A Combination Framework for Tracking Partition Sizes

Sumit Gulwani
Microsoft Research
sumitg@microsoft.com

Tal Lev-Ami *
Tel-Aviv University
tla@post.tau.ac.il

Mooly Sagiv †
Tel-Aviv University
msagiv@post.tau.ac.il

Abstract

We describe an abstract interpretation based framework for proving relationships between sizes of memory partitions. Instances of this framework can prove traditional properties such as memory safety and program termination but can also establish upper bounds on usage of dynamically allocated memory. Our framework also stands out in its ability to prove properties of programs manipulating both heap and arrays which is considered a difficult task.

Technically, we define an abstract domain that is parameterized by an abstract domain for tracking memory partitions (sets of memory locations) and by a numerical abstract domain for tracking relationships between cardinalities of the partitions. We describe algorithms to construct the transfer functions for the abstract domain in terms of the corresponding transfer functions of the parameterized abstract domains.

A prototype of the framework was implemented and used to prove interesting properties of realistic programs, including programs that could not have been automatically analyzed before.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs; F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Program analysis

General Terms Reliability, Verification

Keywords Combining Analyses, Set Analysis, Numerical Analysis, Shape Analysis, Termination, Space Bounds, Memory Safety

1. Introduction

The theme of this paper is to automatically establish invariants regarding sizes of memory partitions. Such invariants are crucial in order to bound the size of dynamically allocated data (e.g., in embedded systems). They are also necessary in order to infer the shape of the data in programs that manipulate both arrays and dynamically allocated data structures, which is common in many implementations of abstract data types such as hashing, skiplists, BTrees, and string implementations. Moreover, proving such invariants in these programs is required for proving their memory safety.

We describe new algorithms for establishing such invariants by combining two kinds of abstractions: (a) Abstractions that partition the memory into (not necessarily) disjoint parts and (b) Numerical abstractions that can track relationships between numeric variables. Our algorithms are parameterized by both abstractions which allows to leverage existing shape abstractions (e.g., [33, 30, 12, 25]) and existing numerical abstractions (e.g., Polyhedra [8], Octagons [26], Intervals [6]). We call such an analysis a *set cardinality analysis*.

We first formalize the notion of a set abstract domain that provides abstractions to partition the memory into (not necessarily disjoint) parts called *base-sets*. We describe the interface that a set domain should export in order for it to be combinable with a numerical domain (Section 3). A key component of such an interface is the `Witness` operator that relates a given base-set with other base-sets that occur in a given set-domain element. (This relationship is transformed into a numerical relationship over the cardinalities of the base-sets by the combination framework, and is the only window to communicate any information about the meaning of a base-set to the numerical domain, which otherwise views base-sets as uninterpreted and simply uses a fresh variable to denote the cardinality of each base-set). We show that several popular heap/shape analysis domains can be easily made to support such an interface (Section 3.3) – this is one of the contributions of the paper.

We then define the notion of a set cardinality abstract domain that is parameterized by a set domain and a numerical domain (Section 4). An element of the set cardinality domain is a pair composed of a set-domain element and a numerical element. The interesting part here is our formalization of the pre-order between the elements in this domain, which defines the level of reasoning built into our set cardinality domain. The pre-order is defined constructively in terms of the partial orders of the individual domains. Hence, given a decision procedure for the set domain and a decision procedure for the numerical domain, our pre-order construction shows how to convert them into a decision procedure for the set cardinality domain. Such a modular construction of the decision procedure for the set cardinality domain is an independent contribution of the paper, and fits in the stream of work on decision procedures for reasoning about sets and their cardinalities [21]. Though we do not prove any completeness results here, the modular construction allows the use of existing set and numerical domains and is demonstrated to be precise enough in practice.

We then describe algorithms for the transfer functions for the set cardinality domain. This is a key technical contribution of the paper. Each of the transfer functions for the set cardinality domain is described modularly in terms of the corresponding transfer functions of the constituent set domain and the numerical domain. We also prove the soundness and completeness of these transfer functions with respect to the pre-order. The basic idea behind the trans-

* Supported by the Adams Fellowship Program of the Israel Academy of Sciences and Humanities.

† Part of this work was done while visiting University of California Berkeley supported by a donation from IBM and while visiting Cadence Berkeley Research Laboratory.

POPL'09, January 18–24, 2009, Savannah, Georgia, USA.

fer functions is to first apply the corresponding transfer function over the set-domain elements of the inputs to obtain the set-domain element of the output. Then, we use the `Witness` operator exported by the set domain to relate the base-sets in the outputs to the base-sets in the inputs and strengthen the numerical elements in the inputs with this information. We then apply the transfer functions over the numerical elements to obtain the final result. The exact details depend on the specific transfer function and are important to establish soundness and completeness of the transfer functions.

We start by describing interesting applications of our framework, namely termination analysis, memory bounds analysis, and memory safety and functional correctness (Section 2). Our framework enables verification of desirable properties of real-world examples, which to our knowledge, have not been analyzed before. We present experimental results illustrating the feasibility of our approach (Section 6.1). We also present a case-study regarding how the choice of the constituent set domain and the constituent numerical domain affects the precision of the set cardinality domain (Section 6).

2. Applications

Our work has several applications that are mentioned below, and we present experimental results for each of these applications in Section 6.1.

Proving Memory Safety as well as Data-structure Invariants. Often some numeric program variables are related to size of data-structures, and are used to iterate over data-structures. Our analysis can automatically track these relationships between program variables and size of data-structures. These relationships are important to prove memory safety. A common pattern in C where these relationships arise is when lists are converted into arrays and are then iterated over in the same loop that iterates over the corresponding array without having a null-dereference check. These relationships are also important to prove data-structure invariants. This happens frequently in object-oriented code wherein base class libraries maintain length of data-structures like queues or lists. In Section 2.1, we present a procedure from Microsoft product code that illustrates the importance of tracking relationships between numeric variables and sizes of data-structures for proving both memory safety and data-structure invariants. We do not know of any existing technique that can automatically verify the correctness of assertions in this code.

Bounding Memory Allocation. This involves bounding the sizes of the partitions corresponding to the allocation statements in the program. This is especially important in embedded systems, where we would like to prove statically that the amount of memory that the system is shipped with is sufficient to execute desired applications. We present examples of bounding memory allocation in terms of sizes of input data-structures for deep copy routines over a variety of data-structures in Section 6.

Proving Termination. The oldest trick for proving termination of loops has been that of finding a ranking function [34]. A ranking function for a loop is a function whose value decreases in each iteration and is bounded below by some finite quantity. There has recently been a lot of work on discovering fancy forms of numerical ranking functions (lexicographic polyranking functions [4], disjunctively well-founded linear ranking functions [29]). However, for several programs based on iteration over data-structures, the ranking function is actually related to the cardinality of some partition of the data-structure. Our technique can find such ranking functions and can in fact even prove a bound on the loop iterations by instrumenting a counter variable in the loop and discovering invariants that relate the counter variable and sizes of partitions. We illustrate this by means of the `BubbleSort` example in Section 2.2. We do not know of any existing technique that can prove even termination of this example automatically.

2.1 String Buffer Example

This example illustrates the use of our analysis for proving memory safety as well as establishing data-structure invariants.

Consider the string buffer data-structure `StringBuffer` described in Figure 1(a), as taken from Microsoft product code. A string buffer is implemented as a list of chunks (in reverse order, so that appends are fast). A chunk consists of a character array `content` whose total size is `size` and its `len` field denotes the total number of valid characters in the array. This program contains the following features that make the task of analysis/verification challenging: (i) The usage of dynamic memory and pointers with destructive pointer mutations and (ii) The usage of arrays and arithmetic. These features are common in C. Moreover, Java ADT implementations, such as hash-maps, raise similar challenges.

The Remove method over string buffer (Figure 1(b)) takes as input a non-negative start index `startIndex` and a positive integer `count` and deletes `count` characters starting from `startIndex`. The first loop (Lines 3-4) counts the total length of characters inside string buffer and stores it into the variable n. The second loop (Lines 8-9) finds the first chunk `endChunk` from which characters are to be removed, while the third for loop (Lines 11-12) finds the last chunk `startChunk` from which characters are to be removed. Both these loops have a memory safety assertion at lines 9 and 12 respectively. The loop condition for each of these loops indicates that m is positive; it does not explicitly indicate that $y \neq$ `null`. However, the assertions hold because there is a relationship between m and y, namely that the total number of characters in the string buffer before y is equal to m. Hence, if $m > 0$, it implies that $y \neq$ `null`. The framework presented in this paper can be used to automatically discover such relationships between numerical program variables and sizes of appropriate partitions of data-structures. For this purpose, we require a set domain whose base-set constructor can represent the set of all characters in a chunk x and the chunks before it (referred to as $R_1(x)$ in Figure 1(c)). When coupled with a relational numerical domain that can represent linear inequalities between numerical variables, our combination framework yields a set cardinality analysis that can discover the required invariants (shown in Figure 1(c)).

The next loop (Lines 21-23) slides down an appropriate number of characters in `endChunk`. The last loop (Lines 25-26) counts the total number of characters in the string buffer and stores it in the variable n'. Line 27 then asserts that n' (whose value is the total number of characters in the string buffer at the end of the `Remove` method) is less than n (whose value is the total number of characters in the string buffer at the beginning of the `Remove` method) by an amount equal to `count`. The assertion holds because lines 14-20 remove `count` characters from the string buffer by destructively updating the data-structure and adjusting the value of then `len` field of appropriate chunks. The approach presented in this paper can automatically discover such relationships that relate the sizes of various partitions of data-structures. These relationships along with the required invariants at other program points are shown in Figure 1(c).

Figure 1(d) describes the effect of the `Remove` method over an example string buffer x. The filled part (both dark filled part and lightly filled part) of each chunk represents the original characters in the string buffer x. The solid filled part represents the characters to be removed. The solid filled part is identified by the second loop (Lines 8-9) and the third loop (Lines 11-12).

```
Remove(StringBuffer * x, int startIndex, int count) {
1  Assume(startIndex ≥ 0 ∧ count > 0);
2  n := 0;
3  for (y := x; y ≠ null; y := y → previous)
4       n := n + (y → len);
5  if (n < startIndex + count) return;
6  endIndex := startIndex + count;
7  y := x; m := n − (y → len);
8  for (; m > endIndex; m := m − (y → len))
9       Assert(y ≠ null); y := y → previous;
10 endChunk := y; endChunkOff := m;
11 for (; m > startIndex; m := m − (y → len))
12      Assert(y ≠ null); y := y → previous;
13 startChunk := y; startChunkOff := m;
14 if (startChunk ≠ endChunk)
15    endChunk → previous := startChunk;
16    startChunk → len := startIndex − startChunkOff;
17    tmp := endIndex − endChunkOff;
18 else
19    tmp := endIndex − startChunkOff;
20 endChunk → len := (endChunk → len) − tmp;
21 for (i := 0; i < endChunk → len; i := i + 1)
22      i' := i + tmp;
23      endChunk → content[i] := endChunk → content[i'];
24 n' := 0;
25 for (y := x; y ≠ null; y := y → previous)
26      n' := n' + (y → len);
27 Assert(n' = n − count);
28 }
```

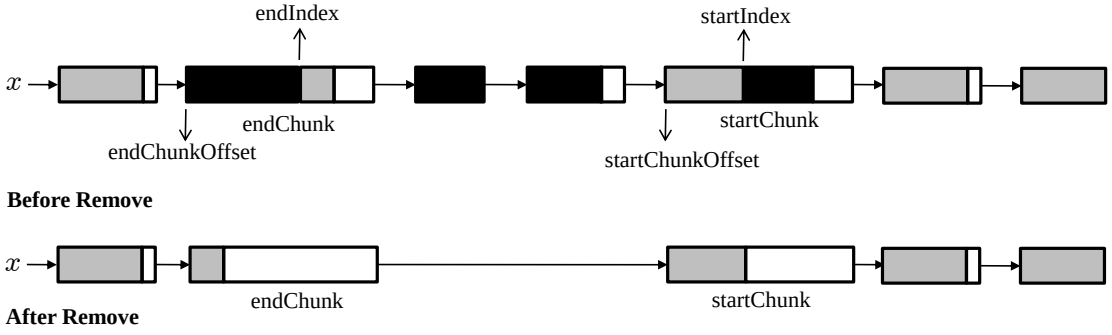
(b) Method to remove `count` elements from string buffer x starting at location `startIndex`

```
typedef struct {
   int len; int size;
   char* content;
   StringBuffer * previous;
} * StringBuffer;
```

(a) String Buffer data-structure

π	Interesting Invariants at program point π
4	$n = \|R_1(x)\| - \|R_1(y)\|$
5	$n = \|R_1(x)\|$
9	$m = \|R_1(y)\| - (y \rightarrow \texttt{len}) \wedge m > 0$
12	$m = \|R_1(y)\| - (y \rightarrow \texttt{len}) \wedge m > 0$
16	$\texttt{startChunkOff} = \|R_1(\texttt{startChunk})\| - (\texttt{startChunk} \rightarrow \texttt{len})$ $\texttt{endChunkOff} = \|R_1(\texttt{endChunk})\| - (\texttt{endChunk} \rightarrow \texttt{len})$ $\|R_1(x)\| = n - (\texttt{endChunkOff} - \texttt{startChunkOff} - (\texttt{startChunk} \rightarrow \texttt{len}))$
20	$(\texttt{startChunk} \rightarrow \texttt{len}) = \texttt{startIndex} - \texttt{startChunkOff}$ $\|R_1(x)\| = n - (\texttt{endChunkOff} - \texttt{startChunkOff} - (\texttt{startChunk} \rightarrow \texttt{len}))$
21	$\|R_1(x)\| = n - count$
26	$n' = \|R_1(x)\| - \|R_1(y)\|$
27	$n' = \|R_1(x)\|$

(c) Interesting invariants at various program points in the `Remove` method that are necessary to prove the given assertions. The invariants given hold right before π is executed.



(d) An example of a string buffer x before and after the remove method.

Figure 1. Remove method of `StringBuffer` data-structure (adapted slightly from Microsoft product code). The method has 2 memory safety assertions and one assertion that relates the sizes of the string buffer at the entry and exit.

2.2 BubbleSort Example

This example illustrates the use of our analysis for proving termination as well as computing a bound on the number of loop iterations.

Consider the BubbleSort procedure shown in Figure 2 that sorts an input array A of length n. Ignore lines 2 and 4 that update a counter variable c. The algorithm works by repeatedly iterating through the array to be sorted, comparing two items at a time and swapping them if they are in the wrong order (Line 8). The iteration through the array (Loop in lines 6-11) is repeated until no swaps are needed (indicated by the `change` boolean variable), which indicates that the array is sorted.

Notice that establishing a bound on the number of iterations of the outer while-loop of this procedure is non-trivial; it is not immediately clear why the outer while loop even terminates. However, note that in each iteration of the while loop, at least one new element "bubbles" up to its correct position in the array (i.e., it is less or equal to all of its successors). Hence, the outer while loop terminates in at most n steps. The set cardinality analysis that we introduce in this paper can automatically establish this fact by computing a relationship between an instrumented loop counter c (to count the number of loop iterations of the outer while loop) and the number of elements that have been put in the correct position. In particular, the set cardinality analysis computes the invariant that c is less than or equal to the size of the set of the array indices that hold elements at their correct position (provided the set cardinality analysis is constructed from a set analysis whose base-set constructor can represent such a set).

```
BubbleSort(int* A, int n)
1   change := true;
2   c := 0;
3   while (change) {
4       c := c + 1;
5       change := false;
6       for(j := 0; j < n − 1; j := j + 1) {
7           if (A[j] > A[j + 1]) {
8               Swap(A[j], A[j + 1]);
9               change := true;
10          }
11      }
12  }
13
```

Figure 2. Bubblesort Routine.

3. Set Domain

In this section, we formalize the notion of a set abstract domain. In particular, we describe the interface that a set domain should support in order for it to be combinable with a numerical domain in our combination framework.

A set domain $\mathcal{P}$ consists of set-domain elements that are related by some partial order $\preceq_{\mathcal{P}}$. A set-domain element P of a set domain should expose some bounded collection of (interpreted) base-sets, referred to as $\texttt{BaseSets}(P)$. Examples for base-sets are $R_1(z)$, where z is a program variable, as used in Section 2.1. The numerical domain tracks relationships of the cardinalities of the base-sets in $\texttt{BaseSets}(P)$.

A set domain exports all the standard operations needed to perform abstract interpretation (namely `Join`, `Widen`, `Eliminate`, `PostPredicate`, as defined in Section 5). Besides these operations, a set domain also needs to support the following operations in order for it to be combinable with a numerical abstract domain to enable tracking of numerical properties over cardinalities of base-sets.

3.1 Witness Operator

The set domain $\mathcal{P}$ exports an operation $\texttt{Witness}_{\mathcal{P}}$ that takes as input a collection S of base-sets and a set-domain element P. Intuitively, $\texttt{Witness}(S, P)$ returns the interpretation of base-sets in S in terms of the base-sets that occur in P using the information from P. This is needed because base-sets are interpreted. Thus, even if the base-sets in S are semantically identical to some base-sets in $\texttt{BaseSets}(P)$ (a common case in our setting), there is no way to infer this equality without the help of the set domain. Without `Witness`, the numerical domain cannot infer anything about cardinalities of base-sets in S (even when it has information about cardinalities of base-sets that occur in P).

The result of $\texttt{Witness}(S, P)$ is in the form of normalized set-inclusion relationships. A *normalized set-inclusion relationship* (implied by P) between a collection T of base-sets is a relationship of the form

$$\bigcup_{i \in I} p_i \subseteq \bigcup_{j \in J} p'_j$$

where $p_i, p'_j \in T$ (for all $i \in I, j \in J$) and P implies that p_i's are all mutually disjoint (i.e., under any concretization of P, the interpretations of p_i's are mutually disjoint). Furthermore, I is maximal and J is minimal. In practice, most of the relationships we get are of the form $p = \bigcup p_i$ where the p_i's are mutually disjoint.

EXAMPLE 1. *Consider a simple set domain whose base sets are of the form q_{nk} where n is a positive integer and k is a natural. We define $x \in q_{nk}$ iff $(x \mod n) \equiv k$. Let $\texttt{BaseSets}(P) = \{q_{30}, q_{20}\}$ and $S = \{q_{60}, q_{63}, q_{40}\}$. We have*

$$\texttt{Witness}(S, P) = \{\ q_{60} \cup q_{63} \subseteq q_{30}, q_{30} \subseteq q_{60} \cup q_{63}, q_{60} \subseteq q_{20}, q_{40} \subseteq q_{20}\ \}$$

Note that although $q_{60} \cup q_{40} \subseteq q_{20}$, this relation is not part of the witness since q_{60} and q_{40} are not disjoint.

The advantage of normalized set-inclusion relationships is that they can be easily translated into numerical relationship over cardinalities of base-sets, which is something that a numerical domain can understand. They will thus be used to relate the cardinalities of the base-sets before and after an abstract operation (see the `P2N` operation in Section 4).

The soundness of the combination framework only requires that the base-sets of the left side of a normalized set-inclusion relationship p_i be all mutually disjoint in P. The soundness does not require I to be maximal or J to be minimal, and neither does it require the `Witness` operator to return all such relationships. However, a more precise collection of such relationships leads to a more precise combined domain.

3.2 Generate Operator

The set domain also has an interface $\texttt{Generate}_{\mathcal{P}}$ to generate information about the cardinality of any base-set in relation to any constant. The function $\texttt{Generate}_{\mathcal{P}}$ takes a set-domain element P, and returns a collection of inequalities of the form $|p| \leq c$ or $|p| \geq c$ (where $p \in \texttt{BaseSets}(P)$ and c is some non-negative integer constant) that are implied by P. In case the set-domain element is inconsistent (i.e., does not represent any concrete element), $\texttt{Generate}_{\mathcal{P}}$ simply returns `false`.

3.3 Examples of Set Domains

In this section, we show that several popular heap/shape analysis domains can be viewed as set domains. In particular, we show that for each of these domains we can easily implement the required operations to be used in the combination framework.

In all three cases the domain is a powerset domain over some base domain. The combination with the numerical domain is performed at the level of the base domain. The construction of the powerset domain is done on top of the combined domain.

3.3.1 Canonical Abstraction

Canonical Abstraction[33] is a powerful domain for shape analysis. The domain is a powerset of abstract shape graphs. Each abstract shape graph is based on equivalence classes of memory locations based on the unary predicates that hold for them. These equivalence classes are called abstract nodes. Canonical abstraction maintains the invariant that each abstract node η must represent at least one memory location. Canonical abstraction can maintain binary information that holds universally on abstract nodes, i.e., formulas of the form $\forall v_1, v_2 \,.\, \eta_1(v_1) \wedge \eta_2(v_2) \rightarrow p(v_1, v_2)$ where p is a binary predicate, v_i ranges over memory locations and $\eta_i(v_i)$ holds when v_i is in the equivalence class defined by η_i. Specifically, canonical abstraction can express the notion of an abstract node η that represents exactly one memory location using the formula $\texttt{unit}(\eta) = \forall v_1, v_2 \,.\, \eta(v_1) \wedge \eta(v_2) \rightarrow v_1 = v_2$.

Required Operations The base-sets of an abstract shape graph are its abstract nodes. Thus,

$$\texttt{BaseSets}(P) \stackrel{\text{def}}{=} \{\eta \mid \eta \in P\}$$

The $\texttt{Witness}_{CA}$ operation is straightforward as the abstract nodes are based on equivalence classes. Thus, abstract nodes have canonical names, which allow to easily relate abstract nodes from different abstract shape graphs. Furthermore, because the base-sets

are equivalence classes of unary predicates, they are necessarily disjoint.

The cardinality constraints arise from the non-emptiness requirement and the definition of `unit`, i.e.,

$$\texttt{Generate}_{\texttt{CA}}(P) \stackrel{\text{def}}{=} \bigwedge(\{|\eta| \geq 1 \mid \eta \in \texttt{BaseSets}(P)\} \cup \{|\eta| = 1 \mid \eta \in \texttt{BaseSets}(P), \texttt{unit}(\eta) \in P\})$$

Finally, the canonical abstraction domain can easily interpret cardinality constraints of the form $|\eta| = 1$ by asserting the formula defining $\texttt{unit}(\eta)$. Note that although this was not part of the standard interface for the canonical abstraction domain, there is an existing mechanism to easily support this constraint. We use this choice of set domain in our experiments in Section 6.1.

3.3.2 Boolean Heaps

The Boolean Heaps Domain [30] is a powerset domain over Boolean Heaps. As in canonical abstraction, each Boolean heap is formed of equivalence classes of unary predicates. However, there is no requirement for non-emptiness of abstract nodes and no extra binary information. Boolean heaps support predicates of the form $v = t$ where t is a ground term. If such a unary predicate holds on an abstract node, this node represents at most one memory location. Thus, we shall define $\texttt{unique}(\eta)$ to hold for any η in which there is a predicate of the form $v = t$.

Required Operations The base-sets of a Boolean heap are its abstract nodes. Thus,

$$\texttt{BaseSets}(P) \stackrel{\text{def}}{=} \{\eta \mid \eta \in P\}$$

The $\texttt{Witness}_{BH}$ operation is very similar to that of canonical abstraction as both abstractions are based on equivalence classes of unary predicates. Because abstract nodes are equivalence classes of unary predicates, the resulting base-sets are necessarily disjoint.

Boolean heaps have the notion of a `unique` base-set but no requirement for non-emptiness. Thus, $\texttt{Generate}_{\texttt{BH}}$ is defined by:

$$\texttt{Generate}_{\texttt{BH}}(P) \stackrel{\text{def}}{=} \bigwedge(\{|\eta| \geq 0 \mid \eta \in \texttt{BaseSets}(P)\} \cup \{|\eta| \leq 1 \mid \eta \in \texttt{BaseSets}(P), \texttt{unique}(\eta)\})$$

Given a cardinality constraint of the form $|\eta| = 0$, the Boolean heap can soundly remove η from the Boolean heap, thus reducing its size and complexity.

3.3.3 Separation Domain

We use the separation domain of Distefano et al [12] as a representative of Separation Logic [31] based abstract domains. The separation domain is a powerset domain of symbolic heaps. A symbolic heap P is a separation logic formula of the form:

$$\exists x'_1, \ldots, x'_n.(\bigwedge_{\pi \in \Pi_P} \pi) \wedge (\bigstar_{Q \in \Sigma_P} Q)$$

Π_P is a set of equalities and dis-equalities between pointer variables (and possibly `nil`). Σ_P can contain either

- `junk` — any non-empty memory,
- $x \mapsto y$ — a memory that contains a single address x whose content is y, or
- $ls(x, y)$ — a non-empty singly-linked list starting at address x and whose last element points to the address y[1].

The formula $\varphi_1 \star \varphi_2$ holds on memories that can be decomposed into two disjoint parts s.t. φ_1 holds on one of them and φ_2 holds on the other.

[1] $ls(x, y)$ is defined recursively as $x \neq y \wedge (x \mapsto y \vee \exists z.\, x \mapsto z \star ls(z, y)$

Required Operations We use the members of Σ_P (a.k.a. star conjuncts) as base-sets, i.e.,

$$\texttt{BaseSets}(P) \stackrel{\text{def}}{=} \Sigma_P$$

The $\texttt{Witness}_{SL}$ operation is computed by finding which star conjuncts are subsets of other star conjuncts. Such algorithms already exist in the domain for performing containment checks. The locality of most operations (i.e., the fact that they modify only small parts of the memory each time) means that most of the star conjuncts will appear verbatim. This allows for efficient implementation of the $\texttt{Witness}_{SL}$ operation. Furthermore, the base-sets are star conjuncts, thus all base-sets of an abstract element are disjoint.

Because all star conjuncts represent non-empty sets and a star conjunct of the form $x \mapsto y$ is of cardinality 1, $\texttt{Generate}_{\texttt{SL}}$ is defined by:

$$\texttt{Generate}_{\texttt{SL}}(P) \stackrel{\text{def}}{=} \bigwedge(\{|\eta| \geq 1 \mid \eta \in \texttt{BaseSets}(P)\} \cup \{|\eta| = 1 \mid \eta \in \texttt{BaseSets}(P), \eta \equiv x \mapsto y\})$$

Finally, if the numerical domain discovers that $|ls(x, y)| = 1$ then $ls(x, y)$ can be strengthened to $x \mapsto y$.

We use the syntax of separation logic in our examples[2].

4. Set Cardinality Domain

In this section, we define the notion of a set cardinality domain that is obtained by a combination of a set domain and a numerical domain. Given a set domain $\mathcal{P}$ and a numerical domain $\mathcal{N}$, we define their combination to be the set cardinality domain $\mathcal{P} \bowtie \mathcal{N}$ whose elements are pairs (P, N), where P is some element that belongs to the set domain $\mathcal{P}$ and N is some element that belongs to the numerical domain $\mathcal{N}$. Furthermore, N uses some special variables, each of which denotes the cardinality of some base-set from $\texttt{BaseSets}(P)$. We denote the special variable corresponding to a base-set p by $\widetilde{p}$. In addition, we use in our examples the shorthand $[p]^A$ to mean $A \equiv \widetilde{p}$. We use the notation $\texttt{SpecialVars}(N)$ to denote the set of all special variables (i.e., non-program variables that corresponds to the cardinality of some base-set) that occur in N. For notational convenience, we overload the notation $\texttt{SpecialVars}(P)$ to denote the set of all special variables that denote the cardinality of some base-set in P, i.e.,

$$\texttt{SpecialVars}(P) = \{\widetilde{p} \mid p \in \texttt{BaseSets}(P)\}$$

Hence, every element (P, N) that belongs to the domain $\mathcal{P} \bowtie \mathcal{N}$ has the property that $\texttt{SpecialVars}(N) \subseteq \texttt{SpecialVars}(P)$. Also, without loss of any generality, we assume that for any two distinct elements (P_1, N_1) and (P_2, N_2), $\texttt{SpecialVars}(P_1) \cap \texttt{SpecialVars}(P_2) = \emptyset$ since we can always rename the special variables that denote the cardinality of some base-sets.

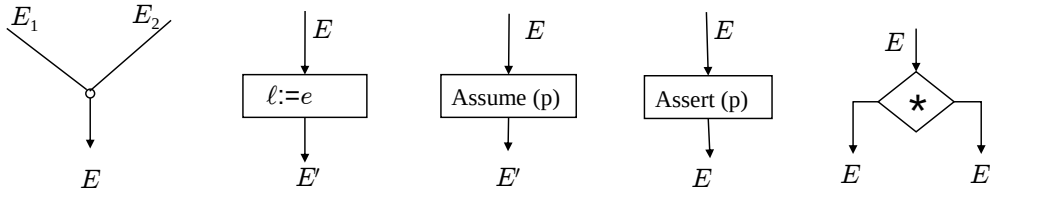
The pre-order $\preceq_{\mathcal{P} \bowtie \mathcal{N}}$ between two elements of the combined domain is defined as follows:

$$\begin{array}{rcl} (P_2, N_2) \preceq_{\mathcal{P} \bowtie \mathcal{N}} (P_1, N_1) & \stackrel{\text{def}}{=} & P_3 \preceq_{\mathcal{P}} P_1 \ \wedge \ N_4 \preceq_{\mathcal{N}} N_1 \\ \text{where } (P_3, N_3) & = & \texttt{Saturate}(P_2, N_2) \\ N_4 & = & \texttt{P2N}(P_3, P_1, N_3) \end{array}$$

The pre-order above has two important components `Saturate` and `P2N`, which are described below.

Saturate The `Saturate` function takes as input an element (P, N) from the combined domain and returns another element (P', N') from the combined domain. P' and N' are obtained from P and N respectively by repeated sharing of information about

[2] Note that the join algorithm we use is a refined version of the one in [12] and uses the ideas in [24].



(a) Join Node (b) Assignment Node (c) Assume Node (d) Assert Node (e) Conditional Node

Figure 3. Flowchart Nodes.

relationships of base-set sizes with integral constants using the $\texttt{Generate}_{\mathcal{P}}$ interface exported by the set domain (as described in Section 3) and the $\texttt{Generate}_{\mathcal{N}}$ interface that can be provided by a numerical domain as follows:

$$\texttt{Generate}_{\mathcal{N}}(N) \stackrel{\text{def}}{=} \bigwedge_{x \in U} \texttt{Eliminate}_{\mathcal{N}}(N, \texttt{Vars}(N) - \{x\})$$
$$\text{where } U = \texttt{SpecialVars}(N)$$

The function $\texttt{Eliminate}_{\mathcal{N}}(N, V)$ eliminates all variables in set V from the abstract element N. The function $\texttt{Eliminate}_{\mathcal{N}}$ is part of the standard abstract interpretation interface that the numerical domain $\mathcal{N}$ comes equipped with. $\texttt{Vars}(N)$ denotes the set of all variables that occur in N.

The `Saturate` function above is inspired by the Nelson-Oppen methodology of combining decision procedures for disjoint theories [27], where elements from different theories share variable equalities (since that is the only information that can be understood by elements from both theories) until no more equalities can be shared. In our case, the information that can be understood by the set-domain element as well as the numerical element involve relating the size of any base-set with a constant. The Nelson-Oppen decision procedure terminates because the number of independent variable equalities that can be shared is bounded above by the number of variables. In our case, the number of relationships that can be shared is potentially unbounded since there is no limit on the size of the constants. To address this issue, we allow for sharing of only those relationships that involve constants up to a bounded size, say 2. This is because, in practice, all instances of set domains can usually only make use of the information whether the size of a base-set is 0, 1, or more than 1. Bounding the size of constants that can be shared during the saturation process guarantees efficient termination of the `Saturate` function.

We show below an example, where the `Saturate` function leads to repeated sharing of information between a set-domain element P and a numerical element N.

EXAMPLE 2. *Consider a program that traverses two linked lists of the same length. One of the lists is pointed to by x and the other by y. Traversing the first list using the statement $z = x \rightarrow \mathit{next}$ will cause the set domain to perform a case split on whether x is a singleton list or not. The case in which x is a singleton yields the element (P, N) where P is $[x \mapsto \mathit{nil}]^A \star [ls(y, \mathit{nil})]^B \wedge z = \mathit{nil}$ and N is $A = B$. Calling $\texttt{Generate}_{\mathcal{P}}(P)$ results in $A = 1 \wedge B \geq 1$, which is used to strengthen N yielding $A = 1 \wedge A = B$. Finally, $\texttt{Generate}_{\mathcal{N}}(A = 1 \wedge A = B)$ is $A = 1 \wedge B = 1$, i.e., $|x \mapsto \mathit{nil}| = 1 \wedge |ls(y, \mathit{nil})| = 1$. When used to strengthen P this yields $x \mapsto \mathit{nil} \star y \mapsto \mathit{nil} \wedge z = \mathit{nil}$. Thus, using the cardinality information we have discovered that the second list is a singleton as well.*

Saturating the input abstract elements is the first step in all the abstract domain operations described in Section 5. However, in the examples there we use saturated elements as inputs to be able to concentrate on other interesting aspects of the algorithms.

***P2N* Operator** Note that the base-sets in P_3 may have different names than those in P_1, yet the base-sets in P_3 might be related to those in P_1 since these are interpreted base-sets. The function $\texttt{P2N}(P_2, P_1, N)$ performs the role of relating the sizes of the base-sets in P_1 and P_3 using the `Witness` operator, translating this into a numerical relationship, and communicating this information to N.

Given any two set-domain elements P, P', and a numerical element N, the function $\texttt{P2N}(P, P', N)$ yields a numerical element that is more precise than N and incorporates information about numerical relationships between sizes of base-sets in P' and those in P.

$$\texttt{P2N}(P, P', N) \stackrel{\text{def}}{=} \texttt{PostPredicate}_{\mathcal{N}}(N, S)$$

where S is the conjunction of linear inequalities, one corresponding to each normalized set-inclusion relationship in $\texttt{WS} = \texttt{Witness}(\texttt{BaseSets}(P'), P)$.

$$S = \{\sum_i \widetilde{p_i} \leq \sum_j \widetilde{p'_j} \mid (\bigcup_i p_i \subseteq \bigcup_j p'_j) \in \texttt{WS}\}$$

The function $\texttt{PostPredicate}_{\mathcal{N}}(N, \texttt{pred})$ strengthens N by assuming the predicate `pred`.

We require that the `P2N` operator satisfies the following transitivity property.

PROPERTY 1. *For any set-domain elements P, P', P'' and any numerical element N, if $P \preceq_{\mathcal{P}} P' \preceq_{\mathcal{P}} P''$, then $\exists V : \mathit{P2N}(P', P'', \mathit{P2N}(P, P', N)) = \mathit{P2N}(P, P'', N)$, where $V = \mathit{SpecialVars}(P')$.*

The operator `P2N` is used extensively in the implementation of the abstract domain operations for the combined domain to relate base-sets coming from different set-domain elements, and strengthen the given numerical element with this information.

THEOREM 1. *The relation $\preceq_{\mathcal{P} \bowtie \mathcal{N}}$ defined above is, in fact, a pre-order.*

The proof of Theorem 1 is given in the full version of the paper [16]. To recap, the pre-order saturates the left element to share information between the set domain and the numerical domain. It then uses `P2N` to relate the base sets of the left element to those of the right element.

The pre-order, as stated above, also defines the logic or formalizes the reasoning power of our combination framework. In that regard it is related to decision procedures that have been described for logics that combine sets, and numerical properties of their cardinalities. However, the two main differences are: (a) Our combination framework is modular wherein any set analysis can be combined with any numerical analysis, and (b) more importantly, we show how to perform abstract interpretation of a program over such a

logic. Performing abstract interpretation requires many more transfer functions besides a decision procedure (such as `Join`, `Widen`, `Eliminate`, etc).

5. Abstract Interpreter for the Set Cardinality Domain

Let $\mathcal{P}$ and $\mathcal{N}$ be any set domain and numerical domain respectively. In this section, we show how to efficiently combine the abstract interpreters that operate over the abstract domains $\mathcal{P}$ and $\mathcal{N}$ to obtain an abstract interpreter that operates over the set cardinality domain $\mathcal{P} \ltimes \mathcal{N}$. Our combination methodology yields the most precise abstract interpreter for the set cardinality domain $\mathcal{P} \ltimes \mathcal{N}$ relative to the pre-order defined in Section 4. The key idea of our combination methodology is to combine the corresponding transfer functions of the abstract interpreters that operate over the domains $\mathcal{P}$ and $\mathcal{N}$ to yield the transfer functions of the abstract interpreter that operates over the domain $\mathcal{P} \ltimes \mathcal{N}$.

An abstract interpreter performs a forward analysis on the program computing invariants (which are elements of the underlying abstract domain over which the analysis is being performed) at each program point. The invariants are computed at each program point from the invariants at the preceding program points in an iterative manner using appropriate transfer functions. We first describe our program model in Section 5.1. In the subsequent sections, we describe the construction of these transfer functions for the set cardinality domain $\mathcal{P} \ltimes \mathcal{N}$ in terms of the transfer functions for the individual domains $\mathcal{P}$ and $\mathcal{N}$.

Further information including proofs can be found in the full version of this paper [16].

5.1 Program Model

We assume that each procedure in a program is abstracted using the flowchart nodes shown in Figure 3.

We allow for assume and assert program statements of the form `assume`(`pred`) and `assert`(`pred`), where `pred` is a predicate that is either understood by the set domain $\mathcal{P}$, or it is a linear inequality predicate. The linear inequality predicate can be over program variables and over special variables that denote the cardinality of some base-set that can be specified using some base-set constructors exported by the set domain $\mathcal{P}$.

Since we allow for `assume` statements, without loss of generality, we can treat all conditionals in the program as non-deterministic (i.e., control can flow to either branch irrespective of the program state before the conditional). A join node has two incoming edges. Note that a join node with more than two incoming edges can be reduced to multiple join nodes each with two incoming edges.

We now describe the transfer functions for each of the flowchart nodes. For lack of space, we leave out the description of the transfer function for `Assert` node, which is quite similar to the definition of the pre-order. See full version [16] for more details.

5.2 Join Node

The abstract element E after a join node (Figure 3(a)) is obtained by computing the join of the elements E_1 and E_2 before the join node using the join operator.

$$E = \texttt{Join}_{\mathcal{D}}(E_1, E_2)$$

The join operator $\texttt{Join}_{\mathcal{D}}$ for a domain $\mathcal{D}$ takes as input two elements E_1 and E_2 from domain $\mathcal{D}$ and computes an optimal upper bound of E_1 and E_2 with respect to the pre-order $\preceq_{\mathcal{D}}$. The following definition makes this more precise.

DEFINITION 1 (Join Operator $\texttt{Join}_{\mathcal{D}}$). *Let $E = \mathit{Join}_{\mathcal{D}}(E_1, E_2)$. Then,*

- *Soundness: $E_1 \preceq_{\mathcal{D}} E$ and $E_2 \preceq_{\mathcal{D}} E$.*
- *Completeness: If E' is such that $E_1 \preceq_{\mathcal{D}} E'$ and $E_2 \preceq_{\mathcal{D}} E'$ and $E' \preceq_{\mathcal{D}} E$, then $E \preceq_{\mathcal{D}} E'$.*

Figure 4 shows how to implement the join operator $\texttt{Join}_{\mathcal{P} \ltimes \mathcal{N}}$ for the set cardinality domain $\mathcal{P} \ltimes \mathcal{N}$ using the join operators $\texttt{Join}_{\mathcal{P}}$ and $\texttt{Join}_{\mathcal{N}}$ for the set and numerical domains in a modular fashion. The implementation also makes use of the eliminate operator $\texttt{Eliminate}_{\mathcal{N}}$ for the numerical domain, which is described in Section 5.3.

EXAMPLE 3. *We explain the implementation of the $\mathit{Join}_{\mathcal{P} \ltimes \mathcal{N}}$ operator by considering the example in Figure 4(b). This is a simplified example of a situation that occurs during an in-place reversal of a linked list. The first input (P_1, N_1) represents two disjoint lists, the list pointed to by x is of length 1 and the list pointed to by y is of length $n-1$. The second input (P_2, N_2) also represents two disjoint lists. Here, the list pointed to by y is of length 1 and the list pointed to by x is of length $n-1$.*

The first step in joining the input elements is to saturate both of them. Remember that in all the examples, we use saturated elements to be able to concentrate on the important issues. Thus, the saturate operation has nothing to add.

Now, the join of the set domain is performed yielding P, which represents two disjoint lists pointed to by x and y. The `P2N` operator is used to strengthen the numerical element by relating the cardinalities of the base-sets of P to the cardinalities of the base-sets in the original elements. This is done using the `Witness` operation, which interprets the base-sets of P using the base-sets of the original elements[3]. For N_1 this means that $A = E$ and $B = F$ and for N_2 this means that $C = E$ and $D = F$. Note that without `P2N` there will be no relation between the base-sets of the two inputs and thus the numerical join would simply return $n \geq 2$. Next, the numerical join is performed. The join loses the fact that one of the lists was a singleton, but retains the important information that the sum of the lengths of the lists is n.

Finally, the numerical variables corresponding to the original base-sets are eliminated to ensure that all the special variables in the numerical element come from the set-domain element. In case of polyhedra join, this has no effect as any information on the original special variables relates to only one of the inputs and is thus lost in the join.

THEOREM 2. *The join operator described in Figure 4(a) satisfies both the soundness and the completeness property stated in Definition 1 (provided the join operators for the base domains, $\mathit{Join}_{\mathcal{P}}$ and $\mathit{Join}_{\mathcal{N}}$, satisfy these properties, and the eliminate operator for the numerical domain, $\mathit{Eliminate}_{\mathcal{N}}$, satisfies the respective soundness and completeness properties stated in Definition 2 on Page 8).*

The proof of Theorem 2 is available in the full-version of the paper [16].

5.3 Assignment Node

The abstract element E' after an assignment node $\ell := e$ (Figure 3(b)) is the strongest postcondition of the element E before the assignment node with respect to the assignment $\ell := e$. It is computed by using an existential quantification operator $\texttt{Eliminate}_{\mathcal{D}}$ as described below.

$$\begin{aligned} E' &= \texttt{Eliminate}_{\mathcal{D}}(E_1, x') \\ \text{where } E_1 &= \texttt{PostPredicate}_{\mathcal{D}}(E[x'/x], x = e[x'/x]) \end{aligned}$$

[3] Because any relation added by `P2N` is based on the `Witness` operation and the original elements are saturated, no new relationships will be added among the original base-sets.

$\texttt{Join}_{\mathcal{P}\bowtie\mathcal{N}}((P_1,N_1),(P_2,N_2))$ =

1 $(P_1', N_1') := \texttt{Saturate}(P_1, N_1)$;
2 $(P_2', N_2') := \texttt{Saturate}(P_2, N_2)$;
3 $P := \texttt{Join}_{\mathcal{P}}(P_1', P_2')$;
4 $N_1'' := \texttt{P2N}(P_1', P, N_1')$;
5 $N_2'' := \texttt{P2N}(P_2', P, N_2')$;
6 $N := \texttt{Join}_{\mathcal{N}}(N_1'', N_2'')$;
7 $V := \texttt{SpecialVars}(N) - \texttt{SpecialVars}(P)$
8 $N' := \texttt{Eliminate}_{\mathcal{N}}(N, V)$;
9 $\texttt{Output}\ (P, N')$;

(a) Algorithm

Inputs:

$P_1 = [x \mapsto \texttt{nil}]^A \star [ls(y, \texttt{nil})]^B$
$N_1 = A = 1 \wedge B = n-1 \wedge B \geq 1$
$P_2 = [ls(x, \texttt{nil})]^C \star [y \mapsto \texttt{nil}]^D$
$N_2 = D = 1 \wedge C = n-1 \wedge C \geq 1$

Trace of $\texttt{Join}_{\mathcal{P}\bowtie\mathcal{N}}((P_1,N_1),(P_2,N_2))$:

$P = [ls(x, \texttt{nil})]^E \star [ls(y, \texttt{nil})]^F$
$N_1'' = A = E \wedge B = F \wedge A = 1 \wedge B = n-1 \wedge B \geq 1$
$N_2'' = C = E \wedge D = F \wedge D = 1 \wedge C = n-1 \wedge C \geq 1$
$N = E + F = n \wedge E \geq 1 \wedge F \geq 1$
$N' = E + F = n \wedge E \geq 1 \wedge F \geq 1$

(b) Example

Figure 4. This figure describes the algorithm for join transfer function for set cardinality domain $\mathcal{P} \bowtie \mathcal{N}$ in terms of the join transfer functions for the domains $\mathcal{P}$ and $\mathcal{N}$ along with an example.

$\texttt{Eliminate}_{\mathcal{P}\bowtie\mathcal{N}}((P,N),\ell)$ =

1 $(P', N') := \texttt{Saturate}(P, N)$;
2 $P_1 := \texttt{Eliminate}_{\mathcal{P}}(P', \ell)$;
3 $N_1 := \texttt{P2N}(P', P_1, N')$;
4 $V := \{\ell\} \cup \texttt{SpecialVars}(N_1) - \texttt{SpecialVars}(P_1)$
5 $N_2 := \texttt{Eliminate}_{\mathcal{N}}(N_1, V)$;
6 $\texttt{Output}\ (P_1, N_2)$;

(a) Algorithm

Inputs:

$P = [ls(x, z)]^A \star [ls(z, \texttt{nil})]^B$
$N = A \geq 1 \wedge B \geq 1 \wedge A = n \wedge B = k$
$\ell = z$

Trace of $\texttt{Eliminate}_{\mathcal{P}\bowtie\mathcal{N}}((P,N),\ell)$:

$P_1 = [ls(x, \texttt{nil})]^C$
$N_1 = A + B = C \wedge A \geq 1 \wedge B \geq 1 \wedge A = n \wedge B = k$
$V = \{z, A, B\}$
$N_2 = C = n + k \wedge n \geq 1 \wedge k \geq 1$

(b) Example

Figure 5. This figure describes the algorithm for existential elimination for the set cardinality domain $\mathcal{P} \bowtie \mathcal{N}$ in terms of the existential elimination algorithms for the domains $\mathcal{P}$ and $\mathcal{N}$ along with an example.

The post-predicate operator $\texttt{PostPredicate}_{\mathcal{D}}$ is defined in Section 5.4. The existential quantification operator $\texttt{Eliminate}_{\mathcal{D}}$ for any domain $\mathcal{D}$ takes as input an element E from $\mathcal{D}$ and an lvalue ℓ, and produces the least element that is above E and does not get affected by any change to ℓ.

DEFINITION 2 (Eliminate Operator $\texttt{Eliminate}_{\mathcal{D}}$). *Let $E' = \mathit{Eliminate}_{\mathcal{D}}(E, \ell)$. Then,*

- *Soundness: $E \preceq_{\mathcal{D}} E'$ and E' does not get affected by any change to ℓ.*
- *Completeness: If E'' is such that $E \preceq_{\mathcal{D}} E''$ and E'' does not get affected by any change to ℓ, then $E' \preceq_{\mathcal{D}} E''$.*

Figure 5 shows how to implement the eliminate operator $\texttt{Eliminate}_{\mathcal{P}\bowtie\mathcal{N}}$ for the set cardinality domain $\mathcal{P} \bowtie \mathcal{N}$ using the eliminate operators $\texttt{Eliminate}_{\mathcal{P}}$ and $\texttt{Eliminate}_{\mathcal{N}}$ for the set and numerical domains in a modular fashion.

EXAMPLE 4. *We demonstrate the implementation of the operator $\mathit{Eliminate}_{\mathcal{P}\bowtie\mathcal{N}}$ by the example in Figure 5(b). The example comes from appending two linked lists. The input is a list pointed to by x whose original length is n and a list pointed to by z whose length is k. The second list has been appended to the first list. Now, we wish to existentially eliminate z.*

First, we eliminate z from the set domain, yielding P_1, a list pointed to by x, losing the information on where z pointed to. Next, we use P2N to express the cardinalities of the base-sets of P in terms of the cardinalities of the base-sets of P_1. In this case, $A + B = C$, i.e., the sum of the lengths of the two parts of the list in P is equal to the length of the list in P_1. Next we eliminate z and variables corresponding to the original base-sets from the numerical element. This loses the original partition of the list and retains the important information that the length of the list is $n+k$.

THEOREM 3. *The Eliminate operator described in Figure 5(a) satisfies both the soundness and the completeness property stated in Definition 2 (provided the eliminate operator for the base domains, $\mathit{Eliminate}_{\mathcal{P}}$ and $\mathit{Eliminate}_{\mathcal{N}}$, satisfy these properties).*

The proof of Theorem 3 is available in the full-version of the paper [16].

5.4 Assume Node

The abstract element E' after an assume node $\texttt{Assume}(\texttt{pred})$ (Figure 3(c)) is obtained by using the post-predicate operator described below.

$$E' = \texttt{PostPredicate}_{\mathcal{D}}(E, \texttt{pred})$$

The post-predicate operator $\texttt{PostPredicate}_{\mathcal{D}}$ for a domain $\mathcal{D}$ takes as input an abstract element E from domain $\mathcal{D}$ and a predicate pred and returns the most precise abstract element E' such that $\gamma_{\mathcal{D}}(E') \supseteq \gamma_{\mathcal{D}}(E) \cap \gamma(\texttt{pred})$, where γ is the concretization operation. The following makes this more precise.

DEFINITION 3 (Post-predicate Operator $\texttt{PostPredicate}_{\mathcal{D}}$). *Let $E' = \mathit{PostPredicate}_{\mathcal{D}}(E, \mathit{pred})$. Let $\gamma_{\mathcal{D}}$ denote the concretization function for domain $\mathcal{D}$. Then,*

- *Soundness: $\gamma_{\mathcal{D}}(E') \supseteq \gamma_{\mathcal{D}}(E) \cap \gamma(\mathit{pred})$.*
- *Completeness: If E'' is such that $\gamma_{\mathcal{D}}(E') \supseteq \gamma_{\mathcal{D}}(E) \cap \gamma(\mathit{pred})$, then $E' \preceq_{\mathcal{D}} E''$.*

Figure 6 shows how to implement the post-predicate operator $\texttt{PostPredicate}_{\mathcal{P}\bowtie\mathcal{N}}$ for the set cardinality domain $\mathcal{P} \bowtie \mathcal{N}$ using the post-predicate operators for the set and numerical domains ($\texttt{PostPredicate}_{\mathcal{P}}$ and $\texttt{PostPredicate}_{\mathcal{N}}$) in a modular fashion.

EXAMPLE 5. *We demonstrate the $\mathit{PostPredicate}_{\mathcal{P}\bowtie\mathcal{N}}$ operator by the example in Figure 6(b). We return to the example of a list of length n pointed to by x to which a list of length k pointed to by z has been appended. We wish to assume the predicate $m = |ls(x, \mathit{nil})|$, i.e., that m is the length of the entire list pointed to by x. Note that the predicate refers to a base-set that P can interpret*

```
PostPredicate_{P⋈N}((P, N), pred) =
1  if pred is an arithmetic predicate:
2     V := SpecialVars(pred) − SpecialVars(P);
3     N_1 := PostPredicate_N(N, pred);
4     N_2 := P2N(P, V, N_1);
5     N_3 := Eliminate_N(N_2, V);
6     return (P, N_3);
7  else // pred is set predicate:
8     P_1 := PostPredicate_P(P, pred);
9     return (P_1, N)
```

(a) Algorithm

Inputs:

$$P = [ls(x,z)]^A \star [ls(z,\texttt{nil})]^B$$
$$N = A \geq 1 \wedge B \geq 1 \wedge A = n \wedge B = k$$
$$\texttt{pred} = |[ls(x,\texttt{nil})]^C| = m$$

Trace of $\texttt{PostPredicate}_{\mathcal{P}\bowtie\mathcal{N}}((P,N),\texttt{pred})$:

$$V = \{C\}$$
$$N_1 = m = C \wedge A \geq 1 \wedge B \geq 1 \wedge A = n \wedge B = k$$
$$N_2 = A + B = C \wedge m = C \wedge A \geq 1 \wedge B \geq 1 \wedge A = n \wedge B = k$$
$$N_3 = A \geq 1 \wedge B \geq 1 \wedge A = n \wedge B = k \wedge m = n + k$$

(b) Example

Figure 6. This figure describes the algorithm for the post-predicate operation (transfer function for assume node) in the set cardinality domain $\mathcal{P} \bowtie \mathcal{N}$ in terms of the post-predicate operations for the individual domains $\mathcal{P}$ and $\mathcal{N}$ along with an example.

but is not in $\texttt{BaseSets}(P)$. The variable that represents the length of the list pointed to by x is C. First we assume that $m = C$. Next, P2N is used to interpret C in terms of the cardinalities of the base-sets in P. In this case, $C = A+B$, i.e., the sum of the lengths of the two parts of the list. Finally, we eliminate the special variables that do not correspond to the base-sets in $\texttt{BaseSets}(P)$. In this case, C is eliminated, retaining the important information that $m = n + k$.

THEOREM 4. *The post-predicate operator described in Figure 6(a) satisfies the soundness property stated in Definition 3 (provided the post-predicate operators for the base domains, $\texttt{PostPredicate}_{\mathcal{P}}$ and $\texttt{PostPredicate}_{\mathcal{N}}$, satisfy the same soundness property, and the eliminate operator for the numerical domain, $\texttt{Eliminate}_{\mathcal{N}}$, satisfies the respective soundness property stated in Definition 2).*

The proof of Theorem 4 follows simply from the observation that the concretization function for the combined domain is the intersection of the concretization functions of the set domain and the numerical domain. However, note that the post-predicate operator described in Figure 6(a) does not necessarily satisfy the completeness property stated in Definition 3 because the pre-order for the combined domain $\preceq_{\mathcal{P}\bowtie\mathcal{N}}$ only accounts for a limited (not necessarily complete) sharing of information between the set domain and the numerical domain. In other words, our pre-order is not the best partial-order that corresponds to the concretization function for the combined domain.

5.5 Fixed-point computation

In presence of loops, the abstract interpreter goes around each loop until a fixed point is reached. A fixed point is said to be reached when the abstract elements E_1, E_2 over domain $\mathcal{D}$ at any program point inside the loop in two successive iterations of that loop represent the same set of concrete elements, i.e., $E_1 \preceq_{\mathcal{D}} E_2$ and $E_2 \preceq_{\mathcal{D}} E_1$.

If the domains $\mathcal{P}$ or $\mathcal{N}$ have infinite chains, then fixed point for a loop may not be reached in a finite number of steps. In that case, a widening operation may be used to over-approximate the analysis results at loop heads.

A widening operator for a domain $\mathcal{D}$ takes as input two elements from $\mathcal{D}$ and produces an upper bound of those elements (which may not necessarily be the least upper bound). A widening operator has the property that it guarantees fixed point computation across loops terminates in a finite number of steps even for infinite height domains.

A widen operator $\texttt{Widen}_{\mathcal{D}}$ for a domain $\mathcal{D}$ takes as input two elements E_1 and E_2 from domain $\mathcal{D}$ and returns an element E with the following property:

DEFINITION 4 (Widening Operator $\texttt{Widen}_{\mathcal{D}}$). *Let $E = \texttt{Widen}_{\mathcal{D}}(E_1, E_2)$. Then,*

- *Soundness: $E_1 \preceq_{\mathcal{D}} E$ and $E_2 \preceq_{\mathcal{D}} E$.*
- *Convergence: The sequence of widen operations converges in a bounded number of steps, i.e., for any strictly increasing sequence $E_0, E_1, \ldots$ (such that $E_i \preceq_{\mathcal{D}} E_{i+1}$ but $E_{i+1} \not\preceq_{\mathcal{D}} E_i$ for all i) , if we define $E'_0 := E_0$, $E'_1 := \texttt{Widen}_{\mathcal{D}}(E'_0, E_1)$, $E'_2 := \texttt{Widen}_{\mathcal{D}}(E'_1, E_2), \ldots$, then there exists $i \geq 0$ such that $E'_j \preceq_{\mathcal{D}} E'_i$ and $E'_i \preceq_{\mathcal{D}} E'_j$ for all $j > i$.*

Figure 7 shows how to implement the widen operator $\texttt{Widen}_{\mathcal{P}\bowtie\mathcal{N}}$ for the set cardinality domain $\mathcal{P} \bowtie \mathcal{N}$ using the widen operators $\texttt{Widen}_{\mathcal{P}}$ and $\texttt{Widen}_{\mathcal{N}}$ for the set and numerical domains in a modular fashion.

EXAMPLE 6. *We demonstrate the $\texttt{Widen}_{\mathcal{P}\bowtie\mathcal{N}}$ operator by the example in Figure 7(b). This example is taken from a program that creates a list of length n. After the first iteration, the list pointed to by x is a singleton and the loop counter $i = 1$. After the second iteration, the length of the list pointed to by x is between 1 and 2 and i equals the length of the list. Note that using Join, the constant 2 will keep increasing without ever converging. The Widen operator for the set domain is equivalent to Join for this set domain and returns a non-empty list pointed to by x. The P2N operator expresses the cardinalities of the original base-sets in terms of this new list. Eliminating the special variables for the original base-sets yields numerical elements in which all the information is in terms of the new base-set. This elimination is necessary to ensure the widening operator precondition that N''_2 is weaker than N''_1. Finally, the numerical widen operator loses the possible range of the length of the list, but retains the important information that the length of the list equals the iteration number and that $i \leq n$, which will allow us to prove that after the loop terminates the length of the list is n.*

The Widen operation described in Figure 7 clearly satisfies the soundness property described above. It also satisfies the convergence property; as stated in the theorem below.

THEOREM 5. *The Widen operation described in Figure 7 satisfies the convergence property stated in Definition 4 (provided the Widen operations for the base domains, $\texttt{Widen}_{\mathcal{P}}$ and $\texttt{Widen}_{\mathcal{N}}$, satisfy Definition 4).*

PROOF: We give a brief sketch of the proof. (Details are in the full version of the paper [16].) Let $(P_1, N_1), (P_2, N_2), \ldots$ be any chain of elements in the set cardinality domain that arise at a given program point during successive loop iterations. Let $(Q_1, M_1) = (P_1, N_1)$ and $(Q_{i+1}, M_{i+1}) = \texttt{Widen}_{\mathcal{P}\bowtie\mathcal{N}}((Q_i, M_i), (P_i, N_i))$. Then, it can be shown that: (1) $Q_i \preceq_{\mathcal{P}} Q_j$ for all $i < j$, and (2) If $Q_j \preceq_{\mathcal{P}} Q_i$ for some $i < j$, then $M_k \preceq_{\mathcal{N}} M_{k+1}\sigma$ and $M_{k+1}\sigma \not\preceq_{\mathcal{N}} M_k$ for all $i \leq k < j$ (otherwise the fixed-point computation converges), where σ is some bijective variable renaming. The result now

$\texttt{Widen}_{\mathcal{P}\bowtie\mathcal{N}}((P_1,N_1),(P_2,N_2))$ =

1. $(P_1',N_1') := (P_1,N_1)$;
2. $(P_2',N_2') := \texttt{Saturate}(P_2,N_2)$;
3. $P := \texttt{Widen}_{\mathcal{P}}(P_1',P_2')$;
4. $N_1'' := \texttt{P2N}(P_1',P,N_1')$;
5. $N_2'' := \texttt{P2N}(P_2',P,N_2')$;
6. $N_1''' := \texttt{Eliminate}_{\mathcal{N}}(N_1'', \texttt{SpecialVars}(N_1') - \texttt{SpecialVars}(P))$;
7. $N_2''' := \texttt{Eliminate}_{\mathcal{N}}(N_2'', \texttt{SpecialVars}(N_2') - \texttt{SpecialVars}(P))$;
8. $N := \texttt{Widen}_{\mathcal{N}}(N_1''', N_2''')$;
9. Output (P,N);

(a) Algorithm

Inputs:

$$\begin{array}{rcl} P_1 & = & [x \mapsto \texttt{nil}]^A \\ N_1 & = & A = 1 \wedge i = 1 \wedge i \leq n \\ P_2 & = & [ls(x,\texttt{nil})]^B \\ N_2 & = & B \geq 1 \wedge B \leq 2 \wedge i = B \wedge i \leq n \end{array}$$

Trace of $\texttt{Widen}_{\mathcal{P}\bowtie\mathcal{N}}((P_1,N_1),(P_2,N_2))$:

$$\begin{array}{rcl} P & = & [ls(x,\texttt{nil})]^C \\ N_1'' & = & C = A \wedge A = 1 \wedge i = 1 \wedge i \leq n \\ N_2'' & = & C = B \wedge B \geq 1 \wedge B \leq 2 \wedge i = B \wedge i \leq n \\ N_1''' & = & C = 1 \wedge i = C \wedge i \leq n \\ N_2''' & = & C \geq 1 \wedge C \leq 2 \wedge i = C \wedge i \leq n \\ N & = & C \geq 1 \wedge i = C \wedge i \leq n \end{array}$$

(b) Example

Figure 7. This figure describes the algorithm for widening for set cardinality domain $\mathcal{P} \bowtie \mathcal{N}$ in terms of the widening algorithms for the domains $\mathcal{P}$ and $\mathcal{N}$ along with an example.

follows from the observation that the length of the chain is bounded above by the product of the number of times Q_i can strictly increase and the number of times M_i can strictly increase up to variable renaming.

□

6. Case Study

The choice of which set analysis and which numerical analysis to combine depends on what data-structures and what properties of those data-structures we want to analyze. Different data-structures typically require different base-set constructors, while different operations on the same data-structure typically require different numerical domains. We illustrate this by two sets of examples.

Table 1 shows the loop invariants required to establish relationships between the size of the output data-structure and the size of the input data-structure for `Copy` function. These examples require different set domains, but the same numerical domain, namely Karr's linear equalities domain, works for all of these examples. The base-set constructors given here are used to informally demonstrate the type of invariants the set domain should be able to represent. The exact way that these base-sets are defined changes according to the set domain used.

Table 2 shows the loop invariants required to establish relationships between the sizes of the output data-structure and the size of the input data-structure on various functions for an acyclic list (see Table 1 for the definition of $R(x,y)$). These examples require different numerical domains, but the same set analysis domain, namely one that provides a "reachability via next link" base-set constructor, works for all of these examples.

It is also interesting to note that in order to validate a given program, one can either choose a more precise set domain or a more precise numerical domain. For example, consider proving the data-structure invariant for the list copy example. The inductive invariant required to prove that the size of the copied list is the same as the size of the input list can either be expressed as $|R(x',\texttt{null})| = |R(x,y)|$ (see Copy example in Table 2) or $|R(x')| = |R(x)| - |R(y)|$ (see Acyclic List example in Table 1). The former is an element in the set cardinality domain built from a relatively more precise set domain (i.e., one that supports $R(x,y)$ as a base-set constructor as opposed to simply supporting $R(x)$), but a relatively less precise numerical domain (i.e., one that supports variable equalities, as opposed to arbitrary linear equalities).

The above observations are not supposed to imply that for each program, we need to work with a different combination of a set domain and a numerical domain. Certain set domains and certain numerical domains are more precise than several others; but precision comes at the cost of efficiency. Hence, we need to estimate the least precise set domain and the least precise numerical domain that would be good enough to reason about desired properties of desired programs.

6.1 Experimental Results

We have implemented an instance of the combination framework by combining the TVLA system [22] with the Polyhedra abstract domain [8] as implemented by the PPL library [2] using some extra widening heuristics. We chose these domains as they can together handle all of the benchmarks described below. Using less precise domains it would be possible to prove some of these benchmarks with better efficiency. Table 3 summarizes the results of running the tool on a set of benchmarks. In all cases, the properties specified were proven without false alarms. The benchmarks were run on a 2.4GHz E6600 Core 2 Duo processor with 2 GB of memory running Linux. For each program we give the time, the overhead factor over running TVLA without cardinality support, and the total number of abstract shape graphs generated in the analysis. Note that the overhead is rather low with an average of 60%. In some cases, the analysis with cardinality is even faster, as it can prune some of the search space using the more precise domain. In some of the examples, running without cardinality support yields memory safety false alarms.

The benchmarks are divided into the five categories detailed below.

StringBuffer We analyzed the two most challenging methods from among the methods supported by the String Buffer class (as implemented in Microsoft product code): SBRemove (see Figure 1) and SBToString. SBToString converts a StringBuffer to a single string by allocating an array of the appropriate size and copying the characters in the correct order. We prove memory safety and data structure invariants on both examples. In SBRemove, we prove that the number of characters in the resulting StringBuffer is in sync with the number of characters removed. In SBToString, we prove that the size of the resulting string equals the number of characters in the original StringBuffer. These examples combine recursively defined data structures with arrays and demonstrate how the domain can track non-trivial relationships between the heap and the numerical variables in the programs.

Termination We prove termination of two non-trivial examples: BubbleSort (see Figure 2) and Mark. The Mark example performs a DFS scan of a graph marking nodes as they are visited and using a stack for pending nodes. The scan terminates when the pending stack is empty. Proving termination in this case is non trivial as the pending stack can grow as well as shrink in each iteration. Our technique is able to prove termination of this example by establishing the inductive invariant that the instrumented loop

Program	Loop Invariant	Base-set Constructor
Acyclic List	$\lvert R(x')\rvert = \lvert R(x)\rvert - \lvert R(y)\rvert$	$R(x) = \emptyset$, if $x = \texttt{null}$ $= \{x\} \cup R(x \rightarrow \texttt{next})$, otherwise
Cyclic List	$\lvert R(x', y')\rvert = \lvert R(x, y)\rvert$	$R(x, y) = \emptyset$, if $x = y$ $= \{x\} \cup R(x \rightarrow \texttt{next}, y)$, otherwise
Tree	$\lvert R_t(x')\rvert = \lvert R_t(x)\rvert$	$R_t(x) = \emptyset$, if $x = \texttt{null}$ $= \{x\} \cup R_t(x \rightarrow \texttt{left}) \cup R_t(x \rightarrow \texttt{right})$, otherwise
n-ary Tree	$\lvert R_n(x', 0, i)\rvert = \lvert R_n(x, 0, i)\rvert$	$R_n(x) = R_n(x, 0, x \rightarrow \texttt{nrChild})$ $R_n(x, \texttt{low}, \texttt{high}) = \emptyset$, if $x = \texttt{null}$ $= \{x\} \cup \bigcup_{\texttt{low} \leq k < \texttt{high}} R_n(x \rightarrow \texttt{children}[k])$
List of Lists	$\lvert R_\ell(x')\rvert = \lvert R_\ell(x)\rvert - \lvert R_\ell(y)\rvert$	$R_\ell(x) = \emptyset$, if $x = \texttt{null}$ $= R(x \rightarrow \texttt{list}) \cup R_\ell(x \rightarrow \texttt{nextL})$, otherwise
List of Arrays (e.g., StringBuffer)	$\lvert R_a(x')\rvert = \lvert R_a(x)\rvert - \lvert R_a(y)\rvert$	$A(x, \texttt{l}, \texttt{h}) = \{x[k] \mid \texttt{l} \leq k < \texttt{h}\}$ $R_a(x) = \emptyset$, if $x = \texttt{null}$ $= A(x \rightarrow \texttt{arr}, 0, x \rightarrow \texttt{len}) \cup R_a(x \rightarrow \texttt{next})$, otherwise
Array of Lists (e.g., Hashtable)	$\lvert T(x', 0, i)\rvert = \lvert T(x, 0, i)\rvert$	$T(x, \texttt{low}, \texttt{high}) = \bigcup_{\texttt{low} \leq k < \texttt{high}} R(x[k])$

Table 1. This table describes the loop invariants (and hence illustrates the choice of the set analysis domain) required to analyze the `Copy` routine of various data-structures. The property discovered is the relationship between the size of the output data-structure with the size of the input data-structure.

Program	Loop Invariant	Numerical Domain
Copy	$\lvert R(x', \texttt{null})\rvert = \lvert R(x, y)\rvert$	Variable Equalities
Reverse	$\lvert R(x', \texttt{null})\rvert = \lvert R(x, y)\rvert$	Variable Equalities
Filter	$\lvert R(x', \texttt{null})\rvert \leq \lvert R(x, y)\rvert$	Difference Constraints
Merge	$\lvert R(x', y')\rvert = \lvert R(x_1, y_1)\rvert + \lvert R(x_2, y_2)\rvert$	Karr's Domain [20] (arbitrary linear equalities)
Merge Without Duplicates	$\lvert R(x', y')\rvert \leq \lvert R(x_1, y_1)\rvert + \lvert R(x_2, y_2)\rvert$	Polyhedra Domain [8] (Linear Inequalities)

Table 2. This table describes the loop invariants (and hence illustrates the choice of the numerical analysis domain) required to analyze various routines of acyclic list data-structure. The property discovered is the relationship between the size of the output data-structure with the size of the input data-structure.

counter is bounded above by the cardinality of the set of nodes whose visited flag has been set to `true`.

Linked List Examples This includes all examples in Table 2. We prove memory safety and data-structure invariants for all examples. In Reverse we prove that the length of the reversed list equals the length of the original list. In filter we prove that the length of the list is less or equal to the length of the original list. In Merge we prove that the length of the resulting list is the sum of lengths of the original lists. In MergeNoDups we prove that the length of the resulting list is less or equal to the sum of lengths of the original lists.

Data Structure Copy This includes all examples in Table 1. These examples illustrate the power of our technique for proving bounds on memory allocation in terms of inputs. We prove that the size of the copied data-structure is equal to the size of the original input data-structure. Note that since a deep copy is performed, the relationship between the memory locations of the original data structure and the copied one cannot be expressed using set comparison operators (like set equality or set inclusion).

JDK Collections Library We have used the tool to analyze most functions of the LinkedList and HashMap classes of JDK 5.0 [1]. The LinkedList class implements a circular doubly-linked list and the HashMap class implements an array of disjoint singly-linked lists. These functions, listed in Table 3, include the ones used to add a single element, add multiple elements and remove an element. We prove memory safety, data-structure invariants, and correct maintenance of the size field in all the examples (i.e., the size field corresponds to the number of elements in the collection). In addition, for HMPutAll we prove that the size of the resulting HashMap is greater or equal to the sizes of the original HashMaps, and less or equal to their sum.

7. Related Work

Combining Abstractions

The seminal paper by Cousot and Cousot in [7] introduces different methods for combining abstract domains including reduced product, which can be used to explain our domain construction (see [9] for further elaboration on domain constructors). However, the problem of developing an effective procedure for computing abstract transformers for reduced products has been addressed only in specific settings. Gulwani and Tiwari gave algorithms for constructing the transfer functions for reduced products for a special case of abstract domains (called logical abstract domains) with the further restriction that the abstract domains being combined should be over convex theories with disjoint signatures [17]. Their methodology is not applicable in our setting since the abstract domains that we consider in this paper, namely set domains and numerical domains, do not fit the required restrictions: the set domain is not convex, and

Category	Program	Time (secs)	Over-head	States
String Buffer	SBRemove	295.21	2.83	50,615
	SBToString	79.53	3.15	10,176
Termination	BubbleSort	3.57	0.54	886
	Mark	2.44	3.02	1,530
Linked List	Reverse	0.34	1.64	90
	Filter	0.76	0.54	238
	Merge	1.08	1.88	341
	MergeNoDups	4.06	2.53	1,838
Data Structure Copy	AcyclicListCopy	0.39	1.44	74
	CyclicListCopy	4.54	1.20	155
	TreeCopy	4.15	1.45	642
	NaryTreeCopy	138.20	N/A	5,439
	ListOfListsCopy	39.95	1.44	5,353
	ListOfArraysCopy	12.67	1.02	2,260
	ArrayOfListsCopy	7.99	0.30	1,628
JDK Collections Library	LLAdd	1.45	2.23	17
	LLAddAll	10.93	0.02	215
	LLRemove	2.51	1.20	173
	HMPut	9.45	1.02	3,132
	HMPutAll	111.84	2.59	22,431
	HMRemove	2.13	1.92	725

Table 3. Experimental results for the set cardinality benchmarks

furthermore, the set domain and the numerical domain both share the cardinality function symbol. Our work thus extends the line of work on constructive synthesis of abstract transformers for reduced product domains (from the abstract transformers of individual domains) for an important class of domains.

Combining Heap and Numerical Abstractions

The idea to combine numeric and pointer analysis for establishing properties of memory was pioneered by Alain Deutsch [10, 11]. Deutsch's abstraction deals with may-aliases in a rather precise way but loses most of the information when the program performs destructive memory updates.

In [19] a type and effect system is suggested for a variant of ML that allows to bound the size of memory used by the program with applications to embedded code. There, the type system allows verifying bounds on memory usage while our analysis can be used to infer the bound. Furthermore, their type system is for a functional language while our analysis is appropriate for an imperative language with destructive pointer updates.

In [18] linear typing and linear programming based inference are used to statically infer linear bounds on heap space usage of first-order functional programs running under a special memory mechanism. In contrast, our method handles imperative programs which use destructive updates.

In [35] an algorithm for inferring sizes of singly-linked lists was presented. This algorithm uses the fact that the number of uninterrupted list segments in singly-linked lists is bounded. This limits the applicability of the method for showing specific properties of singly-linked lists. Similar restrictions apply to [3, 23].

A general method for combining numeric domains and canonical abstraction was presented in [14]. Their method is orthogonal to ours, as it addresses the problem of abstracting values of numerical fields. On the other hand, our work is concerned with cardinalities of memory partitions. Combining the methods can be very useful and is the subject of future work.

Rugina [32] presents a static analysis that can infer quantitative properties (namely height and skewness) of tree-like heaps. Rugina does not address the issue of sizes of data structures and is limited to tree-like heaps. On the other hand, Rugina can handle properties such as height, which are beyond the scope of this paper.

In [5] a method is presented for analyzing a memory allocator by interpreting memory segments as both raw buffers and structured data. However, their method presents a limited way of treating sizes of chunks of memory since they are limited to contiguous chunks of memory and cannot handle sizes of recursive data structures.

In [15], a specialized canonical abstraction was applied to analyze properties of arrays. Arrays are partitioned into the parts before, at, and after a given index. This gives a way to track sizes of specific partitions. However, it does so only in the special case of arrays. Furthermore, it cannot track sizes of partitions other than the ones formed by index variables. Specifically, their method would not be able to handle examples such as StringBuffer remove.

Reducing Pointer to Integer Programs

In [13, 3, 23] it was proposed to conduct pointer analysis in a prepass and then to convert the program into an integer program to allow integer analysis to check the desired properties. This "reduction based approach" allows using different integer analyzers on the resulting program. Furthermore, for proving simple properties of singly-linked lists it was shown in [3], that there is no loss of precision. However, it may lose precision in cases where the heap and numerics interact in complicated ways. Also, the reduction may be too expensive. Our transformers avoid these issues by iterating between the two abstractions and allowing information flow in both directions. Furthermore, our framework allows for an arbitrary set domain (it 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 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.

In [21] an algorithm for combining Boolean algebra and quantifier free Presburger arithmetic is presented. Their approach presents a complete decision procedure for their specific combined domain. In contrast, our method supports set domains that go beyond Boolean algebra formulas and can thus express more complicated invariants. More significantly our approach provides an effective method for computing transformers for performing abstract interpretation, which their method does not. Fortunately, by careful design of the interface between the abstract domains, we avoid solving the complex constraints which their algorithm handles.

In [28] a logic based approach that involves providing an entailment procedure is presented. Their logic allows for user-defined well-founded inductive predicates for expressing shape and size properties of data-structures. They can express invariants that involve other numeric properties of data structures such as height of trees. However, their approach is limited to separation logic while ours can be used in a more general context. In addition their approach does not infer invariants, requiring a heavy annotation burden, while our approach is based on abstract interpretation and can thus infer loop and recursive invariants.

Acknowledgements

We would like to thank Nurit Dor, Denis Gopan, and Michal Segalov for reading an earlier draft of this paper. We thank Denis Gopan for his help with the implementation. Special thanks to Bruno Blanchet and the anonymous reviewers for their insightful comments and pointing of some errors and fixes in an earlier draft of this paper.

References

[1] Annotated outline of collections framework. Sun Microsystems. Available at http://java.sun.com/j2se/1.5.0/docs/guide/collections/reference.html.

[2] R. Bagnara, P. M. Hill, and E. Zaffanella. The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Science of Computer Programming*, 2008. To appear.

[3] A. Bouajjani, M. Bozga, P. Habermehl, R. Iosif, P. Moro, and T. Vojnar. Programs with lists are counter automata. In *CAV*, pages 517–531, 2006.

[4] A. R. Bradley, Z. Manna, and H. B. Sipma. The polyranking principle. In *ICALP*, pages 1349–1361, 2005.

[5] C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang. Beyond reachability: Shape abstraction in the presence of pointer arithmetic. In *SAS*, pages 182–203, 2006.

[6] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.

[7] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *POPL*, pages 269–282, 1979.

[8] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL*, pages 84–96, 1978.

[9] A. Deutsch. On determining lifetime and aliasing of dynamically allocated data in higher-order functional specifications. In *POPL*, pages 157–168, 1990.

[10] A. Deutsch. *Operational Models of Programming Languages and Representations of Relations on Regular Languages with Application to the Static Determination of Dynamic Aliasing Properties of Data*. PhD thesis, LIX, The Comp. Sci. Lab of École Polytechnique, 1992.

[11] A. Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *PLDI*, pages 230–241, 1994.

[12] D. Distefano, P. W. O'Hearn, and H. Yang. A local shape analysis based on separation logic. In *TACAS*, pages 287–302, 2006.

[13] 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.

[14] D. Gopan, F. DiMaio, N. Dor, T. W. Reps, and M. Sagiv. Numeric domains with summarized dimensions. In *TACAS*, pages 512–529, 2004.

[15] D. Gopan, T. Reps, and M. Sagiv. A framework for numeric analysis of array operations. In *POPL*, pages 338–350, 2005.

[16] S. Gulwani, T. Lev-Ami, and M. Sagiv. A combination framework for tracking partition sizes. Technical Report MSR-TR-2008-158, Microsoft Research, Oct. 2008.

[17] S. Gulwani and A. Tiwari. Combining abstract interpreters. In *PLDI*, pages 376–386, 2006.

[18] M. Hofmann and S. Jost. Static prediction of heap space usage for first-order functional programs. In *POPL*, pages 185–197, 2003.

[19] J. Hughes and L. Pareto. Recursion and dynamic data-structures in bounded space: Towards embedded ML programming. In *ICFP*, pages 70–81, 1999.

[20] M. Karr. Affine relationships among variables of a program. *Acta Inf.*, 6:133–151, 1976.

[21] V. Kuncak and M. C. Rinard. Towards efficient satisfiability checking for boolean algebra with presburger arithmetic. In *CADE*, pages 215–230, 2007.

[22] T. Lev-Ami and M. Sagiv. TVLA: A system for implementing static analyses. In *SAS*, pages 280–301, 2000.

[23] S. Magill, J. Berdine, E. M. Clarke, and B. Cook. Arithmetic strengthening for shape analysis. In *SAS*, pages 419–436, 2007.

[24] R. Manevich, M. Sagiv, G. Ramalingam, and J. Field. Partially disjunctive heap abstraction. In *SAS*, pages 265–279, 2004.

[25] R. Manevich, E. Yahav, G. Ramalingam, and M. Sagiv. Predicate abstraction and canonical abstraction for singly-linked lists. In *VMCAI*, pages 181–198, 2005.

[26] A. Miné. The octagon abstract domain. In *WCRE*, pages 310–319, 2001.

[27] G. Nelson and D. Oppen. Fast decision procedures based on congruence closure. *JACM*, 27(2):356–364, Apr. 1980.

[28] H. 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.

[29] A. Podelski and A. Rybalchenko. Transition invariants. In *LICS*, pages 32–41, 2004.

[30] A. Podelski and T. Wies. Boolean heaps. In *SAS*, pages 268–283, 2005.

[31] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74, 2002.

[32] R. Rugina. Quantitative shape analysis. In *SAS*, pages 228–245, 2004.

[33] M. Sagiv, T. W. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM TOPLAS*, 24(3):217–298, 2002.

[34] A. Turing. Checking a large routine. In *The early British computer conferences*, pages 70–72. MIT Press, Cambridge, MA, USA, 1989.

[35] T. Yavuz-Kahveci and T. Bultan. Automated verification of concurrent linked lists with counters. In *SAS*, pages 69–84, 2002.