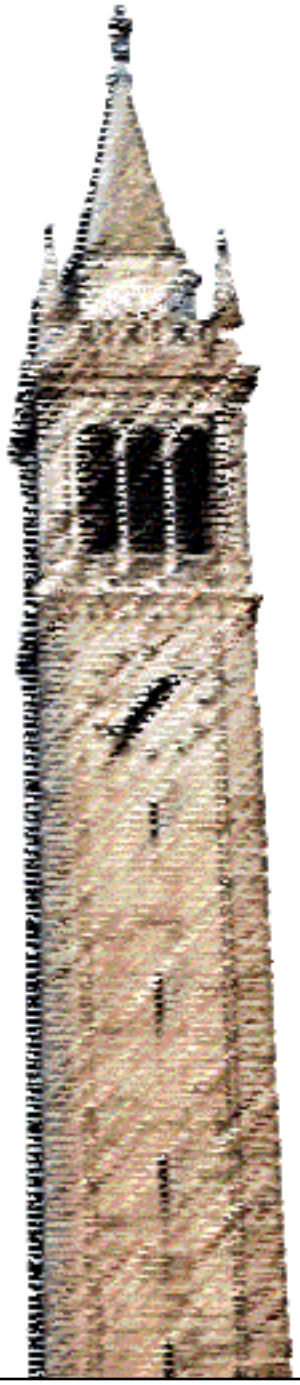
Combining Quantified Domains



Bill McCloskey
Mooly Sagiv

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2009-106
http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-106.html

July 30, 2009

Combining Quantified Domains (Full Version)

Bill McCloskey
U.C. Berkeley
billm@cs.berkeley.edu

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

Abstract

We develop general algorithms for reasoning about numerical properties of programs manipulating the heap via pointers. We automatically infer quantified invariants regarding unbounded sets of memory locations and unbounded numeric values. As an example, we can infer that for every node in a data structure, the node's length field is less than its capacity field. We can also infer per-node statements about cardinality, such as that each node's count field is equal to the number of elements reachable from it. This additional power allows us to prove properties about reference counted data structures and B-trees that were previously unattainable. Besides the ability to verify more programs, we believe that our work sheds new light on the interaction between heap and numerical reasoning.

Our algorithms are parametric in the heap and the numeric abstractions. They permit heap and numerical abstractions to be combined into a single abstraction while maintaining correlations between these abstractions. In certain combinations not involving cardinality, we prove that our combination technique is complete, which is surprising in the presence of quantification.

1. Introduction

This paper presents a general mechanism for combining together two abstract domains that may use quantification. The main application of our technique is in verifying heap-based programs where integer reasoning is required to prove invariants. Our combination technique permits arbitrary predicates to be shared between the domains as well as arbitrary forms of quantification between domains. Besides presenting a useful program analysis technique, we believe that our work helps to explain how integer and heap properties interact in program verification.

Heap and integer reasoning are usually treated separately. However, many program invariants require simultaneous reasoning about the two. We can divide these invariants into a number of classes, depending on the complexity of reasoning involved.

1. Some invariants simply state integer properties that apply to a class of heap objects. For example, if there are two integer fields a and b, we can relate them, as in $\forall o.\ o.a \leq o.b$. Note that this is a mixed formula: the quantifier quantifies over heap objects (really locations), but the $\leq$ predicate is over numbers.
2. It may also be necessary to quantify over both heap objects and integers in a single predicate, particularly when dealing with object fields that are arrays. For example, we might state that for every internal node in a complete n-ary tree, every element of the array of child pointers is non-null.
3. Other invariants require cardinality reasoning. For example, we may want to record the fact that a global variable holds the number of elements in a list.
4. Cardinality reasoning may also be applied per-node. For example, an n-ary tree may store its children in a linked list, while keeping the number of children in a *count* field. This is a more complex form of cardinality reasoning, since there is a different set of objects being counted for each tree node, rather than a single global set as above.

This space of invariants is clearly quite complex. Previous program analysis algorithms have tackled some of these classes of invariants, but we are unaware of a single technique that handles them all. For example, the methods in [11, 10, 13] handle some invariants of class 1, although not all. A recent paper on cardinality reasoning [12] handles invariants from class 3, but not any of the other classes.

In this paper we propose a domain construction that handles all of these classes. Handling them all is important because, surprisingly, some standard data structures require all these forms. For example, the data structure invariants for B-trees require invariants of type 1 to relate the number of keys in each tree node with the number of children. Also, invariants of type 2 allow statements to be made about each child pointer and invariants of types 3 and 4 permit statements about the tree being balanced.

Besides obtaining a more expressive analysis, handling a richer set of invariants gives us a deeper understanding of the interaction between heap and numerical reasoning. We found that some aspects of the interaction, like quantification and predicate sharing, are *fundamental*, while other aspects can be delegated to either the heap or the numerical domain.

When combining two nontrivial reasoning procedures, there is always information that can be shared between the domains to improve the precision of both of them. Equality sharing in Nelson Oppen-style cooperating decision procedures is a typical example [19]. In deciding what information to share between our domains, we asked two questions: (i) what are the "objects" under consideration, and (ii) what properties are guaranteed about those objects? (We use the word "objects" in the most abstract possible sense.) In the case of Nelson-Oppen, the objects are variables and the properties are equalities. For our combined domain, we chose a different point in the space.

Objects Each domain controls a fixed universe of individuals: heap objects for the heap domain and numbers for the numerical domain. We could simply say that these are the "objects," but we want domains to be able to make statements about possibly unbounded sets of individuals. Thus, we require each domain to divide its individuals into a set of named *classes*. For the

numerical domain, a class might contain the integers between 5 and 10. The heap domain might have a class of all objects reachable from a program variable.

At any time, an abstract element must have a finite number of classes whose union contains all the individuals. However, the set of classes may change over time as the individuals are modified.

Classes from one domain are exposed to the other domain. Information about the size of a class and how it intersects other classes is also exposed if available. The other domain treats these class as nothing more than names, since it is unaware of which individuals are contained within them.

Properties Domains also expose a set of predicates. For example, a numerical domain might define a predicate $\mathsf{P}(n) \stackrel{\text{def}}{=} \mathit{offset}(n) < \mathit{length}(n)$. This definition is internal to the numerical domain. However, the numerical domain does shares information about *where* P holds using classes. If the heap domain exposes a class C, then the numerical domain could expose the fact that $\forall n \in C.\ \mathsf{P}(n)$. Even though predicates are "opaque" to the other domain, they can still be useful. A domain may define a predicate in terms of the other domain's predicates.

To ensure termination, the set of predicates a domain exposes must be finite and fixed ahead of time. Typically we expect that the user will determine the set of predicates that are shared based on the properties to be verified.

Thus, the interface between the domains is defined in terms of classes and predicates. The domains share information about which predicates hold over which classes. One benefit of this interface is that two very useful features come "for free." That is, although they must be implemented by the base domains, no additional complexity is required in the combined domain to support them.

Quantification Each domain exposes sufficient information about its classes that the other can quantify over its individuals. Let the two domains be called D_1 and D_2. Assume that an abstract element of D_1 has classes named C and C', both of which satisfy the unary predicate P over all their individuals. Then D_2 can infer that the property $\forall x : D_1.\ \mathsf{P}(x)$ holds without ever consulting D_1. (The notation means that x quantifies over all of D_1's individuals.) Similarly, since domains expose size information, D_2 can check an existential quantifier by finding a non-empty class where P holds.

Cardinality Since rudimentary size information is shared about classes, a numerical domain can inductively track relationships between local variables and the sizes of a heap domain's classes. For example, it might preserve the invariant that integer variable n is equal to the size of all the classes that are reachable from a given pointer variable. Reachability itself is tracked by a shared heap domain predicate. The numerical domain simply needs to record the fact that as each new node becomes reachable (or unreachable), n is incremented (or decremented) by one.

The combination of predicates, classes, inter-domain quantification, and cardinality reasoning allow some very powerful facts to be proved. Consider an example where the heap domain exposes a shared binary predicate E that goes from a parent node in a tree to all the children. The numerical domain can use this predicate to count the number of children a parent has. It may realize that the number of children is equal to the parent node's *nchild* field, and it may expose this equality via a shared predicate C. The heap domain may then determine that all tree nodes reachable from the root satisfy C, meaning that every node has the correct number of children. We know of no other domain that can reason in this way.

The remainder of the paper describes how predicates and classes are shared, and how the shared information is propagated through operations like join and assignment. It should be fairly clear that the amount of sharing (that is, the expressiveness of the language used to share facts between domains) determines the precision of the combined domain. We describe a form of sharing that is fairly simple and easy for the subdomains to implement. Despite this, we prove that our combination is complete in some common circumstances as long as cardinality reasoning is excluded.

In summary, this paper presents two contributions. First, we describe algorithms for combining abstract domains for heap objects and for integer reasoning into a single domain. This domain proves properties about examples like B-trees, skip lists [21], and reference counting implementations, that previous abstract interpreters could not handle. Second, besides proving that our domain is sound, we prove it is complete in some useful cases.

The rest of this paper is organized as follows. Sect. 2 provides an overview of our method using an example of analyzing B-tree implementations. Sect. 3 discusses example heap and numeric domains that can be combined with our construction. Sect. 4 shows the result of applying the analysis to proving correctness of code that uses reference counting for garbage collection. The basic algorithms for combining abstractions are defined and explained in Sect. 5. The semantics of our domain combination is defined in Sect. 6. This semantics is used in Sect. 7 which provides algorithms for computing transfer functions for executing atomic statements on the combined domain and proves soundness. In Sect. 8 we prove that our method is complete under certain requirements. Sect. 9 and Sect. 10 include related work and preliminary conclusions.

2. Overview

We take a logical view of programs. We assume that all memory locations can be addressed using uninterpreted functions. Variables themselves are simply nullary functions. Pointer fields are written as unary functions; for example, `x.f` is written $f(x)$. Array fields are binary functions, where the first argument is the object and the second argument is the index. This, `o.a[i]` becomes $a(o, i)$. Programs consist of accesses and assignments to these functions, as well as the normal control-flow structures. The following is an example program that recursively searches through a B-tree for a given key.

```
search(n, k):
  i := 0
  while i < numchild(n) and k > key(n, i):
    i := i+1
  if i < numchild(n) and k == key(n, i):
    return (n, i)
  if isleaf(n): return null
  else: return search(child(n, i), k)
```

The first argument, a node, is assumed to have scalar fields `numchild` and `isleaf` and array fields `key` and `child`. Note that although `isleaf` is a boolean variable, we treat it as a number (as is done in the C language) so we can use the numerical domain to reason about it.

Be aware that uninterpreted functions are used *only* to describe memory locations. Operations like + are interpreted in the standard way; + is handled by the numerical domain.

As stated in the introduction, each domain exposes a set of predicates. The definitions of these predicates use uninterpreted functions to constrain program locations. Each domain may also use its own symbols in the definition of its predicates, like $+$ and $\leq$ in the numerical domain. There are very few restrictions on what a predi-

	Formula	Intended Meaning
$\mathsf{BT}_1(n{:}\mathbb{H})$	$\mathit{isleaf}(n) \neq 0 \Rightarrow \mathit{numchild}(n) = 0$	Leaves have no children
$\mathsf{BT}_2(n{:}\mathbb{H})$	$\mathit{isleaf}(n) = 0 \Rightarrow 0 \leq M/2 \leq \mathit{numchild}(n) \leq M$	Internal nodes have right number of children
$\mathsf{BT}_3()$	$\mathit{numchild}(r) \neq 0 \Rightarrow \mathit{numchild}(r) \geq 2$	The root has at least two children if it is not a leaf
$\mathsf{BT}_4(n{:}\mathbb{H})$	$\forall i{:}\mathbb{Z}.\ 0 \leq i < \mathit{numchild}(n) \Rightarrow \mathit{child}(n,i) \neq \texttt{nil}$	Children are non-null
$\mathsf{BT}_5(n{:}\mathbb{H})$	$\forall n'{:}\mathbb{H}.\ \forall i,j{:}\mathbb{Z}.\ 0 \leq i < \mathit{numchild}(n) \wedge 0 \leq j < \mathit{numchild}(n') \wedge \mathit{child}(n,i) = \mathit{child}(n',j) \Rightarrow n = n' \wedge i = j$	No sharing
$\mathsf{E}(n_1{:}\mathbb{H}, n_2{:}\mathbb{H})$	$\exists i{:}\mathbb{Z}.\ \mathit{child}(n_1,i) = n_2$	Edge predicate (not an invariant)
$\mathsf{BT}_6(n{:}\mathbb{H})$	$\mathsf{RTC}(\mathsf{E}, r, n)$ (transitive closure of E from r to n)	Every node is reachable from the root
$\mathsf{BT}_7(n{:}\mathbb{H})$	$\forall i{:}\mathbb{Z}.\ 0 \leq i \leq \mathit{numchild}(n) \Rightarrow \mathit{child}(n,i) \neq r$	No incoming pointers to the root

Table 1. Data structure invariants for a B-tree.

cate can express since the domain is responsible for interpreting the predicate. Some sample predicates describing invariants of a B-tree are shown in Table 1. This table assumes that the subdomains are the canonical abstraction [23] and polyhedra [5].

For example, we might like to check is the fact that when `isleaf` is true, `numchild` is zero. We write this predicate as $\mathsf{BT}_1(n{:}\mathbb{H}) \stackrel{\text{def}}{=} \mathit{isleaf}(n) \neq 0 \Rightarrow \mathit{numchild}(n) = 0$. This predicate needs to hold on every node in the data structure.

Recall from the introduction that each domain controls a set of individuals. We write the individuals of the numerical domain as $\mathbb{Z}$ and those of the heap domain as $\mathbb{H}$. In general, for subdomains D_1 and D_2 we call the individuals A_1 and A_2.

Notice that each of the predicates in Table 1 has its parameters annotated with $\mathbb{H}$. This is because the parameters of these predicates are heap nodes. In general, it is possible for predicates to have parameters that range over the individuals of either domain. Thus, as in a many-sorted logic, we can give each predicate a signature, such as $\mathbb{H} \times \mathbb{Z}$.

If its defining subdomain supports it, a predicate may quantify over individuals of either domain. Thus, we annotate the variables in each quantifier with a sort.

Predicate BT_6 may require some additional explanation. The predicate E holds between n_1 and n_2 when n_2 is a child of n_1 (that is, when it appears in n_1's array of children). BT_6 uses a reflexive transitive closure operation, which is supported by the canonical abstraction domain [23]. BT_6 holds of every heap node that is reachable from the root via a series of child edges. As long as this predicate holds of every heap node, we can guarantee that there are no memory leaks.

Like predicates, functions also have a signature. For example, the *child* function maps a heap node an an integer to another heap node, so its signature is $\mathbb{H} \times \mathbb{Z} \rightarrow \mathbb{H}$. Unlike predicates, functions are hidden within a particular domain. This domain can use the function within its shared predicate definitions, but the other domain cannot directly reference the function in any way. Note that, like predicates, the signature of the function has nothing to do with which domain controls it.

If we need to define a predicate in one domain that uses the other domain's functions, there is usually a solution. Since predicates from different subdomains may refer to each other, it is possible to *normalize* a given predicate definition into one that uses only functions from its own domain. As an example, consider predicate BT_7 from Table 1. Assume that *numchild* belongs to the numerical domain while *child* and *r* belong to the heap domain. We can create an auxiliary predicate $\mathsf{Aux}(n{:}\mathbb{H}, i{:}\mathbb{Z}) \stackrel{\text{def}}{=} 0 \leq i < \mathit{numchild}(n)$. Then we can rewrite $\mathsf{BT}'_7(n{:}\mathbb{H}) \stackrel{\text{def}}{=} \forall i{:}\mathbb{Z}.\ \mathsf{Aux}(n,i) \Rightarrow \mathit{child}(n,i) \neq r$. Now both predicates are normalized: Aux only uses functions from the numerical domain (*numchild*) and $\mathsf{BT}'_7(n)$ only uses symbols from the heap domain (*child* and *r*). Even though it is an integer, i does not affect normalization because it is a quantified variable and not a function symbol.

3. Example Domains

In this section we describe some useful subdomains. We use them as examples throughout the paper.

Canonical abstraction. The canonical abstraction domain implemented in TVLA [23] fits well into our framework because it already supports quantification and predicates. The classes of a TVLA domain element are the abstract heap nodes, which are maintained by the canonical abstraction.

TVLA classes are always disjoint and their cardinality is always ≥ 1. Non-summary nodes have cardinality exactly 1.

Although TVLA uses three-valued logic, we can adapt it to a two-valued setting by having both a positive and a negative version of every predicate. For example, if $\mathsf{P} = 0$, then we say that $\neg\mathsf{P}$ holds.

Support for functions is not built into TVLA; it normally handles only predicates. However, we can construct predicates corresponding to each function and keep the two synchronized.

TVLA is not designed to handle quantification over other sorts, such as integers. However, we can treat classes from the other domain as additional TVLA nodes. We use two special TVLA unary predicates, TypeH and TypeZ, to differentiate between the two kinds of nodes. TypeH applies to all normal TVLA nodes and TypeZ applies to nodes representing the other domain's classes. Then we translate quantifiers in formulas as follows. A formula $\forall n : \mathbb{H}.\ \phi$ becomes $\forall n.\ \mathsf{TypeH}(n) \Rightarrow \phi$. The formula $\forall n : \mathbb{Z}.\ \phi$ is transformed to $\forall n.\ \mathsf{TypeZ}(n) \Rightarrow \phi$. Existentials are converted similarly, so $\exists n{:}\mathbb{H}.\ \phi$ becomes $\exists n.\ \mathsf{TypeH}(n) \wedge \phi$.

Separation logic domain. A separation logic domain breaks the heap into pieces. Each piece is typically a list segment, written $ls(x, y)$ for a segment from x to y, or a single list node, written $[x \mapsto y]$. Additional facts may track equalities or disequalities between variables. We place each heap component (list segment or node) into a class. These classes are necessarily disjoint, and like in canonical abstraction, some information about their cardinality is available.

We track nullary functions as variables in the separation domain. Since arrays are not supported, there is no need to consider non-nullary functions.

The theoretical framework of separation logic permits arbitrary predicates. Typically only a fixed set are supported (primarily ls for linked lists). However, it would be possible to generate new predicates, such as $ls(x, y, \mathsf{P})$, which would denote a list segment where all elements satisfy a property P. If P is a shared predicate, then numerical statements could be made about heap elements.

Polyhedra. Simple support for numerical reasoning is possible without any modifications to the standard polyhedra domain. The

domain simply ignores all information from the other domain. It is only able to reason about program variables, since reasoning about fields of objects would require it to understand the heap domain's classes.

However, it is possible to expose numerical predicates to the heap domain. These predicates will only be over program variables, and will thus be nullary predicates. However, they do provide a small measure of interaction between the domains.

Polyhedra with quantification. Reasoning about the fields of objects requires extending the polyhedra domain to understand classes from the heap domain. The challenge is to take a domain that understand facts like $x < 10$ and transform it into a domain that understands $\forall x \in C.\ x < 10$.

Summarizing numerical domains offer a solution to this problem [11]. They treat each class as its own variable, so that the fact above becomes $C < 10$. This approach permits domains like polyhedra to reason about universal quantifiers almost unchanged (two additional operations are required, but they are already provided by polyhedra implementations like Parma [1]).

Support for existential quantifiers is not available, but it can be simulated by taking advantage of the fact that an existential quantifier over C is equivalent to a universal quantifier when $|C| = 1$. Support for empty classes also requires some additional effort.

Polyhedra with cardinality reasoning. Any kind of cardinality reasoning requires the polyhedra domain to track a cardinality variable for each heap class. This variable holds the cardinality of the class. When a new class comes into existence, it typically has cardinality 1. Each time one class is merged with another, we treat the new class's cardinality as the sum of the merged cardinalities. This allows us to track relationships between local integer variables and class cardinalities as data structures are constructed and maintained.

Cardinality variables allow us to compute quantities like $|\{n : \mathsf{P}(n)\}|$. We assume that the predicate P is shared between the heap and polyhedra domains. Thus, the other domain will inform polyhedra of the classes over which P holds. This information can be stored separately from normal polyhedra information, since it is uninterpreted. To evaluate the cardinality, we sum up the sizes of the classes where P holds using their cardinality variables. We must take into account the fact that these classes may overlap.

4. Reference Counting Examples

In this section we use the combination of the canonical abstraction domain and a polyhedra domain augmented with cardinality reasoning to check the correctness of code that uses reference counting for garbage collection. Reference counting is widespread both in the implementation of interpreted languages like Perl and Python as well as in systems code, where it is sometimes managed with wrappers like C++'s `shared_ptr`.

In a basic reference counting system, every object has a *refcount* field. When an object is allocated, this field is initialized to 1. Each time a new reference is created to the object, the field is incremented (an `incref`). When a reference is destroyed (by assigning over it, say), the reference count is decremented (a `decref`). If the count reaches 0, the object is deallocated.

Efficient reference counting systems such as the ones supported in Python interpreters minimize the number of `incref` and `decref` operations by classifying some references as being "owners" and others as "borrowers." Only an owning reference causes the count to increment or decrement. For simplicity, we ignore this aspect, although it can be supported in our system.

Each object in a reference counted system must describe which of its fields should be reference counted. In our example, we assume that each object has a single field *ptr* that is counted. We will analyze two pieces of code, both shown in Figure 1. The first function, `incref`, takes pointers x and y to objects. It assigns y into the *ptr* field of x (which is assumed to be null before the assignment) and increments the reference count of y.

```
  incref(x, y):  // ptr(x) == null
1   ptr(x) := y
2   refcount(y) := refcount(y) + 1

  decref(x):  // ptr(x) != null
1   y := ptr(x)
2   ptr(x) := null
3   if refcount(y) == 1:
4     free(y)
5   else:
6     refcount(y) := refcount(y) - 1
```

Figure 1. Reference counting example.

The second function, `decref`, is responsible for eliminating a reference through the *ptr* field of its argument. Let y be the object originally pointed to, which should not be null. If x is the only pointer to y, then y is freed. Otherwise, the reference is eliminated and y's count is decremented.

It is assumed that `incref` and `decref` will be used throughout a system whenever a reference is created or destroyed in the heap. To track references from local variables, we can treat each local variable as a heap object, much as one has to do in C anyway due to the address-of (`&`) operator. One can think of the variables x and y above as mere temporaries.

Our goal is to check that once an object is freed, it will never be referenced again. We do this in two steps. First, we prove that each `incref` operation preserves the invariant that the number of heap references to an object via the *ptr* field is equal to its reference count. Second, we use this invariant to prove that when an object is freed, there are no heap references to it (so it will never be used again). We summarize these two steps as "verifying `incref`" and "verifying `decref`."

4.1 Verifying `incref`

The invariant that we are interested in tracking can be written as follows.

$$\mathsf{RC}(n{:}\mathbb{H}) \stackrel{\text{def}}{=} \mathit{refcount}(n) = |\{n'{:}\mathbb{H} \mid \mathit{ptr}(n') = n\}|$$

That is, the reference count of an object is equal to the number of objects whose *ptr* field points to it. To state this invariant in the language of the previous section, we will assume that the heap domain tracks the *ptr*, x, and y functions while the integer domain tracks *refcount*. Additionally, we will assume that the heap domain exposes a predicate $\mathsf{Ptr}(n, n') \stackrel{\text{def}}{=} \mathit{ptr}(n) = n'$ that is accessible from the integer domain. Then, assuming that the integer domain has been instrumented for cardinality reasoning (as will be described later), we can define the predicate RC in the integer domain in normal form as follows.

$$\mathsf{RC}(n{:}\mathbb{H}) \stackrel{\text{def}}{=} \mathit{refcount}(n) = |\{n'{:}\mathbb{H} \mid \mathsf{Ptr}(n', n)\}|$$

We assume that when `incref` begins, the *ptr* field of x is null and that y points to an object satisfying RC. In order to allow both domains to make statements about the objects pointed to by x and y, we assume the heap domain exposes singleton classes C_x and C_y containing only these objects. We write the invariants stored in the heap and numeric domains separately. Thus, the invariants at

line 1 are as follows.

$$I_1^H : C_x = \{x\} \wedge C_y = \{y\} \wedge \mathit{ptr}(x) = \texttt{nil} \wedge \neg\mathsf{Ptr}(x,y) \wedge \mathsf{RC}(y)$$
$$I_1^Z : \forall y' \in C_y.\ \mathit{refcount}(y') = |\{n'{:}\mathbb{H} \mid \mathsf{Ptr}(n',y')\}| \\ \wedge\ \forall x' \in C_x, y' \in C_y.\ \neg\mathsf{Ptr}(x',y')$$

Note that the numerical domain is aware that the predicate Ptr is false between C_x and C_y since that predicate is shared.

Executing the statement at line 1 happens in several steps. First, the heap domain is updated to reflect the new value of $\mathit{ptr}(x)$. Besides the direct fact about $\mathit{ptr}(x)$, the heap domain also recognizes that $\mathsf{Ptr}(x,y)$ now holds.

$$I^H : C_x = \{x\} \wedge C_y = \{y\} \wedge \mathit{ptr}(x) = y \wedge \mathsf{Ptr}(x,y) \wedge \mathsf{RC}(y)$$

Since the Ptr predicate is shared between the heap and the numerical domain, the numerical domain needs to be informed. Sharing predicate values between the two domains is an important aspect of the combined domain. The heap domain tells the numerical domain that $\forall x' \in C_x, y' \in C_y.\ \mathsf{Ptr}(x',y')$ has gone from false to true. The numerical domain then realizes that the value of $|\{n'{:}\mathbb{H} \mid \mathsf{Ptr}(n',y')\}|$ has now increased by 1, the size of C_x. Thus, the numerical portion is updated as follows.

$$I^Z : \forall y' \in C_y.\ \mathit{refcount}(y') = |\{n'{:}\mathbb{H} \mid \mathsf{Ptr}(n',y')\}| - 1 \\ \wedge\ \forall x' \in C_x, y' \in C_y.\ \mathsf{Ptr}(x',y')$$

Finally, the numerical domain must inform the heap domain that $\forall y' \in C_y.\ \mathsf{RC}(y')$ no longer holds. Thus, we get the following invariants at the beginning of statement 2.

$$I_2^H : C_x = \{x\} \wedge C_y = \{y\} \wedge \mathit{ptr}(x) = y \wedge \mathsf{Ptr}(x,y)$$
$$I_2^Z : \forall y' \in C_y.\ \mathit{refcount}(y') = |\{n'{:}\mathbb{H} \mid \mathsf{Ptr}(n',y')\}| - 1 \\ \wedge\ \forall x' \in C_x, y' \in C_y.\ \mathsf{Ptr}(x',y')$$

The assignment operation in statement 2 is handled by the numerical domain. It gives us this new invariant.

$$I^Z : \forall y' \in C_y.\ \mathit{refcount}(y') = |\{n'{:}\mathbb{H} \mid \mathsf{Ptr}(n',y')\}| \\ \wedge\ \forall x' \in C_x, y' \in C_y.\ \mathsf{Ptr}(x',y')$$

This, of course, contains the fact $\forall y' \in C_y.\ \mathsf{RC}(C_y)$. Since RC is a shared predicate, we transmit this information to the heap domain, which is updated to give the following final invariants.

$$I_3^H : C_x = \{x\} \wedge C_y = \{y\} \wedge \mathit{ptr}(x) = y \wedge \mathsf{Ptr}(x,y) \wedge \mathsf{RC}(y)$$
$$I_3^N : \forall y' \in C_y.\ \mathit{refcount}(y') = |\{n'{:}\mathbb{H} \mid \mathsf{Ptr}(n',y')\}| \\ \wedge\ \forall x' \in C_x, y' \in C_y.\ \mathsf{Ptr}(x',y')$$

We have proved that the RC invariant is maintained by `incref`, which was our goal.

4.2 Verifying `decref`

In this function our goal is to ensure that if an object is freed it is never accessed again. Assuming that all accesses happen through *ptr* fields of other objects, we need only ensure that no other objects can point to the one being freed. Thus, when verifying `decref`, we track "ghost objects" that abstract all other objects that might be in the system. We will prove that the ghost objects do not point to the object being freed.

Consider the invariant before statement 3 executes. We assume that the heap invariant maintains three classes. Two classes, C_x and C_y, are the same as before. The other class, C_g, holds all the ghost objects (it may have arbitrary cardinality). We assume that the object pointed to by y satisfies RC.

```
Saturate(E1, E2):
  F := empty
  repeat:
    F0 := F
    F := F union Consequences1(E1) union Consequences2(E2)
    E1 := Assume1(E1, F)
    E2 := Assume2(E2, F)
  until F0 = F
  return (E1, E2)
```

Figure 2. Saturation algorithm.

$$I_3^H : C_x = \{x\} \wedge C_y = \{y\} \wedge \mathit{ptr}(x) = y \wedge \mathsf{Ptr}(x,y) \wedge \mathsf{RC}(y)$$
$$I_3^Z : \forall y' \in C_y.\ \mathit{refcount}(y') = |\{n'{:}\mathbb{H} \mid \mathsf{Ptr}(n',y')\}| \\ \wedge\ \forall x' \in C_x, y' \in C_y.\ \mathsf{Ptr}(x',y')$$

Executing statement 3 allows us to assume that $\mathit{refcount}(y) = 1$. This fact is added to the numerical domain.

$$I^Z : \forall y' \in C_y.\ \mathit{refcount}(y') = 1 \wedge |\{n'{:}\mathbb{H} \mid \mathsf{Ptr}(n',y')\}| = 1 \\ \wedge\ \forall x' \in C_x, y' \in C_y.\ \mathsf{Ptr}(x',y')$$

Since Ptr is a shared predicate, the numerical domain knows where it holds. In particular, it knows that $\forall x' \in C_x, y' \in C_y.\ \mathsf{Ptr}(x',y')$. Since $|C_x| = 1$, this accounts for one node in the set of nodes with Ptr edges into C_y. The numerical domain knows that the total number should be one, meaning that no other classes can have edges into C_y. Thus, the numerical domain infers that $\forall x' \in C_x, g' \in C_g.\ \neg\mathsf{Ptr}(x',g')$. It passes this fact to the heap domain. Thus, the combined domain recognizes that none of the ghost objects can possibly point to y, the object to be freed.

5. Basic Operations

Much of the reasoning power of the combined domain is in the base domains. However, the combined domain implements two features of its own: shared predicates and classes. In this section we describe how these features are implemented.

5.1 Saturation

The utility of shared predicates is made manifest only when they are shared. We call the sharing process *saturation*. Saturation is called semantic reduction in [4] as it allows the analysis to convert an abstract element into a more precise one as long as they both represent the same set of states. For example, one domain may discover that a shared predicate is true, which may trigger further inferences in the other domain. The saturation process tries to trigger as many inferences as possible until a fixed point is reached.

The predicate information passed between domains must be expressed in a language that both domains understand. We choose a relatively simple language. A fact is a set of quantifiers followed by a single predicate. The quantifiers can be $\forall$ or $\exists$ and they quantify over all the individuals in a given class. Arguments to the predicate must be quantified variables; no function applications are allowed. The predicate is either a predicate from one domain or the other, or one of the built-in predicates: $=$, or $\neq$. An example fact might be $\forall x \in C_1.\ \exists y \in C_2.\ \mathsf{P}(y,x)$.

The built-in predicates $=$ and $\neq$ are interpreted by both domains. We use them to express cardinality information about classes. To express $C = \emptyset$, we write $\forall n \in C.\ n \neq n$. The negation, $C \neq \emptyset$, is $\exists n \in C.\ n = n$. If we want to say, $|C| \leq 1$, we write $\forall n, n' \in C.\ n = n'$. To say C_1 and C_2 are disjoint, we write $\forall n_1 \in C_1.\ \forall n_2 \in C_2.\ n_1 \neq n_2$. Finally, to say $C_1 \subseteq C_2$, we write $\forall n_1 \in C_1.\ \exists n_2 \in C_2.\ n_1 = n_2$.

```
Repartition((E1, E2), R, F):
  E1 := Repartition1(E1, R, F)
  E2 := Repartition2(E2, R, F)
  return (E1, E2)
```

Figure 3. Repartitioning algorithm.

To permit facts of this form to be exchanged, each domain is required to expose an `Assume` function and a `Consequences` function. `Assume` takes a domain element E and a fact f of the form above and returns an element approximating $E \wedge f$. `Consequences` takes a domain element and returns all the set of facts of the form above that it implies.

The code is shown in Figure 2. The variable F stores a set of facts of the form described. They are accumulated via `Consequences` and then passed to the domains with `Assume`. Since there is a bounded number of predicates and a bounded number of classes in a given element, this process is guaranteed to terminate.

5.2 Repartitioning

Classes are the mechanism by which a domain describes it individuals and the predicates that hold over them to the other domain. They allow a domain to finitely describe properties of an unbounded number of individuals. Class names themselves are arbitrary and ephemeral, like variable names in the lambda calculus. Thus, when two elements are merged together in some way (for a join or meet operation, for example), we need to relate the classes of the two elements.

Consider this example, where D_1 is a separation logic domain and D_2 is a polyhedron domain with quantification. The bracket notation allows us to name the class in which a list segment

$$E_1 : [ls(x, \texttt{nil})]^A \qquad E_2 : \forall n \in A.\ \mathit{val}(n) = 10$$
$$E_1' : [ls(x, \texttt{nil})]^B \qquad E_2' : \forall n \in B.\ \mathit{val}(n) = 10$$

Imagine that we want to join these two elements. The first one describes a list where all the elements fall into class A and the *val* field of each list entry equals 10. The second one describes the same but using a different class name, B.

We break the operation into two steps. First, the separation domain must decide on what classes to use in its new element and how those classes relate to the old classes—we describe this step later. Then, all four subdomain elements must be remapped. The mapping is described by a relation R. Assume for now that the separation domain decides to map A and B to a new class C. Then the relation R would be $\{(A, C)\}$ for the unprimed element and $\{(B, C)\}$ for the primed element.

To actually perform the renaming, each domain exposes a function `Repartition`. Both of the subdomains' elements must be repartitioned, since they both make reference to A and B. When we repartition, we get the following.

$$E_1 : [ls(x, \texttt{nil})]^C \qquad E_2 : \forall n \in C.\ \mathit{val}(n) = 10$$
$$E_1' : [ls(x, \texttt{nil})]^C \qquad E_2' : \forall n \in C.\ \mathit{val}(n) = 10$$

Now it is easier to see how a join might be performed. We describe the join in detail later.

The code for repartitioning is shown in Figure 3. Given a relation R, this function simply calls `Repartition` on both subdomains. The F parameter is explained below. Note that repartition is called only once on each subdomain. This is unlike saturation, which iterates to a fixed point. As a consequence, the subdomains should make their repartitioning decisions independent of the other domain's partitioning.

We describe a way that a subdomain might implement the `Repartition` operation. Imagine that $R = \{(C_1, C), (C_2, C)\}$. Perhaps we know that all the elements of C_1 are equal to 10 and all those of C_2 equal 20. To determine what is true of the new class C, we realize that any individual from C_1 or C_2 may now belong to C. Thus, any property that applied to both C_1 and C_2 is true of C. So we take the least upper bound, meaning that in the repartitioned result, the elements of C are between 10 and 20. More information on how to repartition polyhedra is available in Gopan et al. [11]. Repartitioning heap domains is usually trivial; see, for example, Sagiv et al. [23].

The parameter F is a set of facts about the newly created classes. These facts have the same syntax as the ones in saturation. Typically F will contain information about the cardinality of the new partitions or how they overlap. This can make repartitioning more precise.

Example. Consider an integer domain that tracks the cardinalities of another domain's classes. In this case, it may need information about class overlap. For example, assume that we track the fact that $n = |C|$ and we repartition according to $\{(C, C_1), (C, C_2)\}$, where F tell us that C_1 and C_2 are disjoint. Then the `Repartition` function will rewrite to $n = |C_1| + |C_2|$.

6. Semantics

We assume that semantics of the subdomains are given by two predicates, γ_1 and γ_2. Each of these is invoked as follows. S is a program state.

$$\gamma_i(E_i, S, M, P) \stackrel{\text{def}}{=} \text{“}S \text{ satisfies } E_i \text{ with classes } M\text{, predicates } P\text{”}$$

More formally, S is a triple (A_1, A_2, F). A_1 and A_2 give the set of individuals for each domain. F gives an interpretation to each function. The intention is that $F(f)(a_1, a_2) = a_3$ means $f(a_1, a_2) = a_3$. All functions should be total.

M is a mapping from class names to sets of individuals. P is an interpretation for predicates. Its meaning is that if $P(\mathsf{P})(a_1, a_2)$, then $\mathsf{P}(a_1, a_2)$ holds.

γ_i is responsible for checking that all the statements made by E_i are true in S, M, P and also that all the predicate interpretations in P make sense according to their definition. For brevity, we define $\gamma'(E_1, E_2, S, M, P)$ to mean that both γ_1 and γ_2 hold over S, M, P. Thus, we can define the concretization as follows.

$$\gamma(E_1, E_2) = \{S : \exists M, P.\ \gamma'(E_1, E_2, S, M, P)\}$$

We also define a predicate $\text{MOK}_i(E_i, S, M)$. This predicate ensures that three conditions hold: it checks that every class of E_i is in the domain of M, that if C belongs to D_i then $M(C) \subseteq A_i$, and that $\cup_{C \in D_i} M(C) = A_i$. We assume that γ_i automatically checks that MOK_i holds, so in most cases there is no need to use it. Like with γ', we define MOK' that holds if both MOK_1 and MOK_2 hold.

Throughout, we use the predicate γ_f to interpret facts of the kind that appear in saturation. Assume F is a set of facts, as would be returned by `Consequences`. Let $\gamma_f(F, S, M, P)$ hold if all the facts in F are true according to their interpretation in S, M, P. We can define γ_f as follows. (Note that $[\cdot]_S$, which interprets function applications, is also used throughout the paper.)

$$\gamma_f(F, S, M, P) = \forall f \in F.\ \gamma_f(f, S, M, P)$$
$$\gamma_f((\forall x \in C.\ f), S, M, P) = \forall a \in M(C).\ \gamma_f(f, S, M, P)$$
$$\gamma_f((\exists x \in C.\ f), S, M, P) = \exists a \in M(C).\ \gamma_f(f, S, M, P)$$
$$\gamma_f(\mathsf{P}(e_1, \ldots, e_k), S, M, P) = P(\mathsf{P})([e_1]_S, \ldots, [e_k]_S)$$
$$[x]_S = x$$
$$[f(e_1, \ldots, e_k)]_S = S(\mathsf{f})([e_1]_S, \ldots, [e_k]_S)$$

Using these formulas we can define conditions that each of the subdomains' operations must satisfy. These conditions will in turn allow us to verify the soundness of the combined domain.

Saturation. The conditions for the `Consequences` function are fairly clear.

$$\forall S, M, P.\, \gamma_i(E_i, S, M, P) \Rightarrow \gamma_f(\texttt{Consequences}_i(E_i), S, M, P)$$

That is, if a given state satisfies a domain element, then all the facts returned by `Consequences` must be true in the state.

We can define similar conditions for `Assume`. For any set of facts F, the following must hold.

$$\forall S, M, P.\, \gamma_i(E_i, S, M, P) \wedge \gamma_f(F, S, M, P) \Rightarrow \gamma_i(\texttt{Assume}_i(E_i, F), S, M, P)$$

Using these properties, we can prove the following property of the `Saturate` algorithm in Figure 2. Here, the inputs are (E_1, E_2) and the outputs are (E_1', E_2').

$$\forall S, M, P.\, \gamma'(E_1, E_2, S, M, P) \Rightarrow \gamma'(E_1', E_2', S, M, P)$$

Repartition. The correctness condition for `Repartition` is somewhat unintuitive. We tend to think of classes as having an interpretation, so it might seem reasonable that some mappings R between classes are illegal because they break our intuitive interpretation. In fact, *any* R is legal as long as it does not "lose" individuals. For example, an R that fails to map an original class to any new classes is illegal, since it does not account for all individuals.

We formalize this notion here. Let $\texttt{Repartition}(E_i, R, F) = E_i'$. If a state S, M, P satisfies E_i, then we require that any new partitioning M' that "conforms" to R and F (in a sense to be defined later) must satisfy E_i'. We can write this logically as follows.

$$\forall S, M, P, M'.\, (\gamma_i(E_i, S, M, P) \wedge \text{ROK}_i(E_i, R, S, M, M') \wedge \gamma_f(F, S, M', P)) \Rightarrow \gamma_i(E_i', S, M', P)$$

We can think of R as describing which original classes the individuals of a new class can be drawn from. This motivates the definition of ROK: M' conforms to R if $M'(C)$ contains only individuals that from classes C where $R(C, C')$. We also check MOK, since we want to ensure M' doesn't lose any individuals.

$$\text{ROK}_i(E_i, R, S, M, M') \stackrel{\text{def}}{=} \forall C'.M'(C') \subseteq \bigcup_{C:R(C,C')} M(C) \wedge \text{MOK}_i(E_i, S, M')$$

As before, we assume that ROK' checks ROK_1 and ROK_2.

With these pieces, we can prove the following fact about the combined domain's `Repartition` function.

$$\forall S, M, P, M'.\, (\gamma'(E_1, E_2, S, M, P) \wedge \text{ROK}'(E_1, E_2, R, S, M, M') \wedge \gamma_f(F, S, M', P)) \Rightarrow \gamma'(E_1', E_2', S, M', P)$$

7. Combined Domain

In this section we present the partial order and transfer functions for the combined domain. Generally, the transfer functions are split into two pieces. One pieces performs any repartitioning that may be necessary and the other piece applies the actual transfer function. In practice a combined domain would be used disjunctively, as is done in virtually all shape analyses.

```
Implies((E1, E2), (E1', E2')):
  (E1, E2) := Saturate(E1, E2)

  (R1, F1) := MatchClasses1(E1, E1')
  (R2, F2) := MatchClasses2(E2, E2')

  (E1'', E2'') := Repartition((E1, E2), R1 union R2,
                              F1 union F2)

  return (Implies1(E1'', E1') and Implies2(E2'', E2'))
```

Figure 4. Implies algorithm.

7.1 Implication Pre-Order

The main difficulty in defining a partial order is that the two elements may use different partitionings. Thus, we require each domain to expose a function, `MatchClasses`, which maps the classes of the right-hand element to the classes of the left-hand element. See Figure 4. `MatchClasses` returns R and F that are used to repartition (E_1, E_2). After repartitioning, the sub-domains' `Implies` tests are invoked.

Soundness. The soundness requirements on `MatchClasses` mirror the assumptions needed to apply `Repartition`. They are fairly lax, meaning that `MatchClasses` can choose almost any R as every class maps to something. Let $\texttt{MatchClasses}(E_i, E_i') = (R, F)$. Then we require the following.

$$\forall S, M, P.\, \gamma_i(E_i, S, M, P) \Rightarrow \exists M'.\, \text{ROK}_i(E_i, R, S, M, M') \wedge \gamma_f(F, S, M', P)$$

Note that E_i' does not even appear!

Using this statement, we can prove the soundness of the implies algorithm. First, suppose we are given state S so that

$$\exists M, P.\, \gamma'(E_1, E_2, S, M, P).$$

Then after the saturate step, we continue to have

$$\exists M, P.\, \gamma'(E_1, E_2, S, M, P).$$

After calling `MatchClasses`, we get the following.

$$\exists M, P.\, \gamma'(E_1, E_2, S, M, P) \wedge \exists M'.\, \text{ROK}_1(E_1, R_1, S, M, M') \wedge \gamma_f(F_1, S, M', P) \wedge \exists M'.\, \text{ROK}_2(E_2, R_2, S, M, M') \wedge \gamma_f(F_2, S, M', P)$$

These statements make the call to `Repartition` valid, yielding states that are still valid.

$$\exists M, P.\, \gamma'(E_1'', E_2'', S, M, P)$$

We can state correctness conditions for the subdomains' `Implies` functions as well. We write $\sqsubseteq_i$ for $\texttt{Implies}_i$.

$$\forall S, M, P.\, E_i \sqsubseteq_i E_i' \Rightarrow (\gamma_i(E_i, S, M, P) \Rightarrow \gamma_i(E_i', S, M, P))$$

Thus, if $\texttt{Implies}_1$ and $\texttt{Implies}_2$ return true, then S satisfies (E_1', E_2').

In total, we have proved that if a state satisfies (E_1, E_2) and `Implies` returns true, then the state satisfies (E_1', E_2'). Thus, `Implies` is sound.

Transitivity. To ensure that `Implies` is transitive, we need some restrictions. Assume that A, B, and C are elements of the combined domain. Let $A = (A_1, A_2)$ and the same for B and C. Let $A' = \texttt{Saturate}(A_1, A_2)$ and the same for the others.

To begin, we need to restrict `MatchClasses`. Assume that

$$\texttt{MatchClasses}(A, B) = \mathcal{M}_{AB}$$
$$\texttt{MatchClasses}(B, C) = \mathcal{M}_{BC}$$
$$\texttt{MatchClasses}(A, C) = \mathcal{M}_{AC}.$$

We define the notation that $\mathcal{M}(E) = \texttt{Repartition}(E, R, F)$ when $\mathcal{M} = (R, F)$. We also define the notation that $A \sqsubseteq_* B$ when $A_1 \sqsubseteq_1 B_1$ and $A_2 \sqsubseteq_2 B_2$.

Our first requirement is that for any A, B, and C, the results of `MatchClasses`, when fed to `Repartition`, are transitive.

$$\mathcal{M}_{AC}(A) \sqsubseteq_* \mathcal{M}_{BC}(\mathcal{M}_{AB}(A)).$$

Additionally, we place a monotonicity requirement on repartitioning, that for any $\mathcal{M}$,

$$A \sqsubseteq_* B \Rightarrow \mathcal{M}(A) \sqsubseteq_* \mathcal{M}(B).$$

Another pair of monotonicity properties is also needed. For any E,

$$B \sqsubseteq_* C \Rightarrow \mathcal{M}_{AB}(E) \sqsubseteq_* \mathcal{M}_{AC}(E)$$
$$A \sqsubseteq_* B \Rightarrow \mathcal{M}_{AC}(E) \sqsubseteq_* \mathcal{M}_{BC}(E).$$

We also need saturation to commute with repartitioning. To ensure this, we require,

$$\mathcal{M}(\texttt{Assume}(E, F)) = \texttt{Assume}(\mathcal{M}(E), F).$$

Here we abuse notation so that `Assume` applies $\texttt{Assume}_1$ and $\texttt{Assume}_2$ to a combined domain element. Finally, we need a monotonicity condition on `Assume` and `Consequences`.

$$A_i \sqsubseteq_i B_i \Rightarrow \texttt{Consequences}_i(B_i) \subseteq \texttt{Consequences}_i(A_i)$$
$$A_i \sqsubseteq_i B_i \wedge F \subseteq F' \Rightarrow \texttt{Assume}_i(A_i, F') \sqsubseteq_i \texttt{Assume}_i(B_i, F)$$

We begin by proving a lemma, that if $A' \sqsubseteq_* B$, then $A' \sqsubseteq_* B'$. We first note the following.

$$A'_1 = \texttt{Assume}_1(A'_1, \texttt{Consequences}_2(A'_2))$$

By the monotonicity of `Consequences`, $\texttt{Consequences}_2(B_2) \subseteq \texttt{Consequences}_2(A'_2)$. The by the monotonicity of `Assume`,

$$\texttt{Assume}_1(A'_1, \texttt{Consequences}_2(A'_2)) \sqsubseteq_1 \texttt{Assume}_1(B_1, \texttt{Consequences}_2(B_2)).$$

Therefore,

$$A'_1 \sqsubseteq_1 \texttt{Assume}_1(B_1, \texttt{Consequences}_2(B_2)).$$

We can prove a similar fact about A'_2. Let B^1 be the result of one round of saturating B. Then $A' \sqsubseteq_* B^1$. We can use a similar argument to prove that if $A' \sqsubseteq_* B^i$, then $A' \sqsubseteq_* B^{i+1}$. So by induction, $A' \sqsubseteq_* B'$.

Now we can prove transitivity. We start knowing that $A \sqsubseteq B$ and $B \sqsubseteq C$, which we can rewrite as $\mathcal{M}_{A'B}(A') \sqsubseteq_* B$ and $\mathcal{M}_{B'C}(B') \sqsubseteq_* C$. We know that $B' \sqsubseteq_* B$ by monotonicity of `Assume`. Therefore, by monotonicity of matchings, $\mathcal{M}_{A'B'}(A') \sqsubseteq_* \mathcal{M}_{A'B}(A')$. If we use monotonicity of repartitioning and apply $\mathcal{M}_{B'C}$ to both sides of this, we get

$$\mathcal{M}_{B'C}(\mathcal{M}_{A'B'}(A')) \sqsubseteq_* \mathcal{M}_{B'C}(\mathcal{M}_{A'B}(A')).$$

By transitivity of matchings, we know

$$\mathcal{M}_{A'C}(A') \sqsubseteq_* \mathcal{M}_{B'C}(\mathcal{M}_{A'B'}(A')).$$

Combining this with the last fact (since $\sqsubseteq_*$ is transitive),

$$\mathcal{M}_{A'C}(A') \sqsubseteq_* \mathcal{M}_{B'C}(\mathcal{M}_{A'B}(A')).$$

Now we take the fact $\mathcal{M}_{A'B}(A') \sqsubseteq_* B$ and apply commutativity of saturation and repartitioning to get $\mathcal{M}_{A'B}(A)' \sqsubseteq_* B$. By the lemma we proved above, $\mathcal{M}_{A'B}(A)' \sqsubseteq_* B'$. Applying commutativity again, $\mathcal{M}_{A'B}(A') \sqsubseteq_* B'$. Finally, we apply $\mathcal{M}_{B'C}$ to both sides, getting $\mathcal{M}_{B'C}(\mathcal{M}_{A'B}(A')) \sqsubseteq_* \mathcal{M}_{B'C}(B')$. Then using transitivity of $\sqsubseteq_*$ along with the initial assumption that $\mathcal{M}_{B'C}(B') \sqsubseteq_* C$, we get $\mathcal{M}_{B'C}(\mathcal{M}_{A'B}(A')) \sqsubseteq_* C$.

```
Prepare((E1, E2), (E1', E2')):
  (E1, E2) := Saturate(E1, E2)
  (E1', E2') := Saturate(E1', E2')

  (R1, R1', F1) := MergeClasses1(E1, E1')
  (R2, R2', F2) := MergeClasses2(E2, E2')

  (E1'', E2'') := Repartition((E1, E2),
                              R1 union R2, F1 union F2)
  (E1''', E2''') := Repartition((E1, E2),
                              R1' union R2', F1 union F2)

  return ((E1'', E2''), (E1''', E2'''))

Join((E1, E2), (E1', E2')):
  ((E1, E2), (E1', E2')) := Prepare((E1, E2), (E1', E2'))
  return (Join1(E1, E1'), Join2(E2, E2'))

Meet((E1, E2), (E1', E2')):
  ((E1, E2), (E1', E2')) := Prepare((E1, E2), (E1', E2'))
  return (Meet1(E1, E1'), Meet2(E2, E2'))
```

Figure 5. Join and meet algorithms.

Reflexivity. We need only one condition on `MatchClasses` to guarantee reflexivity: $\mathcal{M}_{AA}(A) = A$. Then since $A' \sqsubseteq_* A$, we use monotonicity to get $\mathcal{M}_{A'A}(A) \sqsubseteq_* A$. Then we use a different monotonicity property to get $\mathcal{M}_{A'A}(A') \sqsubseteq_* A$, which is the desired result.

7.2 Join and Meet

Join and meet are similar to implies checking. The main difference is that instead of matching the classes of the left input to the classes of the right input, we allow both inputs to be repartitioned into a new set of classes that may be more precise. Thus, we require that base domains to expose a `MergeClasses` operation that returns a mapping from either element's classes to new classes. The code is given in Figure 5.

Example. Consider joining the following elements. We use as our subdomains the separation logic domain and an integer domain instrumented with cardinality reasoning. The superscripts are used to name classes.

$$E_1 = [x \mapsto \texttt{nil}]^A \star [ls(y, \texttt{nil})]^B$$
$$E_2 = |A| = 1 \wedge |B| = n - 1 \wedge |B| \geq 1$$
$$E'_1 = [ls(x, \texttt{nil})]^C \star [y \mapsto \texttt{nil}]^D$$
$$E'_2 = |C| = n - 1 \wedge |D| = 1 \wedge |C| \geq 1$$

The calls to `MergeClasses` yield these results.

$$R_1 = \{(A, G), (B, H)\} \qquad R'_1 = \emptyset \qquad F_1 = \emptyset$$
$$R_2 = \{(C, G), (D, H)\} \qquad R'_2 = \emptyset \qquad F_2 = \emptyset$$

Next, we call the Repartition function.

$$E_1 = [x \mapsto \texttt{nil}]^G \star [ls(y, \texttt{nil})]^H$$
$$E_2 = |G| = 1 \wedge |H| = n - 1 \wedge |H| \geq 1$$
$$E'_1 = [ls(x, \texttt{nil})]^G \star [y \mapsto \texttt{nil}]^H$$
$$E'_2 = |G| = n - 1 \wedge |H| = 1 \wedge |G| \geq 1$$

Finally, we do the second join step, which yields the results below.

$$[ls(x, \texttt{nil})]^G \star [ls(y, \texttt{nil})]^H$$
$$|G| + |H| = n \wedge |G| \geq 1 \wedge |H| \geq 1$$

Soundness. We first need to place conditions on `MergeClasses`. They are similar to the conditions on `MatchClasses`.

$$\forall S, M, P.\ \gamma_i(E_i, S, M, P) \Rightarrow \exists M'.\ \text{ROK}_i(E_i, R, S, M, M') \wedge \gamma_f(F, S, M', P)$$

$$\forall S, M, P.\ \gamma_i(E_i', S, M, P) \Rightarrow \exists M'.\ \text{ROK}_i(E_i', R', S, M, M') \wedge \gamma_f(F, S, M', P)$$

Assume we are given a state S so that

$$\exists M, P.\ \gamma'(E_1, E_2, S, M, P).$$

Then after the saturate step, we continue to have

$$\exists M, P.\ \gamma'(E_1, E_2, S, M, P)$$

After calling `MergeClasses`, we get the following.

$$\exists M, P.\ \gamma'(E_1, E_2, S, M, P) \wedge \exists M'.\ \text{ROK}_1(E_1, R_1, S, M, M') \wedge \gamma_f(F_1, S, M', P) \wedge \exists M'.\ \text{ROK}_2(E_2, R_2, S, M, M') \wedge \gamma_f(F_2, S, M', P)$$

These statements make the call to `Repartition` valid, yielding states that are still valid.

$$\exists M, P.\ \gamma'(E_1'', E_2'', S, M, P)$$

We can state the correctness conditions for the subdomains' `Join` functions as well. We write $\sqcup_i$ for $\texttt{Join}_i$.

$$\forall S, M, P.\ \gamma_i(E_i, S, M, P) \Rightarrow \gamma_i(E_i \sqcup_i E_i', S, M, P)$$

(A similar fact holds for E_i'.) Using this property on both domains, we realize that S satisfies $(E_1'' \sqcup_1 E_1''', E_2'' \sqcup_2 E_2''')$.

In total, we have proved that if a state satisfies (E_1, E_2) then the state satisfies the result of the combined `Join` function. We could easily prove the symmetric property starting with (E_1', E_2'), which means that `Join` is sound.

7.3 Assume

There are two steps to processing an assume statement. First, we translate the given predicate into a formulas understood by each subdomain, and then these formulas are then passed to $\texttt{Assume}_i$.

This section expands the language of facts that can be passed to `Assume`. Previously every argument to a predicate had to be a quantified variable. Now we allow function applications as well, as long as the subdomain controls the given function. Function applications can be mixed with quantified variables. Thus, the following fact can be passed to $\texttt{Assume}_1$, given that f and c are functions from D_1: $\forall x \in C.\ \mathsf{P}(f(f(x)), c)$.

We explain the code for `Assume`, shown in Figure 6, with an example. Imagine that function f belongs to D_1, that nullary function c belongs to D_2, and that P is a unary predicate. We proceed through the execution of $\texttt{Assume}((E_1, E_2), \mathsf{P}(f(c)))$. First, we call the function $\texttt{TranslateTerm}_1$ on $f(c)$ with an empty environment. Since f is in D_1, we recurse on the argument c. This time, c is in D_2. There are no subterms so we do not recurse. Imagine we select the variable name z for x. D_2 is required to expose a function `Translate`, which we call.

The purpose of `Translate` is to map a function application into some class that the function result belongs to. We can always return a single class, since we can require there to exist a class equal to the union of any set of classes. Some of the arguments to a function application passed to $\texttt{Translate}_i$ will be applications of other functions from D_i. The remaining arguments will be variables from the environment env, which maps the variables to classes from the other domain.

Returning to the example, imagine $\texttt{Translate}_2$ maps c to class C_c. Then the current environment becomes $[z \mapsto C_c]$ and we return the expression z. Popping up the stack, we are done with arguments to f so we return the expression $f(z)$. `Assume` calls `AddExistentials`, which turns $\mathsf{P}(f(z))$ into $\exists z \in C_c.\ \mathsf{P}(f(z))$ using env. This fact is passed to D_1's `Assume`.

We perform similar steps upon calling $\texttt{TranslateTerm}_2$. The execution eventually leads to a call to $\texttt{Translate}_1$ on $f(z')$, with an environment mapping z' to C_c. Assuming this returns a class C_f, we end up calling $\texttt{Assume}_2$ on $\exists z'' \in C_f.\ \mathsf{P}(z'')$.

```
TranslateTerm1(E1, E2, env, f(e1, ..., ek)):
  if f in D1:
    for i in [1..k]:
      (ei', env) := TranslateTerm1(E1, E2, env, ei)
    return (f(e1', ..., ek'), env)
  else:
    for i in [1..k]:
      (ei', env) := TranslateTerm2(E1, E2, env, ei)
    x := some var not in env
    env := env[x -> Translate2(E2, env, f(e1', ..., ek'))]
    return (x, env)

// TranslateTerm2 is defined similarly

AddExistentials(env, fact):
  foreach [x -> C] in env:
    fact := (exists x in C. fact)
  return fact

Assume((E1, E2), P(e1, ..., ek)):
  env1 := []
  for i in [1..k]:
    (e1i, env1) := TranslateTerm1(E1, E2, env1, ei)
  fact1 := AddExistentials(env1, P(e11, ..., e1k))

  // fact2 := ...similar...

  return (Assume1(E1, fact1), Assume2(E2, fact2))
```

Figure 6. Assume algorithm.

Soundness. To prove soundness, we first need to define the semantics of `Translate`. To do so, we define an auxiliary predicate, TOK, that determines whether two expressions containing variables can be equal when their variables are interpreted in an environment. (The first argument to TOK is an environment mapping x_i to C_i.)

$$\text{TEQ}([x_1 \mapsto C_1, \ldots, x_n \mapsto C_n], M, e, e') \Leftrightarrow \exists x_1 \in M(C_1). \cdots \exists x_n \in M(C_n).\ e = e'$$

Now we can put conditions on `Translate`. Assume that $C = \texttt{Translate}_i(E_i, \text{env}, f(e_1', \ldots, e_k'))$. Let env be an environment and x any variable not in env.

$$\forall S, M, P.\ \gamma_i(E_i, S, M, P) \wedge \forall j.\ \text{TEQ}(\text{env}, M, [e_j]_S, [e_j']_S) \Rightarrow \text{TEQ}(\text{env}[x \mapsto C], M, [f(e_1, \ldots, e_k)]_S, x)$$

We prove the following. Assume that $\texttt{TranslateTerm}_i(E_1, E_2, \text{env}, e)$ returns (e', env'). Then

$$\forall S, M, P.\ \gamma'(E_1, E_2, S, M, P) \Rightarrow \text{TEQ}(\text{env}', M, [e]_S, [e']_S)$$

Additionally, we need to restrict the variables that appear in e'. If $\text{FV}(e') = V_1$ and $\text{dom}(\text{env}) = V_2$, then $V_1 \cap V_2 = \emptyset$ and $\text{dom}(\text{env}') = V_1 \cup V_2$.

```
Assign((E1, E2), f(e1, ..., ek), e):
  (E1, E2) := Saturate(E1, E2)

  if f in D1:
    (lv, env) := TranslateTerm1(E1, E2, [], f(e1, ..., ek))
    (rv, env) := TranslateTerm1(E1, E2, env, e)
    (E1', U, D) := Assign1(E1, env, lv, rv)
    E2' := E2
  else:
    (lv, env) := TranslateTerm2(E1, E2, [], f(e1, ..., ek))
    (rv, env) := TranslateTerm2(E1, E2, env, e)
    (E2', U, D) := Assign2(E2, env, lv, rv)
    E1' := E1

  j := 1
  repeat:
    (E1', U, D) = PostAssign1(E1', U, D, j)
    (E2', U, D) = PostAssign2(E2', U, D, j)
    j := j+1
  until j = num_strata

  (R1, F1) := EliminateClasses1(E1')
  (R2, F2) := EliminateClasses2(E2')
  return Repartition((E1', E2'), R1 union R2, F1 union F2)
```

Figure 7. Assignment algorithm.

First consider the case that $f \in D_i$. Then we make a recursive call on each of the subterms. We can use the invariant we are trying to prove about `TranslateTerm` here, inductively. It tells us that for all arguments j, TEQ(env, $M, [e_j]_S, [e'_j]_S$), and additionally, each invocation on e_j adds a distinct set of variables to env. Thus, we can merge all of these statements under a single set of quantifiers to obtain

$$\text{TEQ}(\text{env}, M, [f(e_1, \ldots, e_k)]_S, [f(e'_1, \ldots, e'_k)]_S).$$

The properties on variables from the recursive calls imply the variable properties for the main call.

Now assume $f \notin D_i$. We get the same properties of the arguments e'_j as before. Then we can use the property of `Translate` to get

$$\text{TEQ}(\text{env}[x \mapsto C], M, [f(e_1, \ldots, e_k)]_S, x),$$

assuming C is the result of `Translate`. Since the variable properties are also satisfied, we have proved the goal for `TranslateTerm`.

Now we prove the soundness of `Assume`. Our goal is to show the following.

$$\forall S, M, P.\, \gamma'(E_1, E_2, S, M, P) \wedge P(\mathsf{P})([e_1]_S, \ldots, [e_k]_S) \Rightarrow \gamma'(E'_1, E'_2, S, M, P)$$

The calls to `TranslateTerm` generate an environment and set of expressions such that TEQ($\text{env}_i, M, [e_{ij}]_S, [e'_{ij}]_S$). `TranslateTerm` uses each variable at most once, so we can combine the quantified equalities together to get the following (assuming $\text{env}_i = [x_1 \mapsto C_1, \ldots, x_n \mapsto C_n]$).

$$\exists x_1 \in M(C_1). \cdots \exists x_n \in M(C_n).\, P(\mathsf{P})([e_{i1}]'_S, \ldots, [e'_{ik}]_S)$$

Thus, the fact that we get in return from `AddExistentials` satisfies $\gamma_f(\text{fact}_i, S, M, P)$. Then the soundness condition on $\texttt{Assume}_i$ shows that the goal holds.

7.4 Assignment

Figure 7 has the code to perform assignment. The first step is to translate the expressions involved into a form that the domain can understand. The domain that performs the assignment is the one that understand the function on the left-hand side.

The translated expressions are passed to the domain's `Assign` operation. This operation updates the function being assigned to and returns a new domain element. Additionally, since the assignment may cause some false predicates to become true and other true predicates to become false, the function returns a set of "up" predicates and "down" predicates. The up predicates are standard facts, like the ones returned by `Consequences`. The down predicates are simpler: they are tuples of the form $\mathsf{P}(C_1, \ldots, C_k)$. If such a tuple belongs to D, it means that $\mathsf{P}(a_1, \ldots, a_k)$ if each $a_i \in C_i$.

The up and down predicates are shared with the other domain via its `PostAssign` operation. This operation may expand the set of up and down predicates. The termination condition of the loop is described below.

Note that assignment can change the partitioning of either base domain element, which must then be reflected in the other base domain. To simplify the issue, we require that the domains' `Assign` and `PostAssign` functions not repartition at all. Instead, partitioning changes must be delayed until all facts have been propagated. Then the `EliminateClasses` function returns a relation from old classes to new classes and `Repartition` is called.

Example. Again consider the case of the combination of a separation logic domain and an integer domain with cardinality.

$$E_1 = [ls(x, z)]^A \star [ls(z, \texttt{nil})]^B$$
$$E_2 = |A| \geq 1 \wedge |B| \geq 1 \wedge |A| = n \wedge |B| = k$$

We assign $z := \texttt{nil}$. Since z belongs to the separation logic domain, we generate $E'_1 = z = \texttt{nil} \wedge [ls(x, \texttt{nil})]^C$. The `Repartition` function relates A and B to C, knowing that A and B are disjoint. Therefore it can infer that $|C| = |A| + |B|$, which generates the repartitioned elements below.

$$E''_1 = z = \texttt{nil} \wedge [ls(x, \texttt{nil})]^C$$
$$E''_2 = |C| = n + k \wedge n \geq 1 \wedge k \geq 1$$

Example. Assume we have a TVLA domain and a numerical domain. The numerical domain is equipped with the predicate $\mathsf{Zero}(n) \stackrel{\text{def}}{=} \mathit{fld1}(n) = 0$. The TVLA domain is equipped with a predicates $\mathsf{Init}(n) \stackrel{\text{def}}{=} \exists n'.\, \mathsf{Zero}(n) \wedge \mathit{fld2}(n) = n'$. Consider the following elements.

$$E_1 = \mathsf{Zero}(N) \wedge \mathsf{Init}(N) \wedge \mathit{fld2}(N) = N'$$
$$E_2 = \mathit{fld1}(N) = 0 \wedge \mathsf{Init}(N)$$

Now imagine that we need to assign $\mathit{fld1}(N) := 1$. Doing so generates the following initial results.

$$E'_2 = \mathit{fld1}(N) = 1 \wedge \mathsf{Init}(N); \quad U = \emptyset; \quad D = \{\mathsf{Zero}(N)\}$$

Next we propagate the invalidated predicates P to the TVLA domain, which generates:

$$E'_1 = \mathit{fld2}(N) = N'; \quad U = \emptyset; \quad D = \{\mathsf{Zero}(N), \mathsf{Init}(N)\}$$

Finally we propagate back to the numerical domain, yielding:

$$E'_2 = \mathsf{True}; \qquad U = \emptyset; \qquad D = \{\mathsf{Zero}(N), \mathsf{Init}(N)\}$$

Soundness. We prove that if a state initially satisfies (E_1, E_2), then the same state, updated via $f(e_1, \ldots, e_k) := e$, satisfies the result of `Assign`.

The main requirement is that the shared predicates must be ordered. We assume that they are divided into numbered *strata*, written T_j. Each stratum T_j should include all the predicates from previous strata before j. For convenience we define $T_0 = \emptyset$. We assume that there is some j such that all predicates are contained in T_j, and we let num_strata $= j$.

Intuitively, T_j is the set of predicates that are interpreted correctly after a call to $\texttt{PostAssign}_i(E_i, U, D, j)$. We expect that D_i

is responsible for defining $T_j - T_{j-1}$ and that the definitions of these predicates depend only on predicates in T_{j-1}. The ordering requirement thus means that we cannot have predicates that are defined recursively. However, the possibility for constructs like transitive closure mostly negates the need for recursively defined predicates.

To begin, we make an addendum to the definition of γ_i. We allow the set of predicates passed to γ_i to be a partial function. If γ_i requires some predicate P to have a particular truth value, and if P is not defined by P, then the requirement should be treated as if it were satisfied. We can state this more formally as follows, for any E_i.

$$\forall S, M, P, P'.\ (\forall \mathsf{P} \in \text{dom}(P).\ P(\mathsf{P}) = P'(\mathsf{P})) \\ \wedge \gamma_i(E_i, S, M, P') \Rightarrow \gamma_i(E_i, S, M, P)$$

That is, if P' is an extension of P, and γ_i holds over P', then it must hold over P as well.

An unusual facet of the proof is that we also require each subdomain to provide an invariant, $I_i(E_i, E_i', j)$, which describes how E_i' changes as it is updated by `PostAssign`. Typically, this invariant will say that E_i' makes the same statements as E_i about predicates that have not been considered yet (those not in T_j).

We define some additional notation as well. We write $T_{j,i}$ to mean $T_j \cap \text{Preds}(D_i)$, where $\text{Preds}(D_i)$ is the set of predicates that D_i is responsible for defining. For a given interpretation of predicates P, we write $P \downarrow S$ to mean P with its domain restricted to the set of predicates S. That is, $(P \downarrow S)(\mathsf{P}) = P(\mathsf{P})$ if $\mathsf{P} \in S$ and otherwise is undefined. Finally, we write $\text{Similar}(P, P', D)$ to mean that P and P' are equal except over D. More formally, $\text{Similar}(P, P', D)$ holds when

$$\forall \mathsf{P}.\ \forall a_1, \ldots, a_k. \\ (\nexists C_1, \ldots, C_k.\ \bigwedge_i a_i \in M(C_i) \wedge \mathsf{P}(C_1, \ldots, C_k) \in D) \Rightarrow \\ P(\mathsf{P})(a_1, \ldots, a_k) \Leftrightarrow P'(\mathsf{P})(a_1, \ldots, a_k)$$

To begin, we make some assumptions about `Assign`. We use the notation $A(S)$ to model how the assignment changes the state. If $S = (A_1, A_2, F)$, then $A(S) = (A_1, A_2, F')$ where

$$F' = F[f \mapsto F(f)[([e_1]_S, \ldots, [e_k]_S) \mapsto [e]_S]].$$

We require the subdomain to provide an `Assign` operation satisfying the following.

$$\forall S, M, P.\ \gamma_i(E_i, S, M, P) \wedge \text{TEQ}(\text{env}, M, [e]_S, [\text{rv}]_S) \\ \wedge \text{TEQ}(\text{env}, M, [f(e_1, \ldots, e_k)]_S, [\text{lv}]_S) \Rightarrow \\ \exists P'.\ \gamma_i(E_i', A(S), M, P') \wedge I_i(E_i, E_i', 1) \\ \wedge \text{Similar}(P, P', D) \wedge \gamma_f(U, A(S), M, P') \\ \wedge T_1 \subseteq \text{dom}(P')$$

The `PostAssign` operation must satisfy these conditions. Let $(E_i'', U', D') = \texttt{PostAssign}(E_i', U, D, j)$. Then,

$$\forall S, M, P, P_0.\ \gamma_i(E_i, S, M, P_0) \wedge \gamma_i(E_i', A(S), M, P \downarrow T_{j-1}) \\ \wedge \gamma_f(U, A(S), M, P) \wedge \text{Similar}(P_0, P, D) \\ \wedge T_j \subseteq \text{dom}(P) \wedge I_i(E_i, E_i', j) \Rightarrow \\ \exists P'.\ \gamma_i(E_i'', A(S), M, P') \wedge I_i(E_i, E_i'', j+1) \\ \wedge \gamma_f(U', A(S), M, P') \wedge \text{Similar}(P_0, P', D') \\ \wedge T_{j+1,i} \subseteq \text{dom}(P') \\ \wedge P = P' \downarrow (\text{dom}(P') - T_{j+1,i})$$

With these conditions in place, soundness follows directly.

```
Widen((E1, E2), (E1', E2')):
  (E1', E2') := Saturate(E1', E2')

  (R1, F1) := MatchClasses1(E1', E1)
  (R2, F2) := MatchClasses2(E2', E2)

  (E1'', E2'') := Repartition((E1', E2'), R1 union R2,
                              F1 union F2)

  return (Widen(E1, E1''), Widen(E2, E2''))
```

Figure 8. Widening algorithm.

7.5 Widening

Although the set of shared predicates is finite, the subdomains may require widening to terminate. Thus, we need a widening algorithm for the combined domain. Figure 8 presents this algorithm. Given combined elements E and E', it saturates E', rewrites the class names of E' to match those of E, and then performs the widening operations provided by the subdomains.

Example. Consider the case of the combination of a separation logic domain and an integer domain with cardinality.

$$E_1 = [x \mapsto \texttt{nil}]^A \qquad E_2 = |A| = 1 \wedge i = 1 \wedge i \leq n$$
$$E_1' = [ls(x, \texttt{nil})]^B \qquad E_2' = |B| \geq 1 \wedge |B| \leq 2 \wedge i = |B| \wedge i \leq n$$

Saturation does nothing. When we call `MatchClasses`, it tells us that class B in E' should be mapped to class A in E. Repartitioning E' in this way yields the following element, which has simply been renamed.

$$E_1'' = [ls(x, \texttt{nil})]^A$$
$$E_2'' = |A| \geq 1 \wedge |A| \leq 2 \wedge i = |A| \wedge i \leq n$$

Performing a widening operation between E_1 and E_1'' yields E_1'' as a result. Widening on E_2 and E_2'' is similar to a join except that the upper bound on $|A|$ is dropped to ensure termination. Thus, we get the following result.

$$E_1^R = [ls(x, \texttt{nil})]^A \qquad E_2^R = |A| \geq 1 \wedge i = |A| \wedge i \leq n$$

Soundness. To prove soundness, we really just need to show that widening is an upper bound operator. First, assume we are given a state S so that element (E_1, E_2) is satisfied. Since E_1 and E_2 are passed to the subdomains' widening operators unchanged, it is clear that we S also satisfies the result of our widening.

Assume we are given a state S so that

$$\exists M, P.\ \gamma'(E_1', E_2', S, M, P).$$

Then after the saturate step, we continue to have

$$\exists M, P.\ \gamma'(E_1', E_2', S, M, P).$$

After calling `MatchClasses`, we get the following.

$$\exists M, P.\ \gamma'(E_1', E_2', S, M, P) \\ \wedge \exists M'.\ \text{ROK}_1(E_1', R_1, S, M, M') \wedge \gamma_f(F_1, S, M', P) \\ \wedge \exists M'.\ \text{ROK}_2(E_2', R_2, S, M, M') \wedge \gamma_f(F_2, S, M', P)$$

These statements make the call to `Repartition` valid, yielding a state that is still valid.

$$\exists M, P.\ \gamma'(E_1'', E_2'', S, M, P)$$

Since the subdomains' widening operators are upper bound operators, we know that S must satisfy the result, which proves our goal.

8. Completeness

This section describes requirements on the subdomains that, when met, guarantee the relative completeness of the partial order, the join operation, and assignment. These requirements can be met in practice, as we describe below.

8.1 Requirements

We first describe the restrictions informally.

- (Disjointness) The classes of each domain should be pairwise disjoint.
- (Emptiness) None of the classes of the domain should be empty.
- (Uniformity) Any statements made by D_1 about individuals of D_2 should state properties of whole classes of D_2, rather than individual elements. Thus, D_1 mandate that a predicate P hold for *all* elements of a class C, but it cannot say that P must hold for *some* element of C.
- (No Shared Predicates) Neither subdomain should expose any shared predicates.

The disjointness and emptiness restrictions are not very limiting. If we have a set of overlapping classes, we can always form the boolean combination of these classes via conjunction and negation; this set will always be disjoint. To deal with emptiness, we can use the disjunctive completion of the combined domain, as is done in TVLA.

Stated syntactically, the uniformity restriction means that all constraints on shared predicates should be universally quantified over a given class. As we noted earlier in the paper, cardinality reasoning cannot be expressed this way: it requires existential reasoning in some cases. Thus, our completeness result does not apply when either base domain reasons about cardinality. In exploring the subject, we have found cardinality reasoning to be the most difficult aspect of heap/numerical combination.

The lack of shared predicates does not forbid domains from tracking predicate information internally. Also, domains are permitted to expose their classes; the other domain may track properties of these classes, but the properties must be managed internally. Below, we give futher details about what domains satisfy this restriction.

The restrictions above can be formalized in terms of γ.

- (Disjointness) For any state (S, M, P) and any element E_i such that $\gamma_i(E_i, S, M, P)$, if C and C' are classes, then $M(C) \cup M(C') = \emptyset$.
- (Emptiness) For any state (S, M, P) and any element E_i such that $\gamma_i(E_i, S, M, P)$, if C is a class, then $M(C) \neq \emptyset$.
- (Uniformity) We say $(S_1, M_1, P_1) \sim_2 (S_2, M_2, P_2)$ when the following conditions are satisfied. Assume $S_1 = (A_{11}, A_{12}, F_1)$ and $S_2 = (A_{21}, A_{22}, F_2)$. We require $A_{11} = A_{21}$. We define an auxiliary relation $a_1 \sim a_2$ that is true when either of the following occur: (1) $a_1 \in A_{11}$, $a_2 \in A_{21}$, and $a_1 = a_2$, or (2) $a_1 \in A_{12}$, $a_2 \in A_{22}$, and $M_1^{-1}(a_1) = M_2^{-1}(a_2)$.

  For $\sim_2$, we require two main conditions. First, for every predicate P of arity k, and all individuals $a_1, \ldots, a_k, a'_1, \ldots, a'_k$, if each $a_i \sim a'_i$, then we have that $P_1(\mathsf{P})(a_1, \ldots, a_k) \Leftrightarrow P_2(\mathsf{P})(a'_1, \ldots, a'_k)$. Second, we require for all functions f and all individuals $a_1, \ldots, a_k, a'_1, \ldots, a'_k$, if each $a_i \sim a'_i$, then we have $F_1(f)(a_1, \ldots, a_k) \sim F_2(f)(a'_1, \ldots, a'_k)$.

  We require that if $(S_1, M_1, P_1) \sim_2 (S_2, M_2, P_2)$ holds, then $\gamma_1(E, S_1, M_1, P_1) \Leftrightarrow \gamma_1(E, S_2, M_2, P_2)$. We define a similar relation $\sim_1$ as well. For it, we require that if $(S_1, M_1, P_1) \sim_1 (S_2, M_2, P_2)$, then $\gamma_2(E, S_1, M_1, P_1) \Leftrightarrow \gamma_2(E, S_2, M_2, P_2)$.
- (No Shared Predicates) The `Consequences` function must not return any facts.

8.2 Basic Construction

Assume we have S_1, M_1, and P_1 so that $\gamma_1(E_1, S_1, M_1, P_1)$. And we have S_2, M_2, and P_2 so that $\gamma_2(E_2, S_2, M_2, P_2)$. We wish to define a function, g_1, that converts these inputs into S, M, P such that $\gamma_1(E_1, S, M, P) \wedge \gamma_2(E_2, S, M, P)$. We assume that both E_1 and E_2 have already been saturated.

Additionally, we wish to define a function g_2. Given S, M, P so that $\gamma_1(E'_1, S, M, P) \wedge \gamma_2(E'_2, S, M, P)$, as well as S_1 and S_2, it returns M_1, P_1, M_2, P_2 so that $\gamma_1(E'_1, S_1, M_1, P_1)$ and $\gamma_2(E'_2, S_2, M_2, P_2)$.

First, we describe g_1. We construct $S = (A_1, A_2, F)$ as follows. A_1 comes from S_1 and A_2 from S_2. Similarly, for a given class C, $M(C) = M_1(C)$ if C is a class of D_1 and otherwise $M(C) = M_2(C)$.

We construct a new F. Given a function f with arity k and elements $a_1, \ldots, a_k$, we need to define a value for $F(f)(a_1, \ldots, a_k)$. Without loss of generality, assume that f is from D_1. We construct an alternate set of parameters $a'_1, \ldots, a'_k$. If argument i comes from D_1 (according to the signature of f) then let $a'_i = a_i$. Otherwise, let C be the class of a_i according to M_1. We let a'_i be some element of $M_2(C)$ (such an element exists because classes are non-empty). Now we use $r = F_1(f)(a'_1, \ldots, a'_k)$ to determine the value of $F(f)(a_1, \ldots, a_k)$. If r is an individual from D_1, then $F(f)(a_1, \ldots, a_k) = r$. Otherwise, let C be the class of r according to M_1. Let r' be some element of $M_2(C)$. Then we let $F(f)(a_1, \ldots, a_k) = r'$.

By assumption, there are no shared predicates, so the definition of P is trivial.

We need to show $\gamma_1(E_1, S, M, P)$ holds. Recall that $\gamma_1(E_1, S_1, M_1, P_1)$. We prove the fact simply by showing that $(S, M, P) \sim_2 (S_1, M_1, P_1)$ and applying the uniformity assumption. The reader can verify that all the conditions of $\sim_2$ are satisfied by the construction. Using a symmetric argument, $\gamma_2(E_1, S, M, P)$ also holds.

The construction and proof of the function g_2 are similar to those for g_1.

8.3 Completeness of Implication

As a notational convenience, we use $\sqsubseteq$ to denote `Implies` and $\sqsubseteq_i$ to denote $\texttt{Implies}_i$.

We require the implication operation for the subdomain to be complete (we are proving relative completeness). Formally, this means the following.

$$[\forall S.\ (\exists M, P.\ \gamma_i(E_i, S, M, P)) \Rightarrow (\exists M, P.\ \gamma_i(E'_i, S, M, P))] \Rightarrow E_i \sqsubseteq_i E'_i$$

The condition that we seek to prove for the combined domain is similar.

$$[\forall S.\ (\exists M, P.\ \gamma'(E_1, E_2, S, M, P)) \Rightarrow (\exists M, P.\ \gamma'(E'_1, E'_2, S, M, P))] \Rightarrow (E_1, E_2) \sqsubseteq (E'_1, E'_2)$$

We will assume the antecedent for the condition on the combined domain holds. Our goal is to prove that $\texttt{Implies}_1$ and $\texttt{Implies}_2$ hold for each S.

First, we need to deal with the issue of repartitioning. We make a fairly strong requirement on the `Repartition` function: that it not affect precision. Without this requirement, it is difficult to make any statements at all about completeness. We can state the requirement

formally as follows, with R and F being arbitrary.

$$\forall S.\ (\exists M, P.\ \gamma_i(E_i, S, M, P)) \Leftrightarrow \\ (\exists M, P.\ \gamma_i(\texttt{Repartition}_i(E_i, R, F), S, M, P))$$

This property allows us to ignore repartitioning when checking completeness.

The saturation step, however, is quite important. In the construction of g_1 and g_2 above, we had to assume that both input elements are saturated.

Now the proof. Assume arbitrary S. We do case analysis depending on whether M and P exist so that $\gamma_1(E_1, S, M, P)$ and $\gamma_2(E_2, S, M, P)$ hold. If both hold, then we use g_1 to find a single S', M, P where $\gamma'(E_1, E_2, S', M, P)$ hold. Then we use the assumption from `Implies` to obtain $\gamma'(E_1', E_2', S', M', P')$ for some M' and P'. Finally, we use g_2 to obtain M_1, P_1, M_2, P_2 so that $\gamma_1(E_1', S, M_1, P_1)$ and $\gamma_2(E_2', S, M_2, P_2)$. These allow us to prove that the conditions for $\texttt{Implies}_1$ and $\texttt{Implies}_2$ are satisfied at S.

Now consider the case where only one of $\gamma_i(E_i, S, M, P)$ holds. Without loss of generality, assume it's $\gamma_1(E_1, S, M, P)$. Then there are two cases, depending on whether there exist S', M', P' so that $\gamma_2(E_2, S', M', P')$ holds. If not, then E_2 is equivalent to False, which we expect to be propagated by `Saturate` to E_1, making it also unsatisfiable, which is a contradiction. So assume we have $\gamma_2(E_2, S', M', P')$. Then we use g_1. to find S'', M'', P'' so that $\gamma'(E_1, E_2, S'', M'', P'')$. Then we use the assumption from `Implies` to obtain $\gamma'(E_1', E_2', S''', M''', P''')$. Finally, we use g_2 to obtain M_1, P_1, M_2, P_2 so that $\gamma_1(E_1', S, M_1, P_1)$ and $\gamma_2(E_2', S', M_2, P_2)$. The first allows us to prove that the conditions for $\texttt{Implies}_1$ are satisfied at S. The conditions for $\texttt{Implies}_2$ are trivially satisfied at S, since the antecedent is false.

Finally, we consider the case where $\gamma_1(E_1, S, M, P)$ and $\gamma_2(E_2, S, M, P)$ are both unsatisfiable for S. In this case, the conditions for $\texttt{Implies}_1$ and $\texttt{Implies}_2$ are both trivially satisfied at S.

8.4 Completeness of Join

As above, we will assume that the subdomains' join procedures are complete.

$$E_A \sqsubseteq_i E \wedge E_B \sqsubseteq_i E \Rightarrow (E_A \sqcup E_B) \sqsubseteq_i E$$

We want to prove the same fact for our combined domain.

$$(E_{A1}, E_{A2}) \sqsubseteq (E_1, E_2) \wedge (E_{B1}, E_{B2}) \sqsubseteq (E_1, E_2) \Rightarrow \\ ((E_{A1}, E_{A2}) \sqcup (E_{B1}, E_{B2})) \sqsubseteq (E_1, E_2)$$

As in the previous proof, it is safe to ignore the effects of repartitioning. We denote the effect of saturating an element E as E'. In that case, we can simplify the assumptions above to the following.

$$E_{A1}' \sqsubseteq_1 E_1 \quad E_{B1}' \sqsubseteq_1 E_1 \\ E_{A2}' \sqsubseteq_1 E_2 \quad E_{B2}' \sqsubseteq_1 E_2$$

The fact that we have to prove is as follows.

$$(E_{A1}' \sqcup_1 E_{B1}')' \sqsubseteq_1 E_1 \\ (E_{A2}' \sqcup_1 E_{B2}')' \sqsubseteq_2 E_2$$

Using the assumptions, as well as the fact that the subdomains' join operations are complete, we can prove the following.

$$(E_{A1}' \sqcup_1 E_{B1}') \sqsubseteq_1 E_1 \\ (E_{A2}' \sqcup_1 E_{B2}') \sqsubseteq_2 E_2$$

Since saturation can't make an element any *less* precise (due to the completeness of `Assume`), we have proved our goal.

8.5 Completeness of Assignment

Since there are no shared predicates, the completeness of assignment depends on two conditions: (1) the $\texttt{Assign}_i$ functions must be complete, and (2) the $\texttt{Translate}_i$ functions must return classes of cardinality 1. Guaranteeing the second conditions requires that the locations used in an assignment operation be brought into "focus" before the assignment takes place, which is outside of the scope of this paper.

To formalize these conditions, we define some auxiliary predicates. The first predicate, Exact, tells us when two states S and S' differ at exactly one place (that is, at one point in the range of one function). We define it as follows.

$$\text{Exact}(S, S') \stackrel{\text{def}}{=} \exists f, a_1, \ldots a_k.\ \forall g, a_1', \cdots a_k'. \\ f \neq g \vee a_1 \neq a_1' \vee \ldots \vee a_k \neq a_k' \Rightarrow \\ F_S(g)(a_1', \ldots, a_k') = F_{S'}(g)(a_1', \ldots, a_k')$$

We define a second auxiliary predicate that tells us when an expression interpreted in a given environment maps to a class whose cardinality is one.

$$\text{Single}(\text{env}, M, e) \stackrel{\text{def}}{=} \forall v \in \text{FV}(e).\ |M(\text{env}(v))| \leq 1$$

We formalize the first condition as follows. We require that if $(E_i', U, D) = \texttt{Assign}_i(E_i, \text{env}, lv, rv)$, then the following must hold.

$$[\forall S, M, P.\ \gamma_i(E_i, S, M, P) \Rightarrow \\ \text{Single}(\text{env}, M, lv) \wedge \text{Single}(\text{env}, M, rv)] \Rightarrow \\ [\forall S, M, P.\ \gamma_i(E_i', S, M, P) \Rightarrow \\ (\exists S_0, M_0, P_0.\ \text{Exact}(S_0, S) \wedge \gamma_i(E_i, S_0, M_0, P_0))]$$

This condition tells us that as long as lv and rv are singleton classes, `Assign` will return an element that differs from the original one in exactly one place. In truth, this place will be lv, but we do not state this because doing so would require us to incorporate some of the soundness proof.

We also place conditions on `Translate`. They essentially say that it must return a singleton class. Assume that $\texttt{Translate}_i(E_i, \text{env}, \texttt{f}(e_1, \ldots$ C and that $x \notin \text{dom}(\text{env})$. Then we require the following.

$$\forall S, M, P.\ \gamma_i(E_i, S, M, P) \wedge \forall j.\ \text{Single}(\text{env}, M, e_i) \Rightarrow \\ \text{Single}(\text{env}[x \mapsto C], M, x)$$

Finally, we place some simple conditions on `PostAssign`. Since there are not shared predicates, we simply require that it preserve the semantics of the input element.

$$\forall S, M, P.\ \gamma_i(E_i, S, M, P)) \Leftrightarrow \gamma_i(E_i', S, M, P)$$

We begin by proving the following fact about `TranslateTerm` by induction. Assume that `TranslateTerm` returns (e', env').

$$\forall S, M, P.\ \gamma'(E_1, E_2, S, M, P) \Rightarrow \text{Single}(\text{env}', M, e')$$

We will lazily make the same assumptions about variable scoping as we have done previously. There are two cases to the proof. First assume that $f \in D_1$. Then, by the induction hypothesis, Single holds of each e_i'. Clearly, then, Single also holds of $f(e_1', \ldots, e_k')$. In the second case, $f \notin D_1$. Again, Single holds of each e_i' by the induction hypothesis. Therefore, the conditions of `Translate` are satisfied, meaning that Single also holds of x, as desired.

Now we can assemble a completeness proof for the combined `Assign` operation. We prove the following about it.

$$\forall S.\ (\exists M, P.\ \gamma'(E_1', E_2', S, M, P)) \Rightarrow \\ (\exists S_0, M_0, P_0.\ \gamma'(E_1, E_2, S_0, M_0, P_0) \wedge \text{Exact}(S_0, S))$$

Since there are no shared predicates, `Consequences` returns the empty set and therefore `Saturate` is the identity function. In `TranslateTerm`, let S, M, P be a state that satisfies (E_1, E_2). Then we know that Single(env, M, lv) and Single(env, M, rv) both hold. These allow us to apply the condition for `Assign`, which essentially gives us exactly what we need. Neither the `PostAssign` or `Repartition` function are semantically meaningful, so we have proven the goal.

8.6 Complete Subdomains

The four requirements in Section 8.1 are restrictive. However, they are satisfied by some of the domains in Section 3. We disregard the disjointness and emptiness restrictions because they are easy to satisfy. The separation logic domain and the polyhedra domain with quantification satisfy the uniformity condition. The canonical abstraction domain may satisfy the uniformity restriction if one is careful about what predicates to expose as shared.

The lack of shared predicates is not an issue in combinations where one of the domains cannot understand shared predicates, such as the combination of a heap domain with the quantified polyhedra domain.

Thus, the separation domain combined with the polyhedra domain with quantifiers satisfies our constraints. Of course, these domains are not themselves complete. However, the relative completeness theorem for the combined domain means that if there is a precision problem, the combined domain cannot be blamed. More generally, the requirements on completeness give us some intuition about what makes combining domains difficult. Namely, cardinality seems to be difficult to handle precisely.

9. Related Work

Combining Abstractions

The seminal paper by Cousot and Cousot in [4] introduces different methods for combining abstract domains and [6] elaborates on domain constructors. Our combination of abstractions is a refinement of the reduced product as it allows the partitions in the different domains to refine each other. This increases the state space but provides further distinctions which are useful for verifying invariants which quantify over both the heap and numeric individuals. Also, we provide sound algorithms for computing transformers.

Combining Heap and Numerical Abstractions

The idea to combine numeric and pointer analysis for establishing properties of memory was pioneered by Alain Deutsch [7, 8]. 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. Elaborations of Deutsch's work appear in [24]

In [15] a type and effect system is suggested for a variant of ML that bounds the size of memory used by the program with applications to embedded code. Their type system checks bounds on memory usage while our analysis can be used to infer the bounds. Furthermore, their type system is for a functional language while our analysis is appropriate for an imperative languages with destructive updates.

In [14] 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 that use destructive updates.

In [25] 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 to showing specific properties of singly-linked lists. Similar restrictions apply to [2, 17].

Rugina [22] 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.

In [3] a method is presented for analyzing a memory allocator by interpreting memory segments as both raw buffers and structured data. Their method presents a limited way of treating sizes of chunks of memory. However, they are limited to contiguous chunks of memory and cannot handle sizes of recursive data structures.

In [10], 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. 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.

In [13], a technique to reason about the contents of arrays, and the relationship between array elements, is presented. This technique infers many useful array properties, but it performs no cardinality reasoning and it cannot reason about recursive data structures.

A general method for combining numeric domains and canonical abstraction was presented in [11]. A general method for tracking partition sizes was recently presented in [12]. These are two orthogonal methods; the former addresses the problem of abstracting values of numerical fields and the latter addresses the problem of inferring partition sizes. The work presented in this paper was inspired by these two works and generalizes both of them in several fundamental ways to establish a useful system. In contrast [11], we support many kinds of partition based abstractions which makes the result more accessible, general and allows more scalable heap abstractions. Also, we prove that our reasoning is complete for numeric fields and incomplete for partition sizes. This sheds some light on the difficulties of these problems.

Reducing Pointer to Integer Programs

In [9, 2, 17] 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" uses different integer analyzers on the resulting program. Furthermore, for proving simple properties of singly-linked lists it was shown in [2] that there is no loss of precision. However, it may lose precision in cases where the heap and integers 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 heap 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 [16] 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 [20] 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.

10. Conclusion

In this paper we presented a technique for simultaneously reasoning about integers and the heap. Our combined domain is parametric in the heap domain and the numerical domain. The crux of our approach is that we permit a great deal of sharing between the domains via predicates and classes of individuals. At the same time, the interface between the domains is very generic; although our intended target is a combination of heap and numerical reasoning, our technique supports arbitrary quantified reasoning over two disjoint sorts.

We have implemented a limited form of our combined domain, where predicates and classes are shared between a canonical abstraction and a domain of bounded difference constraints. We can reason about simple properties, such as "all elements of an array have been initialized to zero." However, our results are limited because we have not yet implemented any cardinality reasoning.

References

[1] 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.

[2] Ahmed Bouajjani, Marius Bozga, Peter Habermehl, Radu Iosif, Pierre Moro, and Tomás Vojnar. Programs with lists are counter automata. In *CAV*, pages 517–531, 2006.

[3] Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. Beyond reachability: Shape abstraction in the presence of pointer arithmetic. In *SAS*, pages 182–203, 2006.

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

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

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

[7] 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.

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

[9] 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.

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

[11] Denis Gopan, Frank DiMaio, Nurit Dor, Thomas W. Reps, and Mooly Sagiv. Numeric domains with summarized dimensions. In *TACAS*, pages 512–529, 2004.

[12] Sumit Gulwani, Tal Lev-Ami, and Mooly Sagiv. A combination framework for tracking partition sizes. In *POPL*, pages 239–251, 2009.

[13] Nicolas Halbwachs and Mathias Péron. Discovering properties about arrays in simple programs. In *PLDI*, pages 339–348, 2008.

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

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

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

[17] Stephen Magill, Josh Berdine, Edmund M. Clarke, and Byron Cook. Arithmetic strengthening for shape analysis. In *SAS*, pages 419–436, 2007.

[18] Bill McCloskey and Mooly Sagiv. Combining quantified domains (full version). `http://www.cs.berkeley.edu/~billm/tr-combining.pdf`, 2009.

[19] Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM Trans. Program. Lang. Syst.*, 1(2):245–257, 1979.

[20] Huu Hai Nguyen, Cristina David, Shengchao Qin, and Wei-Ngan Chin. Automated verification of shape and size properties via separation logic. In *VMCAI*, pages 251–266, 2007.

[21] William Pugh. Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM*, 33(6):668–676, 1990.

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

[23] Mooly Sagiv, Thomas W. Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *ACM TOPLAS*, 24(3):217–298, 2002.

[24] Arnaud Venet. Automatic analysis of pointer aliasing for untyped programs. *Sci. Comput. Program*, 35(2):223–248, 1999.

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