Statically Inferring Complex Heap, Array, and Numeric Invariants

Bill McCloskey[1], Thomas Reps[2,3⋆], and Mooly Sagiv[4,5⋆⋆]

[1] University of California; Berkeley, CA, USA
[2] University of Wisconsin; Madison, WI, USA
[3] GrammaTech, Inc.; Ithaca, NY, USA
[4] Tel-Aviv University; Tel-Aviv, Israel
[5] Stanford University; Stanford, CA, USA

Abstract. We describe Deskcheck, a parametric static analyzer that is able to establish properties of programs that manipulate dynamically allocated memory, arrays, and integers. Deskcheck can verify quantified invariants over mixed abstract domains, e.g., heap and numeric domains. These domains need only minor extensions to work with our domain combination framework.

The technique used for managing the communication between domains is reminiscent of the Nelson-Oppen technique for combining decision procedures, in that the two domains share a common predicate language to exchange shared facts. However, whereas the Nelson-Oppen technique is limited to a common predicate language of shared equalities, the technique described in this paper uses a common predicate language in which shared facts can be quantified predicates expressed in first-order logic with transitive closure.

We explain how we used Deskcheck to establish memory safety of the `thttpd` web server's cache data structure, which uses linked lists, a hash table, and reference counting in a single composite data structure. Our work addresses some of the most complex data-structure invariants considered in the shape-analysis literature.

1 Introduction

Many programs use data structures for which a proof of correctness requires a combination of heap and numeric reasoning. Deskcheck, the tool described in this paper, is targeted at such programs. For example, consider a program that uses an array, *table*, whose entries point to heap-allocated objects. Each object has an *index* field. We want to check that if $table[k] = obj$, then $obj.index = k$. In verifying the correctness of the `thttpd` web server [22], this invariant is required

⋆ Supported, in part, by NSF under grants CCF-{0810053, 0904371}, by ONR under grant N00014-{09-1-0510}, by ARL under grant W911NF-09-1-0413, and by AFRL under grant FA9550-09-1-0279.

⋆⋆ Supported, in part, by grants NSF CNS-050955 and NSF CCF-0430378 with additional support from DARPA.

even to prove memory safety. Formally, we write the following (ignoring array bounds for now):

$$\forall k{:}\mathbb{Z}.\ \forall o{:}\mathbb{H}.\ \mathit{table}[k] = o \Rightarrow (o.\mathit{index} = k \vee o = \mathit{null}) \tag{1}$$

We call this invariant $\mathsf{Inv1}$. It quantifies over both heap objects and integers. Such quantified invariants over mixed domains are beyond the power of most existing static analyzers, which typically infer either heap invariants or integer invariants, but not both.

Our approach is to combine existing abstract domains into a single abstract interpreter that infers mixed invariants. In this paper, we discuss examples using a particular heap domain (canonical abstraction) and a particular numeric domain (difference-bound matrices). However, the approach supports a wide variety of domain combinations, including combinations of two numeric domains, and a combination of the separation-logic shape domain [9] and polyhedra.

Our goal is for the combined domain to be more than the sum of its parts: to be able to infer facts that neither domain could infer alone. As in previous research on combining domains, communication between the two domains is the crucial ingredient. The combined domain of Gulwani and Tiwari [15], based on the Nelson-Oppen technique for combining decision procedures [20], shares equalities between domains. Our technique also uses a common predicate language to share facts; however, in our approach shared facts can be predicates from first-order logic with transitive closure.

Approach. We assume that each domain being combined reasons about a distinct collection of abstract "individuals" (heap objects, or integers, say). Every domain is responsible for grouping its individuals into sets, called *classes*. A heap domain might create a class of all objects belonging to a linked list, while an integer domain may have a class of numbers between 3 and 10.

Additionally, each domain D exposes a set of n-ary predicates to other domains. Every predicate has a definition, such as "$\mathsf{R}(o_1, o_2)$ holds if object o_1 reaches o_2 via *next* edges." Only the defining domain understands the meaning of its predicates. However, quantified atomic facts are shared between domains: a heap domain D might share with another domain the fact that $(\forall o_1 \in C_1, o_2 \in C_2.\ \mathsf{R}(o_1, o_2))$, where C_1 and C_2 are classes of list nodes. Other domains can define their own predicates in terms of R. They must depend on shared information from D to know where R holds because they are otherwise ignorant of R's semantics.

Chains of dependencies can exist between predicates in different domains. A predicate $\mathsf{P2}$ in domain D' can refer to a predicate $\mathsf{P1}$ in D. Then a predicate $\mathsf{P3}$ in D can refer to $\mathsf{P2}$ in D'. The only restriction is that dependencies be acyclic. As transfer functions execute, atomic facts about predicates propagate between domains along the dependency edges. This flexibility enables our framework to reason precisely about mixed heap and numeric invariants.

A Challenging Verification Problem. We have applied DESKCHECK to the cache module of the `thttpd` web server [22]. We chose this data structure because it relies on several invariants that require combined numeric and heap reasoning. We believe this data structure is representative of many that appear in systems code, where arrays, lists, and trees are all used in a single composite data structure, sometimes with reference counting used to manage deallocation. Along with DESKCHECK, our model of `thttpd`'s cache is available online for review [18].

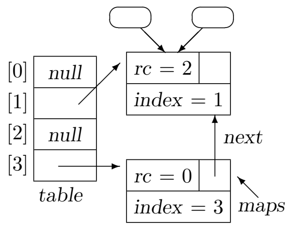


Fig. 1. `thttpd`'s cache data structure.

The `thttpd` cache maps files on disk to their contents in memory. Fig. 1 displays an example of the structure. It is a composite between a hash table and a linked list. The linked list of cache entries starts at the *maps* variable and continues through *next* pointers. These same cache entries are also pointed to by elements of the *table* array. The *rc* field records the number of incoming pointers from external objects (i.e., not counting pointers from the *maps* list nor from *table*), represented by rounded rectangles. The reference count is allowed to be zero.

Fig. 2 shows excerpts of the code to add an entry to the cache. Besides the data structures already discussed, the variable *free_maps* is used to track unused cache entries (to avoid calling `malloc` and `free`). Our goal is to verify that this code, as well as the related code for releasing and freeing cache entries, is memory-safe. One obvious data-structure invariant is that *maps* and *free_maps* should point to acyclic singly linked lists of cache entries. However, there are two other invariants that are more complex but required for memory safety.

Inv1 (from Eqn. (1)): When a cache entry e is freed, `thttpd` nulls out its hash table entry via `table[e.index] = null` (this code is not shown in Fig. 2). If the wrong element were overwritten, then a pointer to the freed entry would remain in *table*, later leading to a segfault when accessed. Inv1 guarantees that if $table[i] = e$, where e is the element being freed, then $e.index = i$, so the correct entry will be set to null.

Inv2: This invariant relates to reference counting. The two main entry points to the cache module are called `map` and `unmap`. The `map` call creates a cache entry if it does not already exist and returns it to the caller. The caller can use the entry until it calls `unmap`. The cache keeps a reference count of the number of outstanding uses of each entry; when the count reaches zero, it is legal (although not necessary) to free the entry. Outstanding references are shown as rounded rectangles in Fig. 1. The cache must maintain the invariant that the number

```
1  Map * map(...)
2  { /* Expand hash table if needed */
3    check_hash_size();
4    m = find_hash(...);
5    if (m != (Map*)0) {
6      /* Found an entry */
7      ++m->refcount;
8      ...
9      return m;
10   }
11   /* Find a free Map entry
12      or make a new one. */
13   if (free_maps != (Map*)0) {
14     m = free_maps;
15     free_maps = m->next;
16   } else {
17     m = (Map*)malloc(sizeof(Map));
18   }
19   m->refcount = 1;
20   ...
21   /* Add m to hashtable */
22   if (add_hash(m) < 0) {
23     /* error handling code */
24   }
25   /* Put m on active list. */
26   m->next = maps;
27   maps = m;
28   ...
29   return m;
30 }
31 static int add_hash(Map* m)
32 { ...
33   int i = hash(m);
34   table[i] = m;
35   m->index = i;
36   ...
37 }
```

Fig. 2. Excerpts of the `thttpd` `map` and `add_hash` functions.

of outstanding references is equal to the value of an entry's reference count (*rc*) field—otherwise an entry could be freed while still in use. We can write this invariant formally as follows. Assuming that cache entries are stored in the *entry* field of the caller's objects (the ones shown by rounded rectangles), we wish to ensure that the number of *entry* pointers to a given object is equal to its *rc* field.

$$\mathsf{Inv2} \overset{\text{def}}{=} \forall o{:}\mathbb{H}.\ o.rc = |\{p{:}\mathbb{H} \mid p.entry = o\}| \tag{2}$$

Verification. We give an example of how Inv1 is verified. §4.3 has a more detailed presentation of this example. The program locations of interest are lines 34 and 35 of Fig. 2, where the hash table is updated. Recall that Inv1 requires that if $table[k] = e$ then $e.index = k$. After line 34, Inv1 is broken, although only "locally" (i.e., at a single index position of *table*). As a first step, we parametrize Inv1 by dropping the quantifier on k, allowing us to distinguish between index positions at which Inv1 is broken and those where it continues to hold.

$$\mathsf{Inv1}(k{:}\mathbb{Z}) \overset{\text{def}}{=} \forall o{:}\mathbb{H}.\ table[k] = o \Rightarrow (o.index = k \lor o = null)$$

After line 34 we know that $\mathsf{Inv1}(x)$ holds for all $x \neq i$. Line 35 restores $\mathsf{Inv1}(i)$.

Neither domain fully understands the defining formula of Inv1: as we will see, the variable *table* is understood only by the heap domain whereas the field *index* is understood only by the integer domain. Consequently, we factor out the integer portion of Inv1 into a separate predicate, as follows.

$$\begin{aligned} \mathsf{Inv1}(k{:}\mathbb{Z}) &\overset{\text{def}}{=} \forall obj{:}\mathbb{H}.\ table[k] = o \Rightarrow (\mathsf{HasIdx}(o,k) \vee o = null) \\ \mathsf{HasIdx}(o{:}\mathbb{H}, k{:}\mathbb{Z}) &\overset{\text{def}}{=} o.index = k \end{aligned}$$

Now Inv1 is understood by the heap domain and HasIdx is understood by the integer domain.

Deskcheck splits the analysis effort between the heap domain and the numeric domain. Line 34 is initially processed by the heap domain because it assigns to a pointer location. However, the heap domain knows nothing about i, an integer. Before executing the assignment, the integer domain is asked to find an integer class containing i. Call this class N_i. Assume that all other integers are grouped into a class $N_{\neq i}$. Then the heap domain essentially treats the assignment on line 34 as $table[N_i] := m$. Since the predicate $\mathsf{HasIdx}(m, i)$ is false at this point, the assignment causes Inv1 to be falsified at N_i. Given information from the integer domain that N_i and $N_{\neq i}$ are disjoint, the heap domain can recognize that remains true at $N_{\neq i}$.

Line 35 is handled by the integer domain because the value being assigned is an integer. The heap domain is first asked to convert m to a class, H_m, so that the integer domain knows where the assignment takes place. After performing the assignment as usual, the integer domain informs the heap domain that $(\forall o \in H_m, n \in N_i.\ \mathsf{HasIdx}(o, n))$ has become true. The heap domain then recognizes that Inv1 becomes true at N_i, restoring the invariant.

Limitations. It is important to understand the limitations of our work. The most important limitation is that *shared predicates, like Inv1 and HasIdx, must be provided by the user of the analysis.* Without shared predicates, our combined domain is no more (or less) precise than the work of Gulwani et al. [14]. The predicates that we supply in our examples tend to follow directly from the properties we want to prove, but supplying their definitions is still an obligation left to the Deskcheck user. Another limitation, which applies to our implementation, is that the domains we are combining sometimes require annotations to the code being analyzed. These annotations do not affect soundness, but they may affect precision and efficiency. We describe both the predicates and the annotations we use for the `thttpd` web server in §5.

Two more limitations affect our implementation. First, it handles calls to functions via inlining. Besides not scaling to larger codebases, inlining cannot handle recursive functions. The use of inlining is not fundamental to our technique, but we have not yet developed a more effective method of analyzing procedures. We emphasize, though, that *we do not require any loop invariants or procedure pre-conditions or post-conditions from the user.* All invariants are inferred by abstract interpretation. We seed the analysis with an initially empty heap.

The final limitation is that our tool requires the user to manually translate C code to a special analysis language similar to BoogiePL [7]. This step could easily be automated, but we have not had time to do it.

Contributions. The contributions of our work can be summarized as follows: **(1)** We present a method to infer quantified invariants over mixed domains while using separate implementations of the different domains. **(2)** We describe an instantiation of Deskcheck based on canonical abstraction for heap properties and difference constraints for numeric properties. We explain how this analyzer is able to establish memory-safety properties of the `thttpd` cache. The system is publicly available online [18]. **(3)** Along with the work of Berdine et al. [2], our work addresses the most complex data-structure invariants considered in the shape-analysis literature. The problems addressed in the two papers are complementary: Berdine et al. handle complex *structural* invariants for nests of linked structures (such as "cyclic doubly linked lists of acyclic singly linked lists"), whereas our work handles complex *mixed-domain* invariants for data structures with both linkage and numeric constraints, such as the structure depicted in Fig. 1.

Organization. §2 summarizes the modeling language and the domain-communication mechanism on which Deskcheck relies. §4 describes how Deskcheck infers mixed numeric and heap properties. §5 presents experimental results. §6 discusses related work.

2 Deskcheck Architecture

2.1 Modeling of Programs

Programs are input to Deskcheck in an imperative language similar to BoogiePL [7]. We briefly describe the syntax and semantics, because this language is used in all this paper's examples. The syntax is Pascal-like. An example program is given in Fig. 3. This program checks that each entry in a linked list has a data field of zero; this field is then set to one.

Line 1 declares a type `T` of list nodes. Lines 3–5 define a set of *uninterpreted functions*. Our language uses uninterpreted functions to model variables, fields, and arrays uniformly. The *next* function models a field: it maps a list node to another list node, so its *signature* is `T` $\to$ `T`. The *data* function models an integer field of list nodes. And *head* models a list variable; it is a nullary function. Note that an array `a` of type `T` would be written as `a[int]:T`. At line 8, *cur* is a procedure-local nullary uninterpreted function (another `T` variable).

The semantics of our programs is similar to the semantics of a many-sorted logic. Each type is a sort, and the type `int` also forms a sort. For each sort there is an infinite, fixed universe of individuals. (We model allocation and deallocation with a free list.) A concrete program state maps uninterpreted function names to mathematical functions having the correct signature. For example, if $U_{\mathtt{T}}$ is the universe of `T`-individuals, then the semantics of the *data* field is given by some function drawn from $U_{\mathtt{T}} \to \mathbb{Z}$.

```
type T;

global next[T]:T;
global data[T]:int;
global head:T;

procedure iter()
  cur:T;
{ cur := head;
  while (cur != null) {
    assert(data[cur] = 0);
    data[cur] := 1;
    cur := next[cur];
  }
}
```

Fig. 3. A program for traversing a linked list.

2.2 Base Domains

Deskcheck combines the power of several abstract domains into a single combined domain. In our experiments, we used a combination of canonical abstraction for heap reasoning and difference-bound matrices for numeric reasoning. However, combinations using separation logic or polyhedra are theoretically possible.

Canonical abstraction [24] partitions heap objects into disjoint sets based on the properties they do or do not satisfy. For example, canonical abstraction might group together all objects reachable from a variable x but not reachable from y. When two objects are grouped together, only their common properties are preserved by the analysis. A canonical abstraction with many groups preserves more distinctions between objects but is more expensive. Using fewer groups is faster but less precise.

Canonical abstraction is a natural fit for Deskcheck because it already relies on predicates. Each canonical name corresponds fairly directly to a class in the Deskcheck setting. Deskcheck allows each domain to decide how objects are to be partitioned into classes: in canonical abstraction we use predicates to decide. We use a variant of canonical abstraction in which a summary node summarizes 0 or more individuals [1] (rather than 1 or more as in most other systems).

Our numeric domain is the familiar domain of difference-bound matrices. It tracks constraints of the form $t_1 - t_2 \leq c$, where t_1 and t_2 are uninterpreted function terms such as $f[\mathrm{x}]$. We use a summarizing numeric domain [12], which is capable of reasoning about function terms as dimensions in a sound way.

The user is allowed to define numeric predicates. These predicates are defined using a simple quantifier-free language permitting atomic numerical facts, conjunction, and disjunction. A typical predicate might be $\mathsf{Bounded}(n) := n \geq$

$0 \wedge n < 10$. Similar to canonical abstraction, we use these numeric predicates to partition the set of integers into disjoint classes. These integer classes permit array reasoning, as explained later in §4.2.

2.3 Combining Domains

In the DESKCHECK architecture, work is partitioned between n domains. Typically $n = 2$, although all of our work extends to an arbitrary number of base domains. Besides the usual operations like join and assignment, these domains must be equipped to share quantified atomic facts and class information.

Each domain is responsible for some of the sorts defined above. In our implementation, the numeric domain handles `int` and the heap domain handles all other types. An uninterpreted function is associated with an abstract domain according to the type of its range. In Fig. 3, *next* and *head* are handled by the heap domain and *data* by the numeric domain. Assignments statements to uninterpreted functions are initially handled by the domain with which they are associated.

Predicates are also associated with a given domain. Each domain has its own language in which its predicates are defined. Our heap domain supports universal and existential quantification and transitive closure over heap functions. Our numeric domain supports difference constraints over numeric functions along with cardinality reasoning. A predicate associated with one domain may refer to a predicate defined in another domain, although cyclic references are forbidden. The user is responsible for defining all predicates. The precision of an analysis depends on a good choice of predicates; however, soundness is guaranteed regardless of the choice of predicates.

Classes. A class, as previously mentioned, represents a set of individuals of a given sort (integers, heap objects of some type, etc.). A class can be a singleton, having one element, or a summary class, having an arbitrary number of elements (including zero). Summary classes are written in bold, as in $\boldsymbol{N_{\neq i}}$, to distinguish them.

The grouping of individuals into classes may be flow-sensitive—we do not assume that the classes are known prior to the analysis. At any time a domain is allowed to change this grouping, in a process called *repartitioning*. Classes of a given sort are repartitioned by the domain to which that sort is assigned. When a domain repartitions its classes, other domains are informed as described below.

Semantics. Each domain D_i can choose to represent its abstract elements however it desires. To define the semantics of a combined element $\langle E_1, E_2 \rangle$, we require each domain D_i to provide a meaning function, $\widehat{\gamma}_i(E_i)$, that gives the meaning of E_i as a logical formula. This formula may contain occurrences of uninterpreted functions that are managed by D_i as well as classes and predicates managed by any of the domains.

We will define a function $\gamma(\langle E_1, E_2 \rangle)$ that gives the semantics of a combined abstract element. Instead of evaluating to a logical formula, this function returns a set of concrete states that satisfy the constraints of E_1 and E_2. A concrete state is an interpretation that assigns values to all the uninterpreted functions used by the program.

Naively, we could define $\gamma(\langle E_1, E_2\rangle)$ as the set of states that satisfy formulas $\widehat{\gamma}_1(E_1)$ and $\widehat{\gamma}_2(E_2)$. However, these formulas refer to classes and predicates, which do not appear in the state. To solve the problem, we let $\gamma(\langle E_1, E_2\rangle)$ be the set of states satisfying $\widehat{\gamma}_1(E_1)$ and $\widehat{\gamma}_2(E_2)$ for *some* interpretation of predicates and classes. We can state this formally using second-order quantification. Here, each P_i is a predicate defined by D_1 or D_2. Each C_i is a class appearing in E_1 or E_2. The number of classes, $n(E_1, E_2)$, depends on E_1 and E_2.

$$\gamma(\langle E_1, E_2\rangle) \stackrel{\text{def}}{=} \{S : S \models \exists P_1. \cdots \exists P_m.\ \exists C_1. \cdots \exists C_{n(E_1,E_2)}.\ \widehat{\gamma}_1(E_1) \wedge \widehat{\gamma}_2(E_2)\}$$

Typically, $\widehat{\gamma}_i(E_i)$ is the conjunction of three subformulas. One subformula gives meaning to the predicates defined by D_i and another gives meaning to the classes defined by D_i. The third subformula, the only one specific to E_i, gives meaning to the constraints in E_i.

We can be more specific about the forms of these three subformulas. A subformula defining a unary predicate P that holds when its argument is positive would look as follows.

$$\forall x.\ \mathsf{P}(x) \iff x > 0$$

In our implementation of the analysis, all predicate definitions must be given by the user. Note that a predicate definition may refer to another predicate (possibly one defined by another base domain). For example, the following predicate might apply to heap objects, stating that their *data* field is positive.

$$\forall o.\ \mathsf{Q}(o) \iff \mathsf{P}(data[o])$$

A subformula that defines a class C containing the integers from 0 to n would look as follows.

$$C = \{x : 0 \leq x < n\}$$

Our implementation uses canonical abstraction [24] to decide how individuals are grouped into classes. Therefore, the definition of a class will always have the following form:

$$C = \{x : \mathsf{P}(x) \wedge \mathsf{Q}(x) \wedge \neg\mathsf{R}(x) \wedge \cdots\}$$

That is, the class contains exactly those object satisfying a set of unary predicates and not satisfying another set of unary predicates. Such unary predicates are called *abstraction predicates.* The user chooses which subset of the unary predicates are abstraction predicates. In theory there can be one class for every subset of the abstraction predicates, but in practice most of these classes are empty and thus not used. Because each class is defined by the abstraction predicates it satisfies (the non-negated ones), this subset of predicates is called the class's *canonical name.*

Subformulas that give meaning to the constraints in E_i are specific to the domain D_i. For example, an integer domain would include constraints like $x - y \leq$

c. A heap domain might include constraints about reachability. Both domains will often include quantified facts of the following form:

$$\forall o \in C.\ \mathsf{Q}(o)$$

A domain may quantify over a class defined by any of the domains and it may use predicates from any of the domains. The predicate that appears may optionally be negated. Facts like this may be exchanged freely between domains because they are written in a common language of predicates and classes. To distinguish the more domain-specific facts like $x - y \leq c$ from the ones exchanged between domains, we surround them in angle brackets. A fact $\langle\ \cdot\ \rangle_H$ is specific to a heap domain and $\langle\ \cdot\ \rangle_N$ is specific to a numeric domain.

3 Domain Operations

This section describes the partial order and join operation of the combined domain and also the transfer function for assignment. These operations make use of their counterparts in the base domains as well as some additional functions that we explain below.

3.1 Partial Order

We can define a very naive partial-order check for the combined domain as follows.

$$\langle E_1^A, E_2^A \rangle \sqsubseteq \langle E_1^B, E_2^B \rangle \iff (E_1^A \sqsubseteq_1 E_1^B) \wedge (E_2^A \sqsubseteq_2 E_2^B)$$

Here, we have assumed that $\sqsubseteq_1$ and $\sqsubseteq_2$ are the partial orders for the base domains.

However, there are two problems with this approach. The first problem is illustrated by the following example. (Assume that class C and predicate P are defined by D_1.)

$$\begin{aligned} E_1^A &= \forall x \in C.\ \mathsf{P}(x) \qquad & E_1^B &= \text{true} \\ E_2^A &= \text{true} & E_2^B &= \forall x \in C.\ \mathsf{P}(x) \end{aligned}$$

If we work out $\gamma(\langle E_1^A, E_2^A \rangle)$ and $\gamma(\langle E_1^B, E_2^B \rangle)$, they are identical. Thus, we should obtain $\langle E_1^A, E_2^A \rangle \sqsubseteq \langle E_1^B, E_2^B \rangle$. However, the partial-order check given above does not, because it is not true that $E_2^A \sqsubseteq_2 E_2^B$.

To solve this problem, we *saturate* E_1^A and E_2^A before applying the base domains' partial orders. That is, we strengthen these elements by exchanging any facts that can be expressed in a common language. (Note that E_1^A and E_2^A are individually strengthened but $\gamma(\langle E_1^A, E_2^A \rangle)$ remains the same; saturation is a semantic reduction.) In the example, the fact $\forall x \in C.\ \mathsf{P}(x)$ is copied from E_1^A to E_2^A.

Any fact drawn from the following grammar can be shared.

$$F ::= \forall x \in C.\ F \mid \exists x \in C.\ F \mid \mathsf{P}(x, y, \ldots) \mid \neg\mathsf{P}(x, y, \ldots) \tag{3}$$

Here, C is an arbitrary class and P is an arbitrary predicate. All variables appearing in $\mathsf{P}(x, y, \ldots)$ must be bound by quantifiers.

```
function Saturate($E_1, E_2$):
    $F := \emptyset$
    repeat:
        $F_0 := F$
        $F := F \cup \mathit{Consequences}_1(E_1) \cup \mathit{Consequences}_2(E_2)$
        $E_1 := \mathit{Assume}_1(E_1, F)$
        $E_2 := \mathit{Assume}_2(E_2, F)$
    until $F_0 = F$
    return $\langle E_1, E_2 \rangle$
```

Fig. 4. Implementation of combined-domain saturation.

To implement sharing, each domain D_i is required to expose an Assume_i function and a $\mathit{Consequences}_i$ function. $\mathit{Consequences}_i$ takes a domain element and returns all facts of the form above that it implies. Assume_i takes a domain element E and a fact f of the form above and returns an element that approximates $E \wedge f$. The pseudocode in Fig. 4 shows how facts are propagated. They are accumulated via $\mathit{Consequences}_i$ and then passed to the domains with Assume_i. Because we require that the number of predicates and classes in any element is bounded, this process is guaranteed to terminate.

We update the naive partial-order check as follows. If $\langle E_1^{A^*}, E_2^{A^*} \rangle = \text{Saturate}(E_1^A, E_2^A)$, then

$$\langle E_1^A, E_2^A \rangle \sqsubseteq \langle E_1^B, E_2^B \rangle \iff (E_1^{A^*} \sqsubseteq_1 E_1^B) \wedge (E_2^{A^*} \sqsubseteq_2 E_2^B)$$

Note that we only saturate the left-hand element; strengthening the right-hand element is sound, but it does not improve precision.

This ordering is still too imprecise. The problem is that the A and B elements may use different class names to refer to the same set of individuals. As an example, consider the following.

$$E_1^A = \forall x \in C.\ \mathsf{P}(x) \qquad E_1^B = \forall x \in C'.\ \mathsf{P}(x)$$
$$E_2^A = (C = \{x : x > 0\}) \qquad E_2^B = (C' = \{x : x > 0\})$$

It's clear that C and C' both refer to the same sets. Therefore, $\gamma(\langle E_1^A, E_2^A \rangle)$ is equal to $\gamma(\langle E_1^B, E_2^B \rangle)$; the difference in naming between C and C' is irrelevant to γ because it projects out class names using an existential quantifier. However, our naive partial-order check cannot discover the equivalence.

To solve the problem, we rename the classes appearing in $\langle E_1^A, E_2^A \rangle$ so that they match the names used in $\langle E_1^B, E_2^B \rangle$. This process is done in two steps: (1) match up the classes in the A element with those in the B element, (2) rewrite the A element's classes according to step 1. In the example above, we get the rewriting $\{C \mapsto C'\}$ in step 1, which is used to rewrite E_1^A and E_2^A as follows.

$$\begin{array}{ll} E_1^A = \forall x \in \mathbf{C'}.\ \mathsf{P}(x) & \quad E_1^B = \forall x \in C'.\ \mathsf{P}(x) \\ E_2^A = (\mathbf{C'} = \{x : x > 0\}) & \quad E_2^B = (C' = \{x : x > 0\}) \end{array}$$

We only rewrite the A elements because rewriting may weaken the abstract element and it is unsound to weaken the B elements in a partial order check. Our partial order is sound with respect to γ, but it may be incomplete. Its completeness depends on the completeness of the base domain operations like $\mathtt{MatchClasses}_i$, and typically these operations are incomplete.

Recall that each class is managed by one domain but may still be referenced by other domains. In the matching step, each domain is responsible for matching its own classes. In our implementation, we match up classes according to their canonical names. Then the rewritings for all domains are combined and every domain element is rewritten using the combined rewriting. In the example above, D_2 defines classes C and C', so it is responsible for matching them. But both E_1^A and E_2^A are rewritten.

```
**function** $\langle E_1^A, E_2^A \rangle \sqsubseteq \langle E_1^B, E_2^B \rangle$:
    $\langle E_1^A, E_2^A \rangle := \text{Saturate}(E_1^A, E_2^A)$

    $R_1 := \mathtt{MatchClasses}_1(E_1^A, E_1^B)$
    $R_2 := \mathtt{MatchClasses}_2(E_2^A, E_2^B)$

    $E_1^{A'} := \mathtt{Repartition}_1(E_1^A, R_1 \cup R_2)$
    $E_2^{A'} := \mathtt{Repartition}_2(E_2^A, R_1 \cup R_2)$

    **return** $(E_1^{A'} \sqsubseteq_1 E_1^B) \wedge (E_2^{A'} \sqsubseteq_2 E_2^B)$
```

Fig. 5. Pseudocode for combined domain's partial order.

Pseudocode that defines the partial-order check for the combined domain is shown in Fig. 5. First, E^A is saturated and its classes are matched to the classes in E^B. Each domain is required to expose a $\mathtt{MatchClasses}_i$ operation that matches the classes it manages. The rewritings R_1 and R_2 are combined and then E^A is rewritten via the $\mathtt{Repartition}_i$ operations that each domain must also expose. Finally, we apply each base domain's partial order to obtain the final result.

3.2 Join and Widening

The join algorithm is similar to the partial-order check. We perform saturation, rewrite the class names, and then apply each base domain's join operation independently. The difference is that join is handled symmetrically: both elements are saturated and rewritten. Instead of matching the classes of E^A to the classes of E^B, we allow both inputs to be repartitioned into a new set of classes that may be more precise than either of the original sets of classes. Thus, we require domains to expose a *MergeClasses$_i$* operation that returns a mapping from either element's original classes to new classes.

```
**function** $\langle E_1^A, E_2^A \rangle \sqcup \langle E_1^B, E_2^B \rangle$:
    $\langle E_1^A, E_2^A \rangle :=$ Saturate$(E_1^A, E_2^B)$
    $\langle E_1^B, E_2^B \rangle :=$ Saturate$(E_1^B, E_2^B)$

    $\langle R_1^A, R_1^B \rangle := \mathit{MergeClasses}_1(E_1^A, E_1^B)$
    $\langle R_2^A, R_2^B \rangle := \mathit{MergeClasses}_2(E_2^A, E_2^B)$

    $E_1^{A'} := \mathit{Repartition}_1(E_1^A, R_1^A \cup R_2^A)$
    $E_2^{A'} := \mathit{Repartition}_2(E_2^A, R_1^A \cup R_2^A)$

    $E_1^{B'} := \mathit{Repartition}_1(E_1^B, R_1^B \cup R_2^B)$
    $E_2^{B'} := \mathit{Repartition}_2(E_2^B, R_1^B \cup R_2^B)$

    **return** $\langle (E_1^{A'} \sqcup_1 E_1^{B'}), (E_2^{A'} \sqcup_2 E_2^{B'}) \rangle$
```

Fig. 6. Pseudocode for combined domain's join algorithm.

The pseudocode for join is shown in Fig. 6. First, E^A and E^B are saturated. Then *MergeClasses$_1$* and *MergeClasses$_2$* are called to generate four rewritings. The rewriting R_i^A describes how to rewrite the classes in E^A that are managed by D_i into new classes. Similarly, R_i^B describes how to rewrite the classes in E^B that are managed by D_i. Finally, E^A and E^B are rewritten and the base domains' joins are applied. When rewriting E^A, we need both R_1^A and R_2^A because classes managed by one base domain can be referenced by the other.

We must define a widening operation for the combined domain as well. The widening algorithm is very similar to the join algorithm. Recall that the purpose of widening is to act like a join while ensuring that fixed-point iteration will terminate eventually. Due to the termination requirement, we make some changes to the join algorithm.

The challenging part of widening is that some widenings that are "obviously correct" may fail to terminate. Miné [19] describes how this can occur in an integer domain. Widening typically works by throwing away facts, producing a less precise element, to reach a fixed point more quickly. The problem occurs if we try to saturate the left-hand operand. Saturation will put back facts that we might have thrown away, thereby defeating the purpose of widening. So to ensure that a widened sequence terminates, we never saturate the left-hand operand. The code is in Fig. 7.

```
**function** $\langle E_1^A, E_2^A \rangle \;\nabla\; \langle E_1^B, E_2^B \rangle$:
    $\langle E_1^B, E_2^B \rangle :=$ Saturate$(E_1^B, E_2^B)$

    $R_1 := \mathtt{MatchClasses}_1(E_1^B, E_1^A)$
    $R_2 := \mathtt{MatchClasses}_2(E_2^B, E_2^A)$

    $E_1^{B'} := \mathtt{Repartition}_1(E_1^B, R_1 \cup R_2)$
    $E_2^{B'} := \mathtt{Repartition}_2(E_2^B, R_1 \cup R_2)$

    **return** $\langle (E_1^A \;\nabla_1\; E_1^{B'}), (E_2^A \;\nabla_2\; E_2^{B'}) \rangle$
```

Fig. 7. Combined domain's widening algorithm.

This code is very similar to the code for the join algorithm. Besides avoiding saturation of E^A, we also avoid repartitioning E^A. Our goal is to avoid any changes to E^A that might cause the widening to fail to terminate. Because we do not repartition E^A, we use $\mathtt{MatchClasses}_i$ instead of $\mathtt{MergeClasses}_i$.

3.3 Assignment

Assignment in the combined domain must solve two problems. First, each base-domain element must be updated to account for the assignment. Second, any changes to the shared predicates and classes must be propagated between domains. We simplify the matter somewhat by declaring that an assignment operation cannot affect classes. That is, the set of individuals belonging to a class is not affected by assignments. However, a predicate that once held over the members of a class may no longer hold, and vice versa.

Base facts. We deal with updating the base domains first, and we deal with predicates later. We require each base domain to provide an assignment transfer function to process assignments. An assignment operation has the form $f[e_1, \ldots, e_k] := e$, where f is an uninterpreted function and $e, e_1, \ldots, e_k$ are all terms made up of applications of uninterpreted functions. The assignment transfer function of domain D_i is invoked as $\mathtt{Assign}_i(E_i, f[e_1, \ldots, e_k], e)$. Each uninterpreted function is understood by only one base domain; we use the transfer function of the domain that understands f. The other domain is left unchanged.

Assume that D_1 understands f so that $\texttt{Assign}_1$ is invoked. The problem is that any of e or $e_1, \ldots, e_k$ may use uninterpreted functions that are understood by D_2 and not by D_1. In this case, D_1 will not know the effect of the assignment. To overcome this problem, we ask D_2 to replace any "foreign" term appearing in e and $e_1, \ldots, e_k$ with a class that is guaranteed to contain the individual to which the term evaluates. Because classes have meaning to both domains, it is now possible for D_1 to process the assignment.

Replacement of foreign terms with classes must be done recursively, because function applications may contain other function applications. The process is shown in pseudocode in Fig. 8 via the TranslateFull$_i$ functions. The function TranslateFull$_1$ replaces any D_2 terms with classes. When it sees a D_2 function application, it translates the arguments of the function application to terms understood by D_2 and then asks D_2, via the $\texttt{Translate}_2$ function that it must expose, to replace the entire application with a class.

As an example, consider the term $f[c]$, where f is understood by D_1 and c is understood by D_2. If we call TranslateFull$_1$ on this term, then c is converted by D_2 to a class, say C, that contains the value of c. The resulting term is $f[C]$, which is understandable by D_1. If, instead, we called TranslateFull$_2$ on $f[c]$, we would again convert c to a class C. Then we would ask D_1 to convert $f[C]$ to a class, say F, which must contain the value of $f[x]$ for any $x \in C$. The result is a class, say F, which is understood by D_2.

Predicates. Besides returning an updated domain element, we require that the $\texttt{Assign}_i$ transfer function return information about how the predicates defined by D_i were affected by the assignment. As an example, suppose that the assignment sets $\mathsf{x} := 0$ and predicate P is defined as $\mathsf{P}() := \mathsf{x} \geq 0$. If the old value of x was negative, then the assignment causes P to go from false to true. The other domain should be informed of the change because it may contain facts about P that need to be updated.

The changes are conveyed via two sets, U and C. The set C contains predicate facts that may have changed. Its members have the form $\mathsf{P}(C_1, \ldots, C_k)$, where each C_i is a class; this means that the truth of $\mathsf{P}(x_1, \ldots, x_k)$ may have changed if $x_i \in C_i$ for all i. If some predicate fact is *not* in C, then it is safe to assume that its truth is not affected by the assignment.

The set U holds facts that are known to be true after the assignment. Its members have same form as facts returned by $\texttt{Consequences}_i$. For example, if an assignment causes P to go from true to false for all elements of a class C_0, then C would contain $\mathsf{P}(C_0)$ and U would contain $\forall x \in C_0.\ \neg\mathsf{P}(x)$.

The $\texttt{Assign}_i$ transfer functions are required to return U and C. However, when one predicate depends on another, $\texttt{Assign}_i$ may not know immediately how to update it. For example, if D_1 defines the predicate $\mathsf{P}() := \mathsf{x} \geq 0$ and D_2 defines $\mathsf{Q}() := \neg\mathsf{P}()$, then $\texttt{Assign}_1$ has no way to know that a change in x might affect Q, because it is unaware of the definition of Q.

We use a post-processing step to update predicates like Q. We require predicates to be *stratified.* A predicate in the j^{th} stratum can depend only on predicates in strata $< j$. Each domain must provide a function

```
**function** TranslateFull$_1$($E_1$, $E_2$, $f[e_1, \ldots, e_k]$):
  **if** $f \in D_1$:
    **for** $i \in [1..k]$:  $e'_i :=$ TranslateFull$_1(E_1, E_2, e_i)$
    **return** $f[e'_1, \ldots, e'_k]$
  **else**:
    **for** $i \in [1..k]$:  $e'_i :=$ TranslateFull$_2(E_1, E_2, e_i)$
    **return** $\mathit{Translate}_2(E_2, f[e'_1, \ldots, e'_k])$

**function** TranslateFull$_2$($E_1$, $E_2$, $f[e_1, \ldots, e_k]$):
  *defined similarly to TranslateFull$_1$*

**function** Assign($\langle E_1, E_2\rangle, f[e_1, \ldots, e_k], e$):
  $\langle E_1, E_2\rangle :=$ Saturate$(E_1, E_2)$

  **if** $f \in D_1$:
    $l :=$ TranslateFull$_1(E_1, E_2, f[e_1, \ldots, e_k])$
    $r :=$ TranslateFull$_1(E_1, E_2, e)$
    $\langle E'_1, U, C\rangle := \mathit{Assign}_1(E_1, l, r)$
    $E'_2 := E_2$
  **else**:
    $l :=$ TranslateFull$_2(E_1, E_2, f[e_1, \ldots, e_k])$
    $r :=$ TranslateFull$_2(E_1, E_2, e)$
    $\langle E'_2, U, C\rangle := \mathit{Assign}_2(E_2, l, r)$
    $E'_1 := E_1$

  $j := 1$
  **repeat**:
    $\langle E'_1, U, C\rangle = \mathit{PostAssign}_1(E_1, E'_1, j, U, C)$
    $\langle E'_2, U, C\rangle = \mathit{PostAssign}_2(E_2, E'_2, j, U, C)$
    $j := j + 1$
  **until** $j =$ num_strata

  **return** $\langle E'_1, E'_2\rangle$
```

Fig. 8. Pseudocode for assignment transfer function. num_strata is the total number of shared predicates.

$\mathit{PostAssign}_i(E_i, E'_i, j, U, C)$. Here, E_i is the domain element before the assignment and E'_i is the element that accounts for updates to base facts and to predicates in strata $< j$. U and C describe how predicates in strata $< j$ are affected by the assignment. The function's job is to compute updates to predicates in the j^{th} stratum, returning new values for E'_i, U, and C. Fig. 8 gives the full pseudocode. It assumes that variable num_strata holds the number of strata.

4 Examples

4.1 Linked Lists

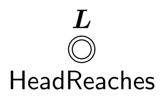
We begin by explaining how we analyze the code in Fig. 3. Although analysis of linked lists using canonical abstraction is well understood [24], this section illustrates our notation. First, some predicates must be specified by the user. These are standard predicates for analyzing singly linked lists with canonical abstraction [24]. The definition formulas use two forms of quantification: `tc` for irreflexive transitive closure and `ex` for existential quantification. All of these predicates are defined in the heap domain.

```
1 predicate NextTC(n1:T, n2:T) := tc(n1, n2) next;
2 predicate HeadReaches(n:T) := head = n || NextTC(head, n);
3 predicate CurReaches(n:T) := cur = n || NextTC(cur, n);
4 predicate SharedViaHead(n:T) := ex(n1:T) head = n && next[n1] = n;
5 predicate SharedViaNext(n:T) :=
6   ex(n1:T, n2:T) next[n1] = n && next[n2] = n && n1 != n2;
```

The predicate in line 1 holds between two list nodes if the second is reachable from the first via *next* pointers. The `Reaches` predicates hold when a list node is reachable from *head*/*cur*. The `Shared` predicates hold when a node has two incoming pointers, either from *head* or from another node's *next* field; they are usually false. These five predicates can constrain a structure to be an acyclic singly linked list.

On entry to the `iter` procedure in Fig. 3, we assume that *head* points to an acyclic singly linked list whose *data* fields are all zero. We abstract all the linked-list nodes into a summary heap class $\boldsymbol{L}$.

We describe the classes and shared predicates of the initial analysis state graphically as follows. Nodes represent classes and predicates are attached to these nodes.



This diagram means that there is a single class, $\boldsymbol{L}$, whose members satisfy the HeadReaches predicate and do not satisfy the CurReaches, SharedViaHead, or SharedViaNext predicates. The double circle means the node represents a summary class. We could write this state more explicitly as follows.

$$\begin{aligned}\forall x \in \boldsymbol{L}.\ & \mathsf{HeadReaches}(x) \wedge \neg\mathsf{CurReaches}(x)\\ & \wedge \neg\mathsf{SharedViaHead}(x) \wedge \neg\mathsf{SharedViaNext}(x)\end{aligned}$$

This state exactly characterizes the family of acyclic singly linked lists. Predicate HeadReaches ensures that there are no unreachable garbage nodes abstracted by $\boldsymbol{L}$, and the two sharing predicates exclude the possibility of cycles. Note that no elements are reachable from *cur* because *cur* is assumed to be invalid on entry to `iter`.

In addition to these shared predicate facts, each domain also records its own private facts. In this case, we assume that the numeric domain records that the data field of every list element is zero: $\langle\ \forall x \in \boldsymbol{L}.\ \mathit{data}[x] = 0\ \rangle_N$. The remainder of the analysis is a straightforward application of canonical abstraction.

4.2 Arrays

In this section, we consider a loop that initializes to null an array of pointers (Fig. 9). The example demonstrates how we abstract arrays. A similar loop is used to initialize a hash table in the `thttpd` web server that we verify in §5.

```
type T;
global table[int]:T;

procedure init(n:int)
  i:int;
{ i := 0;
  while (i < n) {
    table[i] := null;
    i := i+1;
  }
}
```

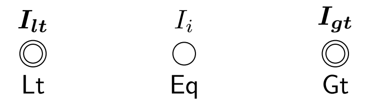
Fig. 9. Initialize an array.

Most of this code is analyzed straightforwardly by the integer domain. It easily infers the loop invariant that $0 \le i < n$. Only the update to *table* is interesting.

Just as the heap domain partitions heap nodes into classes, the integer domain partitions integers into classes. We define predicates to help it determine a good partitioning.

```
predicate Lt(x:int) = 0 <= x && x < i;
predicate Eq(x:int) = x = i;
predicate Gt(x:int) = i < x && x < n;
```

With these predicates, we obtain four integer classes via canonical abstraction, $\boldsymbol{I_{lt}}$, I_i, $\boldsymbol{I_{gt}}$, and $\boldsymbol{X}$. The first three classes contain elements satisfying the three predicates above, respectively. The last class contains all other integers (those that are negative or $\ge n$). Given these classes, we infer the following loop invariant.



$$\langle\ \forall x \in \boldsymbol{I_{lt}}.\ \mathit{table}[x] = \mathit{null}\ \rangle_H$$

The fact on the right is a private heap-domain fact but it can still refer to the integer class $\boldsymbol{I_{lt}}$. The ability of one domain to refer to another domain's classes is what enables mixed quantification in our system.

Using abstract interpretation, our analysis makes several passes over the loop before it infers this invariant. We write P_n to denote the state resulting from analyzing the n^{th} iteration of the loop. In state P_0, $i = 0$ and so $\boldsymbol{I_{lt}}$ is empty. The fact $\langle\ \forall x \in \boldsymbol{I_{lt}}.\ \mathit{table}[x] = \mathit{null}\ \rangle_H$ is vacuously true here, but our analysis does not infer facts about empty classes, so it is not included in P_0. However, it is *implied* by P_0 because $\boldsymbol{I_{lt}}$ is empty.

In state P_1, where $i = 1$, $\boldsymbol{I_{lt}}$ is non-empty and $\langle\ \forall x \in \boldsymbol{I_{lt}}.\ \mathit{table}[x] = \mathit{null}\ \rangle_H$ is inferred from the assignment. To obtain a loop invariant, we join P_0 and P_1. Our join algorithm recognizes that the fact $\langle\ \forall x \in \boldsymbol{I_{lt}}.\ \mathit{table}[x] = \mathit{null}\ \rangle_H$, which is present in P_1, is implied by P_0 (because $\boldsymbol{I_{lt}}$ is empty there) and so it includes this fact in the join result.

The assignment to *table* on line 8 of Fig. 9 proceeds as follows. Because the function *table* is heap-defined while *i* is defined in the numeric domain, the combined domain asks the numeric domain to "translate" *i* into a class. Ideally, the translation should generate the smallest possible class containing the value of *i*. In this case, the numeric domain can return the singleton class I_i, because it knows that I_i satisfies the Eq predicate. Then the heap domain can add $\langle\ \forall x \in I_i.\ \mathit{table}[x] = \mathit{null}\ \rangle_H$ to the analysis state.

The increment to *i* re-arranges the class structure (although this happens outside the assignment transfer function, which requires classes to remain constant). The numeric domain materializes a new class for $i + 1$, which becomes I_i and merges the existing I_i with $\boldsymbol{I_{lt}}$. The resulting domain element implies the loop invariant.

After the loop exits, the loop invariant implies that *table* is null at all indexes in $\boldsymbol{I_{lt}}$, which now includes all valid array indexes.

4.3 Numeric Predicates

We now show how Inv1 (Eqn. (1)) is established in `thttpd`. The code contains the following variable definitions and predicates.

```
global table[int]:T, index[T]:int, size:int;
predicate HasIdx(e:T, x:int) := index[e] = x;
predicate Inv1(x:int) := all(e:T) table[x]=e => HasIdx(e, x) || e=null;
```

The intent is that $\mathit{table}[k] = e$ should imply $\mathit{index}[e] = k$. Variable *size* is the size of the *table* array. Note that HasIdx is defined in the numeric domain because it references *index*, while Inv1 is defined in the heap domain.

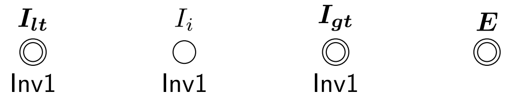
The procedures of interest to us are those that add and remove elements from the table. Our goal will be to prove that `add` preserves Inv1 and that `remove`, assuming Inv1 holds initially, does not leave any dangling pointers.

```
procedure add(i:int)
  o:T;
```

```
 3  { o := new T;
 4    table[i] := o;
 5    index[o] := i;
 6  }
 7  procedure remove(o:T)
 8    i:int;
 9  { i := index[o];
10    table[i] := null;
11    delete o;
12  }
```

Addition. Besides the predicates above, we create numeric predicates to partition the integers into five classes: $\boldsymbol{I_{lt}}$, I_i, $\boldsymbol{I_{gt}}$. Respectively, these are the integers between 0 and $i-1$, equal to i, greater than i but less than *size*. As before, class $\boldsymbol{X}$ holds the out-of-bounds integers.

Assume that upon entering the `add` procedure, we infer the following invariant (recall that we treat all functions via inlining).



$\langle\ \forall x \in I_i.\ \mathit{table}[x] = \mathit{null}\ \rangle_H$

All existing `T` objects are grouped into the class $\boldsymbol{E}$. *table* is unconstrained at $\boldsymbol{I_{lt}}$ and $\boldsymbol{I_{gt}}$and we do not have any information about the HasIdx predicate.

Initially, Inv1 holds at I_i because *table* is null there. When *table* is updated in line 4, Inv1 is potentially broken because *index*$[o]$ may not be i. The assignment on line 5 correctly sets *index*$[o]$, restoring Inv1 at I_i.

The object allocated at line 3 is placed in a fresh class E'. We do not have information about HasIdx for this new class. When line 4 sets $\mathit{table}[i] := \mathit{obj}$, the assignment is initially handled by the heap domain because *table* is a heap function. In order for Inv1 to continue to hold after line 4, we would need to know that $\forall x \in E'.\ \forall y \in I_i.\ \mathsf{HasIdx}(x, y)$. But this fact does not hold because E' is a new object whose *index* field is undefined.

Inv1 is restored in line 5. The assignment is handled by the numeric domain. Besides the private fact that $\langle\ \forall x \in E'.\ \mathit{index}[x] = i\ \rangle_N$, it recognizes that $\forall x \in E'.\ \forall y \in I_i.\ \mathsf{HasIdx}(x, y)$. This information is shared with the heap domain in the $\boldsymbol{\mathit{PostAssign}}_i$ phase of the assignment transfer function. The heap domain then recognizes that Inv1 has been restored at I_i. Thus, procedure `add` preserves Inv1.

Removal. We use the same numeric abstraction used for procedure `add`. On entry we assume that the object that o points to is contained in a singleton class E'. All other `T` objects are in a class $\boldsymbol{E}$. All *table* entries are either *null* or members of $\boldsymbol{E}$ or E'. The verification challenge is to prove that $\langle\ \forall x \in (\boldsymbol{I_{lt}} \cup \boldsymbol{I_{gt}}).\ \forall y \in E'.\ \mathit{table}[x] \neq y\ \rangle_H$. Without this fact, after E' is deleted, we might have pointers from *table* to freed memory. These pointers might later be accessed, leading to a segfault.

Luckily, Inv1 implies the necessary disequality, as follows. We start by analyzing line 9. The integer domain handles this assignment and shares the fact that $\forall x \in E'.\ \forall y \in I_i.$ HasIdx(x, y) holds afterwards. Importantly, because the integer domain knows that i is not in either $\boldsymbol{I_{lt}}$ or $\boldsymbol{I_{gt}}$, it also propagates $\forall x \in E'.\ \forall z \in (\boldsymbol{I_{lt}} \cup \boldsymbol{I_{gt}}).\ \neg$HasIdx$(x, z)$. We assume as a precondition to `remove` that Inv1 holds of $\boldsymbol{I_{lt}}$, I_i, and $\boldsymbol{I_{gt}}$. The contrapositives of the implications in these Inv1 facts, together with the negated HasIdx facts, imply that $\langle\ \forall x \in (\boldsymbol{I_{lt}} \cup \boldsymbol{I_{gt}}).\ \forall y \in E'.\ table[x] \neq y\ \rangle_H$.

The assignment on line 10 is straightforward to handle in the heap domain. It recognizes that $\langle\ \forall x \in I_i.\ table[x] = null\ \rangle_H$ while preserving Inv1 at I_i(because the definition of Inv1 has a special case for null). Finally, line 11 deletes E', Because the heap domain knows that $\langle\ \forall x \in (\boldsymbol{I_{lt}} \cup I_i \cup \boldsymbol{I_{gt}}).\ \forall y \in E'.\ table[x] \neq y\ \rangle_H$, there can be no dangling pointers.

4.4 Reference Counting

In this final example, we demonstrate the analysis of the most complex feature of `thttpd`'s cache: reference counting. To analyze reference counting we have augmented the integer domain in two ways.

The first augmentation allows the numeric domain to make statements about the cardinality of a class. For each class $\boldsymbol{C}$ we introduce a numeric dimension $\#\boldsymbol{C}$, called a *cardinality variable*. Thus, we can make statements like $\langle\ \#\boldsymbol{C} \leq n{+}1\ \rangle_N$. This augmentation was described by Gulwani et al. [14]. Usually, information about the cardinality of a class is accumulated as the class grows. The typical class starts as a singleton, so we infer that $\#C = 1$. As it is repeatedly merged with other singleton classes, its cardinality increments by one. Often we can derive relationships between the cardinality of a class and loop-iteration variables as a data structure is constructed.

Besides cardinality variables, we also introduce *cardinality functions*. These functions are private to the numeric domain. We give an example below in the context of reference counting.

```
type T, Container;
global rc[T]:int, contains[Container]:T;

predicate Contains(c:Container, o:T) := contains[c] = o;
function RealRC(o:T) := card(c:Container) Contains(c, o); // see below
predicate Inv2(o:T) := rc[o] = RealRC[o];
```

There are two types here: `Container` objects hold references to `T` objects. Each `Container` object has a *contains* field to some `T` object. Each `T` object records the number of incoming *contains* edges in its *rc* field.

The heap predicate Contains merely exposes *contains* to the numeric domain. The cardinality function *RealRC* is private to the numeric domain. *RealRC*$[e]$ equals the number of incoming *contains* edges to e. It equals the cardinality of the set $\{c : \texttt{Container} \mid \text{Contains}(c, e)\}$. The Inv2 predicate holds if $rc[e]$ equals this value.

Our goal is to analyze the functions that increment and decrement an object's reference count. We check for memory safety.

```
1  procedure incref(c:Container, o:T)
2  { assert(contains[c]=null);
3    rc[o]:=rc[o]+1;
4    contains[c]:=o;
5  }
6
7  procedure decref(c:Container)
8    o:T;
9  { o := contains[c];
10   contains[c]:=null;
11   rc[o]:=rc[o]-1;
12   if (rc[o]=0)
13     delete o;
14 }
```

Increment. When we start, we assume that class C' holds the object pointed to by `c` and E' holds the object pointed to by o. Class $\boldsymbol{E}$ holds all the other `T` objects and class $\boldsymbol{C}$ contains all the other `Container` objects. Then $\mathit{contains}[c]$, for any $c \in \boldsymbol{C}$, points to an object from either $\boldsymbol{E}$ or E', while $\mathit{contains}[c']$, for $c' \in C'$, is null. We also assume reference counts are correct, so $\mathsf{Inv2}$ at $\boldsymbol{E}$ and E'. This fact implies $\langle\ \forall x \in E'.\ \mathit{RealRC}[x] = \mathit{rc}[x]\ \rangle_N$. The assignment on line 3 updates this fact to $\langle\ \forall x \in E'.\ \mathit{RealRC}[x] = \mathit{rc}[x] - 1\ \rangle_N$ and makes $\mathsf{Inv2}$ false at E'.

The assignment on line 4 is initially handled by the heap domain, which recognizes that $\forall x \in C'.\ \forall y \in E'.\ \mathsf{Contains}(x, y)$ now holds. When this new fact is shared with the numeric domain, it realizes that RealRC increases by 1 at E', thereby restoring $\mathsf{Inv2}$ at E' as desired.

Decrement. Analysis of lines 9, 10, and 11 are similar to `incref`. We assume that the singleton class E' holds the object pointed to by obj. Similarly, C' holds the object pointed to by c. Other `Container` objects belong to the class $\boldsymbol{C}$ and other `T` objects belong to $\boldsymbol{E}$. Line 10 breaks $\mathsf{Inv2}$ at E' and line 11 restores it.

However, lines 12 and 13 are different. After line 12, the numeric domain recognizes that $\langle\ \forall x \in E'.\ \mathit{rc}[x] = 0\ \rangle_N$ holds. Therefore, it knows that $\langle\ \forall x \in E'.\ \mathit{RealRC}[x] = 0\ \rangle_N$ holds, based on the just-restored $\mathsf{Inv2}$ invariant at E'. Given the definition of RealRC, it is then able to infer $\forall x \in (\boldsymbol{C} \cup C').\ \forall y \in E'.\ \neg\mathsf{Contains}(x, y)$. Therefore, when obj is freed at line 13, we know that there are no pointers to it, which guarantees that there will be no accesses to this freed object in the future.

5 Experiments

Our experiments were conducted on the caching code of the `thttpd` web server discussed in §1. Interested readers can find our complete model of the cache, as well as the code for DESKCHECK, online [18]. The web-server cache has four entry-points. The `map` and `unmap` procedures are described in §1. Additionally, the `cleanup` entry-point is called optionally to free cache entries whose reference counts are zero; this happens in `thttpd` only when memory is running low. Finally, a `destroy` method frees all cache entries regardless of their reference count.

This functionality corresponds to 531 lines of C code, or 387 lines of code in the modeling language described in §2.1. The translation from C was done manually. The model is shorter because it elides the system calls for opening files and reading them into memory; instead, it simply allocates a buffer to hold the data. It also omits logging code and comments.

Our goal is to check that the cache does not contain any memory errors—that is, the cache does not access freed memory or fail to free unreachable memory. We also check that all array accesses are in bounds, that unassigned memory is never accessed, and that null is never dereferenced. We found no bugs in the code.

We verify the cache in the context of a simplified client. This client keeps a linked list of ongoing HTTP connections, and each connection stores a pointer to data retrieved from the cache. In a loop, the client calls either `map`, `unmap`, or `cleanup`. When the loop terminates, it calls `destroy`. At any time, many connections may share the same data.

All procedure calls are handled via inlining. There is no need for the user to specify function preconditions or postconditions. Because our analysis is an abstract interpretation, there is no need for the user to specify loop invariants either. This difference distinguishes DESKCHECK from work based on verification conditions.

All of the invariants described in §1 appear as predicate definitions in the verification. In total, thirty predicates are defined. Fifteen of them define common but important linked-list properties, such as reachability and sharing. These are all heap predicates. Another ten predicates are simple numeric range properties to define the array abstraction that is used to check the hash table. The final five are a combination of heap and numeric predicates to check $\mathsf{Inv1}$ and $\mathsf{Inv2}$; they are identical to the ones appearing in §4.3 and §4.4.

Deciding which predicates to provide to the analysis was a fairly simple process. However, the entire verification process took several weeks because it was intermingled with the development and debugging of DESKCHECK itself. It is difficult to estimate the effort that would be required for future verification work in DESKCHECK.

The experiments were performed on a laptop with a 1.86 GHz Pentium M processor and 1 GB of RAM (although memory usage was trivial). Tab. 1 shows the performance of the analysis. The total at the bottom is slightly larger than the sum of the entry-point times because it includes analysis of the client code as well. We currently handle procedure calls via inlining, which increases the cost of the analysis.

Entry-point	Analysis time
`map`	28.23 s
`unmap`	9.08 s
`cleanup`	76.81 s
`destroy`	5.80 s
Total	123.47 s

Table 1. Analysis times of `thttpd` analysis.

Annotations. Currently, we require some annotations from the user. These annotations never compromise the soundness of the analysis. Their only purpose is to improve efficiency or precision. One set of annotations marks a predicate as an abstraction predicate in a certain scope. There are 5 such scopes, making for 10 lines of annotations. We also use annotations to decide when to split an integer class into multiple classes. There are 14 such annotations. It seems possible to infer these annotations with heuristics, but we have not done so yet. All of these annotations are accounted for in the line counts above, as are the predicate definitions.

To give an example of the sorts of annotations required, we present our model of the `mmc_map` function in Fig. 10. The C code for this function is in Fig. 2. Note that *all* of our models are available online [18].

Virtually all of the code in Fig. 10 is a direct translation of Fig. 2 to our modeling language. The only annotations are at lines 14 and 23. These annotations temporarily designate `free_maps` as an abstraction predicate. This means that the node pointed to by `free_maps` is distinguished from other nodes in the canonical abstraction. Outside the scope of the annotations, every node reachable from the `free_maps` linked list is represented by a summary node. Because lines 16–18 remove the head of the list, it is necessary to treat this node separately or else the analysis will be imprecise. These two annotations are typical of all the abstraction-predicate annotations.

As a side note, a previous version of our analysis required loop invariants and function preconditions and postconditions from the user. We used this version of the analysis to check only the first two entry points, `map` and `unmap`. We found the annotation burden to be excessive. These two functions, along with their callees, required 1613 lines of preconditions, postconditions, and loop invariants. Undoubtedly a more expressive language of invariants would allow for more concise specifications, but more research would be required. This heavy annotation burden motivated us to focus on inferring these annotations as we do now via joins and widening.

6 Related Work

There are several methods for implementing or approximating the reduced product [6], which is the most precise refinement of the direct product. Granger's

```
procedure mmc_map(key:int):Buffer
  m:Map;
  b:Buffer;
{
  check_hash_size();

  m := find_hash(key);
  if (m != null) {
    Map_refcount[m] := Map_refcount[m]+1;
    b := Map_addr[m];
    return b;
  }

  @enable(free_maps);
  if (free_maps != null) {
    m := free_maps;
    free_maps := Map_next[m];
    Map_next[m] := null;
  } else {
    m := new Map;
    Map_next[m] := null;
  }
  @disable(free_maps);

  Map_key[m] := key;
  Map_refcount[m] := 1;
  b := new Buffer;
  Map_addr[m] := b;

  add_hash(m);

  Map_next[m] := maps;
  maps := m;

  return b;
}
```

Fig. 10. Our model of the `mmc_map` function from Fig. 2.

method of *local descending iterations* [13] uses a decreasing sequence of reduction steps to approximate the reduced product. The method provides a way to refine abstract *states*; in abstract *transformers*, domain elements can only interact either before or after transformer application. The *open-product* method [5] allows domain elements to interact *during* transformer application. Reps et al. [23] present a method that can implement the reduced product, for either abstract states or transformers, provided that one has a sat-solver for a logic that can express the meanings of both kinds of domain elements.

Combining Heap and Numeric Abstractions. The idea to combine numeric and pointer analysis to establish properties of memory was pioneered by Deutsch [8]. His abstraction deals with may-aliases rather precisely, but loses almost all information when the program performs destructive memory updates.

A general method for combining numeric domains and canonical abstraction was presented by Gopan et al. [12] (and was subsequently broadened to a general domain construction for functions [16]). A general method for tracking partition sizes (along with a specific instantiation of the general method) was presented by Gulwani at al. [14]. The work of Gopan et al. and Gulwani et al. are orthogonal methods: the former addresses how to abstract values of numeric fields; the latter addresses how to infer partition sizes. The present paper was inspired by these two works and generalizes both of them in several ways. For instance, we support more kinds of partition-based abstractions than the work of Gopan et al. [12], which makes the result more general, and may allow more scalable heap abstractions.

Gulwani and Tiwari [15] give a method for combining abstract interpreters, based on the Nelson-Oppen method for combining decision procedures. Their method also creates an abstract domain that is a refinement of the reduced product. As in Nelson-Oppen, communication between domains is solely via equalities, whereas in our method communication is in terms of classes and quantified, first-order predicates.

Emmi et al. [11] handle reference counting using auxiliary functions and predicates similar to the ones discussed in §4.4. As long as only a finite number of sources and targets are updated in a single transition, they automatically generate the corresponding updates to their auxiliary functions. For abstraction, they use Skolem variables to name single, but arbitrary, objects. Their combination of techniques is specifically directed at reference counting; it supports a form of universal quantification (via Skolem variables) to track the cardinality of reference predicates. In contrast, we have a parametric framework for combining domains, as well as a specific instantiation that supports universal and existential quantification, transitive closure, and cardinality. Their analyzer supports concurrency and ours does not. Because their method is unable to reason about reachability, their method would not be able to verify our examples (or `thttpd`).

Reducing Pointer to Integer Programs. In [10, 3, 17], an initial transformation converts pointer-manipulating programs into integer programs to allow integer analysis to check the desired properties. These "reduction-based approaches" uses various integer analyzers on the resulting program. For proving simple properties of singly linked lists, it was shown in [3] that there is no loss of precision; however, the approach may lose precision in cases where the heap and integers interact in complicated ways. The main problem with the approach is that the proof of the integer program cannot use any quantification. Thus, while it can make statements about the size of a local linked list, it cannot make a statement about the size of every list in a hash table. In particular, Inv1 and Inv2 both lie outside the capabilities of reduction-based approaches. Our approach alternates between the two abstractions, allows information to flow in both directions, and can use quantification in both domains. Furthermore, the framework is parametric; in particular, it can use a separation-logic domain [9] or canonical abstraction [24] (and is not restricted to domains that can represent only singly linked lists). Finally, proving soundness in our case is simpler.

Decision Procedures for Reasoning about the Heap and Arithmetic. One of the challenging problems in the area of theorem proving and decision procedures is to develop methods for reasoning about arithmetic and quantification.

Nguyen et al. [21] present a logic-based approach that involves providing an entailment procedure. The logic allows for user-defined, well-founded inductive predicates for expressing shape and size properties of data structures. Their approach can express invariants that involve other numeric properties of data structures, such as heights of trees. However, their approach is limited to separation logic, while ours is parameterized by the heap and numeric abstractions and can be used in more general contexts. In addition, their approach cannot handle quantified cardinality properties, such as the refcount property from `thttpd`:

$$\forall v\colon v.rc = |\{u : u.f = v\}|.$$

Finally, their approach does not infer invariants, which means that a heavy annotation burden is placed on the user. In contrast, our approach is based on abstract interpretation, and can thus infer invariants of loops and recursive procedures.

The logic of Zee et al. [26, 25] also permits verification of invariants involving pointers and cardinality. However, as above, this technique requires user-specified loop invariants. Additionally, the logic is sufficiently expressive that user assistance is required to prove entailment (similar to the partial order in an abstract interpretation). Because the invariants that we infer are more structured, we can prove entailment automatically. However, our abstraction annotations are similar to the case-splitting information required by their analysis.

Work by Lahiri and Qadeer also uses a specialized logic coupled with the verification-conditions approach. They use a decidable logic, so their is no need for assistance in proving entailment. However, they still require manual loop invariants.

Parameterized Model Checking. For concurrent programs, Clarke et al. [4] introduce *environment abstraction*, along with model-checking techniques for formulas that support a limited form of numeric universal quantification (the variable expresses the problem size, à la parameterized verification) together with variables that are universally quantified over non-numeric individuals (which represent processes). Our methods should be applicable to broadening the mixture of numeric and non-numeric information that can be used to model check concurrent programs.

References

1. G. Arnold. Specialized 3-valued logic shape analysis using structure-based refinement and loose embedding. In *SAS*, 2006.

2. J. Berdine, C. Calcagno, B. Cook, D. Distefano, P. O'Hearn, T. Wies, and H. Yang. Shape analysis for composite data structures. In *CAV*, 2007.
3. A. Bouajjani, M. Bozga, P. Habermehl, R. Iosif, P. Moro, and T. Vojnar. Programs with lists are counter automata. In *CAV*, 2006.
4. E. Clarke, M. Talupur, and H. Veith. Proving Ptolemy right: The environment abstraction framework for model checking concurrent systems. In *TACAS*, 2008.
5. A. Cortesi, B. L. Charlier, and P. V. Hentenryck. Combinations of abstract domains for logic programming. *SCP*, 38(1–3):27–71, 2000.
6. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *POPL*, pages 269–282, 1979.
7. R. DeLine and K. Leino. BoogiePL: A typed procedural language for checking object-oriented programs. Technical Report MSR-TR-2005-70, Microsoft Research, 2005.
8. A. Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *PLDI*, pages 230–241, 1994.
9. D. Distefano, P. O'Hearn, and H. Yang. A local shape analysis based on separation logic. In *TACAS*, pages 287–302, 2006.
10. N. Dor, M. Rodeh, and M. Sagiv. CSSV: towards a realistic tool for statically detecting all buffer overflows in C. In *PLDI*, pages 155–167, 2003.
11. M. Emmi, R. Jhala, E. Kohler, and R. Majumdar. Verifying reference counting implementations. In *TACAS*, 2009.
12. D. Gopan, F. DiMaio, N. Dor, T. Reps, and M. Sagiv. Numeric domains with summarized dimensions. In *TACAS*, pages 512–529, 2004.
13. P. Granger. Improving the results of static analyses programs by local decreasing iteration. In *FSTTCS*, 1992.
14. S. Gulwani, T. Lev-Ami, and M. Sagiv. A combination framework for tracking partition sizes. In *POPL*, pages 239–251, 2009.
15. S. Gulwani and A. Tiwari. Combining abstract interpreters. In *PLDI*, 2006.
16. B. Jeannet, D. Gopan, and T. Reps. A relational abstraction for functions. In *SAS*, 2005.
17. S. Magill, J. Berdine, E. Clarke, and B. Cook. Arithmetic strengthening for shape analysis. In *SAS*, pages 419–436, 2007.
18. B. McCloskey. Deskcheck 1.0. `http://www.cs.berkeley.edu/~billm/deskcheck`.
19. A. Miné. A new numerical abstract domain based on difference-bound matrices. In *PADO '01: Proceedings of the Second Symposium on Programs as Data Objects*, pages 155–172, London, UK, 2001. Springer-Verlag.
20. G. Nelson and D. Oppen. Simplification by cooperating decision procedures. *TOPLAS*, 1(2):245–257, 1979.
21. H. Nguyen, C. David, S. Qin, and W.-N. Chin. Automated verification of shape and size properties via separation logic. In *VMCAI*, pages 251–266, 2007.
22. J. Poskanzer. thttpd - tiny/turbo/throttling http server. `http://acme.com/software/thttpd/`.
23. T. Reps, M. Sagiv, and G. Yorsh. Symbolic implementation of the best transformer. In *VMCAI*, pages 252–266, 2004.
24. M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *TOPLAS*, 24(3):217–298, 2002.
25. K. Zee, V. Kuncak, and M. Rinard. Full functional verification of linked data structures. In *ACM Conf. Programming Language Design and Implementation (PLDI)*, 2008.
26. K. Zee, V. Kuncak, and M. Rinard. An integrated proof language for imperative programs. In *PLDI*, pages 338–351, 2009.