Abstraction Refinement for Separation Logic Program Analyses

Matko Botinčan[1], Mike Dodds[2], and Stephen Magill[3]

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

Abstract. Abstract domains based on separation logic allow the compositional analysis of heap-manipulating programs, and their effectiveness on real-world software has been extensively demonstrated. Tractability is achieved by applying abstraction, restricting invariants to a finite set of formulae. In standard analyses this set cannot vary, and loss of information through abstraction can cause an analysis to fail even when a proof is possible in the underlying logic. In this paper, we propose a method for automated refinement of separation logic analyses, geared towards checking the existential shape-based properties of data structures. Our approach is based on abduction, a technique for inferring sufficient preconditions for commands. We use abduction to work backwards from a spurious error, identify the location at which necessary information was lost, and refine the forward analysis so that this information is preserved in further iterations. We demonstrate our approach with several case studies, including code adapted from Redis, Azureus and FreeRTOS.

1 Introduction

Abstraction is often needed to automatically prove safety properties of programs. Finding the right abstraction is difficult, however. Techniques based on CEGAR (**c**ounter-**e**xample-**g**uided **a**bstraction **r**efinement) [10, 19] have gained a lot of attention as they can automatically synthesise an abstraction that is sufficient for proving a given property. Particularly successful has been the application of CEGAR to predicate abstraction [14], enabling automated verification of a wide range of (primarily control-flow driven) safety properties [1, 9, 18].

Meanwhile, separation logic has emerged as a useful domain for verifying shape-based safety properties [2, 3, 8, 13, 25]. 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 [12]. Although this approach has proved effective in practice, it does not provide a means to recover from spurious errors caused by loss of precision during abstraction.

This paper proposes a method for automated tuning of abstractions in separation logic analyses. We focus on proving existential properties, where we need to track some elements of a data structure more precisely than the others. Instead of a single abstraction, our method works with families of abstractions parameterised by multisets, and, in a nutshell, tries to find the multiset parameter for which the analysis will converge. Expanding the multiset refines the abstraction by making 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 (see Fig. 1). 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 [8]. We use the difference in symbolic states generated during forward and the backward analysis to select new elements to add to the multiset.

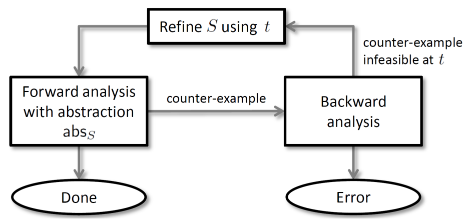

Fig. 1. Proposed abstraction refinement method.

1.1 Related Work

As in Berdine et al. [4] we wish to automate the process of 'tweaking' shape abstractions. [4] passes abstract counter-examples to an SMT solver, which produces concrete counter-example traces. These traces determine so-called *doomed states*, which are the same states singled out for refinement by our procedure. The advantage of using abduction is that it already forms the basis of an approach to compositional verification with demonstrated efficiency [8].

Our approach operates lazily, but in contrast to lazy abstraction [18], 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 [21], 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 the approach of Naik et al. [22], however, instead of analyzing concrete tests, we analyze abstract counter-examples.

The notion of abstraction function that we use is similar to widening [11]. However, refining the abstraction by replacing it with the least upper bound (as Gulavani and Rajamani [16]) would not converge in our case due to the presence of recursive data structures. Refinement with an interpolated widen [15], 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 [23] have been adapted for abstraction refinement [5, 20], however, we believe these approaches could not automatically handle verification of existential properties such as those in §6.

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 is shown in Figure 2. The program constructs a linked list of arbitrary length, pointed to by variable `r` (we use $\star$ to denote non-deterministically-chosen values). It picks

```
r = nil;
while (*) {
  r = new Node(r,*);
}
x = *;
r = new Node(r,x);
while (*) {
  r = new Node(r,*);
}
t = r; res = 0;
while(res==0 && t!=nil){
  d = t->data;
  if (d==x) res = 1;
  t = t->next;
}
assert(res==1);
```

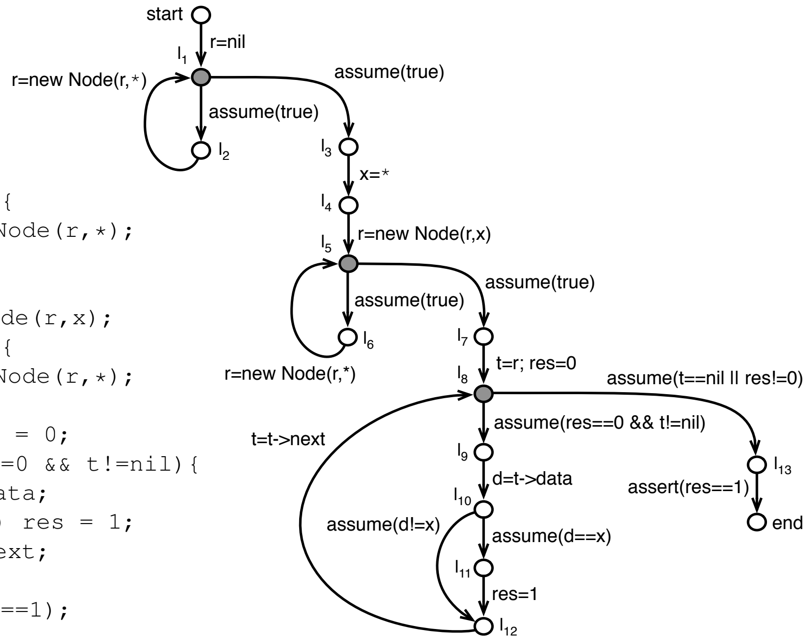

Fig. 2. Left: running example. Right: associated control-flow graph. Nodes where abstraction occurs are shaded.

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.

The problem. Suppose our abstract domain consists of the predicates emp, representing empty heap, $\mathsf{node}(x, y, d)$, representing a linked list node at address x with next pointer y and data contents d, and $\mathsf{list}(x, y)$, 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:

$$\mathsf{list}(x, y) \quad \triangleq \quad \mathsf{node}(x, y, d') \vee (\mathsf{node}(x, n', d') * \mathsf{list}(n', y))$$

(Primed variables—x', y', etc.—indicate logical variables that are existentially quantified). A traditional analysis, e.g. [12], starts with the pre-condition emp and propagates symbolic states over the control-flow graph (right of Fig. 2). 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

$$\mathsf{node}(\mathtt{r}, r', \mathtt{x}) \; * \; \mathsf{node}(r', \mathsf{nil}, d')$$

As is typical, assume the analysis applies the following abstraction step:

$$\mathsf{node}(x, y, d_1) * \mathsf{node}(y, z, d_2) \quad \rightsquigarrow \quad \mathsf{list}(x, z)$$

That is, it forgets list length and data values once there are two nodes in the list. The analysis thus rewrites $\mathsf{node}(\mathtt{r}, r', \mathtt{x}) \; * \; \mathsf{node}(r', \mathsf{nil}, d')$ to $\mathsf{list}(\mathtt{r}, \mathsf{nil})$. At the head of the third while-loop (node l_8) it unfolds $\mathsf{list}(\mathtt{r}, \mathsf{nil})$ back to the single-node case, yielding $\mathsf{node}(\mathtt{r}, \mathsf{nil}, x')$. Since this state is too weak to show that $\mathtt{x} = x'$, the path where `res` is not set to 1 appears feasible. The analysis now faults since it cannot prove the validity of `assert(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, as the analysis would not converge without it. We also cannot pick a tailored abstraction *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 iteratively modify its parameters based on spurious failures, automatically tailoring the abstraction to the property we want to prove.

To fix our example, we augment the domain with a family of predicates $\mathsf{list}(\_, \_, \{\mathtt{d}\})$, representing a list where at least one node holds the value `d` (this multiset-parametric domain is defined in §5.1). Upon failing to prove the final `assert` safe, our backwards analysis looks for *extensions* of symbolic states that *would* satisfy `assert(res==1)`, and so avoid the failure. Technically, this is achieved by posing successive abduction queries along the counter-example. If an extension is found, then the difference between the formulae from forward and the 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`. The analysis refines the abstraction to rewrite nodes containing `x` to $\mathsf{list}(\_, \_, \{\mathtt{x}\})$, "remembering" the existance of `x`. This suffices to avoid the spurious fault, and to prove the program correct.

3 Technical Background

Symbolic heaps. A symbolic heap Δ is a formula of the form $\Pi \wedge \Sigma$ where Π (the pure part) and Σ (the spatial part) are defined by:

$$\Pi ::= \mathsf{true} \mid \mathsf{false} \mid e = e \mid e \neq e \mid p(\bar{e}) \mid \Pi \wedge \Pi$$
$$\Sigma ::= \mathsf{emp} \mid s(\bar{e}) \mid \Sigma * \Sigma$$

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 Δ is denoted by $\mathsf{EVars}(\Delta)$. The separating conjunction $*$ expresses disjointness of partial states—$\Sigma_1 * \Sigma_2$ holds if the state can be split into two parts with disjoint domains, one satisfying Σ_1 and the other Σ_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 Δ. The set of all consistent (resp. disjunctive) symbolic heaps is denoted by SH (resp. $\mathcal{P}(\mathsf{SH})$).

Abstract domain. Our abstract domain is the join-semilattice $(\mathcal{P}(\mathsf{SH})^{\top}, \vdash, \sqcup, \top)$, where $\mathcal{P}(\mathsf{SH})^{\top} \triangleq \mathcal{P}(\mathsf{SH}) \cup \{\top\}$, 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 Δ_1 and Δ_2, find the frame Δ_F such that $\Delta_1 \vdash \Delta_2 * \Delta_F$ holds;
- $\Delta_1 * [\Delta_A] \vdash \Delta_2$ (abduction): given Δ_1 and Δ_2, find the 'missing' assumption Δ_A such that $\Delta_1 * \Delta_A \vdash \Delta_2$ holds.

Specifications. We assume that each atomic command $c \in \mathsf{Cmd}$ 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 $\mathsf{assume}(e) \triangleq \{\mathsf{true}\}\,\{e\}$ and $\mathsf{assert}(e) \triangleq \{e\}\,\{e\}$. 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.

Programs. We represent programs using a variant of intra-procedural control-flow graphs [18] over the set of atomic commands Cmd. A CFG consists of a set of nodes N containing distinguished starting and ending nodes $\mathsf{start}, \mathsf{end} \in \mathsf{N}$, and functions, $\mathsf{succ}\colon \mathsf{N} \to \mathcal{P}(\mathsf{N})$ and $\mathsf{cmd}\colon \mathsf{N} \times \mathsf{N} \rightharpoonup \mathsf{Cmd}$, representing node successors and edge labels. All nodes either have a single successor, or all outgoing edges are labelled with command $\mathsf{assume}(e)$ for the condition e that must hold for that edge to be taken.

Forward transfer function. We define the abstract forward semantics of each atomic command c by a function $[\![c]\!]\colon \mathsf{SH} \to \mathcal{P}(\mathsf{SH})^\top$. The function $[\![c]\!]$, fusing together rearrangement (materialisation) and symbolic execution [23, 2, 12, 8], is defined using the *frame rule*, which allows any triple $\{P\}\, c\, \{Q\}$ to be extended by an arbitrary frame Δ_F that is not modified by c:

$$[\![c]\!](\Delta) \quad \triangleq \quad \begin{cases} \top & \text{if } \nexists \Delta_F.\ \Delta \vdash P * \Delta_F \\ \{Q * \Delta_F \mid \Delta \vdash P * \Delta_F\} & \text{otherwise.} \end{cases}$$

When there is no Δ_F such that $\Delta \vdash P * \Delta_F$, the current heap Δ 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 $\mathcal{P}(\mathsf{SH})^\top \to \mathcal{P}(\mathsf{SH})^\top$ by mapping $\top$ to $\top$ and a set of symbolic heaps to the join of their $[\![c]\!]$-images.

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

$$[\![c]\!]^{\leftarrow}(\Delta) \quad \triangleq \quad \mathit{choose}(\{P * \Delta_A \mid Q * \Delta_A \vdash \Delta\})$$

We use a heuristic function *choose* to select a "good" solution, as generally there can be many solutions to the abduction query (e.g. a trivial one, false). For some fragments best solutions are possible: e.g. the disjunctive points-to fragment with (dis)equalities [8], a variation of which we use in our backward analysis. Along assume-edges we have $[\![\mathsf{assume}(e)]\!]^{\leftarrow}(\Delta) = \mathsf{wp}(\mathsf{assume}(e), \Delta) = \neg e \vee \Delta$.

3.1 Forward Analysis, Abstraction Function, and Parametricity

Forward analysis attempts to compute an inductive invariant $\mathsf{N} \to \mathcal{P}(\mathsf{SH})^{\top}$. It gradually weakens the strongest property by propagating symbolic heaps along CFG edges using the forward transfer, and joining the obtained $[\![c]\!]$-images at each CFG node. Since (1) our abstract domain is infinite (even though the number of program variables is bounded, there can be unboundedly many existential variables), and (2) the transfer functions are not necessarily monotone, forward propagation alone may not reach a fixpoint in a finite number of steps, or may not even converge to a fixpoint.

Abstraction. To ensure termination, (1) propagated symbolic heaps are abstracted into a finite set, and (2) the propagation process is made inflationary.[4] Abstraction is realised by a function $\mathsf{abs}\colon \mathsf{SH} \to \mathsf{CSH}$ whose codomain is a finite subset CSH of SH. At each step,[5] abs replaces the propagated symbolic heap with a logically weaker one in $\mathsf{CSH} \cup \{\top\}$. We require 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 $\Phi\colon (\mathsf{N} \to \mathcal{P}(\mathsf{SH})^{\top}) \to (\mathsf{N} \to \mathcal{P}(\mathsf{SH})^{\top})$, we compute the inflationary fixed-point of the functional $X \mapsto X \cup \Phi(X)$.

Definition 1. *We call a pair* $(\mathsf{SH}, \mathsf{abs})$ *an* analysis*.*

Definition 2 (analysis comparison). *Let* $\mathsf{abs}\colon \mathsf{SH} \to \mathsf{CSH}$ *and* $\mathsf{abs}'\colon \mathsf{SH} \to \mathsf{CSH}'$ *be abstraction functions. We say that* abs' *refines* abs*, written* $\mathsf{abs} \preceq \mathsf{abs}'$*, if* $\mathsf{CSH} \subseteq \mathsf{CSH}'$ *and for every* $\Delta \in \mathsf{SH}$*,* $\mathsf{abs}'(\Delta) \vdash \mathsf{abs}(\Delta)$*. We say that* $(\mathsf{SH}, \mathsf{abs}')$ *is more precise than* $(\mathsf{SH}, \mathsf{abs})$ *if* $\mathsf{abs} \preceq \mathsf{abs}'$*.*

Multiset-parametric analyses. In §4 we work with a family of analyses whose abstraction functions are parameterised by a multiset (such analyses are *parametric* in the sense of [22]). For any such family $(\mathsf{SH}, \mathsf{abs}_S)_{S \in \mathcal{M}}$, where $\mathcal{M}$ is some family of multisets and $\mathsf{abs}_S\colon \mathsf{SH} \to \mathsf{CSH}_S$, we require that if $S \subseteq S'$ then $\mathsf{abs}_S \preceq \mathsf{abs}_{S'}$. In §5 we describe three analyses with this property.

4 Forward-Backward Abstraction Refinement Algorithm

We now define our analysis formally. This version of the analysis is intra-procedural, but we believe it could be made inter-procedural without substantial difficulties (see §7).

[4] A function $f\colon (A, \sqsubseteq) \to (A, \sqsubseteq)$ is *inflationary* if for every a, we have $a \sqsubseteq f(a)$.

[5] In fact, it suffices to ensure that every cycle in the dependency graph contains at least one abstraction step, allowing more efficient iteration strategies [7].

```
S := ∅; t_0 := (start, emp); k := 0; E_T = ∅; T = ({t_0}, E_T, t_0);
while nodes_at_T(k) ≠ ∅ do
    foreach t = (n, Δ) ∈ nodes_at_T(k) do
        foreach n' ∈ succ(n) do
            D' := [[cmd(n, n')]](Δ);
            if D' = ⊤ then
                Add (t, (n', ⊤)) to E_T;
                k, S := Refine((n', ⊤), S);
                Break to the outermost while-loop;
            else
                foreach Δ' ∈ D do
                    Δ'_abs := abs_S(Δ');
                    if Δ'_abs ⋢ inv_T(n') then
                        T_old := T;
                        Add (t, (n', Δ'_abs)) to E_T;
                        inv_T := inv_{T_old}[n' ↦ inv_{T_old}(n') ⊔ Δ'_abs];
    k := k + 1;
```

Algorithm 1: Forward analysis with abstraction refinement.

Let $(\mathsf{SH}, \mathsf{abs}_S)_{S \in \mathcal{M}}$ be a family of analyses parameterised by a multiset. Our method for abstraction refinement starts with the analysis $(\mathsf{SH}, \mathsf{abs}_\emptyset)$, and iteratively refines the abstraction by adding terms to the multiset S. The goal is to eventually obtain S such that using the analysis $(\mathsf{SH}, \mathsf{abs}_S)$ we can compute a sufficient inductive invariant.

Forward analysis. Algorithm 1 shows forward analysis (§3.1) extended with abstraction refinement. The algorithm computes a fixpoint by constructing an abstract reachability tree (ART). An ART is a tree $\mathcal{T} = (T, E, t_0) \in \mathsf{ART}$ where T is the set of nodes, E the set of edges and t_0 the root node. We write $E_\mathcal{T}$ to refer to the set of edges associated with a particular ART $\mathcal{T}$. Nodes in T are of the form $(n, \Delta) \in \mathsf{N} \times \mathsf{SH}$ and represent the abstract states visited during the fixpoint computation. We use the following functions to deal with the ART: $\mathsf{parent}_\mathcal{T} \colon T \setminus \{t_0\} \to T$ returning the unique parent of a node, $\mathsf{depth}_\mathcal{T} \colon T \to \mathbb{N}_0$ returning the length of the path from t_0 to t, and $\mathsf{nodes\_at}_\mathcal{T} \colon \mathbb{N}_0 \to \mathcal{P}(T)$ returning the set of all nodes at the given depth. For $\mathcal{T} = (T, E, t_0)$ and $\mathcal{T}' = (T', E', t_0')$, we write $\mathcal{T} \subseteq \mathcal{T}'$ to indicate that $\mathcal{T}$ is a subtree of $\mathcal{T}'$, i.e., that $T \subseteq T'$, and $E \subseteq E'$, and $t_0 = t_0'$. We write $\mathsf{cmd}(n, n')$ to represent the command labelling the edge between nodes n and n' and $\mathsf{spec}(n, n')$ for the corresponding specification.

The algorithm iteratively propagates $[\![\cdot]\!]$-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 $\mathsf{inv} \colon \mathsf{ART} \to (\mathsf{N} \to \mathcal{P}(\mathsf{SH})^\top)$, reflecting the fact that the invariant at a control point can be recovered from the node labels of the ART. If we have $\mathsf{nodes\_at}(k) = \emptyset$ 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

Refine($t^\top : T,\ S : \mathcal{M}$)$: N_0 \times \mathcal{M}$
begin
  $k := \mathsf{depth}(t^\top) - 1$;
  $t_{\mathsf{curr}} := t^\top;\ \ t_{\mathsf{prev}} := \mathsf{parent}(t^\top)$;
  $\{P\}\ \_\ \{\_\} := \mathsf{spec}(t_{\mathsf{prev}}.n, t_{\mathsf{curr}}.n)$;
  Solve $\overline{t_{\mathsf{prev}}.\Delta} * [\Delta_A] \vdash P * \mathsf{true}$;
  $\Delta'_{\mathsf{prev}} := \overline{t_{\mathsf{prev}}.\Delta} * \Delta_A$;
  $\mathsf{path}_{\mathsf{fwd}} := t_{\mathsf{prev}}.\Delta$;
  $\mathsf{path}_{\mathsf{bwd}} := \Delta'_{\mathsf{prev}}$;
  while $k > 0$ **do**
    $t_{\mathsf{curr}} := t_{\mathsf{prev}};\ \ t_{\mathsf{prev}} := \mathsf{parent}(t_{\mathsf{curr}});\ \ \Delta'_{\mathsf{curr}} := \Delta'_{\mathsf{prev}}$;
    $\Delta'_{\mathsf{prev}} := [\![\mathsf{cmd}(t_{\mathsf{prev}}.n, t_{\mathsf{curr}}.n)]\!]^{\leftarrow}(\Delta'_{\mathsf{curr}})$;
    $\mathsf{path}_{\mathsf{fwd}} := t_{\mathsf{curr}}.\Delta \cdot \mathsf{path}_{\mathsf{fwd}}$;
    $\mathsf{path}_{\mathsf{bwd}} := \Delta'_{\mathsf{curr}} \cdot \mathsf{path}_{\mathsf{bwd}}$;
    if $t_{\mathsf{prev}}.\Delta \vdash \Delta'_{\mathsf{prev}}$ **then**
      $S := S \cup$ **SelectSymbols**($\mathsf{path}_{\mathsf{fwd}}, \mathsf{path}_{\mathsf{bwd}}$);
      Delete t_{curr}-subtree of $\mathcal{T}$;
      return $k - 1, S$;
    $k := k - 1$;
  throw “error”;

Algorithm 2: Backward analysis of counter-example by abduction.

t_0:	$(\mathsf{start}, \mathsf{emp})$	t_8:	$(l_8, \mathsf{list}(\mathtt{r}, \mathsf{nil}) \wedge \mathtt{t} = \mathtt{r} \wedge \mathtt{res} = 0)$
t_1:	$(l_1, \mathtt{r} = \mathsf{nil})$	t_9:	$(l_9, \mathsf{list}(\mathtt{r}, \mathsf{nil}) \wedge \mathtt{t} = \mathtt{r} \wedge \mathtt{res} = 0)$
t_2:	$(l_2, \mathtt{r} = \mathsf{nil})$	t_{10}:	$(l_{10}, \mathsf{node}(\mathtt{r}, \mathsf{nil}, d') \wedge \mathtt{t} = \mathtt{r} \wedge \mathtt{res} = 0 \wedge \mathtt{d} = d')$
t_3:	$(l_1, \mathsf{node}(\mathtt{r}, \mathsf{nil}, \_))$	t_{11}:	$(l_{12}, \mathsf{node}(\mathtt{r}, \mathsf{nil}, d') \wedge \mathtt{t} = \mathtt{r} \wedge \mathtt{res} = 0 \wedge \mathtt{d} = d' \wedge \mathtt{d} \neq \mathtt{x})$
t_4:	$(l_3, \mathsf{node}(\mathtt{r}, \mathsf{nil}, \_))$	t_{12}:	$(l_8, \mathsf{node}(\mathtt{r}, \mathsf{nil}, d') \wedge \mathtt{t} = \mathsf{nil} \wedge \mathtt{res} = 0 \wedge \mathtt{d} = d' \wedge \mathtt{d} \neq \mathtt{x})$
t_5:	$(l_4, \mathsf{node}(\mathtt{r}, \mathsf{nil}, \_))$	t_{13}:	$(l_{13}, \mathsf{node}(\mathtt{r}, \mathsf{nil}, d') \wedge \mathtt{t} = \mathsf{nil} \wedge \mathtt{res} = 0)$
t_6:	$(l_5, \mathsf{list}(\mathtt{r}, \mathsf{nil}))$	t_{14}:	$(\mathsf{end}, \top)$
t_7:	$(l_7, \mathsf{list}(\mathtt{r}, \mathsf{nil}))$		

Fig. 3. Abstract counter-example for the running example (§2).

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 2, **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 check its feasibility. Once a point in the path is found where forward analysis agrees with the backward analysis, the mismatch between the symbolic heaps from forward and backward analyses is used to update the multiset S that determines the abstraction.

Definition 3. *An* abstract counter-example *is a sequence* $(n_0, \Delta_0) \ldots (n_k, \Delta_k)$ *with:*

– $n_0 = \mathsf{start}$ *and for all* $0 < i \le k$, $n_i \in \mathsf{succ}(n_{i-1})$*;*
– $\Delta_0 = \mathsf{emp}$*, for all* $0 < i \le k$, $\Delta_i \in \mathsf{abs}_S(\llbracket \mathsf{cmd}(n_{i-1}, n_i) \rrbracket(\Delta_{i-1}))$ *and* $\Delta_k = \top$.

Figure 3 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 $\mathsf{cmd}(n_{k-1}, n_k) = \{P_k\}\, c_k\, \{Q_k\}$. Since $\llbracket c_k \rrbracket(\Delta_{k-1}) = \top$, Δ_{k-1} misses some assumption required to satisfy P_k. To find this, **Refine** solves the following abduction query (line 6)—here $\overline{\Delta_{k-1}}$ is a rearrangement of Δ_{k-1}, for example to expose particular memory cells:

$$\overline{\Delta_{k-1}} * [\Delta_A] \vdash P_k * \mathsf{true}.$$

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

Letting $\Delta'_{k-1} := \overline{\Delta_{k-1}} * \Delta_A$, **Refine** computes a sufficient resource for the preceding state (line 12):

$$\Delta'_{k-2} := \llbracket \mathsf{cmd}(n_{k-2}, n_{k-1}) \rrbracket^{\leftarrow}(\Delta'_{k-1}).$$

If $\Delta_{k-2} \vdash \Delta'_{k-2}$ then in the step from Δ_{k-2} to Δ_{k-1} a loss of precision has happened, and we use the additional information in Δ'_{k-1} to refine the abstraction (line 15). Otherwise, we continue pushing backwards, and generate Δ'_{k-3}, Δ'_{k-4}, etc.

Eventually, **Refine** either halts with $\Delta_i \vdash \Delta'_i$ for some $i \ge 0$, or in the last step obtains $\Delta_0 \nvdash \Delta'_0$. 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 16). 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 20). Note that in this case the computed heap Δ'_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 17). 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* $\mathsf{inv}_{\mathcal{T}}$ *is an inductive invariant not containing* $\top$.

Proof. The refinement of abstraction in Alg. 1 is achieved by augmenting the multiset S with new elements selected by **SelectSymbols**. Since $\mathsf{abs}_S \preceq \mathsf{abs}_{S'}$ for $S \subseteq S'$, our analysis is immediately sound. Any remaining nodes in the ART not recomputed straight away will either be recomputed at some later stage, or never be discarded. The latter case implies that the analysis may find an invariant weaker than the strongest invariant expressible under a more refined abstraction—but still sufficient for proving the safety property. □

Refinement heuristic. **SelectSymbols** is the heuristic function which refines the abstraction. It takes two sequences of symbolic heaps, $\mathsf{path}_{\mathsf{fwd}}$ and $\mathsf{path}_{\mathsf{bwd}}$: the former corresponds to the current counter-example, and the latter to a path sufficient to avoid the error. By comparing the two, **SelectSymbols** tries to identify symbols which would prevent the parametric abstraction function from causing the error.

As **SelectSymbols** is a heuristic function, it could be implemented many different ways (in a sense, it is a simpler analogue of predicate discovery heuristics [1, 18]). Our implementation works by examining the structure of formulae Δ and Δ' that are associated with the same ART node. The heuristic exhaustively searches for equalities in Δ' (transitively) containing program variables that can be used to strengthen Δ. It preserves the common syntactic parts of Δ and Δ' by explicitly maintaining the substitutions, and searches for equalities in the congruence class associated with each term. Intuitively, because they are mentioned in the calculated sufficient resource, the identified equalities will likely be significant for program correctness. The variables are added to the multiset to strengthen the abstraction. We found this heuristic to work well in our case studies (see §6).

Running example revisited. In §2 we saw a spurious error caused by over-abstracting values in the list. To fix this, we augmented the domain with predicates $\mathsf{list}(\_,\_,\{\mathtt{d}\})$, representing a list that has at least one node with value $\mathtt{d}$. We now show the refinement step in this domain. The backward analysis begins by solving the abduction query

$$(\mathsf{node}(\mathtt{r},\mathsf{nil},d') \wedge \mathtt{x}=d \wedge \mathtt{t}=\mathsf{nil} \wedge \mathtt{res}=0) * [\Delta_A] \vdash \mathtt{res}=1 * [\_]$$

This yields $\Delta_A = \mathsf{false}$ as the only solution. The analysis then generates the following sequence of symbolic heaps (compare with the abstract counter example in Fig. 3; here Δ'_i corresponds to node t_i):

$$\begin{aligned}
\Delta'_{13} &= \mathsf{false}\\
\Delta'_{12} &= (\mathtt{t}\neq\mathsf{nil} \wedge \mathtt{res}=0 \wedge \mathsf{true})\\
\Delta'_{11} &= (t'\neq\mathsf{nil} \wedge \mathtt{res}=0 \wedge \mathsf{node}(\mathtt{t},t',d') * \mathsf{true})\\
\Delta'_{10} &= ((\mathtt{d}=\mathtt{x} \wedge \mathsf{true}) \vee (t'\neq\mathsf{nil} \wedge \mathtt{res}=0 \wedge \mathsf{node}(\mathtt{t},t',d') * \mathsf{true}))\\
\Delta'_{9} &= ((\mathsf{node}(\mathtt{t},t',\mathtt{x}) * \mathsf{true}) \vee (t'\neq\mathsf{nil} \wedge \mathtt{res}=0 \wedge \mathsf{node}(\mathtt{t},t',d') * \mathsf{true}))\\
\Delta'_{8} &= (((\mathtt{res}\neq 0 \vee \mathtt{t}=\mathsf{nil}) \wedge \mathsf{true}) \vee (\mathsf{node}(\mathtt{t},t',\mathtt{x}) * \mathsf{true}) \vee\\
&\qquad (t'\neq\mathsf{nil} \wedge \mathtt{res}=0 \wedge \mathsf{node}(\mathtt{t},t',d') * \mathsf{true}))\\
\Delta'_{7} &= ((\mathtt{r}=\mathsf{nil} \wedge \mathsf{true}) \vee (\mathsf{node}(\mathtt{r},t',\mathtt{x}) * \mathsf{true}) \vee (t'\neq\mathsf{nil} \wedge \mathsf{node}(\mathtt{r},t',d') * \mathsf{true}))\\
\Delta'_{6} &= ((\mathtt{r}=\mathsf{nil} \wedge \mathsf{true}) \vee (\mathsf{node}(\mathtt{r},t',\mathtt{x}) * \mathsf{true}) \vee (t'\neq\mathsf{nil} \wedge \mathsf{node}(\mathtt{r},t',d') * \mathsf{true}))\\
\Delta'_{5} &= (\mathtt{r}=t' \wedge \mathsf{true})
\end{aligned}$$

The algorithm stops at Δ'_5, since $\Delta_5 \vdash \Delta'_5$, and calls **SelectSymbols** to augment the abstraction. Our implementation looks for equalities in each Δ' that can be used to strengthen Δ. In this case, in Δ'_{10} the heuristic identifies $\mathtt{d}=\mathtt{x}$ to strengthen the corresponding $\Delta_{10} = \mathsf{node}(\mathtt{r},\mathsf{nil},d') \wedge \mathtt{t}=\mathtt{r} \wedge \mathtt{res}=0 \wedge \mathtt{d}=d'$. Thus the heuristic selects the variable $\mathtt{x}$ to augment the abstraction's multiset.

In the unrefined analysis, any predicate $\mathsf{list}(\_,\_,S)$ will be abstracted to $\mathsf{list}(\_,\_,\emptyset)$ (equivalent to $\mathsf{list}(\_,\_)$). Adding $\mathtt{x}$ to the multiset means that predicates of the form $\mathsf{list}(\_,\_,\{\mathtt{x}\})$ will be protected from abstraction.

We restart the forward analysis from t_5 with the new abstraction. This time the error is avoided. We obtain the following abstract states:

$$t'_6 = (l_5, \mathsf{list}(\mathtt{r}, \mathsf{nil}, \{\mathtt{x}\})) \ \ldots$$
$$t'_9 = (l_9, \mathsf{list}(\mathtt{r}, \mathsf{nil}, \{\mathtt{x}\}) \wedge \mathtt{t} = \mathtt{r} \wedge \mathtt{res} = 0)$$

Executing from l_9 to l_{10} gives two possible post-states: $\mathsf{node}(\mathtt{r}, r', d') * \mathsf{list}(r', \mathsf{nil}, \{\mathtt{x}\}) \wedge \mathtt{t} = \mathtt{r} \wedge \mathtt{res} = 0 \wedge \mathtt{d} = d'$ and $\mathsf{node}(\mathtt{r}, r', \mathtt{x}) * \mathsf{list}(r', \mathsf{nil}, \emptyset) \wedge \mathtt{t} = \mathtt{r} \wedge \mathtt{res} = 0 \wedge \mathtt{d} = \mathtt{x}$. From the first state we reach a contained state at the head of the loop. From the second, we get the following abstract states:

$$t'_{10} = (l_{10}, \mathsf{node}(\mathtt{r}, r', \mathtt{x}) * \mathsf{list}(r', \mathsf{nil}, \emptyset) \wedge \mathtt{t} = \mathtt{r} \wedge \mathtt{res} = 0 \wedge \mathtt{d} = \mathtt{x}) \ \ldots$$
$$t'_{15} = (\mathsf{end}, \mathsf{node}(\mathtt{r}, \mathtt{t}, \mathtt{x}) * \mathsf{list}(\mathtt{t}, \mathsf{nil}, \emptyset) \wedge \mathtt{res} = 1 \wedge \mathtt{d} = \mathtt{x})$$

Ending with t'_{15}, the analysis has explored all successors of t_5 without reaching an error. In fact, the new abstraction is sufficient to prove the absence of errors on all paths, and complete the analysis. Other examples may need multiple refinement steps, of course.

5 Example Multiset-Parametric Analyses

We describe in detail linked lists with value refinement (used in our running example) and sketch two other 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 §6. Some abstractions proposed by other authors could be straightforwardly formulated as multiset-parametric, e.g. [24].

5.1 Linked Lists with Value Refinement

This is the analysis family used in our running example (§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 $\mathsf{SH}^{\mathsf{mls}}$ contains spatial predicates $\mathsf{node}(\cdot, \cdot, \{d\})$ and $\mathsf{list}(\cdot, \cdot, S)$ for all S and $d \in S$. Here x, y are locations, d is a data value, S is a multiset:

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

The recursive definition of list in $\mathsf{SH}^{\mathsf{mls}}$ is shown in Fig. 4. We use these equivalences as folding and unfolding rules when solving entailment queries in $\mathsf{SH}^{\mathsf{mls}}$.

$$
\begin{array}{lrcl}
\text{case } S = \emptyset: & \mathsf{list}(e, f, \emptyset) & \triangleq & \mathsf{node}(e, f, \_) \vee (\mathsf{node}(e, x', \_) * \mathsf{list}(x', f, \emptyset)) \\
\text{case } S = \{d\}: & \mathsf{list}(e, f, \{d\}) & \triangleq & \mathsf{node}(e, f, \{d\}) \vee (\mathsf{node}(e, x', \{d\}) * \mathsf{list}(x', f, \emptyset)) \\
 & & & \vee (\mathsf{node}(e, x', \_) * \mathsf{list}(x', f, \{d\})) \\
\text{case } |S| > 1, d \in S: & \mathsf{list}(e, f, \{S\}) & \triangleq & \mathsf{node}(e, x', \{d\}) * \mathsf{list}(x', f, S \setminus \{d\}) \vee \\
 & & & \mathsf{node}(e, x', \_) * \mathsf{list}(x', f, S)
\end{array}
$$

Fig. 4. Recursive definition of the list predicate in domain $\mathsf{SH}^{\mathsf{mls}}$.

$$
\begin{array}{rcl}
\Delta \wedge x' = e & \rightsquigarrow_T^{\mathsf{mls}} & \Delta[e/x'] \\
\Delta * \sigma(x', e, \_) & \rightsquigarrow_T^{\mathsf{mls}} & \Delta * \mathsf{true} \qquad \text{if } x' \notin \mathsf{EVars}(\Delta) \\
\Delta * \sigma_1(x', y', \_) * \sigma_2(y', x', \_) & \rightsquigarrow_T^{\mathsf{mls}} & \Delta * \mathsf{true} \qquad \text{if } x', y' \notin \mathsf{EVars}(\Delta) \\
\Delta * \sigma_1(e_1, x', S_1) * \sigma_2(x', e_2, S_2) & \rightsquigarrow_T^{\mathsf{mls}} & \Delta * \mathsf{list}(e_1, \mathsf{nil}, \mathsf{pr}_T(S_1 \cup S_2, \Pi)) \\
 & & \text{if } x' \notin \mathsf{EVars}(\Delta, e_1, e_2) \wedge \Delta \vdash e_2 = \mathsf{nil} \\
\begin{pmatrix} \Delta * \sigma_1(e_1, x', S_1) * \\ \sigma_2(x', e_2, S_2) * \sigma_3(e_3, f, S_3) \end{pmatrix} & \rightsquigarrow_T^{\mathsf{mls}} & \begin{pmatrix} \mathsf{list}(e_1, e_2, \mathsf{pr}_T(S_1 \cup S_2, \Pi)) \\ * \Delta * \sigma_3(e_3, f, S_3) \end{pmatrix} \\
 & & \text{if } x' \notin \mathsf{EVars}(\Delta, e_1, e_2, e_3, f) \wedge \Delta \vdash e_2 = e_3 \\
\Delta * \mathsf{list}(e, f, S) & \rightsquigarrow_T^{\mathsf{mls}} & \mathsf{list}(e, f, \mathsf{pr}_T(S, \Pi))
\end{array}
$$

Fig. 5. Abstract reduction system $\rightsquigarrow_T^{\mathsf{mls}}$ defining the abstraction function $\mathsf{abs}_T^{\mathsf{mls}}$. (In the rules, σ, σ_i range over $\{\mathsf{node}, \mathsf{list}\}$. The pure assumption Π is supplied by the analysis.)

Abstraction. Let T be a finite multiset of program variables and constants. In Fig. 5, we define a parametric reduction system $\rightsquigarrow_T^{\mathsf{mls}}$, which rewrites symbolic heaps from $\mathsf{SH}^{\mathsf{mls}}$ to canonical heaps whose data and multiset values are congruent to elements of T. Except for the final rule, the relation $\rightsquigarrow_T^{\mathsf{mls}}$ resembles the abstraction for plain linked lists developed by Distefano et al. [12, table 2].

The final reduction rule replaces every predicate $\mathsf{list}(e, f, S)$ with the bounded predicate $\mathsf{list}(e, f, \mathsf{pr}_T(S, \Pi))$. The operator pr_T extracts the maximal subset of S such that no element appears more frequently than it does in T (modulo given pure assumptions Π). Let $\sim_\Pi$ be the equivalence relation $x \sim_\Pi y \triangleq \Pi \vdash x = y$. Fix a representative for each equivalence class of $\sim_\Pi$, and for a multiset S, denote by $S/_\Pi$ the multiset of $\sim_\Pi$-representatives where the multiplicity of a representative x is $\sum_{x \sim_\Pi y} S(y)$. Writing $x \cdot n$ for a multiset element x occurring with multiplicity n, we define pr_T by

$$\mathsf{pr}_T(S, \Pi) \triangleq \{x \cdot n \mid x \cdot k' \in S/_\Pi \wedge \exists d' \cdot m' \in T/_\Pi .\, \Pi \vdash x = d' \wedge n = \min(k', m')\}.$$

It is easy to show that $\rightsquigarrow_T^{\mathsf{mls}}$ has no infinite reduction sequences. Thus, $\rightsquigarrow_T^{\mathsf{mls}}$ gives rise to an abstraction function $\mathsf{abs}_T^{\mathsf{mls}}$ which is obtained by exhaustively applying the rules to a given symbolic heap until no more rules apply.

Lemma 1 (Finiteness). *If T is finite and there are only finitely many program variables then the domain $\mathsf{CSH}_T^{\mathsf{mls}} \triangleq \{\Delta \mid \Delta \not\vdash \mathsf{false} \wedge \Delta \not\rightsquigarrow_T^{\mathsf{mls}}\}$ is finite.*

Lemma 2 (Soundness). *As $\Delta \rightsquigarrow_T^{\mathsf{mls}} \Delta'$ implies $\Delta \vdash \Delta'$, $\mathsf{abs}_T^{\mathsf{mls}} \colon \mathsf{SH} \to \mathsf{CSH}_T^{\mathsf{mls}}$ is a sound abstraction function.*

Lemma 3 (Monotonicity). *If $T_1 \subseteq T_2$ then $\mathsf{abs}_{T_1}^{\mathsf{mls}} \preceq \mathsf{abs}_{T_2}^{\mathsf{mls}}$.*

5.2 Linked Lists with Address Refinement

Rather than preserving certain values in the list, we might need to preserve nodes at particular *addresses*. For example, to remove a node from a linked list we might use the procedure shown on the right. Given pre-condition $\mathsf{list}(\mathtt{r}, \mathtt{x}) * \mathsf{node}(\mathtt{x}, n', \_) * \mathsf{list}(n', \mathsf{nil})$ 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.

```
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;
}
```

To preserve information of this kind, we combine the domain of linked lists, $\mathsf{SH}^{\mathsf{rls}}$, 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 $\rightsquigarrow_T^{\mathsf{rls}}$ protects addresses in the multiset T from abstraction. As before, refinement consists of adding new addresses to the multiset.

5.3 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 $\mathsf{SH}^{\mathsf{sls}}$ contains the predicate $\mathsf{list}_{\leq}$, parameterised by an interval of the form $[\alpha, \beta\rangle$, 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 $\rightsquigarrow_T^{\mathsf{sls}}$ works in a similar way to $\mathsf{SH}^{\mathsf{mls}}$: the operator pr_T caps the frequency set S, limiting the number of values that are preserved by abstraction.

6 Experimental Evaluation

To evaluate our abstraction refinement analysis, we implemented Algorithm 1 and abstract domains $\mathsf{SH}^{\mathsf{mls}}$, $\mathsf{SH}^{\mathsf{rls}}$ and $\mathsf{SH}^{\mathsf{sls}}$ 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 ran the analysis against client-oriented specifications [17] describing datastructures from Redis (a key-value store), Azureus (a BitTorrent client) and FreeRTOS (real time operating system). Table 1 shows the obtained results.

***Set, Multiset** and **Map**.* The first three segments in Table 1 correspond to a set of synthetic benchmarks based on client-oriented specifications for Redis [17]. These specifications check various aspects of functional correctness—for example, that following deletion a key is no longer bound in the store. Furthermore, we check these specifications across dynamic updates which may modify the data structures involved—for example, by removing duplicate bindings to optimize for space usage.

The **Set** and **Multiset** benchmarks apply operations add (add an element), del (delete an element) and mem (test for membership) to a list-based set (multiset, respectively) in the order indicated by the benchmark name. The symbols $*$, $*_{\neg\mathsf{del}}$ and $*_{\neg x}$ respectively denote applying all operations any number of times with any argument, all operations

No	Benchmark	Result	Dom	$\|T\|$	#Refn	|ART|	#Quer
Set							
1	$\mathsf{add}(x)$–$*$–$\mathsf{mem}(x)$	✓	$\mathsf{SH}^{\mathsf{mls}}$	1	1	83	162
2	$*$–$\mathsf{add}(x)$–$*_{\neg\mathsf{del}}$–$\mathsf{mem}(x)$	✓	$\mathsf{SH}^{\mathsf{mls}}$	1	1	104	193
3	$*_{\neg\mathsf{del}}$–$\mathsf{mem}(x)$–$*_{\neg x}$–$\mathsf{mem}(x)$	✓	$\mathsf{SH}^{\mathsf{mls}}$	1	1	165	280
4	$*_{\mathsf{add}(x)}$–$\mathsf{all\_equal\_to\_}x$	∞	$\mathsf{SH}^{\mathsf{mls}}$				
5	$*_{\mathsf{add}(x)}$–$\mathsf{all\_sorted}$	$\top$	$\mathsf{SH}^{\mathsf{mls}}$				
Multiset							
6	$\mathsf{add}(x)$–$\mathsf{add}(x)$–$\mathsf{del}(x)$–$\mathsf{mem}(x)$	✓	$\mathsf{SH}^{\mathsf{mls}}$	2	1	67	91
7	$*$–$\mathsf{add}(x)$–$*_{\neg\mathsf{del}}$–$\mathsf{mem}(x)$	✓	$\mathsf{SH}^{\mathsf{mls}}$	1	1	112	205
8	$*_{\neg\mathsf{del}}$–$\mathsf{mem}(x)$–$*_{\neg x}$–$\mathsf{mem}(x)$	✓	$\mathsf{SH}^{\mathsf{mls}}$	1	1	171	312
9	$*$–$\mathsf{add}(x)$–$*_{\neg\mathsf{del}}$–$\mathsf{add}(x)$–$*_{\neg\mathsf{del}}$–$\mathsf{del}(x)$–$\mathsf{mem}(x)$	✓	$\mathsf{SH}^{\mathsf{mls}}$	2	2	219	458
Map							
10	$*$–$\mathsf{put}(k,v)$–$*_{\neg k}$–$\mathsf{get}(k)$	✓	$\mathsf{SH}^{\mathsf{mls}}$	1	1	118	215
11	$*$–$\mathsf{rem}(k)$–$\mathsf{bound}(k)$	✓	$\mathsf{SH}^{\mathsf{mls}}$	1	1	92	168
ByteBufferPool							
12	Property 1	✓	$\mathsf{SH}^{\mathsf{rls}}$	1	1	154	231
13	Property 2	✓	$\mathsf{SH}^{\mathsf{rls}}$	2 (1)	1	189	270
14	Property 3	✓	$\mathsf{SH}^{\mathsf{rls}}$	6 (2)	4	316	511
FreeRTOS list							
15	Property 4	✓	$\mathsf{SH}^{\mathsf{mls}}$	1	1	91	158
16	Property 5	✓	$\mathsf{SH}^{\mathsf{sls}}$	6	5	425	971

Table 1. Experimental Results. The benchmarks that were successfully verified are marked with ✓, those where the analysis threw an error with $\top$ and those where the analysis did not terminate with ∞. *Dom* is the domain used for the analysis, $|T|$ is the size of the multiset T after the final refinement (the number in parentheses denotes the size of the minimal sufficient T), *#Refn* is the number of refinement steps, *|ART|* is the number of symbolic states in the final ART, and *#Quer* is the total number of queries sent to the prover.

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) 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 $\mathsf{SH}^{\mathsf{mls}}$ to loop forever by adding x to T at each refinement step. Benchmark 5 is a universal property for which our analysis in $\mathsf{SH}^{\mathsf{mls}}$ fails to find an inductive invariant due to the ordering predicate (using $\mathsf{SH}^{\mathsf{sls}}$ on the same benchmark loops forever). We discuss existential vs. universal properties in §7.

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. FreeRTOS list, used by FreeRTOS scheduler for task management, is a sorted cyclic list with a sentinel node. 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.

7 Analysis Properties, Limitations and Conclusions

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

Without further assumptions, Algorithm 1 might diverge, or report a spurious counter-example which is in fact not feasible. 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 Δ, $[\![c]\!]([\![c]\!]^{\leftarrow}(\Delta)) \vdash \Delta$) 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.

We have presented an intra-procedural version of our analysis, but believe that it could be made inter-procedural without substantial further research (although it would make our algorithm much more complex). Abduction is already known to work well in an inter-procedural setting [8]. We envisage augmenting the analysis from [8] with abstraction refinement. Counter-example analysis could either reuse the generated procedure summaries or refine them by diving into the procedures using abduction.

Our backward analysis, **Refine**, is heavily dependent on the success of abduction. In the case studies we considered, **Refine** uses abduction in what is effectively the points-to fragment with (dis)equalities, for which abduction can obtain provably-optimal solutions (making the backward transfer precise) and for which there are heuristics that work well in practice [8]. In general (e.g., if we were to introduce procedures), we may end up repeatedly refining the same spurious path, resulting in non-termination.

All the refinement domains considered in §5 can be seen as representing existential properties, e.g. "*the list segment contains at least certain values*". It would be straightforward to define domains for other existential properties, e.g. "*the list contains a particular subsequence*". However, *universal* properties, such as "*all nodes contain a particular value*", are hard to capture in our approach. This bias towards existential properties is a result of our analysis structure. When forward execution fails spuriously, 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.

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. Our work represents a step forward from current separation logic analyses, which typically fix the domain for the whole run of the analysis.

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. Naudžiūnienė, and M. Parkinson. coreStar: The Core of jStar. In *Boogie*, 2011.
7. F. Bourdoncle. Efficient chaotic iteration strategies with widening. In *Formal Methods in Programming and their Applications*, volume 735 of *LNCS*, 1993.
8. C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. *J. ACM*, 58(6), 2011.
9. S. Chaki, E. M. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. In *ICSE*, 2003.
10. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *CAV*, 2000.
11. 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.
12. D. Distefano, P. W. O'Hearn, and H. Yang. A local shape analysis based on separation logic. In *TACAS*, 2006.
13. D. Distefano and M. J. Parkinson. jStar: towards practical verification for Java. In *OOPSLA*, 2008.
14. S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *CAV*, 1997.
15. B. S. Gulavani, S. Chakraborty, A. V. Nori, and S. K. Rajamani. Automatically refining abstract interpretations. In *TACAS*, 2008.
16. B. S. Gulavani and S. K. Rajamani. Counterexample driven refinement for abstract interpretation. In *TACAS*, 2006.
17. 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.
18. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL*, 2002.
19. R. P. Kurshan. *Computer-aided verification of coordinating processes: the automata-theoretic approach*. Princeton University Press, 1994.
20. A. Loginov, T. W. Reps, and S. Sagiv. Abstraction refinement via inductive learning. In *CAV*, 2005.
21. K. L. McMillan. Lazy abstraction with interpolants. In *CAV*, 2006.
22. M. Naik, H. Yang, G. Castelnuovo, and M. Sagiv. Abstractions from tests. In *POPL*, 2012.
23. S. Sagiv, T. W. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *TPLS*, 24(3), 2002.
24. V. Vafeiadis. Shape-value abstraction for verifying linearizability. In *VMCAI*, 2009.
25. 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.

A Relative Progress and Completeness

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., $[\![\cdot]\!]$-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 $\varDelta$, we have $[\![c]\!]([\![c]\!]^{\leftarrow}(\varDelta)) \vdash \varDelta$.
(c) When called with a $(\mathsf{path}_{\mathsf{fwd}}, \mathsf{path}_{\mathsf{bwd}})$-pair of the counter-example and the path sufficient to avoid the error, the procedure call **SelectSymbols**$(\mathsf{path}_{\mathsf{fwd}}, \mathsf{path}_{\mathsf{bwd}})$ picks symbols A for augmenting S such that the spurious counter-example ending with $\mathsf{path}_{\mathsf{fwd}}$ is eliminated by the abstraction $\mathsf{abs}_{S \cup A}$.

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

Theorem 2 (Relative progress). *Let γ_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 γ_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 **Refine**. Since $\mathsf{abs}_{S(j)} \preceq \mathsf{abs}_{S(j+1)}$, no new counter-examples can appear in the part of the ART that is recomputed in the $(j+1)$-th step (invariants computed in $\mathsf{CSH}_{S(j+1)}$ will be at least as strong as those in $\mathsf{CSH}_{S(j)}$). 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. □

Theorem 3 (Relative completeness). *If the safety property is implied by an inductive invariant expressible in CSH_S for some finite multiset S and assuming that those elements would eventually be selected from counter-examples by* **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 $\mathsf{CSH}_{S'}$ for some $S' \supseteq S$. □

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,[6] we in fact have exact forward transfer, meaning spurious counter-examples can always be eliminated.

[6] 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 [8]. 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 [8]) 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.[7] 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 §5.

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 $\mathsf{SH}^{\mathsf{rls}}$, built from plain spatial predicates node and list.

Our abstraction works similarly to the abstraction for plain linked lists [12] except that it can be refined to preserve nodes at particular addresses. Fig. 6 shows rewrite rules realising the abstraction $\mathsf{abs}^{\mathsf{rls}}_T$. 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. $\mathsf{CSH}^{\mathsf{rls}}_T \triangleq \{\Delta \mid \Delta \nvdash \mathsf{false} \wedge \Delta \not\rightsquigarrow^{\mathsf{rls}}_T\}$ *is finite and* $\mathsf{abs}^{\mathsf{rls}}_T : \mathsf{SH} \to \mathsf{CSH}^{\mathsf{rls}}_T$ *is a sound abstraction.*

[7] 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).

$$\varDelta * \sigma_1(e_1, x') * \sigma_2(x', e_2) \rightsquigarrow^{\mathsf{rls}}_T \varDelta * \mathsf{list}(e_1, \mathsf{nil})$$
$$\text{if } x' \notin \mathsf{EVars}(\varDelta, e_1, e_2) \wedge \varDelta \vdash e_2 = \mathsf{nil} \wedge \forall t \in T \,.\, \varDelta \nvdash e_1 = t$$

$$\varDelta * \sigma_1(e_1, x') * \sigma_2(x', e_2) * \sigma_3(e_3, f) \rightsquigarrow^{\mathsf{rls}}_T \varDelta * \mathsf{list}(e_1, e_2) * \sigma_3(e_3, f)$$
$$\text{if } x' \notin \mathsf{EVars}(\varDelta, e_1, e_2, e_3, f) \wedge \varDelta \vdash e_2 = e_3 \wedge \forall t \in T \,.\, \varDelta \nvdash e_1 = t$$

Fig. 6. Abstract reduction system $\rightsquigarrow^{\mathsf{rls}}_T$ defining the abstraction function $\mathsf{abs}^{\mathsf{rls}}_T$. First three rules (not shown) are the same as in Fig. 5. In the shown rules, σ, σ_i range over $\{\mathsf{node}, \mathsf{list}\}$ and the data field is elided.

$S = \emptyset$:

$$\mathsf{list}_{\leq}(e, f, [\alpha, \beta\rangle, \emptyset) \triangleq \alpha \leq d' < \beta \wedge (\mathsf{node}(e, f, \{d'\}) \vee \mathsf{node}(e, x', d') * \mathsf{list}_{\leq}(x', f, [d', \beta\rangle, \emptyset))$$

$S = \{d\}$:

$$\mathsf{list}_{\leq}(e, f, [\alpha, \beta\rangle, \{d\}) \triangleq \mathsf{node}(e, f, \{d\}) \vee d = \alpha \wedge \mathsf{node}(e, x', \{d\}) * \mathsf{list}_{\leq}(x', f, [d, \beta\rangle, \emptyset) \vee d \neq \alpha \wedge \alpha \leq d' < \beta \wedge \mathsf{node}(e, x', d') * \mathsf{list}_{\leq}(x', f, [d', \beta\rangle, \{d\})$$

$|S| > 1, d \in S$:

$$\mathsf{list}_{\leq}(e, f, [\alpha, \beta\rangle, S) \triangleq d = \alpha \wedge \mathsf{node}(e, x', \{d\}) * \mathsf{list}_{\leq}(x', f, [d, \beta\rangle, S \setminus \{d\}) \vee d \neq \alpha \wedge \alpha \leq d' < \beta \wedge \mathsf{node}(e, x', d') * \mathsf{list}_{\leq}(x', f, [d', \beta\rangle, S)$$

Fig. 7. Recursive definition of the $\mathsf{list}_{\leq}$ predicate in domain $\mathsf{SH}^{\mathsf{sls}}$.

Lemma 5. *If* $T_1 \subseteq T_2$ *then* $\mathsf{abs}^{\mathsf{rls}}_{T_1} \preceq \mathsf{abs}^{\mathsf{rls}}_{T_2}$.

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 §5.1. However, the domain consists of ordered lists segments.

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

$$\mathsf{list}_{\leq}(e, f, [\alpha, \beta\rangle, S) = \mathsf{list}_{\leq}(e, x', [\alpha, \gamma\rangle, S \cap [\alpha, \gamma\rangle) * \mathsf{list}_{\leq}(x', f, [\gamma, \beta\rangle, S \cap [\gamma, \beta\rangle).$$

Folding/unfolding rules for exposing/hiding are similar to the rules for list (Fig. 4), but in addition keep track of the involved inequalities. New rules for $\mathsf{list}_{\leq}$ are shown in Fig. 7. Note that each rule maintains the invariant I.

Abstraction. In the abstraction, we proceed similarly as in Fig. 5 but also maintain the invariant I. Fig. 8 shows rewrite rules corresponding to the fourth rule of Fig. 5 for $\sigma_1 = \sigma_2 = \mathsf{node}$ and $\sigma_1 = \sigma_2 = \mathsf{list}_{\leq}$. The rest of the cases for σ_i are analogous to the fourth rule, and the fifth rule of Fig. 5. The resulting abstraction $\mathsf{abs}^{\mathsf{sls}}_T$ satisfies the following lemmas:

$$\Delta * \mathsf{node}(e_1, x', \{d_1\}) * \mathsf{node}(x', e_2, \{d_2\}) \rightsquigarrow^{\mathsf{sls}}_T$$
$$\Delta * \mathsf{list}_{\leq}(e_1, \mathsf{nil}, [d_1, d_2 + 1\rangle, \mathsf{pr}_T(\{d_1, d_2\}, \Pi))$$
$$\text{if } x' \notin \mathsf{EVars}(\Delta, e_1, e_2) \wedge \Delta \vdash e_2 = \mathsf{nil}$$

$$\Delta * \mathsf{list}_{\leq}(e_1, x', [\alpha_1, \beta_1\rangle, S_1) * \mathsf{list}_{\leq}(x', e_2, [\alpha_2, \beta_2\rangle, S_2) \rightsquigarrow^{\mathsf{sls}}_T$$
$$\Delta * \mathsf{list}_{\leq}(e_1, \mathsf{nil}, [\alpha_1, \beta_2\rangle, \mathsf{pr}_T(S_1 \cup S_2, \Pi))$$
$$\text{if } x' \notin \mathsf{EVars}(\Delta, e_1, e_2) \wedge \Delta \vdash e_2 = \mathsf{nil} \wedge \beta_1 \leq \alpha_2$$

Fig. 8. Selected rules of the abstract reduction system $\rightsquigarrow^{\mathsf{sls}}_T$ defining the abstraction function $\mathsf{abs}^{\mathsf{sls}}_T$.

Lemma 6. *For* $\mathsf{CSH}^{\mathsf{rss}}_T \triangleq \{\Delta \mid \Delta \nvdash \mathsf{false} \wedge \Delta \not\rightsquigarrow^{\mathsf{sls}}_T\}$, $\mathsf{abs}^{\mathsf{sls}}_T : \mathsf{SH} \rightarrow \mathsf{CSH}^{\mathsf{rls}}_T$ *is a sound abstraction. If the domain of values is finite then* $\mathsf{CSH}^{\mathsf{rss}}_T$ *is also finite.*

Lemma 7. *If* $T_1 \subseteq T_2$ *then* $\mathsf{abs}^{\mathsf{sls}}_{T_1} \preceq \mathsf{abs}^{\mathsf{sls}}_{T_2}$.

For infinite value domains, the set $\mathsf{CSH}^{\mathsf{rss}}_T$ is infinite since we have infinite ascending chains of intervals as parameters to $\mathsf{list}_{\leq}$. We could recover convergence in such cases by using widening on the interval domain [11].