast \text{del} \rightarrow \text{del}(x) \rightarrow \text{mem}(x) \) | \checkmark | \( \text{SH}^\text{mls} \) | 2          | 2          | 219           | 458            |
| Map | \( \ast \rightarrow \text{put}(k, v) \rightarrow \ast \rightarrow \text{get}(k) \) | \checkmark | \( \text{SH}^\text{mls} \) | 1          | 1          | 118           | 215            |
| 11 | \( \ast \rightarrow \text{rem}(k) \rightarrow \text{bound}(k) \) | \checkmark | \( \text{SH}^\text{mls} \) | 1          | 1          | 92            | 168            |
| ByteBufferPool | 12 Property 1 | \checkmark | \( \text{SH}^\text{rls} \) | 1          | 1          | 154           | 231            |
| 13 | Property 2 | \checkmark | \( \text{SH}^\text{rls} \) (1) | 1          | 1          | 189           | 270            |
| 14 | Property 3 | \checkmark | \( \text{SH}^\text{rls} \) (2) | 4          | 1          | 316           | 511            |
| FreeRTOS list | 15 Property 4 | \checkmark | \( \text{SH}^\text{mls} \) | 1          | 1          | 91            | 158            |
| 16 | Property 5 | \checkmark | \( \text{SH}^\text{mls} \) | 6          | 5          | 425           | 971            |

Table 1. Experimental Results. Benchmarks verified by the analysis are marked with \( \checkmark \), those where it threw an error with \( \top \) and those where it did not terminate with \( \infty \). Dom is the domain used for the analysis, \( |T| \) is the size of the multiset \( T \) after the final refinement (number in parentheses denotes the size of the minimal sufficient \( T \)), \#Refn is the no. of refinement steps, \#ART the no. symbolic states in the final ART, and \#Quer the no. queries sent to the prover.
are to a list-based map. For benchmarks 1,2,6,7,9,10 the goal was to prove that the last operation returns true; for benchmark 11 that it returns false; and, for benchmarks 3 and 8 that the two mem operations return the same value. Benchmark 4 illustrates a universal property that causes our analysis in $SH^{\text{mls}}$ to loop forever by adding $x$ to $T$ at each refinement step. Benchmark 5 is a universal property for which our analysis in $SH^{\text{mls}}$ fails to find an inductive invariant due to the ordering predicate (using $SH^{\text{mls}}$ on the same benchmark loops forever).

**ByteBufferPool.** Azureus uses a pool of ByteBuffer objects to store results of network transfers. In early versions, free buffers in this pool were identified by setting the buffer position to a sentinel value. The **ByteBufferPool** benchmarks check properties of this pool. Property 1 checks that if the pool is full and a buffer is freed, that just-freed buffer is returned the next time a buffer is requested. Property 2 checks that if the pool has some number of free buffers, then no new buffers are allocated when a buffer is requested. Property 3 checks that if the pool has at least two free buffers, then two buffer requests can be serviced without allocating new buffers.

**FreeRTOS list.** This is a sorted cyclic list with a sentinel node, used task management in the scheduler. The value of the sentinel marks the end of the list—for instance, on task insertion the list is traversed to find the right insertion point and the guard for that iteration is the sentinel value. To check correctness of the shape after insertion (Property 4) it suffices to remember that the sentinel value is in the list. To check that tasks are also correctly sorted according to priorities (Property 5) we need to keep track of list sortedness and all possible priorities as splitting points.

### 6 Conclusions and Limitations

We have presented a CEGAR-like abstraction refinement scheme for separation logic analyses, aimed at refining existential properties of programs, in which we want to track some elements of a data structure more precisely than others.

Our prototype tool is built on coreStar [6], and we expect our approach would combine well with other separation logic tools, e.g. [7,3]. In particular, abduction is known to work well in an inter-procedural setting [7] and we thus believe our approach could be made inter-procedural without substantial further research.

Minimizing incompleteness is more challenging, as without further assumptions Algorithm [1] might diverge, or fail to recognize a spurious counter-example as infeasible. If the forward transfer function is exact (i.e., returns the strongest post-condition) and the backward transfer function is precise (i.e., for any $c$ and $\Delta$, $[c]([c]^{-1} (\Delta)) \vdash \Delta$) then the algorithm makes progress relative to the refinement heuristic. Intuitively, if SelectSymbols always picks a symbol such that the refined abstraction rules out the spurious counter-example, then that counter-example will never reappear in subsequent iterations. However, we are skeptical that our current heuristic satisfies this condition. For a more formal discussion, see Appendix A.

Note that Berdine et al. [4] similarly do not establish progress for their analysis. Predicate abstraction techniques that do not a priori fix the set of predicates have the same issue, as do interpolation-based procedures that do not constrain the language of acceptable interpolants. In both cases, the restrictions that ensure termination also limit the set of programs that can be proved correct.
References

1. T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In SPIN, 2001.
2. J. Berdine, C. Calcagno, and P. W. O’Hearn. Symbolic execution with separation logic. In APLAS, 2005.
3. J. Berdine, B. Cook, and S. Ishtiaq. Slayer: Memory safety for systems-level code. In CAV, 2011.
4. J. Berdine, A. Cox, S. Ishtiaq, and C. M. Wintersteiger. Diagnosing abstraction failure for separation logic-based analyses. In CAV, 2012.
5. D. Beyer, T. A. Henzinger, and G. Théoduloz. Lazy shape analysis. In CAV, 2006.
6. M. Botinčan, D. Distefano, M. Dodds, R. Grigore, D. Naudžiūniene, and M. Parkinson. coreStar: The Core of jStar. In Boogie, 2011.
7. C. Calcagno, D. Distefano, P. W. O’Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. J. ACM, 58(6), 2011.
8. S. Chaki, E. M. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. In ICSE, 2003.
9. B. Y. E. Chang, X. Rival, and G. C. Necula. Shape analysis with structural invariant checkers. In SAS, pages 384–401, 2007.
10. W.-N. Chin, C. David, H. H. Nguyen, and S. Qin. Automated verification of shape, size and bag properties via user-defined predicates in separation logic. In Science of Computer Programming, volume 77:9, 2012.
11. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In CAV, 2000.
12. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In POPL, 1977.
13. D. Distefano, P. W. O’Hearn, and H. Yang. A local shape analysis based on separation logic. In TACAS, 2006.
14. D. Distefano and M. J. Parkinson. jStar: towards practical verification for Java. In OOPSLA, 2008.
15. S. Graf and H. Sáidi. Construction of abstract state graphs with PVS. In CAV, 1997.
16. B. S. Gulavani, S. Chakraborty, A. V. Nori, and S. K. Rajamani. Automatically refining abstract interpretations. In TACAS, 2008.
17. B. S. Gulavani and S. K. Rajamani. Counterexample driven refinement for abstract interpretation. In TACAS, 2006.
18. C. M. Hayden, S. Magill, M. Hicks, N. Foster, and J. S. Foster. Specifying and verifying the correctness of dynamic software updates. In VSTTE, 2012.
19. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In POPL, 2002.
20. R. P. Kurshan. Computer-aided verification of coordinating processes: the automata-theoretic approach. Princeton University Press, 1994.
21. A. Loginov, T. W. Reps, and S. Sagiv. Abstraction refinement via inductive learning. In CAV, 2005.
22. K. L. McMillan. Lazy abstraction with interpolants. In CAV, 2006.
23. M. Naik, H. Yang, G. Castelnuovo, and M. Sagiv. Abstractions from tests. In POPL, 2012.
24. S. Sagiv, T. W. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. TPLS, 24(3), 2002.
25. V. Vafeiadis. Shape-value abstraction for verifying linearizability. In VMCAI, 2009.
26. H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. W. O’Hearn. Scalable shape analysis for systems code. In CAV, 2008.
Without further assumptions, the abstraction refinement algorithm might diverge, or report a spurious counter-example which is in fact not feasible. The following idealised assumptions suffice to ensure progress and completeness (we are skeptical that condition (c) holds for our current realisation of the analysis—see below).

(a) The forward transfer function is exact (i.e., $\llbracket J \rrbracket$-image is the strongest post-condition in the given abstract domain).
(b) The backward transfer function is precise (so we are able to identify spurious counter-examples). Formally, for any $c$ and $\Delta$, we have $\llbracket c \rrbracket (\llbracket c \rrbracket^+(\Delta)) \vdash \Delta$.
(c) When called with a $(p_{fwd}, p_{bwd})$-pair of the counter-example and the path sufficient to avoid the error, the procedure call $\text{SelectSymbols}(p_{fwd}, p_{bwd})$ picks symbols $A$ for augmenting $S$ such that the spurious counter-example ending with $p_{fwd}$ is eliminated by the abstraction $\text{abs}_{S \cup A}$.

Alg. 1 then makes progress by ensuring that a counter-example, once eliminated, remains eliminated in all subsequent iterations.

**Theorem 2 (Relative progress).** Let $\gamma_j$ be the counter-example processed in the $j$-th refinement step. Then for all $j \geq 1$, $|\gamma_j| \leq |\gamma_{j+1}|$, where $| \cdot |$ denotes the length of the counter-example. In addition, if $\gamma_j$ is processed with value $k$ in the while-loop on line 2 of Alg. 1 then the program being analysed has no counter-examples of length less than $k$.

**Proof.** Let $S(j)$ denote the multiset from the $j$-th iteration of $\text{Refine}$. Since $\text{abs}_{S(j)} \preceq \text{abs}_{S(j+1)}$, no new counter-examples can appear in the part of the ART that is recomputed in the $(j+1)$-th step (invariants computed in $\text{CSH}_{S(j+1)}$ will be at least as strong as those in $\text{CSH}_{S(j)}$). Since (c) guarantees that the previous counter-example has been eliminated, if a new counter-example is found then the corresponding value of $k$ in the while-loop will be either the same as in the $j$-th step or larger. $\square$

**Theorem 3 (Relative completeness).** If the safety property is implied by an inductive invariant expressible in $\text{CSH}_S$ for some finite multiset $S$ and assuming that those elements would eventually be selected from counter-examples by $\text{SelectSymbols}$ then Alg. 1 terminates without throwing an error.

**Proof.** Since Alg. 1 proceeds in a breadth-first fashion and counter-examples to safety properties are finite, all counter-examples leading to picking elements of $S$ will eventually be processed, enabling Alg. 1 to compute an invariant in $\text{CSH}_{S'}$, for some $S' \supseteq S$. $\square$

Assumptions (a) and (b) can be satisfied (although for implementation efficiency we may choose not to). Assumption (c) is more problematic.

**Forward transfer.** Without exactness, a spurious counter-example may never be eliminated, because our analysis refines only the abstraction function. Since separation logic analyses effectively calculate strongest post-conditions, we in fact have exact forward transfer, meaning spurious counter-examples can always be eliminated.

---

5 modulo deallocation—although even for that case the forward transfer is tight in actual implementations.
Backward transfer. In our analysis abduction is performed on finite unfoldings of predicates, modulo an arbitrary frame, fixed along the counter-example. As a result, counter-examples are always expressed as data-structures of a particular size (rather than e.g. general lists which could be of any size). This means that counter-examples can be expressed in the points-to fragment of separation logic, in which optimal solutions are possible [7]. Thus in principle we can satisfy (b) and make backward transfer precise. However, such a complete abductive inference is of exponential complexity since it has to consider all aliasing possibilities. In our implementation, we use a polynomial heuristic algorithm (similar to [7]) which may miss some solutions, but in practice has roughly the same cost as frame inference.

Selecting symbols. Due to its heuristic nature, it is unlikely that our implementation of SelectSymbols satisfies assumption (c). Furthermore, we are unsure whether it is generally possible to construct SelectSymbols that would satisfy (c) for an arbitrary parametric domain. While at least in principle we could employ a trivial heuristic which enumerates all multisets of symbols, that would be impractical. The problem of picking symbols which are certain to eliminate a particular counter-example seems uncomfortably close to selecting predicates for predicate abstraction sufficient to prove a given property. Many effective heuristics used in this area are incomplete (in that they may fail to find an adequate set of predicates when one exists), and there has been only a limited progress in characterising complete methods[6]. Unfortunately, all such complete predicate refinement methods rely on interpolation, a luxury which we do not (yet) have in separation logic. More work is needed to understand the intrinsic complexity of ways for doing refinement in separation logic analyses such as the one proposed in this paper in relation to the logical properties of separation logic domains.

B Details of Other Multiset-Parametric Domains

Here we give detailed definitions of the two analysis families that we sketched in §4.

B.1 Linked Lists with Address Refinement

This analysis allows refinement on protecting particular addresses, rather than values. We work with the domain of linked lists, which we denote $\text{SH}^{\text{rls}}$, built from plain spatial predicates node and list.

Our abstraction works similarly to the abstraction for plain linked lists [13] except that it can be refined to preserve nodes at particular addresses. Fig. 5 shows rewrite rules realising the abstraction $\text{abs}^{\text{rls}}_T$. The rules are guarded by a finite set of terms $T$ representing locations—each rule is enabled only if the spatial object triggering the rule is not among the locations in $T$.

Lemma 4. $\text{CSH}^{\text{rls}}_T \equiv \{ \Delta \mid \Delta \not\vdash \text{false} \land \Delta \not\Rightarrow^{\text{rls}}_T \}$ is finite and $\text{abs}^{\text{rls}}_T : \text{SH} \rightarrow \text{CSH}^{\text{rls}}_T$ is a sound abstraction.

[6] See Ranjit Jhala, Kenneth L. McMillan. A Practical and Complete Approach to Predicate Refinement. In TACAS, 2006, for an instance of such complete predicate refinement method (for difference bound arithmetic over the rationals).
\[
\Delta \rightarrow (e_1, x') \rightarrow \sigma_2(x', e_2) \sim_{RLS} \Delta \rightarrow \text{list}(e_1, \text{nil})
\]
if \(x' \notin \text{EVars}(\Delta, e_1, e_2) \land \Delta \not\vdash e_2 = \text{nil} \land \forall t \in T, \Delta \not\vdash e_1 = t\)

\[
\Delta \rightarrow \sigma_1(e_1, x') \rightarrow \sigma_2(x', e_2) \rightarrow \sigma_3(e_3, f) \sim_{RLS} \Delta \rightarrow \text{list}(e_1, e_2) \rightarrow \sigma_3(e_3, f)
\]
if \(x' \notin \text{EVars}(\Delta, e_1, e_2, e_3, f) \land \Delta \not\vdash e_2 = e_3 \land \forall t \in T, \Delta \not\vdash e_1 = t\)

**Fig. 5.** Abstract reduction system \(\sim_{RLS}\) defining the abstraction function \(\text{abs}_{RLS}\). First three rules (not shown) are the same as in Fig. 4. In the shown rules, \(\sigma, \sigma_1\) range over \(\{\text{node}, \text{list}\}\) and the data field is elided.

\[
S = 0;
\text{list}_{\leq}(e, f, [\alpha, \beta], \emptyset) \triangleq \alpha \leq d' < \beta \land (\text{node}(e, f, \{d'\}) \lor \text{list}_{\leq}(e', f, [\alpha', \beta], \emptyset))
\]

\[
S = \{d\};
\text{list}_{\leq}(e, f, [\alpha, \beta], \{d\}) \triangleq \text{node}(e, f, \{d\}) \lor \text{node}(e, x', \{d\}) \lor \text{list}_{\leq}(e', f, [\alpha', \beta], \emptyset) \lor \text{node}(e, x', [d', \beta], \emptyset) \lor \text{node}(e, x', [d', \beta], \emptyset) \lor \text{list}_{\leq}(e', f, [\alpha', \beta], \{d\})
\]

\[
|S| > 1, d \in S;
\text{list}_{\leq}(e, f, [\alpha, \beta], S) \triangleq \text{node}(e, x', \{d\}) \lor \text{node}(e, x', [d, \beta], S) \lor \text{node}(e, x', [d', \beta], S) \lor \text{list}_{\leq}(e', f, [\alpha', \beta], S) \lor \text{node}(e, x', [d', \beta], S) \lor \text{list}_{\leq}(e', f, [\alpha', \beta], S)
\]

**Fig. 6.** Recursive definition of the \(\text{list}_{\leq}\) predicate in domain \(\text{SH}_{RLS}\).

**Lemma 5.** If \(T_1 \subseteq T_2\) then \(\text{abs}_{RLS}^{T_1} \leq \text{abs}_{RLS}^{T_2}\).

**B.2 Sorted Linked Lists with Value Refinement**

Lastly, we present an analysis that works in the domain of sorted linked lists. Our abstraction can be refined to preserve particular values in the list, as with the analysis described in [4]. However, the domain consists of ordered lists segments.

**Domain.** The predicate \(\text{list}_{\leq}(x, y, [\alpha, \beta], S)\) holds if \(x\) points to a sorted non-empty list segment ending with \(y\) whose data values are all greater than or equal to \(\alpha\) and less than \(\beta\), and for each \(d \in \text{dom}(S)\), there are at least \(S(d)\) nodes in the list with value \(d\). Parameters \(\alpha, \beta\) and \(S\) satisfy the invariant \(I: \forall d \in \text{dom}(S), \alpha \leq d < \beta\). Sorted lists can be split according to the following rule:

\[
\text{list}_{\leq}(e, f, [\alpha, \beta], S) = \text{list}_{\leq}(e, x', [\alpha, \gamma], S \cap [\alpha, \gamma]) \cap \text{list}_{\leq}(e', f, [\gamma, \beta], S \cap [\gamma, \beta]).
\]

Folding/unfolding rules for exposing/hiding are similar to the rules for list (Fig. 3), but in addition keep track of the involved inequalities. New rules for \(\text{list}_{\leq}\) are shown in Fig. 6. Note that each rule maintains the invariant \(I\).

**Abstraction.** In the abstraction, we proceed similarly as in Fig. 4 but also maintain the invariant \(I\). Fig. 7 shows rewrite rules corresponding to the fourth rule of Fig. 4 for \(\sigma_1 = \sigma_2 = \text{node}\) and \(\sigma_1 = \sigma_2 = \text{list}_{\leq}\). The rest of the cases for \(\sigma_i\) are analogous to the fourth rule, and the fifth rule of Fig. 4. The resulting abstraction \(\text{abs}_{RLS}^{\text{SH}}\) satisfies the following lemmas:
\[
\Delta \ast \text{node}(e_1, x', \{d_1\}) \ast \text{node}(x', e_2, \{d_2\}) \xrightarrow{\text{sls}_T} \\
\Delta \ast \text{list}_\leq \left( e_1, \text{nil}, [d_1, d_2 + 1], \text{pr}_T(\{d_1, d_2\}, \Pi) \right) \\
\text{if } x' \notin \text{EVars}(\Delta, e_1, e_2) \land \Delta \vdash e_2 = \text{nil} \\
\Delta \ast \text{list}_\leq \left( e_1, x', [\alpha_1, \beta_1], S_1 \right) \ast \text{list}_\leq \left( x', e_2, [\alpha_2, \beta_2], S_2 \right) \xrightarrow{\text{sls}_T} \\
\Delta \ast \text{list}_\leq \left( e_1, \text{nil}, [\alpha_1, \beta_2], \text{pr}_T(S_1 \cup S_2, \Pi) \right) \\
\text{if } x' \notin \text{EVars}(\Delta, e_1, e_2) \land \Delta \vdash e_2 = \text{nil} \land \beta_1 \leq \alpha_2
\]

Fig. 7. Selected rules of the abstract reduction system \( \xrightarrow{\text{sls}_T} \) defining the abstraction function \( \text{abs}_{\text{sls}_T} \).

**Lemma 6.** For \( \text{CSH}_{T_{\text{rss}}} \equiv \{ \Delta \mid \Delta \not\vdash \text{false} \land \Delta \not\xrightarrow{\text{sls}_T} \} \), \( \text{abs}_{\text{sls}_T}: \text{SH} \rightarrow \text{CSH}_{T_{\text{rss}}} \) is a sound abstraction. If the domain of values is finite then \( \text{CSH}_{T_{\text{rss}}} \) is also finite.

**Lemma 7.** If \( T_1 \subseteq T_2 \) then \( \text{abs}_{T_1} \preceq \text{abs}_{T_2} \).

For infinite value domains, the set \( \text{CSH}_{T_{\text{rss}}} \) is infinite since we have infinite ascending chains of intervals as parameters to \( \text{list}_\leq \). We could recover convergence in such cases by using widening on the interval domain [12].