UC Berkeley
UC Berkeley Electronic Theses and Dissertations

Title
Practical Shape Analysis

Permalink
https://escholarship.org/uc/item/8th04445

Author
McCloskey, William Terrence

Publication Date
2010

Peer reviewed|Thesis/dissertation

Practical Shape Analysis

by

William Terrence McCloskey

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Eric Brewer, Chair
Professor George Necula
Professor Leo Harrington

Spring 2010

Practical Shape Analysis

Abstract

Practical Shape Analysis

by

William Terrence McCloskey

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Eric Brewer, Chair

Shape analysis is a program analysis technique used to prove that imperative programs using manual memory management will not crash. In the past, shape analysis has been applied to data structures like linked lists and binary trees. It has also been used on simplified versions of Windows device drivers.

We describe techniques that allow us to apply shape analysis to data structures that occur commonly in systems code. These data structures often use arrays, hash tables, C strings, and buffers of a known size. Sometimes, memory in these data structures is managed by manual reference counting. Analyzing such code is difficult or impossible with existing shape analyses. Most difficult of all, many data structures use several of these patterns at the same time, such as a hash table pointing to reference counted objects through which a doubly linked list threads.

We describe an analysis capable of handling these data structures easily and efficiently. Our technique uses abstract interpretation over the combination of two abstract domains. One, based on three-valued logic, is used for analyzing the heap. The other domain reasons about integers and set cardinality. The key feature of the combined domain is that quantified facts can be shared between the integer and heap domains. The precision we achieve is significantly greater than if either domain were used independently.

Besides improvements in precision, we also describe changes that make both domains more scalable and efficient. We present the results of experiments analyzing the cache data structure of the `thttpd` web server, which uses a hash table, linked lists, and reference counting in a single data structure. We successfully prove the absence of memory errors in about two minutes.

Contents

List of Figures

List of Tables

Acknowledgments

I am indebted to many people for help with this thesis. My advisor, Eric Brewer, read through all the drafts and made many improvements. Over the years, he has given me freedom and encouragement, without which I would not have finished. The research is a product of a fruitful and very enjoyable collaboration with Mooly Sagiv and Tom Reps, who also read through drafts and made suggestions. George Necula served on my committee and has helped me in many ways as a mentor and teacher at Berkeley. And Ras Bodík provided me with years of free lunches and much-needed advice.

I also owe a great deal to my friends at Berkeley, AJ, Manu, Padma, Dave, Evan, Ana, David, and Brian, to my mom and dad, and to Jamie.

Thank you all.

Chapter 1

Introduction

The goal of this thesis is to statically analyze realistic systems programs, such as servers and operating systems. We search for memory errors in these programs. Memory errors can take the form of out-of-bounds memory accesses, accesses to freed memory, or freeing memory more than once or not at all.

The correctness of all programs depends on *invariants*. These are properties that the program ensures are always true. We need to discover a set of invariants that, taken together, imply memory safety. Ideally, this set of invariants would be much smaller than the set required to prove total program correctness. Unfortunately, memory safety is not so well contained—its tendrils reach far deep—so we need to discover some surprisingly sophisticated invariants. We describe some of them in the next few paragraphs and explain why they are necessary to prove memory safety. We emphasize that all these examples are used in real code [39].

Reachability. Reachability plays an important role in the definition of memory safety. In order to prove that there are no memory leaks, we must prove the invariant that all allocated objects are reachable.

However, reachability plays an important role even if we are not interested in guaranteeing the absence of memory leaks. Consider a program where `list1` and `list2` point to two disjoint linked lists. The following code frees the elements of `list1` and then traverses the other list.

```
1  while (list1 != null) {
2    tmp = list1->next;
3    free(list1);
4    list1 = tmp;
5  }
6  p = list2;
7  while (p) { ...; p = p->next; }
```

Without reachability, there is no way to distinguish elements of `list1` from elements of

`list2`. Therefore, we have no way to know that an element freed at line 3 is not accessed at line 7. To solve the problem, we must infer the invariant that elements reachable from `list2` are not reachable from `list1` and therefore will not be freed.

Cardinality. Suppose that `free_list` points to a linked list of nodes that are available to be used when needed. If the list becomes too large, then the following code releases some of the elements to the operating system.

```
1  while (free_count > DESIRED_FREE_COUNT) {
2    tmp = free_maps;
3    free_maps = tmp->next;
4    --free_count;
5    free(tmp);
6  }
```

Proving that this code does not dereference a null pointer at line 3 requires us to infer the invariant that `free_count` holds the length of the linked list.

Index invariants. Suppose we have a data structure like the one below.

```
struct T { int index; char *data; }
struct T *table[32];
...
obj = malloc(sizeof(Obj));
obj->index = i;
table[i] = obj;
obj->data = ...;
```

The array `table` points to objects of type `T`. The index of an object in `table` is stored in its `index` field. When freeing an object, the `index` field is used to remove it from the table:

```
n = obj->index;
table[n] = null;
free(obj);
```

Danger lurks here. If `obj->index` is incorrect, then we will null out the wrong `table` entry. As a consequence, `obj` will remain in the table after it has been freed. Then it is possible that the following code accesses freed memory.

```
if (table[i]) puts(table[i]->data);
```

To prevent this problem, we must ensure that if `table[i] = obj` then `obj->index = i`. This invariant is difficult to infer because it quantifies both over an integer, `i`, and a heap object, `obj`.

Reference counting. Manual reference counting, in which the programmer tracks the number of references to a given object, is very closely related to memory safety. We want to prove that a given reference counting scheme does not access memory that it has already freed. To do so, we must prove the invariant that the reference count tracked by the programmer is equal to the actual number of references. The problem is complicated by the fact that programmers may track only a subset of all references to a given object; other references may be handled in a different way.

Until now, invariants of this complexity have not been checkable by automated analyses. There do exist analyses that can verify *some* of these properties. Heap analyses based on canonical abstraction [46] or separation logic [4] can check reachability properties. A few specialized systems can check cardinality properties [31, 25] or reference counting [21]. However, no system is able to check *all* of these properties simultaneously on a single data structure.

To avoid reinventing the wheel, our goal is to combine the best aspects of existing analyses to create a more powerful tool that can analyze many complex invariants simultaneously. This thesis presents an analysis, called DESKCHECK, that is able to verify all of these invariants. DESKCHECK performs abstract interpretation [15] over a combination of an existing heap domain and an existing integer domain. These domains have been augmented to communicate invariants with each other as they learn them, increasing their individual power. We have used DESKCHECK to verify the memory safety of the cache module of a real-world web server called `thttpd` [39]. All of the invariants listed above are needed to verify the `thttpd` cache.

The next few sections describe the fundamentals of DESKCHECK, followed by an overview of our combination technique and a summary of the rest of the thesis.

1.1 Programming Language

Programs are provided to DESKCHECK in a language called PBJ. The syntax of the language is close to Pascal. The semantics is modeled on logic, much as in other modeling languages like Boogie [30]. Eventually, we would like to use techniques from tools like CCured [38] and Deputy [11] to translate C and C++ programs to PBJ automatically. For now, we must be content with manual translations.

A PBJ program begins with a list of type declarations, such as "`type List;`" or "`type IntArray;`". Values in PBJ are either integers or instances of a user-defined type, like `List`.

All program data is stored in *maps*. A map is a mapping from a sequence of keys to values. Every variable in the program is a map with a given signature. The signature tells us the types of the map's keys and values. A map `x` with the signature "`x[int,int]:int`" maps two integers to an integer. The syntax "`y:int`" is shorthand for a map that requires no keys and returns an integer. The number of keys is called the arity.

The simplest way to describe how this works is with an example.

Example 1 Consider the following program.

```
1  procedure test1()
2    x:int;
3    a[int]:int;
4  {
5    x := 0;
6    a[0] := x;
7    a[1] := x+1;
8  }
```

This program declares two maps, `x` and `a`. The map `x` works much like a variable. We can assign a value to it and access the value later. The map `a` is like an array. The integer key acts as an array index. □

Maps are a general way of expressing typical programming language concepts like variables, arrays, and fields. The following example shows how fields work.

Example 2 Consider this program.

```
1  type List;
2
3  global List_data[List]:int;
4  global List_next[List]:List;
5
6  procedure test2(p:List)
7    sum:int;
8  {
9    sum := 0;
10   while (p != null) {
11     sum := sum + List_data[p];
12     p := List_next[p];
13   }
14 }
```

This program traverses a linked list. First we declare the type of list elements. Then we declare two global maps, `List_data` and `List_next`, representing fields of the list object. We intend `List_data[e]` to refer to the data field of a linked list node `e`. Similarly, `List_next[e]` is a pointer to the next element of the list. All user-defined types include a special `null` value that can be used as a terminator. □

1.1.1 Syntax

The grammar for the PBJ language is shown in Figure 1.1. We omit the grammar for declaring types, maps, and procedures. Statements are conventional. The most common

stmt ::= lvalue := expr ;
| id (expr*) ;
| lvalue := id (expr*) ;
| lvalue := **new** type ;
| **delete** lvalue ;
| **assert** (expr) ;
| **return** expr ;
| **if** (expr) stmt **else** stmt
| **while** (expr) stmt
| **label** L ;
| **goto** L ;
| { stmt* }

lvalue ::= id | id [expr*]

expr ::= lvalue
| int-literal
| expr **+** expr
| **-** expr
| expr **=** expr
| expr **>** expr
| **not** expr
| expr **&&** expr
| expr **||** expr

Figure 1.1: Syntax of the PBJ language.

statement is assignment. Following it is the syntax for procedure calls; the result of a procedure can be stored or ignored. The `new` and `delete` statements are similar to `malloc` and `free` in C. Unlike C++, no constructors or destructors are invoked. The grammar for expressions is utterly conventional.

PBJ is statically typechecked. If, for example, `x` has signature `x[int]:List` and `field` has signature `field[List]:int`, then `field[x[3]]` will be accepted by the type checker while `field[3]` will be rejected.

1.1.2   Semantics

Values in PBJ are either integers or elements of some user-defined type. For each type, we assume that there exists an infinite, fixed universe of values, or "individuals." Initially, all individuals of a type t belong to an "available" set, $A(t)$. When the program asks for a new individual of type t via `new t`, we remove an arbitrary individual from $A(t)$ and return it to the program.

We store the state of a PBJ program as a function, σ. For any map variable m, $\sigma(m)$ stores the value of m. If m has arity k, then $\sigma(m)$ will be a function of arity k. As an example, after the statements in Example 2 execute, the state will be as follows.

$$\sigma(\mathtt{x}) = \{\langle\rangle \mapsto 0\}$$
$$\sigma(\mathtt{a}) = \{\langle 0\rangle \mapsto 0, \langle 1\rangle \mapsto 1\}$$

A map of arity 2 would have keys like $\langle 1, 2\rangle$.

Expressions. We write the semantics for an expression e as $\langle e, \sigma\rangle \rightarrow v$ where σ is the state in which the expression is evaluated and v is the result of the evaluation. We give

some examples of how expressions are evaluated.

$$\frac{\langle e_1, \sigma \rangle \rightarrow v_1 \ \cdots \ \langle e_n, \sigma \rangle \rightarrow v_n \quad (\langle v_1, \ldots, v_n \rangle \mapsto v) \in \sigma(x)}{\langle x[e_1, \ldots, e_n], \sigma \rangle \rightarrow v}$$

$$\frac{\langle e, \sigma \rangle \rightarrow n \quad \langle e', \sigma \rangle \rightarrow n'}{\langle e + e', \sigma \rangle \rightarrow n + n'}$$

$$\frac{\langle e, \sigma \rangle \rightarrow v \quad \langle e', \sigma \rangle \rightarrow v' \quad v'' = \text{if } v = v' \text{ then } 1 \text{ else } 0}{\langle e = e', \sigma \rangle \rightarrow v''}$$

The rule for addition assumes that both operands evaluate to integers. The PBJ type system ensures that this will always be the case. The rule for $x[1]$ succeeds only when the key $\langle 1 \rangle$ is present in $\sigma(x)$. When the key is not present, we are not able to evaluate the expression to any value. We say that we do not "make progress" on such expressions.

Statements. We only give the semantics of a few key statements. Control structures like "if" and "while" are handled in the typical way. We need to track the state σ, the set of available objects A, and the set of freed objects F. We write the meaning of a statement s as $\langle \sigma, A, F \rangle \ s \ \langle \sigma', A', F' \rangle$. This means that, starting in the left state, s finishes in the right state.

$$\frac{\langle e, \sigma \rangle \rightarrow v \quad \langle e_i, \sigma \rangle \rightarrow v_i \quad \sigma' = \sigma[x \mapsto \sigma(x)[\langle v_1, \ldots, v_n \rangle \mapsto v]]}{\langle \sigma, A, F \rangle \ x[e_1, \ldots, e_n] := e \ \langle \sigma', A, F \rangle}$$

$$\frac{\langle e_i, \sigma \rangle \rightarrow v_i \quad v \in A(t) \quad A' = A[t \mapsto A(t) \backslash v] \quad \sigma' = \sigma[x \mapsto \sigma(x)[\langle v_1, \ldots, v_n \rangle \mapsto v]]}{\langle \sigma, A, F \rangle \ x[e_1, \ldots, e_n] := \texttt{new } t \ \langle \sigma', A', F \rangle}$$

$$\frac{\langle e, \sigma \rangle \rightarrow v \quad v \notin F \quad F' = F \cup \{v\}}{\sigma' = \{x \mapsto \{k \mapsto v' : (k \mapsto v) \in \sigma(x) \wedge v \notin k\} : x \in \text{dom}(\sigma)\}}{\langle \sigma, A, F \rangle \ \texttt{delete } e \ \langle \sigma', A, F' \rangle}$$

In the $\texttt{new } t$ statement, an individual from $A(t)$ is arbitrarily chosen, removed from $A(t)$, and returned as the result. In the $\texttt{delete } e$ statement, e is evaluated to some individual v. Then all entries of the form $\langle \ldots, v, \ldots \rangle \mapsto v'$ are removed from all maps. We use the set F to ensure that no individual is ever freed twice.

If we are not able to make progress on a statement's sub-expressions, then we cannot make progress on the statement either. This condition signals that the program is in error. The goal of this thesis is to guarantee the absence of such errors.

Logic connection. The semantics of PBJ programs is similar to the semantics of many-sorted logic. In the rest of the thesis, we use the language of logic when reasoning about the formal semantics of PBJ programs. Map variables are called "uninterpreted functions." (They are called uninterpreted to distinguish them from interpreted functions like $+$.) Types

are called "sorts." Values are called "individuals," and they are drawn from a "universe." Eventually we introduce predicates from logic as well. To distinguish predicates from uninterpreted functions, we write a predicate as Pred and an uninterpreted function as *func*.

1.2   Analysis of Programs

The goal of our program analysis is to statically exclude certain classes of errors. Now that we have explained PBJ semantics more clearly, we can state the errors.

- Accessing a map with a key that is not in its domain.

- Using `delete` twice on the same value.

- Failing an assertion in the program.

To solve this problem, we use the technique of *abstract interpretation* [13]. This section gives a brief overview of abstract interpretation; it can be skipped by those already familiar with it.

Throughout this section we use the following example program. Our goal is to prove that the assertion holds.

```
1  procedure test(n:int)
2    i:int
3  {
4    i := 0;
5    while (i < n)
6      i := i+1;
7    assert(i = n);
8  }
```

Interpretation.   The first step in proving the assertion is to transform the program into a control-flow graph, shown in Figure 1.2. The edges of this graph shown how the program transitions between statements. Each edge corresponds to a point in the execution of the program. For example, this program will pass through the points labeled U and Z once, while passing through X n times.

We can "interpret" the program by starting at the point labeled U in some initial program state where n is positive. We execute it according to the semantics shown in the previous section. Whenever we reach a branch in the graph, we evaluate the condition ($i < n$ in this case) to choose whether to go left or right. We can label each program point p with the set of all states that pass through that program point. We call this set $\Sigma(p)$. For example,
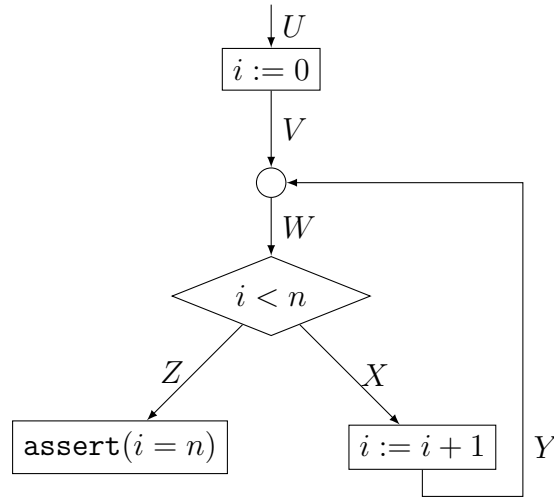
Figure 1.2: Control-flow graph for example program.

$\Sigma(V)$ contains all states where n is a positive number and i is zero. To simplify notation, we write $S(a, b)$ for the following state.

$$\sigma(\mathtt{i}) = \{\langle\rangle \mapsto a\}$$
$$\sigma(\mathtt{n}) = \{\langle\rangle \mapsto b\}$$

(Technically a state also contains the sets A and F described earlier, but we elide them from this example.) The states at V are $\{S(0, n) : n > 0\}$. Continuing in this fashion, we get the following.

$$\Sigma(V) = \{S(0, n) : n > 0\}$$
$$\Sigma(W) = \{S(i, n) : 0 \le i \le n \land n > 0\}$$
$$\Sigma(X) = \{S(i, n) : 0 \le i \le n - 1 \land n > 0\}$$
$$\Sigma(Y) = \{S(i, n) : 1 \le i \le n \land n > 0\}$$
$$\Sigma(Z) = \{S(n, n) : n > 0\}$$

The final equation allows us to prove that the assertion holds: in every state listed, i=n is true.

Abstract interpretation. Unfortunately, the analysis presented above cannot be executed on a computer since the sets $\Sigma(p)$ are infinite. Instead, we must find a way to describe any $\Sigma(p)$ using a finite representation. Let $\mathcal{S}$ be the set of all states that can arise at any program point. For example, $\mathcal{S}$ contains $S(0, 0), S(0, 1), S(1, 0), \ldots$.

We call the finite representation of some set of states an *abstract element*. The collection of all abstract elements is called an *abstract domain*, denoted D. There are many possible

abstract domains. In this section, we use as an example a very simple domain whose job it is to decide whether i is negative, positive, or zero.

There are five abstract elements in $D = \{\texttt{i:}-, \texttt{i:0}, \texttt{i:}+, \top, \bot\}$. The first three elements mean that i is negative, zero, or positive. The $\top$ element means that i can have any sign; its value is unknown. The $\bot$ element will be described later. We can map every element of D to the set of states that it represents. We use the notation $\gamma(e)$ to denote the states represented by abstract element e.

$$\gamma(\texttt{i:}-) = \{S(i,n) : i < 0, n \in \mathbb{Z}\}$$
$$\gamma(\texttt{i:0}) = \{S(0,n) : n \in \mathbb{Z}\}$$
$$\gamma(\texttt{i:}+) = \{S(i,n) : i > 0, n \in \mathbb{Z}\}$$
$$\gamma(\top) = \{S(i,n) : i, n \in \mathbb{Z}\}$$
$$\gamma(\bot) = \emptyset$$

To generate a mapping in the other direction, from $2^{\mathcal{S}}$ to D, we approximate. Given a set of states $S \subseteq \mathcal{S}$, we map it to some element e such that $S \subseteq \gamma(e)$. That is, it might map to an element that represents additional states.[1] We call this map α. For example, $\alpha(\Sigma(V))$ is the abstract element i:0 even though $\gamma(\texttt{i:0})$ is bigger than $\Sigma(V)$ (because $\gamma(\texttt{i:0})$ places no constraints on n). Similarly, $\Sigma(Y)$ and $\Sigma(Z)$ map to i:+. At W and X we are forced to map to $\top$, because i can be either zero or positive.

This form of approximation (called over-approximation because we allow a bigger set to represent a smaller one) is sound because any errors present in the program will still be found in the abstraction. Consider assertion checking. Suppose we approximate the set of states entering an assert statement with an element e. If every state in $\gamma(e)$ passes the assertion, then, since the set of states that can actually reach the assertion is a subset of $\gamma(e)$, there can be no assertion failure.

In order for this formalism to make sense, we require D to be a partial order. We write the order as $\sqsubseteq$. For our example domain the order is shown below.

$$
\begin{array}{c}
\top \\
\diagup \ \mid \ \diagdown \\
\texttt{i:}- \quad \texttt{i:0} \quad \texttt{i:}+ \\
\diagdown \ \mid \ \diagup \\
\bot
\end{array}
$$
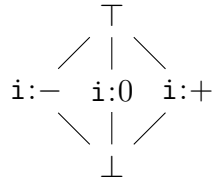
This means that $\bot \sqsubseteq \texttt{i:0} \sqsubseteq \top$. We require that if $e \sqsubseteq e'$, then $\gamma(e) \subseteq \gamma(e')$. Then, when choosing $\alpha(S)$, we pick the one lowest in the order that still over-approximates S. That is why, in the examples above, we did not use $\top$ unless necessary. However, we are required

---

[1]Ideally, we would like to be able to create a one-to-one mapping between sets of states and their finite representations. Unfortunately, these sets have different size. The set D must be at most countably infinite, since all its elements are themselves finite. The set of all sets of states is the power set of $\mathcal{S}$. Since $\mathcal{S}$ is infinite, the power set is uncountably infinite.

to have the $\top$ element in case none of the other elements is large enough to abstract a given set of states, as was the case for W and X.

If everything is defined correctly, then α and γ should form a *Galois insertion*:

$$\forall e \in D.\ \alpha(\gamma(e)) = e$$
$$\forall S \subseteq \mathcal{S}.\ \gamma(\alpha(S)) \supseteq S$$

Based on the semantics we gave earlier, a program state is a triple $\langle \sigma, A, F \rangle$. Thus, elements of $\mathcal{S}$ are triples of this form.

Transfer functions. Given an abstract element e, we need a way to compute the effect of a program statement on the element. For example, suppose that $\gamma(e)$ is the set of states $\{S(0, n) : n > 0\}$. When we execute the statement $\mathtt{i} := \mathtt{i} + 1$, we get a new set of states $\{S(1, n) : n > 0\}$. Given only the original element e and the statement $\mathtt{i} := \mathtt{i} + 1$, we need to compute a new element e' that approximates the new set of states.

There is no way to do this in general. We require each domain to supply a function, called $T(e, s)$, that computes the effect of an assignment s on an element e. In our example sign domain, we define T as follows. Only assignments to $\mathtt{i}$ have any effect at all. The assignment $\mathtt{i} := c$ returns either $\mathtt{i:}-$, $\mathtt{i:}0$, or $\mathtt{i:}+$ depending on whether c is negative, zero, or positive. The assignment $\mathtt{i} := \mathtt{i} + c$ is more difficult. It depends on the original element e. If $e = \mathtt{i:}-$ and c is negative, then the new value of $\mathtt{i}$ is still negative, so we return $\mathtt{i:}-$. Similarly, if e is $\mathtt{i:}+$ and c is positive, then we return $\mathtt{i:}+$. If e is $\mathtt{i:}0$, we return $\mathtt{i:}-$ or $\mathtt{i:}+$ if c is negative or positive, respectively. In other case we must return $\top$.

In order for the transfer function T to be sound, we need it to satisfy the following.

$$\forall \langle \sigma, A, F \rangle \in \gamma(e).\ \forall \sigma', A', F'.\ \langle \sigma, A, F \rangle\ s\ \langle \sigma', A', F' \rangle \Rightarrow \langle \sigma', A', F' \rangle \in \gamma(T(e, s))$$

When this formula holds, we say that the transfer function T is *sound*.

Our handling of branches is similar to our handling of assignments. However, since branches are not of much relevance to this thesis, we omit the details.[2]

The other necessary ingredient of an abstract domain is that the partial order must be computable. We must have a computable test to tell us whether $e \sqsubseteq e'$. We also must be able to compute the least upper bound (join) of two elements in this order, which we write as $e \sqcup e'$. We will see below how these pieces are used.

In summary, we have defined a domain D, a partial order $\sqsubseteq$, a join $\sqcup$, and a transfer function T. These are the essential constituents of any abstract domain.

Analysis engine. The abstract domain defined above is not precise enough to prove the assertion, so we define a more powerful abstract domain. Elements consist of conjunctions of facts. A fact is of the form $e_1 \leq e_2$ or $e_1 < e_2$, where e_i is either an integer (like 7) or a

---

[2]We convert branches to non-deterministic choice points. We place assume statements after the choice points to enforce the branch condition. The assume statements are processed by T like any other statement.

variable (like $\mathtt{n}$). We say that a state $\langle \sigma, A, F \rangle \in \gamma(e)$ if all the facts in e are satisfied by σ. Using our notation above, the state $S(1, 4)$ satisfies $\mathtt{n} \le 4$ and $\mathtt{i} \le \mathtt{n}$ but not $\mathtt{n} \le \mathtt{i}$. To guarantee that an element is finite, we ignore facts of the form $3 \le 4$. Also, if the fact $\mathtt{n} \le 5$ is true, then we leave out facts like $\mathtt{n} \le 6$, $\mathtt{n} \le 7$, etc.

We perform the analysis as a fixed-point computation.[3] We process each statement via its transfer function as if we were executing the program. In the case of loops, we keep iterating the loop until we reach a fixed-point. Our analysis is monotonic (abstractions only get bigger over time), so the Knaster-Tarski theorem ensures that a fixed-point exists and that the least-fixed point is the result of iteratively applying the analysis.

We begin the analysis with the element $1 \le \mathtt{n}$ at U (refer to Figure 1.2). When we apply the transfer function for $i := 0$, we get the following element at V.

$$V : 0 \le \mathtt{i} \wedge \mathtt{i} \le 0 \wedge \mathtt{i} < \mathtt{n} \wedge 1 \le \mathtt{n}$$

The state at V is copied to W without change. The branch has no effect in this case, so the abstraction at X is the same.

The transfer function for the increment returns the following element.

$$Y : 1 \le \mathtt{i} \wedge \mathtt{i} \le 1 \wedge \mathtt{i} \le \mathtt{n} \wedge 1 \le \mathtt{n}$$

To compute the new abstract element at W, we take the join of the element at V and the element at Y.

$$W : 0 \le \mathtt{i} \wedge \mathtt{i} \le 1 \wedge \mathtt{i} \le \mathtt{n} \wedge 1 \le \mathtt{n}$$

Next we apply the transfer function for the branch to D. This generates a new element at X:

$$X : 0 \le \mathtt{i} \wedge \mathtt{i} \le 1 \wedge \mathtt{i} < \mathtt{n} \wedge 1 \le \mathtt{n}$$

We apply the increment transfer function to this element, and it loops back around again.

Continuing in this fashion, we get a succession of elements at W. After k steps:

$$W : 0 \le \mathtt{i} \wedge \mathtt{i} \le k \wedge \mathtt{i} \le \mathtt{n} \wedge 1 \le \mathtt{n}$$

This example illustrates a painful reality: the Knaster-Tarski theorem does not guarantee that we will find a least-fixed point in a finite number of iterations.

Consequently, we introduce a new requirement on all domains D. Besides a computable join operator $\sqcup$, we need a computable *widening* operator. The widening operator ∇ is an over-approximation of the join: $e \sqcup e' \sqsubseteq e \nabla e'$. Thus it is always safe to use widening instead of join. In addition, unlike the join, a succession of widenings is guaranteed to stabilize after a finite number of steps.

If we perform a widening at W of the results of step k and step $k+1$, we get the following:

$$W : 0 \le \mathtt{i} \wedge \mathtt{i} \le \mathtt{n} \wedge 1 \le \mathtt{n}$$

---

[3]We perform fixed-point iteration using the method of Bourdoncle [7].

We have simply dropped the fact that $\mathtt{i} \leq k$. We apply the transfer functions for the true branch and the increment, getting the following.

$$X : 0 \leq \mathtt{i} \wedge \mathtt{i} < \mathtt{n} \wedge 1 \leq \mathtt{n}$$
$$Y : 1 \leq \mathtt{i} \wedge \mathtt{i} \leq \mathtt{n} \wedge 1 \leq \mathtt{n}$$

Then we join Y with V to get the new value for W.

$$W : 0 \leq \mathtt{i} \wedge \mathtt{i} \leq \mathtt{n} \wedge 1 \leq \mathtt{n}$$

This is the same as the previous W, so we have found a fixed point. We are done with the loop.

Once we reach a fixed-point at a given edge, we call the abstract element there an *invariant*. As suggested by the name, an invariant is a property that will hold at that program point no matter what the inputs and no matter how many times the edge is executed. The entire abstract interpretation process amounts to finding the strongest invariant at each program point.

We now address program point Z. We apply the transfer function for the false branch to the element at W. Essentially, we are adding a constraint $\neg(i < n)$ to the element at W.

$$Z : \mathtt{n} \leq \mathtt{i} \wedge \mathtt{i} \leq \mathtt{n} \wedge 1 \leq \mathtt{n}$$

A consequence of this element is that $\mathtt{i} = \mathtt{n}$, so we have proved that the assertion always holds. This was our goal.

To summarize, the basic progress of program analysis is to iteratively execute the transfer functions, using the join operator at merge points in the control-flow graph and using widening to force the iteration to stabilize. When we reach a fixed-point, we are done. A full description of this process can be found in the work of Bourdoncle [7].

1.3   Domain Structure

Our goal is to construct a domain that is capable of inferring all the example invariants listed in the introduction, such as reachability, cardinality, and reference counting. Although no existing domain is able to infer *all* of these invariants, *some* domains can infer *some* of these invariants. Since we would like to reuse previous work, our domain will be a combination of these domains. The existing domains of interest to us fall into two categories.

- Integer domains prove relationships between integer variables. One common domain, called polyhedra, will discover any invariant of the form $a_1 \cdot x_1 + \cdots + a_n \cdot x_n \leq c$, where each x_i is a program variable and the a_is and c are constants.

- Heap domains prove properties of data structures, such as linked lists and binary trees. These domains are less commonly used because they are more complex, more brittle, and run more slowly.

Mixing these domains is difficult. An integer domain can infer integer properties, but it is incapable of reasoning about data structures at all. Consider this program.

```
1  type List;
2  global List_data[List]:int;
3
4  procedure test(p:List)
5    n:int
6  {
7    n := 10;
8    List_data[p] := 10;
9  }
```

Any integer domain easily proves the invariant $n = 10$ at line 7, but no integer domain can prove anything relevant to line 8. Heap domains have a similar drawback—they cannot prove anything about integers.

One solution to this problem is to simultaneously analyze a program using both an integer domain and a heap domain. This approach would allow us to infer both integer and heap invariants. Unfortunately, we would never infer any invariants that require both integer and heap reasoning. For example, consider the index invariant from the introduction. There, an array called `table` points to heap-allocated objects. Each object has an `index` field. We want to ensure that if $\texttt{table}[i] = obj$, then $obj.\texttt{index} = i$. Formally, we write the following (ignoring array bounds for now):

$$\mathsf{Inv1} := \forall i{:}\mathbb{Z}.\ \forall obj{:}\mathbb{H}.\ \texttt{table}[i] = obj \Rightarrow (obj.\texttt{index} = i \vee obj = \texttt{null}). \tag{1.1}$$

$\mathsf{Inv1}$ quantifies over both heap objects and integers. Such quantified invariants over mixed domains are beyond the power of existing abstract domains, even when two are used simultaneously.

The ingredient that is missing from this approach is communication. Each time one domain infers a fact, it must communicate it to the other domain, which may make use of the fact. Together, the two domains can infer mixed invariants with both heap and integer components.

To facilitate the communication between domains, we construct a *combined domain*. This domain relies on an integer and a heap domain. These are called the *base domains*.
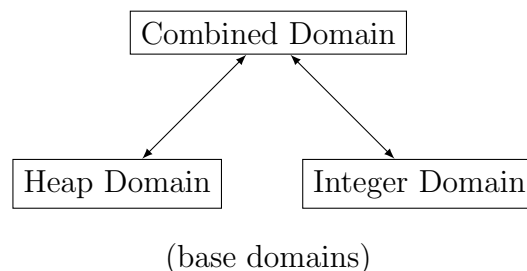
(base domains)

Both base domains have been modified to communicate information to the combined domain whenever they infer a new invariant. The combined domain immediately forwards the new invariants to the other domain.

1.4 Base Domains

As our heap domain we have chosen canonical abstraction, as first implemented in the TVLA tool [46]. However, we have heavily modified this domain, as described in Chapter 2. For our integer domain we use difference-bound matrices [36]. We also have made many changes to this domain, as described in Chapter 4. This section briefly explains the rationale for starting with these two domains.

1.4.1 Heap Domain

There are, at present, only two techniques for automatically inferring heap invariants: canonical abstraction as implemented in TVLA [46] and separation logic [4]. In both techniques, the user provides descriptions of the data structures to be analyzed. Confusingly, the descriptions are called "predicates" in both systems, although they mean different things. In both cases, the analysis infers invariants in terms of the predicates specified by the user. Providing sophisticated predicates is more work, but it leads to stronger invariants.

TVLA predicates are written in first-order logic with transitive closure. The predicates specify properties of data structures that might be of interest. For example, the following predicate specifies a doubly linked list property.

$$\mathsf{Inverse}(n{:}\mathbb{H}) := n.\texttt{next}.\texttt{prev} = n$$

Another property of interest for lists is whether their nodes have multiple incoming pointers.

$$\mathsf{Shared}(n{:}\mathbb{H}) := \exists n_1{:}\mathbb{H}.\ \exists n_2{:}\mathbb{H}.\ n1.\texttt{next} = n \wedge n_2.\texttt{next} = n \wedge n_1 \neq n_2$$

TVLA domain elements are conjunctions of these predicates (or their negations). For example, an element might state that a list satisfies Double and never satisfies Shared.

Separation logic predicates are written recursively. A single separation logic predicate will typically describe an entire data structure. For example, a predicate describing a singly linked list is as follows. (The $\star$ operator is similar to conjunction.)

$$\mathsf{SLL}(x{:}\mathbb{H}) := (x = \texttt{null}) \vee (\exists y{:}\mathbb{H}.\ (x.\texttt{next} = y) \star \mathsf{SLL}(y))$$

Like in TVLA, a separation logic domain element is essentially a conjunction of predicates. The primary difference from TVLA is that each object in the heap can be referenced only once across all predicates. This is how the $\star$ operator differs from conjunction—it implicitly states that the two operands are "separate," that they describe disjoint portions of the heap.

This provision is often useful in reasoning about separation logic formulas. However, it makes it nearly impossible to decompose a data structure description into multiple predicates. Instead, a single predicate is used for the entire data structure.

We chose to use TVLA as our heap abstraction because its predicates are more composable. In TVLA, the description of a complex data structure is the conjunction of many predicates, usually simple ones. In separation logic, a complex data structure must be described using one complex predicate.

Composability is important for two reasons. The simpler reason is that it allows us to describe a complex data structure by combining predicates from simpler structures. For example, a chained hash table can be described using a set of array predicates and a set of linked list predicates.

However, we have a more important reason for favoring composability. When used inside of the combined domain, a heap domain shares information with the integer domain through its predicates. As it discovers that more predicates are true, it sends them to the integer domain. To be effective, the heap domain must communicate each bit of information as it learns it. Separation logic predicates are too coarse-grained to be useful for this purpose. TVLA predicates, which constrain a much smaller portion of the data structure, work well.

We do not wish to argue in this section that TVLA is strictly better than separation logic. It is often slower, less scalable, and more complex to implement. We chose TVLA mainly because it cooperates well with the combination domain. §2.7 has a much more detailed discussion of the strengths and weaknesses of the two approaches.

1.4.2 Integer Domain

There are many integer domains of varying performance and expressiveness. Perhaps the most commonly used is the domain of polyhedra [2]. This domain infers conjunctions of linear inequalities of the form $a_1 \cdot x_1 + \cdots + a_n \cdot x_n \leq c$.

We chose, instead, to implement the domain of difference-bound matrices [36]. This domain infers more restricted facts of the form $x_1 - x_2 \leq c$. That is, it bounds the difference between two variables. Either variable can be missing, permitting invariants like $x \geq c$ and $x \leq c$.

We chose this domain mainly because it is simple to implement while still being sufficiently powerful to infer all the invariants in our test programs. Since we have made many changes to the domain to make it cooperate with the combined domain, simplicity of implementation was very important to us. Another benefit of this domain over polyhedra is that it performs better. Eventually, though, we expect that a more powerful domain will be needed. We expect that most of the changes proposed in this thesis should apply equally well to other, more powerful integer domains.

1.5   Thesis Structure

The main goal of the thesis is to analyze the data structures in real-world systems programs for memory safety. In particular, we want to develop an analysis that can infer invariants like the ones in the introduction. For an application, we focus on the cache module of the `thttpd` web server. We have divided the problem into several parts.

- First, using the framework of abstract interpretation [15], we design a parameterized *combined abstract domain*, into which two existing base abstract domains can be "plugged." The most important aspect of our combined domain is that it supports quantification in both the base domains. The purpose of the combined domain is to share facts between the base domains. As a consequence, the responsibility for analyzing a program is interleaved between the two base domains as they share facts. We describe the combined domain in Chapter 3.

- In order to use our combined domain, we need good base domains. We have selected TVLA [46] as our heap domain for its composability. However, TVLA has several drawbacks. It is not known to scale well and it is also difficult to use. Our heap domain is actually a heavily modified version of TVLA, designed to correct these problems. Chapter 2 describes the heap domain. Chapter 4 explains the (minimal) adaptations to make this domain work well with the combined domain.

- For our integer domain, we use a very simple relational domain based on difference matrices [36]. It is very easy to integrate this domain with our combined domain. However, the result is not very powerful. To achieve the full potential of the combined system, we must improve the integer domain. As the third piece of the thesis, we augment our integer domain to support quantified reasoning and cardinality. These improvements, described in Chapter 4 allow it to reason about index invariants and reference counting.

Based on these pieces, we identify three goals for the thesis. The first is to make a working combined domain that reasons effectively about the invariants of interest to us. The second goal is to improve TVLA so that it is sufficiently scalable and easy to use when analyzing `thttpd`. The final goal is to augment the integer domain to the point that it can reason about complex cardinality and reference counting invariants. In Chapter 5 we present our `thttpd` experiments and in Chapter 6 we discuss the goals in the context of the experiments.

Chapter 2

Heap Domain

2.1 Introduction

This chapter describes the heap domain used in DESKCHECK. This domain is based on the Three-Valued-Logic Analyzer (TVLA) [47]. In this chapter, we review the main ideas of TVLA and present some improvements to the basic algorithm. The improvements fall into four categories.

① We use the technique of finite differencing to generate transfer functions automatically. We extend existing finite differencing research [40] to be more precise and to handle arbitrary uses of transitive closure. (§2.3)

② TVLA requires frequent evaluations of first-order logic formulas. We develop an efficient algorithm for evaluating formulas similar to the way that a relational database optimizes and executes queries. (§2.4)

③ TVLA includes a phase called *sharpening*. Given an abstract program state, it makes the state more precise without changing its concrete denotation. To implement sharpening, TVLA uses a syntactic mechanism that sometimes requires manual intervention. We develop a new, semantic algorithm for sharpening and prove that, in a limited sense, it is complete. (§2.5)

④ TVLA uses a disjunctive abstraction, which occasionally leads to state-space explosion. We continue to use a disjunctive abstraction, but we reduce the need for disjunction in the common case. As an example, TVLA typically requires three disjuncts to represent a singly linked list. Our domain is able to represent the same list using only one disjunct. (§2.6)

To distinguish our domain from vanilla TVLA, we call ours "the heap domain." Both domains implement a form of canonical abstraction [47]. In the remainder of the chapter, we review TVLA (§2.2) and present the extensions above. Related work is presented in §2.7.

2.2 TVLA Overview

This overview of TVLA is an adaptation of material from Sagiv, Reps, and Wilhelm [47]. We first present the concrete semantics and then we describe the abstraction. To describe the concrete semantics, we informally show how to convert a concrete heap to its TVLA representation. We completely ignore the existence of integers throughout this section.

TVLA views each heap object as a *node.* Thus, we generate a node for every object in the concrete heap. Every variable and field is represented by an uninterpreted function. TVLA converts these functions to predicates. For every function $f : T_1 \times T_2 \to T_3$, it generates a predicate $\mathsf{F} : T_1 \times T_2 \times T_3$. If $f(x, y) = z$, then $\mathsf{F}(x, y, z)$ holds; otherwise $\mathsf{F}(x, y, z)$ is false. Predicates that represent functions are called *core predicates.*[1]

Besides core predicates, TVLA also allows the user to define a set of *instrumentation predicates.* Each instrumentation predicate is defined in the language of first-order logic with transitive closure. Formally stated, this language is as follows.

$$
\begin{aligned}
\varphi := \; & \mathsf{P}(x, y, \ldots) && \text{atomic facts} \\
\mid \; & x = y && \text{equality} \\
\mid \; & \neg\varphi && \text{negation} \\
\mid \; & \varphi_1 \wedge \varphi_2 && \text{conjunction} \\
\mid \; & \varphi_1 \vee \varphi_2 && \text{disjunction} \\
\mid \; & \forall v.\; \varphi && \text{universal quantification} \\
\mid \; & \exists v.\; \varphi && \text{existential quantification} \\
\mid \; & \mathrm{TC}(s, t; x, y).\; \varphi && \text{transitive closure}
\end{aligned}
$$

In the last line, φ is expected to have x and y as free variables. If we interpret it as a function, $\varphi(x, y)$, then $(\mathrm{TC}(s, t; x, y).\; \varphi)$ holds if there is a sequence of nodes $n_1, n_2, \ldots, n_k$ such that $s = n_1$, $t = n_k$, and $\varphi(n_i, n_{i+1})$ holds for $i = 1$ to $k - 1$. (Since $k = 1$ is legal, we are using reflexive transitive closure.)

Example 3 We might define an instrumentation predicate as follows.

$$\mathsf{SharedViaNext}(n) := \exists n_1.\; \exists n_2.\; \mathsf{Next}(n_1, n) \wedge \mathsf{Next}(n_2, n) \wedge \neg(n_1 = n_2)$$

This predicate holds at a node if there are multiple incoming *next* pointers, such in as $next(n_1) = n$ and $next(n_2) = n$. □

An instrumentation predicate may have arbitrary arity, including zero. One predicate may, in its definition, refer to another predicate as an atomic fact. We only require that predicates have no cyclic dependencies.

---

[1] In TVLA, core predicates are more general than functions. In our system, all core predicates are functions.

Given a concrete heap, we can evaluate all the instrumentation predicates at all possible places to see where they hold. Since any given heap is finite, there is no difficulty in evaluating quantifiers. Since there is no abstraction, the answer is always precise.

This forms a complete picture of the information stored in a TVLA heap: a set of nodes and the predicates that hold among them. The predicate facts are simply a conjunction of atomic literals, which have the form $\mathsf{P}(n_1, n_2, \ldots)$ or $\neg\mathsf{P}(n_1, n_2, \ldots)$. The combination of nodes and predicate information is called a *structure* in TVLA.

Example 4 Consider a linked list of three elements, headed by a pointer *head* and linked together by *next* pointers. Call the three nodes n_1, n_2, and n_3. Additionally, there is a "null" node z. We have *head* $= n_1$ and *null* $= z$. For $i \in \{1, 2\}$, *next*$(n_i) = n_{i+1}$; also, *next*$(n_3) = z$. Since functions are required to be total, we say that *next*$(z) = z$. Converting these functions to predicates, we get the following.

$$\mathsf{Head}(n_1) \wedge \mathsf{Null}(z) \wedge \mathsf{Next}(n_1, n_2) \wedge \mathsf{Next}(n_2, n_3) \wedge \mathsf{Next}(n_3, z) \wedge \mathsf{Next}(z, z)$$

There are also many negated facts.

$$\neg\mathsf{Head}(n_2) \wedge \neg\mathsf{Head}(n_3) \wedge \cdots \wedge \neg\mathsf{Next}(z, n_3)$$

In addition, we can evaluate the $\mathsf{SharedViaNext}$ predicate defined above.

$$\neg\mathsf{SharedViaNext}(n_1) \wedge \neg\mathsf{SharedViaNext}(n_2) \wedge \cdots \wedge \neg\mathsf{SharedViaNext}(z)$$

We can also define other predicates.

$$\mathsf{SharedViaHead}(n) := \mathsf{Head}(n) \wedge \exists n'.\ \mathsf{Next}(n', n)$$
$$\mathsf{HeadReaches}(n) := \exists n'.\ \mathsf{Head}(n') \wedge \mathrm{TC}(n', n; v, v').\ \mathsf{Next}(v, v')$$

The first predicate holds of a node if it has an incoming *next* pointer and it is also the head. The second predicate holds if it is reachable from the head via *next* edges. If we evaluate these predicates, we see that $\mathsf{SharedViaHead}$ is false at all nodes and $\mathsf{HeadReaches}$ is true at all nodes. □

It quickly becomes awkward to write out all these predicate values. Hence, we adopt a graphical notation.
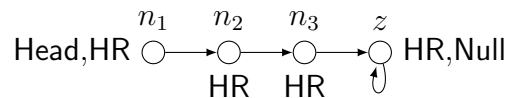
Nodes are circles. Unary predicates are written next to the nodes that satisfy them. We abbreviate $\mathsf{HeadReaches}$ as HR. The Next predicate is shown using edges. If a predicate value is not shown (such as $\mathsf{Next}(n_1, n_3)$ or $\mathsf{SharedViaHead}(n_1)$) then it is false.

2.2.1   Abstraction

Call the structure above L_3 because it represents a three-node list. In a program analysis, the goal is to represent all possible data structures that might occur at a given point in the program. If this set consisted of lists of size 1, 2, or 3, then we could represent this state as a disjunction of structures, $L_1 \vee L_2 \vee L_3$. But what about the more likely case of a list of arbitrary length? This would require an infinite disjunction, which cannot be represented in a computer. Hence, we need *abstraction*. Abstraction is the process of converting something arbitrarily large into something that is guaranteed to have finite size.

Since disjunctions must be finite, we need a single structure that can represent an arbitrary number of nodes. To do this, we introduce the concept of a *summary node*, which represents one or more concrete nodes. Nodes that are not summary nodes are called *singleton nodes*. Using summary nodes, we can begin to construct a structure that represents linked lists of many different sizes.
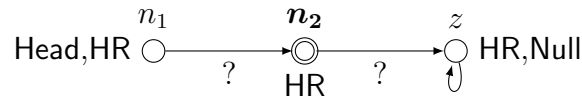
Here, $\boldsymbol{n_2}$ is a summary node. It is written in bold and drawn with a double circle as a reminder.

We are now faced with a conundrum: should the edges shown with question marks be true or false? $\boldsymbol{n_2}$ potentially represents many concrete nodes. n_1 connects to the first one, but not to the rest. Similarly, only the last node represented by $\boldsymbol{n_2}$ connects to z.

To resolve the problem, we allow predicate values to take on the value $1/2$, meaning "don't know." We set $\mathsf{Next}(n_1, \boldsymbol{n_2}) = 1/2$ because some of the nodes represented by $\boldsymbol{n_2}$ have an edge from n_1 but not all of them. The same goes for $\mathsf{Next}(\boldsymbol{n_2}, z)$. We are also forced to set $\mathsf{Next}(\boldsymbol{n_2}, \boldsymbol{n_2}) = 1/2$. The reason is that every node represented by $\boldsymbol{n_2}$ has a *next* edge to its successor node, which may also be represented by $\boldsymbol{n_2}$.

We can now redraw the diagram above as follows. Dashed lines show when a Next predicate value is $1/2$.
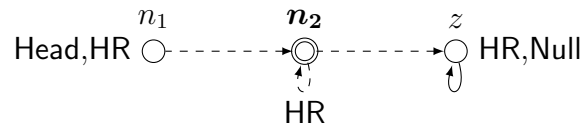
Now we can define a TVLA structure more formally.

Definition 1 A structure S is a tuple, $\langle U, \iota \rangle$. U is the set of nodes and ι is the interpretation of predicates. For each predicate P of arity k, $\iota(\mathsf{P}) : U^k \to \{0, 1, 1/2\}$. 0 means the predicate value is false and 1 means true.   □

We use a clever trick to distinguish summary nodes from singleton nodes. We create a binary predicate, $\mathsf{Eq}(n, n')$, that holds when $n = n'$. For a singleton node n, $\mathsf{Eq}(n, n) = 1$.

For a summary node n, $\mathsf{Eq}(n, n) = 1/2$. Why not 1? Because there may be concrete nodes n_1 and n_2 represented by n and they are not the same. We can track the Eq predicate using ι just like we do any other predicate.

We distinguish between *concrete structures*, which never contain 1/2 predicate values, and *three-valued structures*, which may contain 1/2. We would like to state formally which concrete structures are represented by a three-valued structure. Whenever a structure S' represents another structure S, we say that S *embeds* into S' and write $S \sqsubseteq S'$.

Example 5 Using the same structure as Example 4, we would like to show the following.
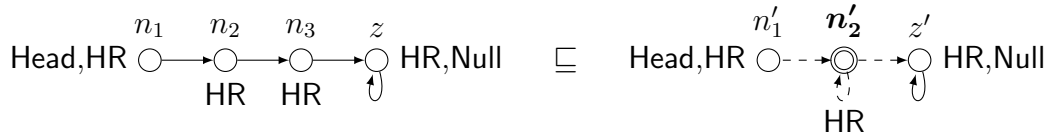
We call the left structure $S = \langle U, \iota \rangle$ and the right structure $S' = \langle U', \iota' \rangle$. The first step in showing that one structure embeds inside another is to match up the nodes. We define a function f from U to U'. In this case, it should be intuitive how the nodes match up.

$$f(n_1) = n'_1 \qquad f(n_2) = \boldsymbol{n'_2} \qquad f(n_3) = \boldsymbol{n'_2} \qquad f(z) = z$$

Then we check that the predicates in the two structures match up. Whenever $\mathsf{P}(n) = 1$ in S, we require that $\mathsf{P}(f(n)) = 1$ or 1/2 in S'. Similarly, whenever a predicate is 0 in S, it must be 0 or 1/2 in S'.

Consider the value of $\iota'(\mathsf{Next})(n'_1, \boldsymbol{n'_2})$. There are two relevant constraints.

$$1 = \iota(\mathsf{Next})(n_1, n_2) \sqsubseteq \iota'(\mathsf{Next})(n'_1, \boldsymbol{n'_2}) = 1/2$$
$$0 = \iota(\mathsf{Next})(n_1, n_3) \sqsubseteq \iota'(\mathsf{Next})(n'_1, \boldsymbol{n'_2}) = 1/2$$

Some values are less constrained. If we consider $\iota'(\mathsf{Next})(n'_1, n'_1)$, the only constraint we get is that $0 = \iota(\mathsf{Next})(n_1, n_1) \sqsubseteq \iota'(\mathsf{Next})(n'_1, n'_1) = 0$.

We also consider the unary predicates, such as $\mathsf{HeadReaches}(\boldsymbol{n'_2})$.

$$1 = \iota(\mathsf{HeadReaches})(n_2) \sqsubseteq \iota'(\mathsf{HeadReaches})(\boldsymbol{n'_2}) = 1$$
$$1 = \iota(\mathsf{HeadReaches})(n_3) \sqsubseteq \iota'(\mathsf{HeadReaches})(\boldsymbol{n'_2}) = 1$$

We omit the other predicate checks, but we do examine the Eq predicate. The condition that the predicates match up says that if n is a summary node in S, then it must map to a summary node in S' (i.e., if $\iota(\mathsf{Eq})(n, n) = 1/2$ then $\iota'(\mathsf{Eq})(f(n), f(n)) = 1/2$). Additionally, if $f(n_1) = f(n_2) = n'$, where n_1 and n_2 are distinct nodes in U, then n' must be a summary node. □

Before giving the formal definition of embedding, we need to define an ordering on $\{0, 1, 1/2\}$. This order formalizes our intuitive notion that 0 and 1 are "more specific" than $1/2$.

$$0 \sqsubseteq 0 \quad 1 \sqsubseteq 1 \quad 1/2 \sqsubseteq 1/2$$
$$0 \sqsubseteq 1/2 \quad 1 \sqsubseteq 1/2$$

We call 0 and 1 "definite values" and call $1/2$ an "indefinite value."

Definition 2 Given two TVLA structures, $S = \langle U, \iota \rangle$ and $S' = \langle U', \iota' \rangle$, we say that S *embeds* into S' if there exists a surjection $f : U \to U'$ (called the *embedding function*) that satisfies the following. For any predicate P,

$$\iota(\mathsf{P})(n_1, \ldots, n_k) \sqsubseteq \iota'(\mathsf{P})(f(n_1), \ldots, f(n_k)). \qquad \square$$

Now we can define the meaning of a three-valued structure in terms of concrete structures. $\gamma(S)$ is the set of concrete structures represented by a three-valued structure S.

$$\gamma(S) = \{S_0 : S_0 \text{ is concrete} \wedge S_0 \sqsubseteq S\}$$

Some embeddings are more precise than others. For example, in Example 5, we could have instead chosen all the predicate values in S' to be $1/2$ and S still would have embedded into it. But given a structure S and an embedding function f, f induces a most precise structure S' such that $S \sqsubseteq S'$. We can characterize S' as follows (where $\sqcup$ is the least upper bound of the $\sqsubseteq$ relation).

$$\iota'(\mathsf{P})(n_1', \ldots, n_k') = \bigsqcup_{f(n_i) = n_i'} \iota(\mathsf{P})(n_1, \ldots, n_k)$$

This formulation identifies all relevant constraints of the form $v \sqsubseteq \iota'(\mathsf{P})(n_1', \ldots, n_k')$ and then takes the least upper bound over all these v.

Canonical abstraction. The preceding material on embeddings offers a way to collapse nodes of a structure together in the most precise possible way (the induced embedding). However, it does not offer any insight in choosing which nodes to collapse (i.e., in choosing f). Ultimately, this choice is a heuristic, but the following technique is easy to understand and works well in practice.

We identify a subset of predicates $\mathcal{A}$ called *abstraction predicates*. These predicates must be unary. Then we form an equivalence relation on nodes. We write $n \equiv n'$ if, for every $\mathsf{P} \in \mathcal{A}$, $\iota(\mathsf{P})(n) = \iota(\mathsf{P})(n')$.

Then we collapse equivalent nodes. More formally, we define $f_\equiv(n) = [n]_\equiv$. That is, a node n is mapped to its equivalence class in $\equiv$. This ensures that $f_\equiv(n) = f_\equiv(n')$ if and only if $n \equiv n'$.

Another way to think of this process is in terms of *canonical names*. Assume that $\mathcal{A} = \{\mathsf{P}_1, \mathsf{P}_2, \ldots, \mathsf{P}_m\}$. Then the canonical name of a node n is a sequence,

$$\langle \iota(\mathsf{P}_1)(n), \iota(\mathsf{P}_2)(n), \ldots, \iota(\mathsf{P}_m)(n) \rangle.$$

Nodes having the same canonical name are collapsed to the same node.

Example 6 In the preceding example, we would have obtained the given f if $\mathcal{A}$ were $\{\mathsf{Head}, \mathsf{Null}\}$. Adding any of the sharing or reachability predicates above to $\mathcal{A}$ would not have made a difference, because they all evaluate to the same value on n_2 and n_3. Using $\mathcal{A} = \emptyset$ would have merged all nodes into a single node.

We could have defined a predicate like the following.

$$\mathsf{Second}(n) = \exists n'. \; \mathsf{Head}(n') \wedge \mathsf{Next}(n', n)$$

This predicate is true on n_2 and false on n_3. The choice $\mathcal{A} = \{\mathsf{Head}, \mathsf{Second}, \mathsf{Null}\}$ would result in no abstraction at all. Assuming the ordering of predicates given here, the canonical names would be:

$$n_1 : \langle 1, 0, 0 \rangle \qquad n_2 : \langle 0, 1, 0 \rangle \qquad n_3 : \langle 0, 0, 0 \rangle \qquad z : \langle 0, 0, 1 \rangle \qquad \square$$

Summary nodes. Examining γ yields a deeper understanding of summary nodes. Let S' be a structure with one summary node, n. Assume there are no predicates besides Eq. Let S be some concrete structure such that $S \sqsubseteq S'$. That is, let $S \in \gamma(S')$. Since $\iota'(\mathsf{Eq})(n, n) = 1/2$, Definition 2 imposes no constraints on predicate values in S. The only constraint at all is that there must be some surjection mapping nodes of S to nodes of S'. If S has no nodes, then no such function can exist. If it has at least one node, then there is such a function (it maps all nodes of S to n).

Consequently, a summary node in an abstract structure must abstract at least one node in a concrete structure. This restriction is useful because it makes some reasoning simpler. However, it also causes problems, which we will address in §2.6.

Joining structures. The embedding relation defines a partial order on structures. We write the least upper bound (or *join*) of this order as $\sqcup$. This operation has the property that $\gamma(S \sqcup S') \supseteq \gamma(S) \cup \gamma(S')$. However, it is not always possible to join two structures. As we saw above, there is no structure S such that $\gamma(S)$ includes both an empty structure and a structure with one or more nodes. Thus, if S_0 is the empty structure and S_{1+} is a structure with a single summary node, it simply is not possible to join S_0 with S_{1+}.

Nevertheless, it is possible to join some structures. If there exists a bijection m mapping the nodes of S to the nodes of S', then we define the join as follows. Let $S = \langle U, \iota \rangle$ and let $S' = \langle U', \iota' \rangle$ (so that m maps U to U'). Then $S \sqcup S' = \langle U, \iota'' \rangle$, where for every predicate P,

$$\iota''(\mathsf{P})(u_1, \ldots, u_k) = \iota(\mathsf{P})(u_1, \ldots, u_k) \sqcup \iota'(\mathsf{P})(m(u_1), \ldots, m(u_k)).$$

Note that both S and S' embed into the joined structure—S via the identity embedding function and S' via m^{-1}.

Since there is no bijection between nodes of S_0 and nodes of S_{i+1}, this join does not work here. (In fact, the existence of a bijection is not a necessary condition for the existence of a join. However, we only perform joins of structures where a bijection exists.)

Domain. For the reasons just mentioned, a single TVLA structure may not be able to represent all the states that arise at a given point in a program. Consequently, we use a disjunction of structures, $S_1 \vee \cdots \vee S_n$. We will define a domain for abstract intepretation [13] where these disjunctions of TVLA structures are the domain elements. The meaning of a domain element is defined as follows.

$$\gamma(S_1 \vee S_2 \vee \cdots \vee S_k) = \gamma(S_1) \cup \gamma(S_2) \cup \cdots \cup \gamma(S_k)$$

We still must define (1) a least upper bound on the domain elements (a join), (2) a partial order on domain elements, and (3) a way of approximating the effect of a program statement on a domain element. To distinguish the partial order and join of domain elements from those over TVLA structures, we write the domain element versions as $\sqsubseteq$ and $\sqcup$.

Before describing these operations, we place a restriction on domain elements. The purpose of this restriction is to ensure that a domain element is always finite in size. We first make the observation that any structure that satisfies the canonical abstraction is finite: there is at most one node for any given canonical name, and there are $2^{|\mathcal{A}|}$ canonical names.

However, the number of disjuncts in the domain element can still grow without bound. To prevent this, we first say that the canonical name of a structure is the set of canonical names of its nodes. So the canonical name of the structure in Example 6 is

$$\{\langle 1, 0, 0\rangle, \langle 0, 1, 0\rangle, \langle 0, 0, 0\rangle, \langle 0, 0, 1\rangle\}.$$

Here, then, is the restriction: a domain element should not contain two structures with the same canonical names. If it does, we join them together via $\sqcup$ into a single structure. The bijection we use for the join matches nodes with the same canonical name.

This completes the definition of the TVLA domain. The domain elements are disjunctions of structures, where each structure has at most one node with a given canonical name, and where no two disjuncts have the same set of canonical node names. We say that

$$S_1 \vee \cdots \vee S_k \sqsubseteq S'_1 \vee \cdots \vee S'_k$$

if for each S_i there is an S'_j with the same canonical node names such that $S_i \sqsubseteq S'_j$. The join operation $\sqcup$ first forms the element $S_1 \vee \cdots \vee S_k \vee S'_1 \vee \cdots \vee S'_k$ and then merges, via $\sqcup$, any two disjuncts with the same set of canonical names.

We have already informally described an algorithm for converting a concrete heap into a three-valued structure by computing predicate values as in Example 4. Let β be the function

that runs this algorithm and then computes the canonical abstraction (i.e., merges nodes with the same canonical name). Given this β, we form α as follows.
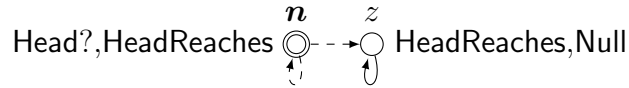
$$\alpha(\mathcal{S}) = \bigsqcup \{\beta(e) : e \in \mathcal{S}\}$$

The γ function we have already seen. The $\bot$ element in TVLA is an empty disjunction. The next section explains the transformer for assignment.

Example 7 Consider the set $\mathcal{S}$ of all acyclic singly linked lists of zero or more elements. We will compute $\alpha(\mathcal{S})$. The empty list is represented by a single null node, z, as follows.

$$\overset{z}{\bigcirc} \; \mathsf{Head,HeadReaches,Null}$$

Using the abstraction predicates $\mathcal{A} = \{\mathsf{Null}\}$, we represent the other cases using a single summary node for all list elements, as follows. The question mark means that $\mathsf{Head}(\boldsymbol{n}) = 1/2$.

$$\mathsf{Head?,HeadReaches} \overset{\boldsymbol{n}}{\circledcirc}\!\dashrightarrow\!\overset{z}{\bigcirc} \mathsf{HeadReaches,Null}$$

If the first structure is S_0 and the second is S_{1+}, then the domain element we desire is $S_0 \vee S_{1+}$. □

The disjunction of these two structures represents *exactly* the set of acyclic singly linked lists. So in this case, $\gamma(\alpha(\mathcal{S})) = \mathcal{S}$. (In general we are guaranteed only that $\gamma(\alpha(\mathcal{S})) \supseteq \mathcal{S}$.) The example illustrates the importance of instrumentation predicates. Without the $\mathsf{HeadReaches}$ predicate, it would be possible for an unreachable node to be abstracted by $\boldsymbol{n}$—essentially forming floating garbage. Similarly, the $\mathsf{SharedViaHead}$ and $\mathsf{SharedViaNext}$ predicates (not shown because they are false everywhere) exclude the possibility of cycles.

2.2.2 Transformers

This section discusses how the abstraction is updated due to program statements. We focus on the assignment operation; other operations work similarly. Assignment is divided into four stages.

Focus An assignment statement may reference objects that are represented by summary nodes in the current abstraction. Operating directly on summary nodes usually leads to imprecision, so the focus operation splits the node in question off from the summary node. This stage is called *materialization* in some papers.

Sharpen Abstraction "forgets" some information about predicates like Next, but it retains the information indirectly in instrumentation predicates like HeadReaches and SharedViaNext. When a node is materialized, the Next information about it may still be imprecise. However, we can use the instrumentation predicates to recover this information. This process is called *sharpening* (or, in some texts, *coerce*).

Update This step actually performs the assignment. Suppose it has the form $f(e) := e'$. The first step is to update the F predicate. Instrumentation predicates may depend on F; these predicates are updated afterwards using a process called finite differencing.

Blur The focus step usually breaks the canonical abstraction because the node it materializes has the same canonical name as the original summary node. The *blur* step restores the canonical abstraction by merging nodes that have the same canonical name.

These steps are illustrated in the following example.

Example 8 Consider the abstract structure from Example 5. It represents a list of two or more elements. We will analyze the statement *head := next[head]*. We consider the four stages mentioned above in turn.
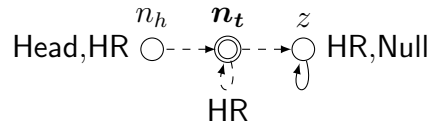
In the **focus** stage, the analysis ensures that both *head* and *next[head]* are represented by singleton nodes. Since $\mathsf{Head}(n_h) = 1$, and n_h is a singleton node, there is no need to do anything for *head*. However, *next[head]* is some object abstracted by the summary node $\boldsymbol{n_t}$ (we know this because $\mathsf{Next}(n_h, \boldsymbol{n_t}) = 1/2$). Thus, we need to materialize $\boldsymbol{n_t}$.

Intuitively, we split $\boldsymbol{n_t}$ into two nodes, n_{t1} and $\boldsymbol{n_{t2}}$. Both nodes satisfy the same set of predicates as $\boldsymbol{n_t}$ did with three exceptions: n_{t1} is a singleton node, and $\mathsf{Next}(n_h, n_{t_1}) = 1$ while $\mathsf{Next}(n_h, \boldsymbol{n_{t_2}}) = 0$. This yields the following structure.
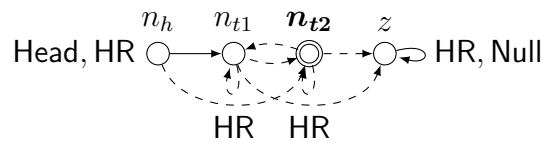
However, the split ignores the possibility that n_t could represent only a single concrete node. In that case, we should not create $\boldsymbol{n_{t_2}}$ at all. Thus, we get a second structure.
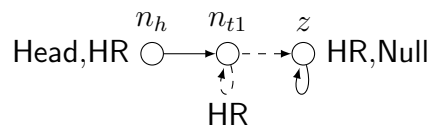
We call these two structures S_2 and S_1, respectively. The result of the focus operation is $S_2 \vee S_1$.

Next comes the **sharpen** operation. It is applied independently to S_2 and S_1. We consider S_1 first because it is simpler. The only indefinite information in this structure are the facts $\mathsf{Next}(n_{t1}, n_{t1}) = 1/2$ and $\mathsf{Next}(n_{t1}, z) = 1/2$, meaning that it is not known whether n_{t1} points to itself or to z. In fact, though, n_{t1} cannot point to itself. To see why, recall the definition of $\mathsf{SharedViaNext}$.

$$\mathsf{SharedViaNext}(n) := \exists n_1.\ \exists n_2.\ \mathsf{Next}(n_1, n) \wedge \mathsf{Next}(n_2, n) \wedge \neg(n_1 = n_2)$$

Because $\mathsf{SharedViaNext}(n_{t1}) = 0$, there cannot be multiple incoming *next* pointers to n_{t1}. Therefore, we can *sharpen* $\mathsf{Next}(n_{t1}, n_{t1})$ to 0. Since *next* is a total function, we can then sharpen $\mathsf{Next}(n_{t1}, z)$ to 1. The resulting structure S_1' is as follows.

We can apply the same techniques to S_2. Using exactly the same reasoning, we recognize that $\mathsf{Next}(n_{t1}, n_{t1}) = 0$ and $\mathsf{Next}(\boldsymbol{n_{t2}}, n_{t1}) = 0$. We also consider the fact $\mathsf{Next}(n_{t1}, z) = 1/2$. We can prove this to be false as follows: if it were true, then $\boldsymbol{n_{t2}}$ would not be reachable from *head*, contradicting the $\mathsf{HeadReaches}(\boldsymbol{n_{t2}}) = 1$ fact. Hence, we obtain the following.
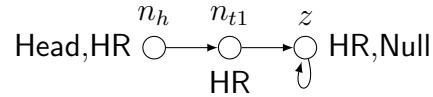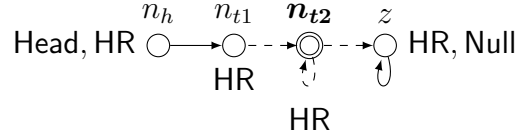

We call the two structures resulting from sharpen S_1' and S_2'.

Next we perform the **update** phase. Recall that the statement is *head* := *next*[*head*]. First the core predicate Head is updated: $\mathsf{Head}(n_h)$ is updated to 0 and $\mathsf{Head}(n_{t1})$ is updated to 1. Next, the instrumentation predicates are updated. The details of this process are described later. The result is that $\mathsf{HeadReaches}(n_h)$ is updated to 0. The result for S_2' is below.
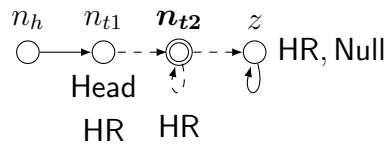
Finally the **blur** phase executes. Its operation depends on the abstraction predicates. If $\mathcal{A} = \{\mathsf{Head}, \mathsf{Null}\}$, then n_h and $\boldsymbol{n_{t2}}$ will be merged together, since neither of them satisfy either abstraction predicate. It's unlikely that this is a desirable outcome; when performing list iteration, it's usually best to keep the nodes already seen separate from those yet to be visited. Thus, we choose $\mathcal{A} = \{\mathsf{Head}, \mathsf{Null}, \mathsf{HeadReaches}\}$. Then no merging will occur. $\quad\Box$

We will consider the update and sharpen operations in greater detail in the next sections, where we propose modifications to them.

2.3 Finite Differencing

The previous section presented an overview of the TVLA approach to shape analysis. The final section of the overview explained how an assignment statement is processed in four phases: focus, sharpen, update, and blur. The update phase is the one that actually implements the assignment. It does so via a process called *finite differencing*, which was described by Reps, Sagiv, and Loginov [40].

This section presents the first contribution of this thesis: an extension to make finite differencing more precise. (This contribution was number ① in the bulleted list in the introduction.) We first present an overview of finite differencing and then describe our extensions starting in Section 2.3.2.

Finite differencing updates the instrumentation predicates according to how the core predicates are affected by the assignment. As an example, suppose we have the predicate $P(x) := A(x) \vee B(x)$, where A and B are core predicates. If $A(n)$ is updated from 0 to 1, and if $B(n)$ remains 0, then we must update $P(n)$ from 0 to 1 as well.

We could, of course, simply re-evaluate all the instrumentation predicates after updating the core predicates. However, the purpose of instrumentation predicates is to retain information about summary nodes that has been lost in the core predicates. As we will see, if we simply re-evaluate them we get an imprecise answer for summary nodes.

Instead, we generate a formula for the "derivative" of an instrumentation predicate. The derivative formula evaluates to true only in places where the original predicate would be affected by the assignment. As long as the nodes relevant to the change are fully materialized, the core predicates will contain sufficient information to get precise results for the instrumentation predicates.

Given a formula φ, the goal is to compute two difference formulas: φ^+ and φ^-. φ^+ evaluates to 1 if changes in the core predicates cause the value of φ to change from 0 to 1. φ^- evaluates to 1 holds if core predicate changes cause φ to change from 1 to 0. We use the terminology that φ "goes up" or "goes down" in these cases.

To be a bit more precise, φ^+ is 1 if φ was 0 in every concrete structure before the assignment and if it is 1 in every concrete structure after the assignment. φ^+ is 1/2 if φ changes from 0 to 1 in some, but not all, structures. And φ^+ is 0 if φ never goes from 0 to 1 in any concrete structure. Similar statements hold for φ^-.

If φ references predicate $A(x)$ as an atomic literal, then φ^+ and φ^- will reference $A^+(x)$ and $A^-(x)$. These formulas evaluate to 1 whenever $A(x)$ goes up or goes down. The next example explains the process.

Example 9 Let A, B, and C be core predicates. Assume that $P(x) := A(x) \vee B(x)$ as above and that $Q(x) := P(x) \wedge C(x)$. Let there be three nodes, n_1, n_2, and $\boldsymbol{n_3}$ (the last one is a summary node). Initially, the predicates are as follows.

	A	B	C	P	Q
n_1	1	0	1	1	1
n_2	0	0	0	0	0
n_3	1/2	1/2	1	1	1

$$\mathsf{P}(x) := \mathsf{A}(x) \vee \mathsf{B}(x)$$
$$\mathsf{Q}(x) := \mathsf{P}(x) \wedge \mathsf{C}(x)$$

Notice that the facts about P and Q store information that could not be obtained by evaluating their defining formulas (shown on the right): although it is not known whether A or B hold at n_3, the $\mathsf{P}(n_3) = 1$ fact requires that each concrete node abstracted by n_3 satisfies either A or B.

Suppose an assignment changes A so that it goes down at n_1 and up at n_2. If we simply re-evaluate P and Q everywhere, we get the following. We show a change from v to v' as in $v \to v'$. No arrow means no change.

	A	B	C	P	Q
n_1	$1 \to 0$	0	1	$1 \to 0$	$1 \to 0$
n_2	$0 \to 1$	0	0	$0 \to 1$	0
n_3	1/2	1/2	1	$1 \to 1/2$	$1 \to 1/2$

The re-evaluation of P and Q at n_3 causes imprecision. Previously, P preserved the fact that either A or B is true at every node abstracted by n_3. Despite the fact that nothing about n_3 is changed by the assignment, we lose this information in the re-evaluation.

Although it is tempting not to re-evaluate P at n_3 since nothing changed there, this does not work in general: if the defining formula for P had contained quantifiers, then failing to re-evaluate would be unsound.

Instead, we use differencing to compute a more precise answer for P and Q everywhere. The first step is to translate the assignment into differences in the core predicates. We can compute the differences directly based on the old and new values of A, B, and C. We only show the values for A, since B and C are not affected by the assignment (meaning that their derivatives are zero everywhere).

	A	A$^+$	A$^-$
n_1	$1 \to 0$	0	1
n_2	$0 \to 1$	1	0
n_3	1/2	0	0

It is important to point out one subtlety of differencing for the core predicates. We can set A$^+$ and A$^-$ to zero only because A is not affected at *any* of the concrete nodes abstracted by n_3. If there is a change at some n_3 such that $\mathsf{A}(n_3)$ remains 1/2, we must set both $\mathsf{A}^+(n_3)$ and $\mathsf{A}^-(n_3)$ to 1/2.

The changes to P, written $\mathsf{P}^+(n)$ and $\mathsf{P}^-(n)$, are computed by evaluating formulas in our logic. Later in the section will will show how to generate these formulas; for now it suffices to say that they are typical first-order logic formulas that depend on A$^+$, A$^-$, B$^+$, and B$^-$ as well as the old values of A and B. Evaluating them yields the following. Ignore the last column.

	A	B	C	P	A^+	A^-	P^+	P^-	$P'(x)$
n_1	$1 \to 0$	0	1	1	0	1	0	1	0
n_2	$0 \to 1$	0	0	0	1	0	1	0	1
$\boldsymbol{n_3}$	$1/2$	$1/2$	1	1	0	0	0	0	1

Using the values for $P^+(n)$ and $P^-(n)$ we can directly compute the new value of P, written P'. It is defined as

$$P'(x) := P^+(x) \vee (P(x) \wedge \neg P^-(x)).$$

Notice that both P^+ and P^- are zero at $\boldsymbol{n_3}$, meaning that no update is required there. This allows us to retain the information saved in P there.

Next we move on the $Q(x) := P(x) \wedge C(x)$. Q^+ and Q^- are defined in terms of the current values of P and C as well as the differences. However, we have all this information since the differences to P have just been computed. In general, it is always possible to order the updates to instrumentation predicates as long as there are no cyclic dependencies. □

Comparison. We can compare our approach to finite differencing with TVLA's. First we review our approach. Given an assignment operation, we compute changes to the core predicates such as A, above. We temporarily store these changes in the heap structure as $\iota(A^+)$ and $\iota(A^-)$. Then we evaluate the difference formulas P^+ and P^- to compute changes to the instrumentation predicate P. These formulas will depend on A^+ and A^-, so we will look up $\iota(A^+)$ and $\iota(A^-)$ during evaluation. We store the evaluation results in $\iota(P^+)$ and $\iota(P^-)$ in case another instrumentation predicate, like Q, depends on P.

To fully illustrate the idea, we show evaluation rules for finite differencing formulas (Table 2.1). Let $[\![\varphi]\!]_{S,A}$ be the result of evaluating φ over three-valued structure S. The assignment A binds free variables to nodes.

φ	$[\![\varphi]\!]_{S,A}$ (assume $S = \langle U, \iota \rangle$)
$P(v_1, \ldots, v_k)$	$\iota(P)(A(v_1), \ldots, A(v_k))$
$P^+(v_1, \ldots, v_k)$	$\iota(P^+)(A(v_1), \ldots, A(v_k))$
$P^-(v_1, \ldots, v_k)$	$\iota(P^-)(A(v_1), \ldots, A(v_k))$
$x = y$	$\iota(Eq)(A(x), A(y))$
$\neg \varphi'$	$\neg\, [\![\varphi']\!]_{S,A}$
$\varphi_1 \wedge \varphi_2$	$[\![\varphi_1]\!]_{S,A} \wedge [\![\varphi_2]\!]_{S,A}$
$\varphi_1 \vee \varphi_2$	$[\![\varphi_1]\!]_{S,A} \vee [\![\varphi_2]\!]_{S,A}$
$\forall v.\ \varphi'$	$\bigwedge_{n \in U} [\![\varphi']\!]_{S,A[v \to n]}$
$\exists v.\ \varphi'$	$\bigvee_{n \in U} [\![\varphi']\!]_{S,A[v \to n]}$

Table 2.1: Semantics of finite difference formulas.

The main difference between TVLA and our system is that TVLA handles differencing *syntactically* while we do it *semantically*. TVLA never stores differences in ι. Instead, it

substitutes one difference formula into another, so that difference formulas do not depend on other differences. The following example illustrates the approach.

Example 10 Suppose that the change to A above was caused by the statement "$a :=$ *next*[a]." Considering only the syntax of this statement, TVLA will generate *formulas* for A^+ and A^-.

$$\mathsf{A}^+(n) := \exists n'. \, \mathsf{A}(n') \wedge \mathsf{Next}(n', n)$$
$$\mathsf{A}^-(n) := \mathsf{A}(n) \wedge \neg\mathsf{Next}(n, n)$$

(The final clause in the down formula is needed in case a points to itself.) TVLA will then syntactically substitute these formulas into the difference formulas it generates for P, so that P^+ and P^- do not include any reference to differences in A or any other formula. □

Our system computes updates "on-demand" given an arbitrary set of changes to core predicates. In contrast, TVLA examines each assignment before the analysis runs and computes customized difference formulas for each instrumentation predicate. TVLA's approach has the advantage that the syntactic substitution may reveal opportunities for simplifying the difference formula that our system would not discover. Typically, though, it results in larger formulas that take longer to evaluate. Additionally, the up-front cost of generating and storing update formulas for every statement in the program can be burdensome.

2.3.1 Difference Formulas

The previous section described how difference formulas are *used*. This section describes how they are *generated*. Differencing is syntax-directed. We begin by computing P^+ and P^-. First consider the up formula: $\mathsf{P}^+(x) = (\mathsf{A}(x) \vee \mathsf{B}(x))^+$. P will go up if either A goes up or B goes up. But we can be more precise than this. If A goes up when B was already equal to 1, then P will not go up. We can formalize this insight as follows.

$$\mathsf{P}^+(x) = (\mathsf{A}^+(x) \wedge \neg\mathsf{B}(x)) \vee (\neg\mathsf{A}(x) \wedge \mathsf{B}^+(x))$$

This formula is evaluated in the "old" state, before any predicates are updated by the assignment statement. Thus, $\mathsf{A}(x)$ does not account for any changes to A from the assignment.

Now we consider $\mathsf{P}^-(x) = (\mathsf{A}(x) \vee \mathsf{B}(x))^-$. If $\mathsf{A}(x)$ goes down, then $\mathsf{P}(x)$ will go down as well, but only if the new value of $\mathsf{B}(x)$ is 0. Similarly, P will go down if B goes down and A is zero in the future. In order to formalize this notion, we need a way to express "the future value of $\mathsf{A}(x)$." We use the *future operator*, denoted $\mathbf{F}[\varphi]$. Normally φ is evaluated in the old state, but $\mathbf{F}[\varphi]$ evaluates φ in the updated state. This can be implemented easily as long as all the predicates mentioned in φ have already been updated. Since we only use $\mathbf{F}[\cdot]$ on sub-expressions of P, this requirement will always be satisfied.

Using the future operator, we can generate P^- based on the intuition above.

$$\mathsf{P}^-(x) = (\mathsf{A}^-(x) \wedge \neg\mathbf{F}[\mathsf{B}(x)]) \vee (\neg\mathbf{F}[\mathsf{A}(x)] \wedge \mathsf{B}^-(x))$$

To formally define the semantics of the future operator, we need to extend the evaluation rules in Table 2.1. Now we must evaluate formulas over *two* structures: the original one S and the updated one S'. We let $[\![\varphi]\!]_{S,S',A}$ be the result of the evaluation. Table 2.2 shows the updated rules.

φ	$[\![\varphi]\!]_{S,S',A}$ (assume $S = \langle U, \iota \rangle$)
$\mathsf{P}(v_1, \ldots, v_k)$	$\iota(\mathsf{P})(A(v_1), \ldots, A(v_k))$
$\mathsf{P}^+(v_1, \ldots, v_k)$	$\iota(\mathsf{P}^+)(A(v_1), \ldots, A(v_k))$
$\mathsf{P}^-(v_1, \ldots, v_k)$	$\iota(\mathsf{P}^-)(A(v_1), \ldots, A(v_k))$
$x = y$	$\iota(\mathsf{Eq})(A(x), A(y))$
$\mathbf{F}[\varphi']$	$[\![\varphi']\!]_{S',S',A}$
$\neg\varphi'$	$\neg\, [\![\varphi']\!]_{S,S',A}$
$\varphi_1 \wedge \varphi_2$	$[\![\varphi_1]\!]_{S,S',A} \wedge [\![\varphi_2]\!]_{S,S',A}$
$\varphi_1 \vee \varphi_2$	$[\![\varphi_1]\!]_{S,S',A} \vee [\![\varphi_2]\!]_{S,S',A}$
$\forall v.\ \varphi'$	$\bigwedge_{n \in U} [\![\varphi']\!]_{S,S',A[v \to n]}$
$\exists v.\ \varphi'$	$\bigvee_{n \in U} [\![\varphi']\!]_{S,S',A[v \to n]}$

Table 2.2: Semantics of the future operator.

Using the same intuition appearing above, we present the complete finite differencing rules for all types of formulas in Table 2.3. Rather than describe them in detail, we give a few examples to show how they are used.

Example 11 In the following examples we depict formulas as trees that grow to the right. Consider the new formula $\mathsf{R}(a_0) := \exists b.\ \mathsf{E}(a_0, b) \wedge \mathsf{C}(b)$. We show this formula as follows.

This formula has a typical structure. It might mean: R holds of a node n if n points to another object (via the field edge E) where C holds.

We can compute the derivatives of R using Table 2.3 (and a few simplifications to be described).

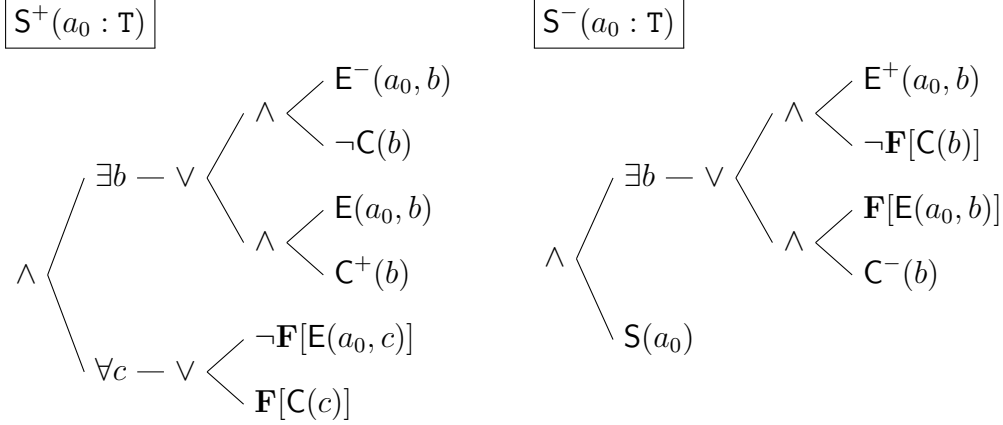| φ	φ^+	φ^-
true, false	false	false
$\mathsf{P}(x)$	$\mathsf{P}^+(x)$	$\mathsf{P}^-(x)$
$x = y$	false	false
$\neg\varphi'$	φ'^-	φ'^+
$\varphi_1 \wedge \varphi_2$	$(\varphi_1^+ \wedge \mathbf{F}[\varphi_2]) \vee (\mathbf{F}[\varphi_1] \wedge \varphi_2^+)$	$(\varphi_1^- \wedge \varphi_2) \vee (\varphi_1 \wedge \varphi_2^-)$
$\varphi_1 \vee \varphi_2$	$(\varphi_1^+ \wedge \neg\varphi_2) \vee (\neg\varphi_1 \wedge \varphi_2^+)$	$(\varphi_1^- \wedge \neg\mathbf{F}[\varphi_2]) \vee (\neg\mathbf{F}[\varphi_1] \wedge \varphi_2^-)$
$\exists x.\ \varphi'$	$(\exists x.\ \varphi'^+) \wedge (\neg\exists x.\ \varphi')$	$(\exists x.\ \varphi'^-) \wedge (\neg\exists x.\ \mathbf{F}[\varphi'])$
$\forall x.\ \varphi'$	$(\exists x.\ \varphi'^+) \wedge (\forall x.\ \mathbf{F}[\varphi'])$	$(\exists x.\ \varphi'^-) \wedge (\forall x.\ \varphi')$

Table 2.3: Finite differencing rules.
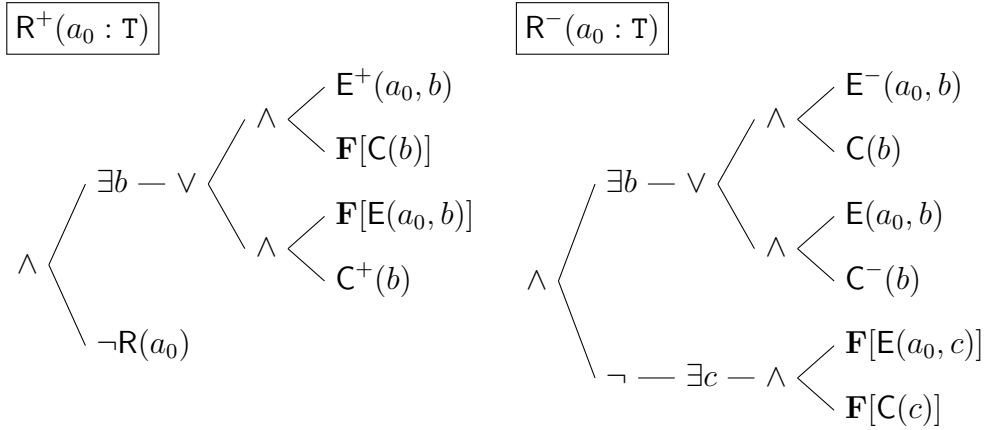
Consider the left formula. The bottom branch says that for R to go up, it must have been zero before. According to the top branch, it will go up if a new E edge to some node is created and if C is true there in the future state. It will also go up if C goes up at a node that has an incoming E edge from a_0.

Now consider the right formula. The bottom branch says that for R to go down, it must be false when re-evaluated the future state. The top branch provides an additional condition. Either there must be a node where C was true and E went down, or where E was true and C went down. The top branch ensures that the existential quantifier goes down somewhere where it originally held; the bottom branch ensures that there are no other places where it still holds.

We make two notes about these formulas. At the top level, R^+ has two branches; according to the rules in the table, the bottom one should be $\neg\exists y.\ \mathsf{E}(a_0, y) \wedge \mathsf{C}(y)$. However, we recognize that this expression contains the defining formula for R, so we replace it with $\mathsf{R}(a_0)$. This makes formulas smaller and more efficient to evaluate as well as more precise.

Secondly, the formula for R^- should contain $\mathbf{F}[\exists y.\ \mathsf{E}(a_0, y) \wedge \mathsf{C}(y)]$. However, it is always valid to push the $\mathbf{F}[\cdot]$ operator down to atomic formula, as we have done. Having $\mathbf{F}[\cdot]$ only apply to atomic formulas simplifies our algorithms.   □
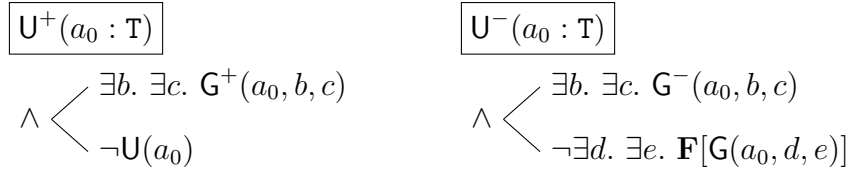
Example 12 Now consider another typical formula, $\mathsf{S}(x) := \forall y.\ \mathsf{E}(x, y) \Rightarrow \mathsf{C}(y)$. We obtain the following derivatives.

$$\boxed{\mathsf{S}^+(a_0 : \mathsf{T})} \qquad\qquad \boxed{\mathsf{S}^-(a_0 : \mathsf{T})}$$

Left tree:
- $\exists b - \vee$:
  - $\wedge$: $\mathsf{E}^-(a_0, b)$, $\neg\mathsf{C}(b)$
  - $\wedge$: $\mathsf{E}(a_0, b)$, $\mathsf{C}^+(b)$
- $\forall c - \vee$: $\neg\mathbf{F}[\mathsf{E}(a_0, c)]$, $\mathbf{F}[\mathsf{C}(c)]$

Right tree:
- $\exists b - \vee$:
  - $\wedge$: $\mathsf{E}^+(a_0, b)$, $\neg\mathbf{F}[\mathsf{C}(b)]$
  - $\wedge$: $\mathbf{F}[\mathsf{E}(a_0, b)]$, $\mathsf{C}^-(b)$
- $\mathsf{S}(a_0)$

On the left, the top branch says that for S to go up, an E edge to some node where C is false must go down. Or C must go up at a node with an incoming E edge. Additionally, the bottom branch re-evaluates S in the future state.

The meaning of the down derivative should be clear by now. □

Example 13 The following example shows another small divergence from Table 2.3. Let $\mathsf{U}(x) := \exists y.\ \exists z.\ \mathsf{G}(x, y, z)$. We get the following derivatives.

$$\boxed{\mathsf{U}^+(a_0 : \mathsf{T})} \qquad\qquad \boxed{\mathsf{U}^-(a_0 : \mathsf{T})}$$

Left tree:
- $\wedge$: $\exists b.\ \exists c.\ \mathsf{G}^+(a_0, b, c)$, $\neg\mathsf{U}(a_0)$

Right tree:
- $\wedge$: $\exists b.\ \exists c.\ \mathsf{G}^-(a_0, b, c)$, $\neg\exists d.\ \exists e.\ \mathbf{F}[\mathsf{G}(a_0, d, e)]$

In the presence of chained quantifiers (of the form $\exists a.\ \exists b.\ \cdots \exists z.\ \varphi$ or $\forall a.\ \forall b.\ \cdots \forall z.\ \varphi$) we apply the quantifier rule from Table 2.3 only once. □

2.3.2 Improved Precision

The formulas given above are correct. However, they often yield imprecise results even in the common case.

Example 14 Consider, again, the predicate $\mathsf{P}(x) := \mathsf{A}(x) \vee \mathsf{B}(x)$. Let the initial values be $\mathsf{A}(n) = 0$ and $\mathsf{B}(n) = 1/2$. Assume that A goes up to 1 at n. Clearly the final value of $\mathsf{P}(n)$ should be 1. However, using the difference formula we get the following.

$$
\begin{aligned}
\mathsf{P}^+(n) &= (\mathsf{A}^+(n) \wedge \neg\mathsf{B}(n)) \vee (\neg\mathsf{A}(n) \wedge \mathsf{B}^+(n)) \\
&= (1 \wedge 1/2) \vee (1 \wedge 0) \\
&= 1/2
\end{aligned}
$$

Using this information, we are forced to update $\mathsf{P}(n)$ to be $1/2$. □

The problem here is that the formulas given above are, in some sense, too precise. We get $\mathsf{P}^+(n) = 1/2$ because, in a concrete structure where $\mathsf{B}(n) = 1$, $\mathsf{P}(n)$ was already 1 and so $\mathsf{P}(n)$ does not actually go up.

More generally, each up formula contains two pieces: a check that the new value is 1 and a check that the old value was 0. The latter check ensures that we only update a predicate when it is actually changing. Unfortunately, if the previous value of a predicate was $1/2$, then we can never say definitively that it goes up or down because we are unsure of its old value.

We would prefer to take a looser interpretation of the "did it change?" check. To do so, we introduce a new operator, $\mathbf{1}[\cdot]$, called the *definite operator*. Its job is to promote $1/2$ values to 1. The value of $\mathbf{1}[\varphi]$ is 0 if φ is 0 and 1 if φ is $1/2$ or 1.

Using the definite operator, we can refine the difference formula for disjunction as follows.
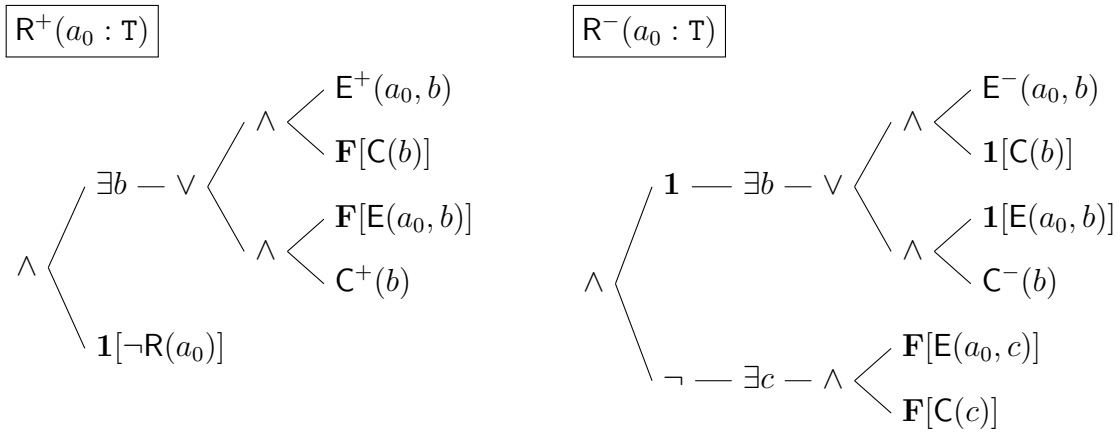
$$(\varphi_1 \vee \varphi_2)^+ = (\varphi_1^+ \wedge \mathbf{1}[\neg\varphi_2]) \vee (\mathbf{1}[\neg\varphi_1] \wedge \varphi_2^+)$$

Using values from the previous example, $\mathsf{P}^+(n)$ now equals 1, as desired. The following table gives the same rules as Table 2.3 but modified to be more precise.

φ	φ^+	φ^-
$\varphi_1 \wedge \varphi_2$	$(\varphi_1^+ \wedge \mathbf{F}[\varphi_2]) \vee (\mathbf{F}[\varphi_1] \wedge \varphi_2^+)$	$(\varphi_1^- \wedge \mathbf{1}[\varphi_2]) \vee (\mathbf{1}[\varphi_1] \wedge \varphi_2^-)$
$\varphi_1 \vee \varphi_2$	$(\varphi_1^+ \wedge \mathbf{1}[\neg\varphi_2]) \vee (\mathbf{1}[\neg\varphi_1] \wedge \varphi_2^+)$	$(\varphi_1^- \wedge \neg\mathbf{F}[\varphi_2]) \vee (\neg\mathbf{F}[\varphi_1] \wedge \varphi_2^-)$
$\exists x.\ \varphi'$	$(\exists x.\ \varphi'^+) \wedge (\mathbf{1}[\neg\exists x.\ \varphi'])$	$(\mathbf{1}[\exists x.\ \varphi'^-]) \wedge (\neg\exists x.\ \mathbf{F}[\varphi'])$
$\forall x.\ \varphi'$	$(\mathbf{1}[\exists x.\ \varphi'^+]) \wedge (\forall x.\ \mathbf{F}[\varphi'])$	$(\exists x.\ \varphi'^-) \wedge (\mathbf{1}[\forall x.\ \varphi'])$

Table 2.4: More precise finite differencing rules.

Example 15 To see the difference, compare these derivatives for $\mathsf{R}(x) := \exists y.\ \mathsf{E}(x, y) \wedge \mathsf{C}(y)$ to the ones in Example 11.

The only change to the up derivative is that the check that R was zero before (the lower branch) is inside $\mathbf{1}[\cdot]$. The down derivative may seem a bit more complex. However, the inner definite operators have no effect, since they are nested inside the outer one. The purpose of the outer one is to ensure that if R is 0 in the future state, then R^- is 1 even if a change only *maybe* occurred. □

Comparison. In TVLA the imprecision explained above is acceptable because it is eventually "fixed up" by the sharpening operation, which TVLA runs after every transformer. However, sharpening is the most expensive operation in the analysis. Using the formulas in Table 2.4 gives precise results, allowing us to sharpen much less frequently.

2.3.3 Transitive Closure

So far we have not discussed finite differencing rules for formulas that use transitive closure. We do so now. First, assume that $\psi := (\mathrm{TC}(s, t; x, y).\ \varphi(x, y))$. Sometimes we write $\psi(\cdot, \cdot)$ to allow substitution of other variables for s and t. We wish to generate ψ^+ and ψ^-. The easiest way to do this is instead to compute the new value of ψ, call it ψ', and then derive ψ^+ and ψ^- from it.

The changes that can affect ψ are either φ going up or down. Thus, we could simply write $\psi' = (\mathrm{TC}(s, t; x, y).\ \mathbf{F}[\varphi])$, but this is not very precise. In many cases, ψ will be known definitely even when no information about φ is available. Thus, we want a formula that reuses ψ whenever possible. The solution is to stitch together a path from s to t consisting of sub-paths from x to y where either (1) $\varphi^+(x, y)$ holds, or (2) $\psi(x, y)$ holds and is "trusted," meaning that no ψ^- edges can affect it.

Example 16 To see what this means, consider the following example.
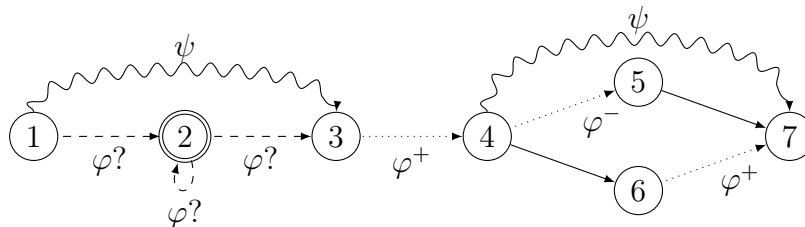
In this figure, an unlabeled solid edge means that φ holds between the two nodes. A dotted edge means that φ goes up or down, depending on the label. A dashed edge means that φ is 1/2 in the original structure. Finally, a squiggle edge means that ψ held in the original state. Node 2 is a summary node while the other nodes are singletons. The goal is to compute where ψ holds in the future.

Between nodes 1 and 3, ψ was known to hold even though φ may not hold in between. Since φ does not go down anywhere between 1 and 3, the existing ψ edge is considered "trusted," so we can be sure that ψ holds in the future as well.

Between nodes 4 and 7 the situation is more difficult. The ψ edge from 4 to 7 cannot be trusted: φ goes down between 4 and 5, and the path originally witnessed by the ψ edge may depend on φ between 4 and 5. However, φ is known to hold between 4 and 6, and φ goes up between 6 and 7. So ψ still holds between 4 and 7 in the future, although it uses a different path of φ edges than it did previously.

Using these inferences, we can stitch together a path from node 1 to node 7: first the trusted ψ edge from 1 to 3 is used, followed by the φ^+ edge to 4, the φ edge to 6 that doesn't go down, and finally the φ^+ edge to 7. □

We formalize the logic behind this example as follows. We will use the transitive closure operator to stitch together our path. We will close over a new formula, φ_{new}, to be defined.

$$\psi'(s,t) := \text{TC}(s,t;x,y).\ \varphi_{\text{new}}(x,y)$$

$\varphi_{\text{new}}(x,y)$ should hold either if φ goes up or if there is a trusted ψ edge.

$$\varphi_{\text{new}}(x,y) := \varphi^+(x,y) \vee \psi_{\text{trusted}}(x,y)$$

To decide whether an existing $\psi(x,y)$ is trusted, we need to know whether it might have used a φ edge that went down. This can happen only if there is a $\varphi^-(a,b)$ edge such that x reaches a and b reaches y.

$$\psi_{\text{trusted}}(x,y) := \psi(x,y) \wedge \neg \exists a,b.\ \psi(x,a) \wedge \varphi^-(a,b) \wedge \psi(b,y)$$

This fully defines ψ'. However, we also need ψ^+ and ψ^- in case ψ is used inside some other formula. We could simply say that $\psi^+ = \psi' \wedge \mathbf{1}[\neg\psi]$. However, this will return 1 even when no change has taken place. We can make it more precise by requiring that φ go up somewhere. For ψ to go up between s and t, there must be a path from s to a place where φ goes up, and then a path from there to t. The path must be along existing ψ edges or else edges where φ goes up. We formalize such a path with ψ_Δ.

$$\psi_\Delta(x,y) = \text{TC}(x,y;a,b).\ \psi(a,b) \vee \varphi^+(a,b)$$
$$\psi^+(s,t) = \psi'(s,t) \wedge \mathbf{1}[\neg\psi(s,t) \wedge \exists a,b.\ \psi_\Delta(s,a) \wedge \varphi^+(a,b) \wedge \psi_\Delta(b,t)]$$

We do essentially the same thing for ψ^-. However, we do not need ψ_Δ edges in this case—we can simply use ψ edges.

$$\psi^-(s,t) = \neg\psi'(s,t) \wedge \mathbf{1}[\psi(s,t) \wedge \exists a,b.\ \psi(s,a) \wedge \varphi^-(a,b) \wedge \psi(b,t)]$$

Comparison. The original finite differencing rules presented in Reps et al. [40] also handled transitive closure using trusted edges. However, those authors wanted to completely eliminate transitive closure from the difference equations.[2] Rather than taking the transitive closure of φ_{new}, as we do, they allow only three steps to be taken along φ_{new} edges.

---

[2]Transitive closure cannot be axiomatized in a complete way in first-order logic. Thus, eliminating transitive closure permits traditional theorem provers to be applied to shape analysis.

The "three-step rule" can be encoded using only existential quantification. However, the three-step rule supports only single-edge changes during an assignment. Our rules allow arbitrary φ edge additions and deletions. Although single-edge changes are the common case, checking that an assignment makes only a single edge change introduces a lot of complexity into the analysis. As a result, our finite differencing code is about 300 lines of ML. TVLA's is about 1,300 lines of Java.

2.4 Formula Evaluation

The previous section described the finite differencing mechanism. It is an easy, declarative way to describe how instrumentation predicates are affected by an assignment. This section describes an efficient, novel way to evaluate the difference formulas (or any formulas). This step can be costly if done naively. We borrow techniques from the database literature to make it run faster. This section is contribution ② from the introduction.

2.4.1 Semantics

Recall the semantics of queries, shown again in Table 2.5. We evaluate a formula φ over two structures: the state before an assignment, $S = \langle U, \iota \rangle$, and the state after an assignment, $S' = \langle U, \iota' \rangle$. Note that the universes are the same. Each formula may evaluate to 0, 1, or 1/2. Since φ may have free variables, we also need an *assignment*, A, which maps free variables of φ to individuals from U. We use the notation $[\![\varphi]\!]_{S,S',A}$ to mean the evaluation of φ in structures S and S' over assignment A.

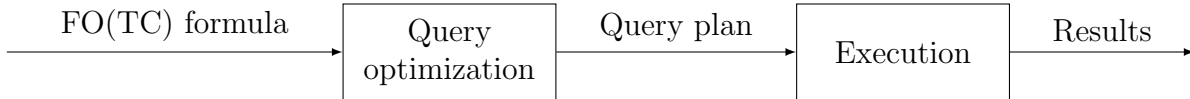| φ	$[\![\varphi]\!]_{S,S',A}$ (assume $S = \langle U, \iota \rangle$)
$\mathsf{P}(v_1, \ldots, v_k)$	$\iota(\mathsf{P})(A(v_1), \ldots, A(v_k))$
$\mathsf{P}^+(v_1, \ldots, v_k)$	$\iota(\mathsf{P}^+)(A(v_1), \ldots, A(v_k))$
$\mathsf{P}^-(v_1, \ldots, v_k)$	$\iota(\mathsf{P}^-)(A(v_1), \ldots, A(v_k))$
$x = y$	$\iota(\mathsf{Eq})(A(x), A(y))$
$\mathbf{F}[\varphi']$	$[\![\varphi']\!]_{S',S',A}$
$\neg\varphi'$	$\neg \, [\![\varphi']\!]_{S,S',A}$
$\mathbf{1}[\varphi']$	$\mathbf{1}[[\![\varphi']\!]_{S,S',A}]$
$\varphi_1 \wedge \varphi_2$	$[\![\varphi_1]\!]_{S,S',A} \wedge [\![\varphi_2]\!]_{S,S',A}$
$\varphi_1 \vee \varphi_2$	$[\![\varphi_1]\!]_{S,S',A} \vee [\![\varphi_2]\!]_{S,S',A}$
$\forall v.\ \varphi'$	$\bigwedge_{n \in U} [\![\varphi']\!]_{S,S',A[v \to n]}$
$\exists v.\ \varphi'$	$\bigvee_{n \in U} [\![\varphi']\!]_{S,S',A[v \to n]}$
$\mathrm{TC}(s, t; x, y).\ \varphi'$	$\iota(\mathsf{Eq})(A(s), A(t)) \vee \bigvee_{P \in \mathrm{Paths}(S, A(s), A(t))} \bigwedge_{\langle n_1, n_2 \rangle \in P} [\![\varphi']\!]_{S,S',A[x \to n_1, y \to n_2]}$

Table 2.5: Semantics of formulas.

The logical operators $\land$, $\lor$, and $\lnot$ appearing on right-hand side of the table are just as one would expect: $0 \land 1 = 0$, $0 \lor 1 = 1$, $0 \land 1/2 = 0$, and $0 \lor 1/2 = 1/2$. The Paths function works as follows: from the complete graph over the nodes U, it returns all subsets of edges that form a path from $A(s)$ to $A(t)$.

One can convert Table 2.5 to an evaluation algorithm in an obvious way. Unfortunately, each nested quantifier introduces an $O(|U|)$ factor to the complexity of evaluation. An assignment statement may require the evaluation of 50 or so difference formulas, each of which may have quantifiers nested two or three levels deep. Even for small programs, $|U| \sim 100$ and there may be thousands of assignment statements to analyze. Consequently, the naive algorithm is quite slow, as we will show in our experiments.

2.4.2 Introduction

The process of optimizing and executing a query is shown below. We briefly describe each step in this section.

FO(TC) formula $\longrightarrow$ [Query optimization] $\xrightarrow{\text{Query plan}}$ [Execution] $\xrightarrow{\text{Results}}$

The goal is to execute formulas from first-order logic with transitive closure (FO(TC)). To make an analogy with databases, we call these formulas *queries*. The first step is to optimize these queries, making them easier to execute. *Query execution* then generates a series of results, such as "$\varphi(x)$ is $1/2$ at $x = n_4$."

Optimizing a query generates a *query plan*. In our system, a query plan is not much different from a query except that a bit more information is included about how to execute it. Query plans are pre-computed, so the cost of optimization can usually be neglected. The goal is to make query execution as fast as possible. Since the purpose of all this machinery is to execute a query, we discuss this step first. Then we describe how queries are optimized.

Query execution. We use as an example the query $\varphi(x) := \exists y.\, \mathsf{E}(x, y) \land \mathsf{A}^+(x)$. When optimized, this query has the following query plan: $(\mathrm{find}(y).\, \mathsf{A}^+(x) \land \mathsf{E}(x, y))$. We will explain later why the query is optimized this way. The query plan differs from the original query in only two ways: the order of the conjunction has been reversed and the existential quantifier has been converted to a special "find" quantifier. For now, assume that find has the same meaning as existential quantification.

Let's execute this query plan. Since changes occur locally, $\mathsf{A}^+(x)$ will usually be true for only one or two values of x. Thus, we can start by enumerating all the x values where $\mathsf{A}^+(x)$ holds. Next, we need to execute $\mathsf{E}(x, y)$. We already have a set of bindings for x, but y is unknown. If we are smart, we keep an *index* on E that tells us, for a given x value, the set of y values where $\mathsf{E}(x, y)$ holds. Then we can extend our result set so that it contains bindings for $\langle x, y \rangle$ where query is true. Finally, the find quantifier projects out the x binding from the result. Our result is a set of x values where φ is true.

One major advantage of this approach over the naive one is that we only get results where φ is true. Most of the queries that we evaluate in practice are false almost everywhere. We save a lot of time by enumerating only the places where they evaluate to true.

Let us consider the execution of this query plan more closely. When we execute $\mathsf{A}^+(x)$, we have no information on the value of x. The result is a set of xs where $\mathsf{A}^+(x)$ is 1 or $1/2$. We can write the result as a set of pairs of the form $\langle A, v \rangle$, where A is an assignment binding x to a node and v is 1 or $1/2$.

Now consider $\mathsf{E}(x, y)$. This time we start with an initial assignment A_0 with a binding for x. We still return a set of pairs $\langle A, v \rangle$, where A contains all the bindings from A_0 as well as a binding for y. In general, when executing a query plan ψ, we start with an initial assignment A_0 and we return a set of pairs, $\langle A, v \rangle$, where A includes the same bindings as A_0 as well as bindings for the remaining free variables of ψ. v is always 1 or $1/2$.

**Query optimization.** Consider the previous query plan again.

$$\text{find}(y).\ \mathsf{A}^+(x) \wedge \mathsf{E}(x, y)$$

Suppose we execute the conjunction in the opposite order. We would enumerate places $\langle x, y \rangle$ where $\mathsf{E}(x, y)$ holds and then verify that A^+ holds there. Unfortunately, there are likely to be many more places where E holds than places where A^+ holds. Consequently, we will generate many intermediate results that are later discarded. This is inefficient. Thus, an important part of query optimization is ordering conjunctions to minimize the size of the result set at any point.

Another aspect of query optimization is quantifier conversion. One easy rule is to convert all universal quantifiers to existentials using De Morgan's law. Each existential quantifier is then converted to either a "find plan" or a "scan plan." A find plan, such as (find(x). $\psi(x)$), relies on $\psi(x)$ to enumerate x values where $\psi(x)$ is 1 or $1/2$. Its only job is to project the x bindings out of the resulting assignments.

Scan plans, of the form (scan(x). $\psi(x)$), are less efficient but sometimes necessary. They iterate over all nodes $n \in U$, executing ψ with an initial assignment where x is bound to n. Consider a query like $\exists x.\ \neg \mathsf{P}(x)$. We often store indexes that enumerate the places where predicates are 1 or $1/2$, but we have not found it profitable to index the places where predicates fail to hold. Therefore, the only way to execute the query is to scan over every node x in the structure, looking for places where $\mathsf{P}(x)$ is 0 or $1/2$. We convert such a query to a scan plan, (scan(x). $\neg \mathsf{P}(x)$). Given an initial assignment A_0, a scan quantifier executes its subquery using the initial assignment $A_0[x \mapsto n]$ for every possible node n.

Returning to the flow chart, we can fill in some details.
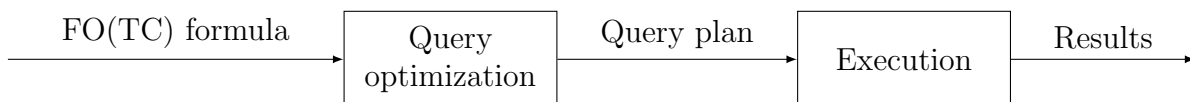
The query optimization stage chooses the best order for executing conjunctions. It also replaces existential quantifiers with either find or scan quantifiers. The result is a query plan. Given a query plan and an initial assignment A_0, the execution engine returns a set of results $\langle A, v \rangle$ containing a complete assignment A and whether the result is 1 or 1/2. The next two sections describe query execution and query optimization in more detail.

2.4.3 Query Execution

As a notational convenience, we write the result of executing query plan ψ with initial assignment A_0 as $(\psi \,?\, A_0)$. The next few paragraphs describe the execution algorithm.

Derivatives. Derivative queries are the easiest to execute. Consider $\mathsf{P}^+(x, y)$. We always keep around the complete set of tuples where a derivative holds because the set is expected to be small. Call this set D. If the initial assignment A_0 contains neither variables x or y, we simply add each tuple to A_0 and put it in the result set:

$$\mathsf{P}^+(x, y) \,?\, A_0 = \{\langle A_0[x \mapsto x', y \mapsto y'], v \rangle : \langle x', y', v \rangle \in D\}.$$

However, it may be that A_0 already contains a binding for x. Then we would return the following.

$$\mathsf{P}^+(x, y) \,?\, A_0 = \{\langle A_0[y \mapsto y'], v \rangle : \langle A_0(x), y', v \rangle \in D\}.$$

We use a similar procedure if y, or both x and y, are in A_0. In all cases, we simply iterate over D.

Atomic formulas. Imagine we are given the query plan $\mathsf{P}(x, y)$ along with an initial assignment A_0. There are several ways to execute this query. If x and y are present in A_0 then we check $\iota(\mathsf{P})(A_0(x), A_0(y))$. If it is a non-zero value v then we return the result $\{\langle A_0, v \rangle\}$. Otherwise we return an empty result.

In some cases, we use indexes to generate the result. Let the index $I_1(n)$ be a set of tuples $\langle n', v \rangle$ where $\iota(\mathsf{P})(n, n') = v$. Similarly, let I_2 be an index over y. We only keep indexes for binary predicates where one argument is known already. This is profitable because, treating the binary predicate as a graph edge, the degree of a given node is usually small. In the case that x is known in A_0, we return the following.

$$\mathsf{P}(x, y) \,?\, A_0 = \{\langle A_0[y \mapsto y'], v \rangle : \langle y', v \rangle \in I_1(A_0(x))\}.$$

When there is no index available, we scan over the entire relation $\iota(\mathsf{P})$, looking for places where it is non-zero that match A_0.

Negation. Negation is the Achilles' heel of our algorithm. Given the query plan $\neg\psi$, we are only able to return results if all the free variables of ψ are already present in A_0. Luckily, the query optimization algorithm guarantees that this condition holds.

To evaluate the negation, we recursively evaluate ψ. Because of the restriction above, there are only three possible results: $\emptyset$, $\{\langle A_0, 1\rangle\}$, and $\{\langle A_0, 1/2\rangle\}$. If we get an empty result, we return $\{\langle A_0, 1\rangle\}$. For a 1 result, we return $\emptyset$. And for the 1/2 result, we return a 1/2 result.

$$(\neg\psi)\,?\,A_0 = \begin{cases} \emptyset & \text{if } (\psi\,?\,A_0) = \{\langle A_0, 1\rangle\} \\ \{\langle A_0, 1\rangle\} & \text{if } (\psi\,?\,A_0) = \emptyset \\ \{\langle A_0, 1/2\rangle\} & \text{if } (\psi\,?\,A_0) = \{\langle A_0, 1/2\rangle\} \end{cases}$$

Definite operator. Given the query plan $\mathbf{1}[\psi]$, we recursively evaluate ψ. Then we iterate over its results $\langle A, v\rangle$, converting a 1/2 values in v to 1.

$$\mathbf{1}[\psi]\,?\,A_0 = \{\langle A, 1\rangle : \langle A, v\rangle \in (\psi\,?\,A_0)\}$$

Conjunctions. We assume that the query optimizer has already decided which conjunct should be executed first. So when given a query plan $\psi_1 \wedge \psi_2$, we first compute $R_1 = (\psi_1\,?\,A_0)$. Then for each resulting tuple $\langle A_1, v_1\rangle$ in R_1, we compute $(\psi_2\,?\,A_1)$. If it returns a tuple $\langle A_2, v_2\rangle$, then we return a tuple $\langle A_2, v_1 \wedge v_2\rangle$. More formally,

$$(\psi_1 \wedge \psi_2)\,?\,A_0 = \{\langle A_2, v_1 \wedge v_2\rangle : \langle A_1, v_1\rangle \in (\psi_1\,?\,A_0) \wedge \langle A_2, v_2\rangle \in (\psi_2\,?\,A_1)\}.$$

Disjunction. We handle disjunctions in a fairly obvious way: we execute both subqueries with the same initial assignment and then merge the results together. However, we have to be careful when merging. For convenience, we define a function to merge results together. Given a set S, possibly containing more than one entry for a given assignment A, merge(S) will return a single result set that is the disjunction of its inputs.

$$\begin{aligned} \text{merge}(S) = \ & \{\langle A, 1\rangle : \langle A, 1\rangle \in S\} \\ & \cup \{\langle A, 1/2\rangle : \langle A, 1/2\rangle \in S \wedge \langle A, 1\rangle \notin S\} \end{aligned}$$

Then we handle disjunctions as follows.

$$(\psi_1 \vee \psi_2)\,?\,A_0 = \text{merge}((\psi_1\,?\,A_0) \cup (\psi_2\,?\,A_0))$$

Finds. To execute a find query $(\text{find}(x).\ \psi)$, we recursively execute ψ. It will enumerate values of x satisfying the query. We project away the x binding from these results and then merge them together using the merge function.

$$(\text{find}(x).\ \psi)\,?\,A_0 = \text{merge}(\{\langle A\backslash x, v\rangle : \langle A, v\rangle \in (\psi\,?\,A_0)\})$$

Scans.   We process scans similarly. However, we explicitly iterate over every node, passing it in with the initial assignment for ψ.

$$(\text{scan}(x).\ \psi)\,?\,A_0 = \text{merge}(\{\langle A\backslash x, v\rangle : \exists n \in U.\ \langle A, v\rangle \in (\psi\,?\,A_0[x \mapsto n])\})$$

Transitive closure.   Finally, we give rules for transitive closure. This step is a bit trickier. Consider the plan $(\text{TC}(s, t; x, y).\ \psi)$. We will assume that $A_0(s)$ is known in the initial assignment but $A_0(t)$ is not. (It is also possible to compute an answer in the symmetric case when $A_0(t)$ is known but $A_0(s)$ is not, or when both are known. The query optimizer ensures that at least one or the other is known.)

We essentially use breadth-first search. In each iteration, we advance a *frontier F*. Throughout the algorithm, we keep track of the reachability set T. Both F and T are sets of pairs, $\langle n, v\rangle$. If $v = 1$, then n is reachable from $A_0(s)$; if $v = 1/2$, it may be reachable. We use a fixed-point computation, beginning with $T_0 = \emptyset$ and $F_0 = \{\langle A_0(s), 1\rangle\}$. Then we advance as follows.

$$\begin{aligned}
T_i' &= \{\langle A'(y), v \wedge v'\rangle : \exists \langle n, v\rangle \in F_i.\ \langle A', v'\rangle \in \varphi\,?\,A_0[x \mapsto n]\} \\
T_{i+1} &= \text{mergenodes}(T_i, T_i') \\
F_{i+1} &= T_{i+1} - T_i
\end{aligned}$$

The mergenodes function is similar to merge, above: it unions the sets, but when a given node that is in both sides, it joins their values via $\vee$. When $F_j = \emptyset$, we stop iterating. Note that these steps compute *irreflexive* transitive closure. We add reflexivity explicitly at the end:

$$\begin{aligned}
(\text{TC}(s, t; x, y).\ \psi)\,?\,A_0 &= \text{merge}(R_1 \cup R_2) \\
R_1 &= \{\langle A_0[t \mapsto n], v\rangle : \langle n, v\rangle \in T_j\} \\
R_2 &= \{\langle A_0[t \mapsto A_0(s)], \iota(\mathsf{Eq})(A_0(s), A_0(s))\rangle\}
\end{aligned}$$

2.4.4   Query Execution Examples

We now give some examples of how realistic query plans are executed. We show the recursive execution of queries as follows.

$\psi_1\,?\,A_1$
$\quad\quad \psi_2\,?\,A_2$
$\quad\quad \rightarrow \{\langle A, v\rangle\}$
$\quad\quad \psi_3\,?\,A_3$
$\quad\quad \rightarrow \{\langle A', v'\rangle\}$
$\rightarrow \{\langle A'', v''\rangle\}$

This notation means that we execute the query ψ_1 with the initial assignment A_1. While executing ψ_1, we need to recursively execute the query ψ_2 with the initial assignment A_2.

This execution terminates with one result, $\langle A, v \rangle$. We perform another recursive evaluation as well, also returning one result. Based on these, we return one result, $\langle A'', v'' \rangle$, for the original query.

Example 17 Consider the query $(\mathrm{find}(x).\,\mathrm{find}(y).\,\mathsf{P}(x, y))$. Suppose that P holds at (n, n_1) and that it may hold at (n, n_2) and at (n', n_3). We recursively execute the query down to the atomic formula, as follows.

$(\mathrm{find}(x).\,\mathrm{find}(y).\,\mathsf{P}(x, y))\,?\,\{\}$
$\qquad(\mathrm{find}(y).\,\mathsf{P}(x, y))\,?\,\{\}$
$\qquad\qquad\mathsf{P}(x, y)\,?\,\{\}$
$\qquad\qquad\rightarrow \{\langle \{x \mapsto n, y \mapsto n_1\}, 1\rangle, \langle \{x \mapsto n, y \mapsto n_2\}, 1/2\rangle, \langle \{x \mapsto n', y \mapsto n_3\}, 1/2\rangle\}$
$\qquad\rightarrow \{\langle \{x \mapsto n\}, 1\rangle, \langle \{x \mapsto n'\}, 1/2\rangle\}$
$\rightarrow \{\langle \{\}, 1\rangle\}$ $\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad$ □

Example 18 We are given the query plan $(\mathrm{find}(x).\,\mathsf{P}(x) \wedge \mathsf{Q}(x))$. Suppose that P holds at n and n' and that Q holds only at n'. We first execute P with an empty initial assignment. Each result forms an initial assignment for executing the Q query.

$(\mathrm{find}(x).\,\mathsf{P}(x) \wedge \mathsf{Q}(x))\,?\,\{\}$
$\qquad(\mathsf{P}(x) \wedge \mathsf{Q}(x))\,?\,\{\}$
$\qquad\qquad\mathsf{P}(x)\,?\,\{\}$
$\qquad\qquad\rightarrow \{\langle \{x \mapsto n\}, 1\rangle, \langle \{x \mapsto n'\}, 1\rangle\}$
$\qquad\qquad\mathsf{Q}(x)\,?\,\{x \mapsto n\}$
$\qquad\qquad\rightarrow \{\}$
$\qquad\qquad\mathsf{Q}(x)\,?\,\{x \mapsto n'\}$
$\qquad\qquad\rightarrow \{\langle \{x \mapsto n'\}, 1\rangle\}$
$\qquad\rightarrow \{\langle \{x \mapsto n'\}, 1\rangle\}$
$\rightarrow \{\langle \{\}, 1\rangle\}$ $\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad$ □

Example 19 Consider $\neg\mathsf{P}(x)$. We are only able to execute this query given an initial assignment that includes an x binding. Suppose that P holds at n but not at n'.

$\neg\mathsf{P}(x)\,?\,\{x \mapsto n\}$
$\qquad\mathsf{P}(x)\,?\,\{x \mapsto n\}$
$\qquad\rightarrow \{\langle \{x \mapsto n\}, 1\rangle\}$
$\rightarrow \{\}$
$\neg\mathsf{P}(x)\,?\,\{x \mapsto n'\}$
$\qquad\mathsf{P}(x)\,?\,\{x \mapsto n'\}$
$\qquad\rightarrow \{\}$
$\rightarrow \{\langle \{x \mapsto n'\}, 1\rangle\}$ $\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad$ □

Example 20 Now take the query $(\forall x.\ (\neg P(x)) \vee Q(x))$. The optimizer will convert this to the query plan $(\neg \text{find}(x).\ P(x) \wedge (\neg Q(x)))$. Suppose that P holds at n and n'. Q does not hold at n but it may hold at n'.

$\neg\text{find}(x).\ P(x) \wedge (\neg Q(x)) \,?\, \{\}$
$\quad\ \text{find}(x).\ P(x) \wedge (\neg Q(x)) \,?\, \{\}$
$\qquad\quad P(x) \wedge (\neg Q(x)) \,?\, \{\}$
$\qquad\qquad\quad P(x) \,?\, \{\}$
$\qquad\qquad\quad \rightarrow \{\langle \{x \mapsto n\}, 1\rangle, \langle \{x \mapsto n'\}, 1\rangle\}$
$\qquad\qquad\quad Q(x) \,?\, \{x \mapsto n\}$
$\qquad\qquad\quad \rightarrow \{\}$
$\qquad\qquad\quad Q(x) \,?\, \{x \mapsto n'\}$
$\qquad\qquad\quad \rightarrow \{\langle \{x \mapsto n'\}, 1/2\rangle\}$
$\qquad\qquad \rightarrow \{\langle \{x \mapsto n'\}, 1/2\rangle\}$
$\qquad\ \rightarrow \{\langle \{\}, 1/2\rangle\}$
$\rightarrow \{\langle \{\}, 1/2\rangle\}$ $\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad$ □

Example 21 Finally, we look at the transitive closure query $(\text{TC}(s, t; x, y).\ E(x, y))$. To execute this query we require that either s or t is bound in the initial assignment. Suppose that s is bound to singleton node n and that E holds at (n, n') and (n', n''). We perform a breadth-first search of the graph, using recursive E queries to determine the edge set.

$(\text{TC}(s, t; x, y).\ E(x, y)) \,?\, \{s \mapsto n\}$
$\quad\ E(x, y) \,?\, \{s \mapsto n, x \mapsto n\}$
$\quad\ \rightarrow \{\langle \{s \mapsto n, x \mapsto n, y \mapsto n'\}, 1\rangle\}$
$\quad\ E(x, y) \,?\, \{s \mapsto n, x \mapsto n'\}$
$\quad\ \rightarrow \{\langle \{s \mapsto n, x \mapsto n', y \mapsto n''\}, 1\rangle\}$
$\quad\ E(x, y) \,?\, \{s \mapsto n, x \mapsto n''\}$
$\quad\ \rightarrow \{\}$
$\rightarrow \{\langle \{s \mapsto n, t \mapsto n\}, 1\rangle, \langle \{s \mapsto n, t \mapsto n'\}, 1\rangle, \langle \{s \mapsto n, t \mapsto n''\}, 1\rangle\}$ $\qquad\qquad$ □

2.4.5 Query Optimization

The process of query optimization converts a formula to a query plan. Since there may be many query plans that generate the same results, we are interested in the one that executes the quickest. The main goal of query optimization is to determine the order in which to execute the operands of a conjunction.

Example 22 Consider the query plan $A^+(x) \wedge E(x, y)$. If we execute the query in this order, we can use derivative information for A^+ to quickly enumerate all the nodes x where $A^+(x)$ holds. Then, for each such x value, we can use an index to find all nodes y where $E(x, y)$ holds.

On the other hand, consider the plan $\mathsf{E}(x, y) \wedge \mathsf{A}^+(x)$. We must iterate over all tuples (x, y) where E holds and then filter out the ones where $\mathsf{A}^+(x)$ holds. Since A^+ typically holds in a small number of places, the second form of the query will be slower than the first. □

This example illustrates a very common way to execute queries in our analysis. Since predicates like P^+ and P^- typically hold in only a few places (for core predicates, usually only one place), we would like to start query execution there. Then we continue along edge predicates. Nodes typically have small degree, so the size of the result set expands only gradually. Ideally, query execution should be no more expensive than traversing a graph, starting at nodes where derivatives hold and walking along edge predicates. As much as possible, we would like to avoid generating a result and then filtering it later.

Strategies. The query optimizer generates many different possible plans for a given query. We classify these plans according to (1) their estimated cost, and (2) the variables that must be in the initial assignment A_0 for the plan to be valid. The reason for (2) is that some efficient strategies may be valid only when some variables are already in A_0. In the extreme case, negations can execute only when all free variables are already in A_0.

The triple containing the required variables, the cost, and the query plan, written $\langle V, c, \psi \rangle$, is called a *strategy*. If an initial assignment A_0 contains bindings for all the variables in V, then we say that the strategy is *feasible*.

We use a recursive process to generate many strategies for a query; at the top level, we choose the cheapest feasible one. We write $\mathcal{O}(\varphi)$ to denote the set of strategies that result from optimizing the query φ. The algorithm for generating strategies is shown in Table 2.6 and the heuristics it uses are described below.

Atomic predicates. Consider derivative $\mathsf{P}^+(x)$ (or $\mathsf{P}^-(x)$). The number of nodes where a derivative holds is typically small, so even when x is not part of the initial assignment, we say that executing $\mathsf{P}^+(x)$ has cost 0, since it will generate few results. This corresponds to the strategy $\langle \emptyset, 0, \mathsf{P}^+(x) \rangle$.

Now consider a binary predicate $\mathsf{E}(x, y)$. The set of pairs of nodes where E holds may be large. When the initial assignment does not contain x or y, we assign a cost of 1 to $\mathsf{E}(x, y)$. This is the strategy $\langle \emptyset, 1, \mathsf{E}(x, y) \rangle$. However, if either x or y is known, we give the query a cost of 0, because the degree of nodes is usually small, so the result size will not increase much. This corresponds to two strategies: $\langle \{x\}, 0, \mathsf{E}(x, y) \rangle$ and $\langle \{y\}, 0, \mathsf{E}(x, y) \rangle$.

Now consider $\mathsf{P}(v_1, \ldots, v_k)$ where $k \neq 2$. If all the variables are known, we use cost 0, generating the strategy $\langle \{v_1, \ldots, v_k\}, 0, \mathsf{P}(v_1, \ldots, v_k) \rangle$. If some variable is unknown we assign the cost to be 1, giving the strategy $\langle \emptyset, 1, \mathsf{P}(v_1, \ldots, v_k) \rangle$.

Disjunction. We consider a disjunction $\varphi_1 \vee \varphi_2$. Since both subformulas are executed with the same initial assignment, they can be optimized independently. We recursively run $\mathcal{O}(\varphi_1)$ and $\mathcal{O}(\varphi_2)$. Suppose that $\langle V_1, c_1, \psi_1 \rangle$ is a strategy for the first subquery and

φ	$\mathcal{O}(\varphi)$
$\mathsf{P}(x,y)$	$\{\langle \emptyset, 1, \varphi \rangle, \langle \{x\}, 0, \varphi \rangle, \langle \{y\}, 0, \varphi \rangle\}$
$\mathsf{P}(v_1, \ldots, v_k)$	$\{\langle \{v_1, \ldots, v_k\}, 0, \varphi \rangle, \langle \emptyset, 1, \varphi \rangle\}$
$\neg \varphi'$	$\{\langle \mathrm{FV}(\varphi'), c, \neg \psi \rangle : \langle V, c, \psi \rangle \in \mathcal{O}(\varphi')\}$
$\mathbf{1}[\varphi']$	$\{\langle V, c, \mathbf{1}[\psi] \rangle : \langle V, c, \psi \rangle \in \mathcal{O}(\varphi')\}$
$\varphi_1 \wedge \varphi_2$	$\{\langle V_1 \cup (V_2 - \mathrm{FV}(\varphi_1)), c_1 + c_2, \psi_1 \wedge \psi_2 \rangle : \langle V_i, c_i, \psi_i \rangle \in \mathcal{O}(\varphi_i)\}$
	$\cup \; \{\langle V_2 \cup (V_1 - \mathrm{FV}(\varphi_2)), c_1 + c_2, \psi_2 \wedge \psi_1 \rangle : \langle V_i, c_i, \psi_i \rangle \in \mathcal{O}(\varphi_i)\}$
$\varphi_1 \vee \varphi_2$	$\{\langle V_1 \cup V_2, c_1 + c_2, \psi_1 \vee \psi_2 \rangle : \langle V_i, c_i, \psi_i \rangle \in \mathcal{O}(\varphi_i)\}$
$\exists x. \; \varphi'$	$\{\langle V, c, \mathrm{find}(x). \; \psi \rangle : \langle V, c, \psi \rangle \in \mathcal{O}(\varphi') \wedge x \notin V\}$
	$\cup \; \{\langle V \backslash \{x\}, c + 10, \mathrm{scan}(x). \; \psi \rangle : \langle V, c, \psi \rangle \in \mathcal{O}(\varphi') \wedge x \in V\}$
$\mathrm{TC}(s, t; x, y). \; \varphi'$	$\{\langle V - \{x\} \cup \{s\}, c, \mathrm{TC}(s, t; x, y). \; \psi \rangle : \langle V, c, \psi \rangle \in \mathcal{O}(\varphi') \wedge y \notin V\}$
	$\cup \; \{\langle V - \{y\} \cup \{t\}, c, \mathrm{TC}(s, t; x, y). \; \psi \rangle : \langle V, c, \psi \rangle \in \mathcal{O}(\varphi') \wedge x \notin V\}$

Table 2.6: Query optimization rules. Note that strategies are pruned after each recursive call.

$\langle V_2, c_2, \psi_2 \rangle$ is returned for the second subquery. In order to execute both queries, all variables in V_1 and V_2 must belong to the initial assignment. Thus, we form the combined strategy $\langle V_1 \cup V_2, c_1 + c_2, \psi_1 \vee \psi_2 \rangle$ (adding the costs is simply a heuristic). Written out formally, we have the following.

$$\mathcal{O}(\varphi_1 \vee \varphi_2) := \{\langle V_1 \cup V_2, c_1 + c_2, \psi_1 \vee \psi_2 \rangle : \langle V_i, c_i, \psi_i \rangle \in \mathcal{O}(\varphi_i)\}$$

Conjunction. Conjunctions are the heart of query optimization. As before, we start by optimizing the subqueries φ_1 and φ_2. For strategy $\langle V_1, c_1, \psi_1 \rangle$ returned for the first subquery and $\langle V_2, c_2, \psi_2 \rangle$ for the second, we return *two* combined strategies: one where ψ_1 is executed first and the other where ψ_2 is done first.

First assume ψ_1 executes first. We generate a strategy $\langle V, c_1 + c_2, \psi_1 \wedge \psi_2 \rangle$, where V is to be determined. To determine V, the set of variables that must be present in the initial assignment, we form constraints. The execution of ψ_1 requires $V_1 \subseteq V$. Naively, we might think $V_2 \subseteq V$ is required as well. However, the execution of ψ_1 added variables $\mathrm{FV}(\varphi_1)$ to the assignments of its results. These results are fed back in as the initial assignment for executing ψ_2. Thus, we can make a looser constraint that $V_2 \subseteq V \cup \mathrm{FV}(\varphi_1)$. Combining this with the $V_1 \subseteq V$ constraint, the smallest solution for V is $V_1 \cup (V_2 - \mathrm{FV}(\varphi_1))$. Formally, we get the following strategies.

$$\{\langle V_1 \cup (V_2 - \mathrm{FV}(\varphi_1)), c_1 + c_2, \psi_1 \wedge \psi_2 \rangle : \langle V_i, c_i, \psi_i \rangle \in \mathcal{O}(\varphi_i)\}$$

We combine these with the symmetric set of strategies where ψ_2 is executed first.

$$\{\langle V_2 \cup (V_1 - \mathrm{FV}(\varphi_2)), c_1 + c_2, \psi_2 \wedge \psi_1 \rangle : \langle V_i, c_i, \psi_i \rangle \in \mathcal{O}(\varphi_i)\}$$

Negation. In §2.4.3, the rule to execute $\neg\varphi$ had a side condition requiring that all free variables of φ be in the initial assignment. When optimizing a negation, we must ensure this condition holds. Thus, for every strategy $\langle V, c, \psi \rangle$ generated for φ, we generate a strategy $\langle \text{FV}(\varphi), c, \neg\psi \rangle$ for the negation. Notice how this strategy requires that all free variables belong to the initial assignment.

Definite operator. Executing $\mathbf{1}[\varphi]$ is no more difficult than executing φ, so we use essentially the same set of strategies:

$$\mathcal{O}(\mathbf{1}[\varphi']) := \{\langle V, c, \mathbf{1}[\psi]\rangle : \langle V, c, \psi \rangle \in \mathcal{O}(\varphi')\}$$

Existentials. As described earlier, there are two possible query plans for an existential ($\exists x.\ \varphi$): find and scan. Find relies on the subquery to generate bindings to x while scan iterates over all nodes, binding x in the initial assignment. We generate strategies for both find and scan.

First we optimize the subquery. Let $\langle V, c, \psi \rangle$ be a resulting strategy. We can generate a strategy that uses find only if x is not in V (i.e., if x is not required in the initial assignment). In this case, we generate the strategy $\langle V, c, \text{find}(x).\ \psi \rangle$. On the other hand, a scan query plan is always feasible, allowing us to generate the strategy $\langle V \backslash \{x\}, c + 10, \text{scan}(x).\ \psi \rangle$. We eliminate x from V because it is not needed in the initial assignment. We add 10 to the cost of the strategy to account for the cost of iterating over all nodes.
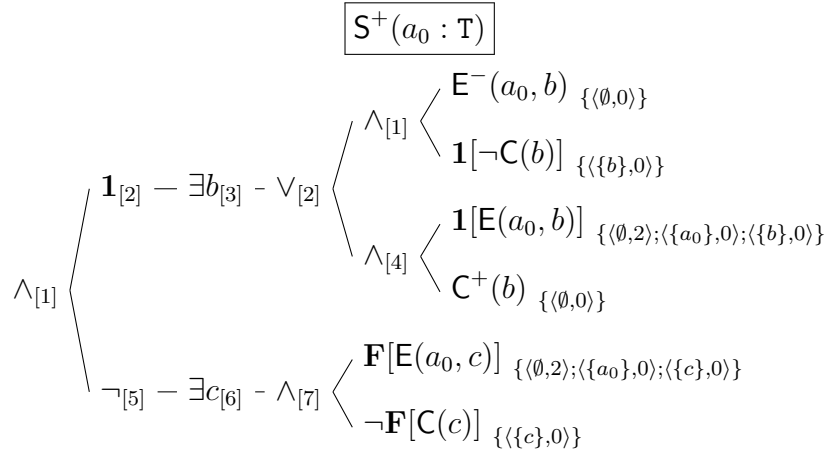
Transitive closure. There are two ways to execute a transitive closure ($\text{TC}(s, t; x, t).\ \varphi$): forward and backward. The forward method starts at the known node s and walks forward along $\varphi(x, y)$ edges. We require that executing φ with x in the initial assignment generate a binding for y. Every node we reach is bound to t in the result. The backward method starts at t and walks backwards along $\varphi(x, y)$ edges, generating x bindings given a y binding, and arriving at bindings for s.

Given a strategy $\langle V, c, \psi \rangle$ for φ, the forward method is feasible if $y \notin V$. In this case we generate a strategy $\langle V - \{x\} \cup \{s\}, c, \text{TC}(s, t; x, y).\ \psi \rangle$. s is added to the set of required variables because we must know the node from which to start walking forward.

The backward method is feasible if $x \notin V$. We generate a similar strategy in this case. See Table 2.6.

Pruning. Without some form of pruning, the set of strategies generated by $\mathcal{O}$ may become quite large. Many of these strategies will be redundant. A strategy $\langle V, c, \psi \rangle$ is redundant if there is some other strategy that dominates it. A strategy $\langle V', c', \psi' \rangle$ dominates it when $V' \subseteq V$ and $c' \leq c$. In this case we remove the dominated strategy. The following function performs this sort of pruning. It is run before returning a given recursive result from $\mathcal{O}$ (meaning that pruning occurs at every level of the formula tree).

$$\text{Prune}(S) = \{\langle V, c, \psi \rangle : \neg\exists \langle V', c', \psi' \rangle \in S.\ V' \subseteq V \wedge c' \leq c \wedge \langle V', c', \psi' \rangle \neq \langle V, c, \psi \rangle\}$$

$$\boxed{\mathsf{S}^+(a_0 : \mathsf{T})}$$

$$\wedge_{[1]} \left\langle \begin{array}{l} \mathsf{E}^-(a_0, b) \quad _{\{\langle\emptyset,0\rangle\}} \\[4pt] \mathbf{1}[\neg\mathsf{C}(b)] \quad _{\{\langle\{b\},0\rangle\}} \end{array} \right.$$

$$\mathbf{1}_{[2]} - \exists b_{[3]} - \vee_{[2]} \left\langle \begin{array}{l} \wedge_{[1]} \\[20pt] \wedge_{[4]} \left\langle \begin{array}{l} \mathbf{1}[\mathsf{E}(a_0, b)] \quad _{\{\langle\emptyset,2\rangle;\langle\{a_0\},0\rangle;\langle\{b\},0\rangle\}} \\[4pt] \mathsf{C}^+(b) \quad _{\{\langle\emptyset,0\rangle\}} \end{array} \right. \end{array} \right.$$

$$\wedge_{[1]} \left\langle \begin{array}{l} \\ \neg_{[5]} - \exists c_{[6]} - \wedge_{[7]} \left\langle \begin{array}{l} \mathbf{F}[\mathsf{E}(a_0, c)] \quad _{\{\langle\emptyset,2\rangle;\langle\{a_0\},0\rangle;\langle\{c\},0\rangle\}} \\[4pt] \neg\mathbf{F}[\mathsf{C}(c)] \quad _{\{\langle\{c\},0\rangle\}} \end{array} \right. \end{array} \right.$$

[1] $\{\langle\emptyset, 0, \text{upper}\rangle\}$ [2] $\{\langle\emptyset, 0\rangle\}$ [3] $\{\langle\emptyset, 0, \text{find}\rangle\}$ [4] $\{\langle\emptyset, 0, \text{lower}\rangle\}$ [5] $\{\langle\{a_0\}, 0\rangle\}$
[6] $\{\langle\emptyset, 2, \text{find}\rangle; \langle\{a_0\}, 0, \text{find}\rangle\}$ [7] $\{\langle\emptyset, 2, \text{upper}\rangle; \langle\{a_0\}, 0, \text{upper}\rangle; \langle\{c\}, 0, \text{upper}\rangle\}$

Figure 2.1: Optimization results for up derivative of $\mathsf{S}(x) := \forall y.\ \mathsf{E}(x, y) \Rightarrow \mathsf{C}(y)$.

2.4.6   Query Optimization Examples

This section describes the optimization of a few complex but typical queries.

Example 23 We begin with the up derivative of $\mathsf{S}(x) := \forall y.\ \mathsf{E}(x, y) \Rightarrow \mathsf{C}(y)$. This formula was explained in Example 12. The optimization results are shown in Figure 2.1. We show query strategies in small print to the left of each formula or else separately, at the bottom of the figure. Recall that the top branch of this formula looks for places where $\mathsf{E}(x, y)$ goes down or $\mathsf{C}(y)$ goes up. The bottom branch verifies that the formula is true in the future. Note that universal quantifiers have been converted to existentials by De Morgan's laws.

Each formula is annotated with a set of strategies. To save space we do not show the ψ component of strategies. Instead, when there is a choice between several query plans, we give a description of which one is chosen, at least at the top level. For conjunctions, we say whether the upper or the lower branch is executed first. For existentials, we say whether we use a scan or a find query. And for transitive closure we say whether we search forward or backward.

Like most queries, this one has only one optimal strategy: given derivative information for E and C, we start with the upper branch of the query. The top-right conjunction generates tuples (a_0, b) satisfying E^-. It filters out the ones that satisfy C. These results are unioned with the results of the lower conjunction, which generates nodes b where C goes up and then finds a_0 nodes by walking across existing E edges. The existential quantifier, converted to a find quantifier, projects b out of the results. Finally, in the lower branch, we check that each result satisfies the query in the future state. Since a_0 is already known, we find c nodes by walking across E edges as before.

In this query the main function of the optimizer is to choose whether to execute the upper or lower branch of a conjunction first. Consider the upper-right conjunction. Executing E^- first requires no variables in the initial assignment. After executing it, variables a_0 and b become known, allowing us to execute the lower conjunct, which requires b to be known. Using the reverse execution order would require b to be known beforehand, meaning that the $\exists b$ quantifier would have to be executed using a scan rather than a find.   □

This query illustrates a very important point. Besides increasing the precision of transformers, finite differencing provides a mechanism for executing queries incrementally. Rather than re-evaluating a query completely for every statement, we have a way to compute only what has changed. Another way to say this is that derivatives make good queries for query optimization because up and down formulas tend to have small result sets.

Example 24 We optimize the up derivative of $\mathsf{Path}(s,t) := \mathrm{TC}(s,t;x,y). \; \mathsf{E}(x,y)$. The results are shown in Figure 2.2.

This query can also be executed without any expensive scans. The lower branch is executed first. Its job is to determine places (a_0, a_1) where Path might go up. Then the upper branch checks that there is indeed a path between those nodes.

The lower branch looks for intermediate nodes (f, g) where $\mathsf{E}^+(f, g)$ and where there is a path from a_0 to f and from g to a_1. The path must be a sequence of existing path edges or else E^+ edges. When executing the query, we first find the single $\mathsf{E}^+(f, g)$ edge. Then we find all a_0 that reach f via a sequence of Path or E^+ edges, as well as all a_1 that g reaches along such edges. After projecting out f and g, we have all the a_0 and a_1 where Path can go up. We check that Path was not already true there in the original structure.

Next, the upper branch executes. It verifies that there is a path between a_0 and a_1 along either E^+ edges or trusted edges. To check if an edge is trusted, we find all d and e where $\mathsf{E}^-(d, e)$ holds and then see if this E^- edge might break the Path edge we are trying to use. If it cannot, then the edge is trusted.

One interesting point to note is that the transitive closure operation labeled [9] is executed backward. The conjunction to the left of it executes its lower branch, an E^+, first. When the transitive closure executes next, variable f is known while a_0 is not. Thus we execute it backward. In a symmetrical way, the transitive closure labeled [11] must be executed in the forward direction.   □

2.4.7   Experiments

Using the web server cache, we compared the performance of the naive evaluation algorithm to the more efficient one presented here. The total analysis time with the naive algorithm was 141.03s. Of this time, 131.26s were spent running queries (93%). In contrast, the total analysis time using the optimized algorithm was 9.71s, of which 22 milliseconds were spent running queries (less than 1%). The efficient algorithm is nearly 6,000 times faster than the naive algorithm. Although it would undoubtedly be possible to further optimize the algorithm, the time would be better spent on other stages of the analysis.

$$\boxed{\mathsf{Path}^+(a_0 : \mathtt{T}, a_1 : \mathtt{T})}$$

Figure 2.2: Optimization results for up derivative of the transitive closure predicate $\mathsf{Path}(s,t) := \mathrm{TC}(s,t;x,y). \, \mathsf{E}(x,y)$.

To get a better understanding of the evaluation algorithm, we instrumented it with counters. We counted the number of evaluations of atomic predicates and derivatives. Note that the efficient algorithm may return multiple results for each evaluation if the initial assignment does not contain all the free variables.

Algorithm	# atomic evaluations	# derivative evaluations
Naive	94,795,212 (0.202)	23,070,446 (0.000784)
Efficient	9,314 (1.11)	10,502 (0.0528)

Table 2.7: Number of query evaluations and average result set size in parentheses).

Not surprisingly, the difference in time between the algorithms is similar to the difference in number of evaluations. The naive algorithm performs 5,947 times as many evaluations and is 5,966 times slower.

The naive algorithm we implemented does differ from Table 2.5 in how it implements transitive closure. It performs a breadth-first search. It maintains a frontier, and at each step of the search it picks a frontier node and another arbitrary node and checks to see if the one can reach the other. The newly reachable nodes become the frontier of the next iteration. This algorithm introduces many nested loops over nodes, so we suspected that it is responsible for many of the evaluations performed by the naive algorithm. The results bear out this hypothesis.

Evaluation context	# atomic evaluations	# derivative evaluations
In TC	80,548,460 (0.204)	20,464,304 (0.000779)
Outside TC	14,246,752 (0.194)	2,606,142 (0.000824)

Table 2.8: Number of query evaluations in naive algorithm inside and outside of transitive closure quantifiers.

Clearly, most atomic and derivative evaluations in the naive algorithm appear as part of a transitive closure query. A better transitive closure algorithm might be able to eliminate some of these evaluations. However, even if it were able to eliminate all evaluations inside transitive closures, it would still perform 850 times as many evaluations as the efficient algorithm. Assuming a constant time per evaluation, the naive query algorithm would still consume the majority of the total analysis time.

One reason that the efficient algorithm is so much faster is that, for the queries used in the web server example, *it never performs a scan.* The query optimizer is able to convert every existential quantifier into a find operation. The naive algorithm uses scans for every quantifier. It also performs implicit scans over every predicate argument, since each instrumentation derivative must be evaluated over its entire domain. The average nesting depth for the web cache, including predicate arguments, is 2.73.

We broke down the atomic predicate evaluations further for the efficient algorithm. For each evaluation of $\mathsf{P}(v_1, \ldots, v_k)$, we counted the number of variables in $\{v_1, \ldots, v_k\}$ that were known (i.e., present in the initial assignment) and unknown.

Query type	# evaluations	Average result set size
1 known, 0 unknown	1,500	0.185
2 known, 0 unknown	3,144	0.413
0 known, 1 unknown	296	1.89
1 known, 1 unknown	4,219	1.69
0 known, 2 unknown	155	6.84
Derivative	10,502	0.0528

Table 2.9: Efficient evaluation statistics.

We can first consider how many "filter" evaluations occur. These are evaluations where all variables are already known. All evaluations performed by the naive algorithm are filter evaluations. The first two rows shows that the efficient algorithm performs 4,644 filter queries versus 4,670 queries where at least one variable is unknown. Of the "filter" queries, 1,576 (or 34%) return a positive answer. Ideally this number would be 100%, meaning that we never generate a result and then discard it, but some filtering is inevitable.

We designed our algorithm based on the hypothesis that derivatives return few results and that nodes have small degree. The data shows that derivatives indeed return very few results. We measure node degree by the number of results returned by a query where one variable is known and the other unknown. The average is 1.69. On the other hand, the average number of results returned for a binary query where neither variable is known is larger: 6.84.

2.5 Sharpening

The previous two sections described new techniques for computing the effect of an assignment statement precisely and efficiently. However, as explained in Section 2.2.2, two preprocessing steps are needed before an assignment can be analyzed: focus and sharpen. We give an example of both steps here; the remainder of this section describes a new algorithm for sharpening, which is contribution number ③ of this chapter.

Example 25 Suppose we are to analyze the statement $x := next[x]$. Also suppose we have the following instrumentation predicate:

$$\mathsf{Shared}(n) := \exists n_1, n_2. \ \mathsf{Next}(n_1, n) \land \mathsf{Next}(n_2, n) \land n_1 \neq n_2.$$

These predicate holds when an object has two or more incoming *next* pointers.

Now consider the following structure. The edges denote the Next predicate.

$$u_1 \quad \boldsymbol{u_2}$$
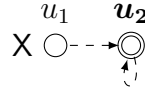
Note that neither u_1 nor $\boldsymbol{u_2}$ satisfies Shared.

Analyzing the assignment $x := next[x]$ directly on the state will not yield anything useful. We will simply get that x may point to some node abstracted by $\boldsymbol{u_2}$. In a typical loop we want a more precise abstraction that distinguishes between nodes that have already been seen and those yet to be seen—without this we will not infer a precise loop invariant.

Consequently, before the assignment we focus on $next[x]$. The focus operation is simply a case analysis—it does not change the meaning of a domain element at all. We get a disjunction of the following two structures.

S_1	S_2
$u_1 \qquad u_2$	$u_1 \qquad u_2 \qquad \boldsymbol{u_2'}$
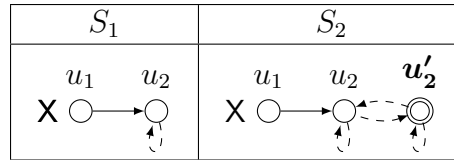
Now we can apply sharpening to these two structures to make them more precise. In S_1, $\mathsf{Shared}(u_2) = 0$, which tells us that u_2 cannot have a Next edge to itself. If it did, then it would have two incoming *next* pointers, contradicting the fact that $\mathsf{Shared}(u_2) = 0$. Similarly, in S_2 we discover that $\mathsf{Next}(u_2, u_2) = 0$.

We can also show that $\mathsf{Next}(\boldsymbol{u_2'}, u_2) = 0$ in S_2. However, the reasoning for this inference is more complex. Since $\boldsymbol{u_2'}$ is a summary node, setting $\mathsf{Next}(\boldsymbol{u_2'}, u_2)$ to 0 means that *no node* abstracted by $\boldsymbol{u_2'}$ can have an edge to u_2. And, indeed, if even one node abstracted by $\boldsymbol{u_2'}$ had a Next edge to u_2 then u_2 would be shared, contradicting the $\mathsf{Shared}(u_2) = 0$ fact. $\square$

The remainder of this chapter describes the algorithm that infers facts like $\mathsf{Next}(\boldsymbol{u_2'}, u_2) = 0$, as we just did in the example.

2.5.1   Integrity Constraints

Instrumentation predicates retain vital information in the heap abstraction. Sharpening exploits that information by bringing core predicates in line with instrumentation predicates. It is considered the most difficult and expensive step in the TVLA algorithm [5].

We can use a formula to express the desire for core predicates and instrumentation predicates to agree. Let the instrumentation predicate P (of arity k) have formula φ as its definition. The following sentence asserts that the core predicates and instrumentation predicates agree.

$$\forall u_1, \ldots, u_k.\ \mathsf{P}(u_1, \ldots, u_k) \iff \varphi(u_1, \ldots, u_k). \tag{2.1}$$

This formula is called an *integrity constraint*.

Formally, we assume we have a set of integrity constraints I. Each integrity constraint is a closed formula in first-order logic with transitive closure (FO(TC)). Although integrity constraints are always quantified if-and-only-if statements, as above, this section allows them to be arbitrary.

Every concrete heap that is abstracted by a structure S must satisfy the integrity constraints for all the instrumentation predicates (it must also embed into S, of course). We define $\gamma_I(S)$ to be the set of all concrete heaps that satisfy the integrity constraints I and embed into S. Relating this to the previous definition, $\gamma(S) = \gamma_\emptyset(S)$.

The sharpening algorithm applies integrity constraints to a heap. It may sharpen a particular predicate value from $1/2$ to 0 or 1, or it may decide that the heap is infeasible and throw it away entirely. Sharpening only affects the interpretation of predicates ι; the universe U is left alone. This restriction is a practical one. Relaxing it would make sharpening more precise, but the problem would become undecidable.

Formally, sharpening is the process of refining S by finding some S', over the same nodes as S, where S' embeds into S and $\gamma_I(S') = \gamma_I(S)$. There will always be a best S' (lowest in the embedding order) as long as $\gamma_I(S) \neq \emptyset$.

Example 26 Consider the following heap structure S.
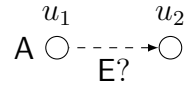
Assume that we have the integrity constraint $\forall n.\ \mathsf{A}(n) \Rightarrow \exists n'.\ \mathsf{E}(n, n')$ in I. Any concrete heap $S' \in \gamma_I(S)$ that satisfies this integrity constraint must have $\mathsf{E}(u_1, u_2) = 1$. Therefore, we sharpen E to 1 there. □

2.5.2 Correctness by Case Analysis

Any sharpening of structure S to S' that satisfies the $\gamma_I(S') = \gamma_I(S)$ condition is correct. However, this property does not provide much insight in designing an algorithm to implement sharpening: we would have to enumerate all the structures in $\gamma(S)$ and then filter out those not satisfying I. Since $\gamma(S)$ is typically infinite, we need a better approach.

We can use case analysis to obtain a more direct technique for sharpening. Consider sharpening the predicate value $p(u_1, \ldots, u_k)$ to v (either 0 or 1). We write this sharpening as $p^v(u_1, \ldots, u_k)$. Suppose we want to decide if the sharpening is correct. We split the original structure S into structures $S_1, \ldots, S_n$. These structures should embed into S via embedding functions f_i. We require the case analysis to be *exhaustive*, meaning

$$\gamma(S) = \gamma(S_1) \cup \cdots \cup \gamma(S_n).$$

In some of the S_i cases, the sharpening will be *realized*, meaning that $p(u'_1, \ldots, u'_k) = v$ for every u'_j that embeds into the original structure's u_j (i.e., where $f_i(u'_j) = u_j$). In the remaining cases, we require the integrity constraint to be violated, meaning it should evaluate

to 0. If we can split the cases in this way, so that in each one either the sharpening is realized or the integrity constraint is violated, then the sharpening is sound.

Consider Example 26. We can split the structure exhaustively into two cases: one where $E(u_1, u_2) = 0$ and the other where $E(u_1, u_2) = 1$. The first structure violates the integrity constraint and the second one realizes the sharpening $E^1(u_1, u_2)$, meaning that this sharpening is valid.

The problem becomes slightly more complex when summary nodes are involved because there are more cases to analyze, as in this example.

Example 27 Consider the following heap structure S.
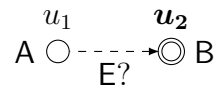
Imagine we have the integrity constraint $\forall n. A(n) \Rightarrow (\forall n'. B(n') \Rightarrow E(n, n'))$. Intuitively, we can see that E can be sharpened to 1 between u_1 and $\boldsymbol{u_2}$. To prove this, we use the following case analysis.
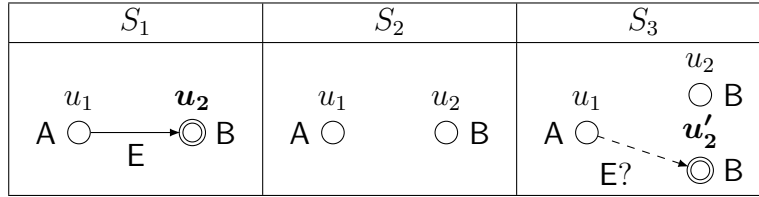
In the first one, S_1, every B node has an edge from the A node. In the latter two cases, there is at least one B node that does not have an incoming edge. (We are forced to partition this situation into two cases because summary nodes in TVLA always represent at least one node.)

These structures form an exhaustive case analysis of S. The sharpening $E^1(u_1, \boldsymbol{u_2})$ (i.e., E set to 1) is realized only in S_1. However, the integrity constraint is violated in S_2 and S_3, so the sharpening is sound. □

Notice that in the case analysis above, we split the node $\boldsymbol{u_2}$ in structure S_3. This appears to contradict our statement earlier that sharpening does not change the universe U. In fact, there is no contradiction. Although case analysis may materialize nodes to justify a sharpening, no nodes are materialized in the heap that results from the sharpening. In the example above, we sharpened E between u_1 and $\boldsymbol{u_2}$; we did not change U at all.

The next section presents an algorithm that will find most sharpenings that require only a finite case analysis. Unfortunately, some sharpenings may be sound but require infinite case analysis to prove correct. This commonly happens with integrity constraints that use transitive closure. When transitive closure is not involved, only a small number of cases are typically necessary to prove a given sharpening.

2.5.3 Sharpening Algorithm

The previous section offers a possible algorithm for sharpening structures: starting with a possible sharpening $p^v(u_1, \ldots, u_k)$, search for a case analysis that proves the sharpening correct. However, even if we ignore the problem of infinite case analysis, the algorithm will still be very slow. This section presents an efficient algorithm that will find many, but not all, sharpenings. Later we will analyze its completeness.

Our algorithm, called M, requires no case analysis. Instead, it traverses the given integrity constraint, φ, in a syntax-directed fashion. We write $M(\varphi, S, A)$ to denote the resulting sharpenings given the constraint φ, a structure S, and an assignment A of free variables to nodes. The sharpenings that are returned have the form $p^v(v_1, \ldots, v_k)$.

As a precondition for using M, we assume that $[\![\varphi]\!]_{S,A} = 1/2$. If φ evaluates to 1 then no sharpenings are possible. If it evaluates to zero, then the entire heap can be discarded. When designing the algorithm M, we can think of the results returned by $M(\varphi, S, A)$ as atomic literals that are implied by the integrity constraint φ.

Atomic literals. We start by considering $\varphi = \mathsf{P}(v_1, \ldots, v_k)$. Clearly this φ logically implies the sharpening $\mathsf{P}^1(A(v_1), \ldots, A(v_k))$. Similarly, if $\varphi = \neg\mathsf{P}(v_1, \ldots, v_k)$ then we return $\mathsf{P}^0(A(v_1), \ldots, A(v_k))$.

Conjunctions. Suppose $\varphi = \varphi_1 \wedge \varphi_2$. We recursively run M on both conjuncts. Assume that $M(\varphi_1, S, A)$ returns a result $p^v(u_1, \ldots, u_k)$. Using the intuition above, we know that $\varphi_1 \Rightarrow p^v(u_1, \ldots, u_k)$. Consequently $\varphi \Rightarrow p^v(u_1, \ldots, u_k)$. Therefore, any sharpening returned for either φ_1 or φ_2 is also valid for φ. We simply union them together to get $M(\varphi, S, A)$.

$$M(\varphi_1 \wedge \varphi_2, S, A) := M(\varphi_1, S, A) \cup M(\varphi_2, S, A).$$

Example 28 Consider the following structure S with only one node.

$$\mathsf{A?,B?} \overset{u}{\bigcirc}$$

Suppose that we are asked to compute $M(\mathsf{A}(x) \wedge \neg\mathsf{B}(x), S, A)$, where $A(x) = u$. We recursively apply M, obtaining the following.

$$M(\mathsf{A}(x), S, A) = \{\mathsf{A}^1(u)\}$$
$$M(\neg\mathsf{B}(x), S, A) = \{\mathsf{B}^0(u)\}$$

Then we return the combined answer $\{\mathsf{A}^1(u), \mathsf{B}^0(u)\}$. □

Note that we cannot always apply M recursively to φ_1 and φ_2. It may be that one of these subformulas evaluates to 1. We cannot apply M to this subformula because M is only valid on formulas evaluating to $1/2$. So we act as if M returned the empty set when applied to that conjunct. In other words, we simply return the result of applying M to the other conjunct.

Disjunctions. Disjunctions are handled somewhat similarly. Let $\varphi = \varphi_1 \vee \varphi_2$. Suppose that both disjuncts evaluate to $1/2$. If a sharpening $p^v(u_1, \ldots, u_k)$ is returned by $M(\varphi_1, S, A)$, that means that $\varphi_1 \Rightarrow p^v(u_1, \ldots, u_k)$. This does *not* mean that $\varphi \Rightarrow p^v(u_1, \ldots, u_k)$ because φ is too weak.

However, if $p^v(u_1, \ldots, u_k)$ is returned by *both* φ_1 and φ_2 then it must be that $\varphi \Rightarrow p^v(u_1, \ldots, u_k)$ and so we can return $p^v(u_1, \ldots, u_k)$ as a sharpening for φ. In other words,

$$M(\varphi_1 \vee \varphi_2, S, A) := M(\varphi_1, S, A) \cap M(\varphi_2, S, A).$$

Example 29 Consider the following structure S.

$$
\begin{array}{cc}
u_1 & u_2 \\
\mathsf{A}?\ \bigcirc & \bigcirc\ \mathsf{A}?
\end{array}
$$

Suppose that we are asked to compute $M(\mathsf{A}(x) \vee \mathsf{A}(y), S, A)$, where $A(x) = u_1$ and $A(y) = u_2$. We recursively apply M, obtaining the following.

$$M(\mathsf{A}(x), S, A) = \{\mathsf{A}^1(u_1)\}$$
$$M(\mathsf{A}(y), S, A) = \{\mathsf{A}^1(u_2)\}$$

Intersecting these answers yields the empty set. And, indeed, it is not sound to perform any sharpenings in this case. Some members of $\gamma_I(S)$ have $\mathsf{A}(u_1) = 0$ and other members have $\mathsf{A}(u_2) = 0$ (although no member has both).

However, given an alternate assignment A' where $A'(x) = u_1$ and $A'(y) = u_1$, we get the following.

$$M(\mathsf{A}(x), S, A') = \{\mathsf{A}^1(u_1)\}$$
$$M(\mathsf{A}(y), S, A') = \{\mathsf{A}^1(u_1)\}$$

In this case, the intersection is $\{\mathsf{A}^1(u_1)\}$. And, indeed, the integrity constraint applied at A' does require $\mathsf{A}(u_1)$ to be 1. □

As we did for conjunctions, if either φ_1 or φ_2 does not evaluate to $1/2$, then we return M applied to the other disjunct as the result.

Quantifiers. We handle universal quantifiers much as we handled conjunctions: by combining the results. Suppose $\varphi = \forall v.\ \varphi'$. For each $n \in U$, we compute $M(\varphi', S, A[v \mapsto n])$. We union together all these sharpenings. As before, we skip over places where φ' does not evaluate to $1/2$.

$$M(\forall v.\ \varphi', S, A) = \bigcup_{u \in U} \{M(\varphi', S, A[v \mapsto u]) : [\![\varphi']\!]_{S, A[v \to u]} = 1/2\}$$

We do the analogous thing for existentials. Instead of union, we use intersection.

$$M(\exists v.\ \varphi', S, A) = \bigcap_{u \in U} \{M(\varphi', S, A[v \mapsto u]) : [\![\varphi']\!]_{S,A[v \to u]} = 1/2\}$$

For transitive closure, we use the definition that it is a disjunction over all paths and a conjunction over the edges in the path. Then we turn conjunctions into unions and disjunctions into intersections, as follows.

$$M(\text{ITC}(s, t; x, y).\ \varphi', S, A) = \bigcap_{P \in \text{Paths}(S, A(s), A(t))} \bigcup_{(u_1, u_2) \in P} \left\{ \begin{array}{l} M(\varphi', S, A[x \mapsto u_1, y \mapsto u_2]) \\ : [\![\varphi']\!]_{S,A[x \mapsto u_1, y \mapsto u_2]} = 1/2 \end{array} \right\}$$

This equation handles *irreflexive* transitive closure, which we write as ITC. For reflexive transitive closure, we convert the formula $(\text{TC}(s, t; x, y).\ \varphi)$ to $(x = y) \vee (\text{ITC}(s, t; x, y).\ \varphi)$.

Putting all this together, we obtain the following algorithm. We assume that integrity constraints are in negated normal form.

| φ	$M(\varphi, S, A)$
$\mathsf{P}(v_1, \ldots, v_k)$	$\{\mathsf{P}^1(A(v_1), \ldots, A(v_k))\}$
$\neg\mathsf{P}(v_1, \ldots, v_k)$	$\{\mathsf{P}^0(A(v_1), \ldots, A(v_k))\}$
$\varphi_1 \wedge \varphi_2$	$\begin{cases} M(\varphi_1, S, A) & \text{if } [\![\varphi_2]\!]_{S,A} \neq 1/2 \\ M(\varphi_2, S, A) & \text{if } [\![\varphi_1]\!]_{S,A} \neq 1/2 \\ M(\varphi_1, S, A) \cup M(\varphi_2, S, A) & \text{otherwise} \end{cases}$
$\varphi_1 \vee \varphi_2$	$\begin{cases} M(\varphi_1, S, A) & \text{if } [\![\varphi_2]\!]_{S,A} \neq 1/2 \\ M(\varphi_2, S, A) & \text{if } [\![\varphi_1]\!]_{S,A} \neq 1/2 \\ M(\varphi_1, S, A) \cap M(\varphi_2, S, A) & \text{otherwise} \end{cases}$
$\forall v.\ \varphi'$	$\bigcup_{u \in U}\{M(\varphi', S, A[v \mapsto u]) : [\![\varphi']\!]_{S,A[v \to u]} = 1/2\}$
$\exists v.\ \varphi'$	$\bigcap_{u \in U}\{M(\varphi', S, A[v \mapsto u]) : [\![\varphi']\!]_{S,A[v \to u]} = 1/2\}$
$\text{ITC}(s, t; x, y).\ \varphi'$	$\bigcap_{P \in \text{Paths}(S, A(s), A(t))} \bigcup_{(u_1, u_2) \in P}$ $\{M(\varphi', S, A[x \mapsto u_1, y \mapsto u_2]) : [\![\varphi']\!]_{S,A[x \mapsto u_1, y \mapsto u_2]} = 1/2\}$

2.5.4 Soundness

This algorithm is straightforward and easy to implement. However, it holds several surprises.

1. When S contains summary nodes and when φ includes existential quantifiers, the algorithm is unsound. That is, it generates incorrect sharpenings. We fix this problem by filtering out any sharpening involving a summary node introduced by an existential quantifier.

2. Even when fixed in this way, the algorithm is still unsound in an even stranger way. If φ includes multiple occurrences of the same variable in the same predicate at different positions (such as $\mathsf{E}(x,x)$) then the results may be unsound. However, in our experience no realistic integrity constraints have this form. So we require that M never be applied to such a constraint.

This section describes the soundness problems in detail and presents solutions for them. We begin by explaining the problem posed by summary nodes and existential quantifiers.

Example 30 We reconsider Example 27.
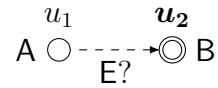
Recall that we can sharpen $\mathsf{E}(u_1, \boldsymbol{u_2})$ to 1 when using the integrity constraint $\forall n.\ \mathsf{A}(n) \Rightarrow (\forall n'.\ \mathsf{B}(n') \Rightarrow \mathsf{E}(n, n'))$. We saw that this was legal based on the following case analysis.

Consider changing the integrity constraint to $\forall n.\ \mathsf{A}(n) \Rightarrow (\exists n'.\ \mathsf{B}(n') \wedge \mathsf{E}(n, n'))$ (we changed the inner universal quantifier to existential). Since there is only one valid binding for n', our algorithm does not care whether the quantifier is universal or existential. It returns the same sharpening either way. However, that sharpening is incorrect for this integrity constraint! The new constraint requires that the A node connect to *some* B node, but not necessarily all the B nodes. In the case analysis, S_3 no longer violates the integrity constraint.

We can look more closely at the real source of the problem. Given the same structure, suppose we are asked to compute $M(\mathsf{E}(n, n'), S, A)$, where $A(n) = u_1$ and $A(n') = \boldsymbol{u_2}$. The problem is that $\boldsymbol{u_2}$ is a summary node. Returning the answer $\mathsf{E}^1(u_1, \boldsymbol{u_2})$, as the algorithm does, means the following: any structure not violating the integrity constraint has $\mathsf{E}(u_1, \boldsymbol{u_2}) = 1$. Is this true?

The crucial issue is whether the integrity constraint $\mathsf{E}(n, n')$ is violated in a structure when E links $A(n)$ to some nodes abstracted by $A(n')$, but not all of them. If it is violated in the "some but not all" case, then the sharpening is valid. Otherwise it is not. And the answer depends on whether n' is a universally or an existentially quantified variable. □

When we have a potential sharpening $p^v(u_1, \ldots, u_k)$ and each u_i is either a singleton node or came from a universally quantified variable, then the sharpening is legal: because the quantifier is universal, the integrity constraint will evaluate to zero if even one of the p

values is changed. However, if there is a u_i summary node that came from an existentially quantified variable, then changing one of the p values will not change the evaluation of the integrity constraint—it will still yield $1/2$.

We take advantage of this insight in the algorithm. We track in the assignment A not only the node mapped to a variable, but also whether it is universally or existentially quantified. For example, we write $A(n) = \langle u_1, \forall \rangle$ and $A(n') = \langle \boldsymbol{u_2}, \exists \rangle$ when n is universally quantified and n' is existentially quantified, as in the example.

Before returning an answer $p^v(u_1, \ldots, u_k)$ based on a literal $\mathsf{P}(v_1, \ldots, v_k)$, we check whether each v_i is existentially quantified, and if it is, we return the answer only if u_i is a singleton node. The following helper function implements this check.

$\mathrm{Filter}(p^v(v_1, \ldots, v_k), S, A)$
    if for all i, $A(v_i) = \langle u_i, \exists \rangle \Rightarrow \iota(\mathsf{Eq})(u_i, u_i) = 1$
    then return $\{p^v(A(v_1), \ldots, A(v_k))\}$ **else** return $\emptyset$

Based on these insights, we can revise the definition of M as follows. The cases not listed below are the same as before. Note that when the augmented A is used by $[\![\varphi]\!]_{S,A}$, we assume the quantifier information is dropped.

| φ	$M(\varphi, S, A)$
$\mathsf{P}(v_1, \ldots, v_k)$	$\mathrm{Filter}(\mathsf{P}^1(v_1, \ldots, v_k), S, A)$
$\neg \mathsf{P}(v_1, \ldots, v_k)$	$\mathrm{Filter}(\mathsf{P}^0(v_1, \ldots, v_k), S, A)$
$\forall v. \varphi'$	$\bigcup_{u \in U} \{M(\varphi', S, A[v \to \langle u, \forall \rangle]) : [\![\varphi']\!]_{S,A[v \to u]} = 1/2\}$
$\exists v. \varphi'$	$\bigcap_{u \in U} \{M(\varphi', S, A[v \to \langle u, \exists \rangle]) : [\![\varphi']\!]_{S,A[v \to u]} = 1/2\}$
$\mathrm{ITC}(s, t; x, y). \varphi'$	$\bigcap_{P \in \mathrm{Paths}(S, A(s), A(t))} \bigcup_{(a_1, a_2) \in P}$ $\{M(\varphi', S, A[x \to a_1, y \to a_2]) : [\![\varphi']\!]_{S,A[x \to a_1, y \to a_2]} = 1/2\}$

The definition of Paths is slightly different here. The paths it returns still consist of sets of node pairs, but now the nodes are annotated with $\forall/\exists$ information, just like A. The first and last nodes ($A(s)$ and $A(t)$) are quantified as determined by A; the nodes in the middle are always existentially quantified, as befits the TC operator. Note that Paths may return paths that are not simple. An edge may occur once as the first edge, meaning one endpoint may be universally quantified, and again as an inner edge where both endpoints are existentially quantified.

With this revision, the algorithm is now correct except for a very unusual corner case. Before describing that case, we present an example.

Example 31 We revisit the example of the Shared predicate from the beginning of the section. Recall that

$$\mathsf{Shared}(n) := \exists n_1, n_2. \ \mathsf{Next}(n_1, n) \wedge \mathsf{Next}(n_2, n) \wedge n_1 \neq n_2.$$

We consider only the structure S_2 from that example, shown below.

$$X \; \bigcirc \xrightarrow{\quad} \bigcirc \;\; \overset{u_2 \quad \boldsymbol{u_2'}}{\dashrightarrow} \;\; \circledcirc$$
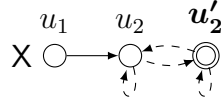
First we form the integrity constraint according to Equation 2.1. For simplicity, we only use one direction of the implication since it is sufficient to obtain the results we need. Desugaring implication and converting to negated normal form yields

$$\varphi = \forall n. \; \mathsf{Shared}(n) \vee (\forall n_1, n_2. \; \neg\mathsf{Next}(n_1, n) \vee \neg\mathsf{Next}(n_2, n) \vee n_1 = n_2).$$

Our goal is to obtain the results $\{\mathsf{Next}^0(u_2, u_2), \mathsf{Next}^0(\boldsymbol{u_2'}, u_2)\}$.

We define ψ as the inner $\forall$-quantified formula so that the integrity constraint is $\varphi = (\forall n. \; \mathsf{Shared}(n) \vee \psi)$. We start by evaluating $M(\forall n. \; \mathsf{Shared}(n) \vee \psi, S, [])$, where $[]$ is the empty assignment. According to the algorithm, we recursively evaluate $M(\mathsf{Shared}(n) \vee \psi)$ at every node and union the results. We concentrate on u_2, since the others simply return $\emptyset$.

Thus we compute $M(\mathsf{Shared}(n) \vee \psi, S, [n \mapsto \langle u_2, \forall \rangle])$. In processing the disjunction, we recognize that $\mathsf{Shared}(u_2) = 0$, so we simply return the result of $M(\psi, S, [n \mapsto \langle u_2, \forall \rangle])$.

ψ is defined as $(\forall n_1, n_2. \; \neg\mathsf{Next}(n_1, n) \vee \neg\mathsf{Next}(n_2, n) \vee n_1 = n_2)$. We first process it at $n_1 = u_1$ and $n_2 = u_2$, causing us to evaluate

$$M(\neg\mathsf{Next}(n_1, n) \vee \neg\mathsf{Next}(n_2, n) \vee n_1 = n_2, S, [n_1 \mapsto \langle u_1, \forall \rangle, n_2 \mapsto \langle u_2, \forall \rangle, n \mapsto \langle u_2, \forall \rangle]).$$

The first disjunct is zero because $\mathsf{Next}(u_1, u_2) = 1$. The third equality disjunct is also zero because $u_1 \neq u_2$. Thus, we simply return

$$M(\neg\mathsf{Next}(n_2, n), S, [n_1 \mapsto \langle u_1, \forall \rangle, n_2 \mapsto \langle u_2, \forall \rangle, n \mapsto \langle u_2, \forall \rangle]) = \{\mathsf{Next}^0(u_2, u_2)\}.$$

Next, we can process ψ at $n_1 = u_1, n_2 = \boldsymbol{u_2'}$. We evaluate

$$M(\neg\mathsf{Next}(n_1, n) \vee \neg\mathsf{Next}(n_2, n) \vee n_1 = n_2, S, [n_1 \mapsto \langle u_1, \forall \rangle, n_2 \mapsto \langle \boldsymbol{u_2'}, \forall \rangle, n \mapsto \langle u_2, \forall \rangle]).$$

As before, only the second disjunct is relevant, giving us

$$M(\neg\mathsf{Next}(n_2, n), S, [n_1 \mapsto \langle u_1, \forall \rangle, n_2 \mapsto \langle \boldsymbol{u_2'}, \forall \rangle, n \mapsto \langle u_2, \forall \rangle]) = \{\mathsf{Next}^0(\boldsymbol{u_2'}, u_2)\}.$$

Note that we were able to return this result only because n_2 was universally quantified.

We union these two results together. No other evaluations of ψ return anything useful, so they form the final result. □

Syntactic restriction.   The algorithm presented is the one that we implement. However, as we already described, it is unsound in an unusual case.

Example 32 Let $\varphi = \forall n. \; \mathsf{E}(n, n)$. Assume S is is a single summary node u with a 1/2 E self-loop.

$$\boldsymbol{u}$$

Let $S^\sharp$ contain two concrete nodes with definite E self-loops.

$$u_1 \qquad u_2$$
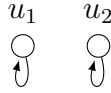
Then algorithm M will infer the sharpening $\mathsf{E}^1(u, u)$, which is invalid in $S^\sharp$. □

The obvious culprit is $\mathsf{E}(n, n)$, where a variable is used twice in the same predicate in an integrity constraint. However, we can craft more sophisticated examples not fitting this model. Consider $\varphi = \forall n, n'.\ \mathsf{E}(n, n') \vee \mathsf{E}(n', n)$, under which M is unsound over the same S, but now $S^\sharp$ must contain an additional E edge from one node to the other (but only in one direction).

To fix this problem, we place a syntactic restriction on the sharpening formulas φ. The restriction says that a universally quantified variable can appear in only one position for a given predicate P. Thus, a formula like $\forall n.\ \mathsf{E}(n, n)$ is invalid because the universally quantified variable n appears in both the first and the second parameter of E. Similarly, $\forall n, n'.\ \mathsf{E}(n, n') \vee \mathsf{E}(n', n)$ is also illegal since n still appears as the first parameter of E and, later, as the second parameter of E.

However, we do allow a variable to appear twice in a predicate if one of them is negated. Thus, the formula $\forall n, n'.\ \mathsf{E}(n, n') \vee \neg\mathsf{E}(n', n)$ is legal because E and $\neg\mathsf{E}$ are treated separately by the restriction. In practice, all the sharpening rules of interest to us meet the restriction.

Given this restriction on integrity constraints, the sharpening algorithm is sound. The proof of this theorem appears in §2.9.

Theorem 1 *Let φ be a sharpening rule that satisfies the syntactic restriction. Let S be a structure, and $S^\sharp$ a concrete structure that embeds into S via f. Assume $[\![\varphi]\!]_S = 1/2$. Assume that $[\![\varphi]\!]_{S^\sharp} = 1$. Assume that $p^v(u_1, \ldots, u_k) \in M(\varphi, S, [\,])$. Then $\iota^\sharp(p)(u'_1, \ldots, u'_k) = v$ for all $\langle u'_1, \ldots, u'_k \rangle$ such that $f(u'_i) = u_i$ for each i.* □

2.5.5   Completeness

Despite the fairly ad-hoc fixes mentioned above, the algorithm is still complete in a limited sense. Its main weakness is that it fails to find every possible sharpening when φ contains positive references to the equality predicate. Equality makes sharpening much more difficult because it allows the sharpening formula to constrain the size of the model. In general, dealing with cardinality is a very difficult problem, one which is not solved by our sharpening algorithm.

Example 33 This examples shows how equalities affect sharpening. Let $\varphi = (\exists n.\ \mathsf{A}(n)) \wedge (\forall n, n'.\ \mathsf{A}(n) \Rightarrow n = n')$. Then consider the structure below.

$$\overset{\boldsymbol{u}}{\textcircled{\scriptsize\bigcirc}} \; \mathsf{A}?$$

The first clause of the integrity constraint says that some node satisfies A. The second clause says that there is at most one node. Thus, it is safe to sharpen A to one here, but algorithm M will not discover this fact. □

Despite this problem, if we ignore constraints involving equality, we can make a useful statement about the completeness of M. The overall goal is to prove that if $p^v(u_1, \ldots, u_k) \notin M(\varphi, S, A)$, then there is some structure S' that embeds into S where p does not equal v and where φ holds.

Ideally, we would like S' to be a concrete structure (meaning no $1/2$ predicate values) and we would like to show that $[\![\varphi]\!]_{S', A'} = 1$. However, this is a much more difficult problem, since it requires us to determine if the model S is satisfiable under the constraint φ. Since φ is an arbitrary formula in first order logic with transitive closure, this problem is undecidable. Instead, we allow S' to be an abstract structure—one that itself may be unsatisfiable as well.

A simple way to explain our completeness result is as follows: "If some sharpening result p^v is not returned by M, then by materializing some nodes in S and changing some predicate values from $1/2$ to 1 we can obtain a structure S' where p^v is violated and where φ is not falsified." This is a much weaker result, but it is still quite useful for understanding whether M will generate a given sharpening. The full statement of the theorem (which is complex) and its proof appear in §2.9.

2.5.6 Use In Practice

We have found that this algorithm works very will in practice. Because it is essentially an instrumented version of the evaluation algorithm, we can use the techniques from §2.4, including query optimization, to run sharpening quickly.

In only a few cases has the algorithm failed to find a correct sharpening. These cases all related to transitive closure, where infinite case analysis is required to prove the sharpening correct. Our completeness result does not apply here. To alleviate the problem, we augment the integrity constraints that we generate for transitive closure predicates. Consider the predicate $\mathsf{NextTC}(s, t) := \mathrm{TC}(s, t; x, y). \; \mathsf{Next}(x, y)$. Normally we would generate the following integrity constraint, which follows from Equation 2.1.

$$\forall s, t. \; \mathsf{NextTC}(s, t) \iff \mathrm{TC}(s, t; x, y). \; \mathsf{Next}(x, y)$$

To improve sharpening, we generate additional integrity constraints.

- $\forall s, t. \; (\mathrm{TC}(s, t; x, y). \; \mathsf{NextTC}(x, y)) \Rightarrow \mathsf{NextTC}(s, t)$

- If Next is known to be a function:

$$\forall x, y, z. \; \mathsf{NextTC}(x, y) \wedge \mathsf{NextTC}(x, z) \wedge \neg\mathsf{NextTC}(z, y) \Rightarrow \mathsf{NextTC}(y, z)$$

- If Next is known to be a function:

$$\forall x, y, z.\ \mathsf{NextTC}(x, y) \wedge x \neq y \wedge \mathsf{Next}(x, z) \Rightarrow \mathsf{NextTC}(z, y)$$

Aside from these forms added for any transitive closure predicate, we have found no need to manually add additional integrity constraints.

Comparison. The TVLA algorithm for sharpening [46] tries to rewrite each integrity constraint into rules of the form

$$\forall v_1, \ldots, v_k.\ \psi \Rightarrow \mathsf{P}(v_1, \ldots, v_k).$$

Then it searches for places where ψ has the value 1 and sets P to 1 at those places. Originally the algorithm searched all tuples $\langle u_1, \ldots, u_k \rangle$, and thus had exponential complexity in k, but Bogulov et al. [5] present an improvement that dramatically increases performance for most rules via incremental evaluation. However, both of these approaches rely on simple syntactic rewriting of the integrity constraints. The automatically generated integrity constraints typically need to be augmented with rules that the user provides. These rules are not checked for soundness, so they may make the analysis unsound.

2.6   Abstraction

This section presents the contributions of this thesis related to abstraction (contribution ④ of the introduction). Abstraction is the process of summarizing an infinite set of items using only a finite description. For example, an abstraction of a linked list is a finite description of the infinitely many linked lists of all possible lengths. The abstraction we present in this section is called the DESKCHECK abstraction.

Recall that TVLA uses a disjunctive abstraction. An abstract element in the TVLA domain is a disjunction of three-valued structures. If $E = S_1 \vee \cdots \vee S_n$, then the meaning of E is $\gamma(E) = \gamma(S_1) \cup \cdots \cup \gamma(S_n)$.

When computing the join of two abstract elements, E and E', TVLA tries to avoid generating large disjunctions. Rather than simply concatenating the disjuncts E and E', TVLA will compute the canonical node names of each structure in E and E'. Each structure S gets a canonical structure name $N(S)$ equal to the canonical names of its nodes. If $E = S_1 \vee \cdots \vee S_n$ and $E' = S'_1 \vee \cdots \vee S'_n$ and if there are some S_i and S'_j so that $N(S_i) = N(S'_j)$, then a single disjunct appears in the result to represent both S_i and S'_j. Consequently, at most one disjunct appears in the result for each canonical structure name. This ensures a finite abstraction. (See the introduction for more information.)

Example 34 Consider joining the following two abstract elements, $E = S_0 \vee S_2$ and $E' = S_3$. Let S_i be a structure abstracting a linked list of i elements. We assume the abstraction predicates are $\mathcal{A} = \{\mathsf{Head}, \mathsf{HeadReaches}\}$ (assuming *head* points to the head of the list). We show the structures here, with abbreviated predicate names.

S_0	S_2	S_3
(empty)	u_1 u_2 H,HR ◯⟶◯ HR	u_1 $\boldsymbol{u_2}$ H,HR ◯- - ⟶◎ HR

Note that the canonical abstraction has already been applied, causing the second and third nodes of S_3 to be collapsed into a summary node.

The canonical structure names are as follows.

$$N(S_0) = \emptyset$$
$$N(S_2) = \{\{\mathsf{Head}, \mathsf{HeadReaches}\}, \{\mathsf{HeadReaches}\}\}$$
$$N(S_3) = \{\{\mathsf{Head}, \mathsf{HeadReaches}\}, \{\mathsf{HeadReaches}\}\}$$

Since the canonical structure name of S_3 is the same as the name of S_2, the result of the join operation is $S_0 \vee (S_2 \sqcup S_3)$. Note that $(S_2 \sqcup S_3) = S_3$. □

Scalability. The canonical abstraction guarantees that TVLA's abstractions are finite, but it does not guarantee that they are small. In practice, descriptions may be very large, involving many disjuncts.

Consider a single linked list with an unknown number of nodes. We need three disjuncts to represent it. The first one represents the empty case; this structure has canonical name $\emptyset$. The second one represents a single node; this structure has canonical name $\{\{\mathsf{Head}, \mathsf{HeadReaches}\}\}$. The third structure is the "two or more nodes" case; it has canonical name $\{\{\mathsf{Head}, \mathsf{HeadReaches}\}, \{\mathsf{HeadReaches}\}\}$.

With multiple lists, the number of disjuncts increases exponentially. Consider two list pointers, x and y, pointing to distinct lists. Let there be predicates X and Y for the head pointers and RX and RY for reachability. Then there are nine disjuncts with the following canonical structure names.

$$N(S_{00}) = \emptyset$$
$$N(S_{10}) = \{\{\mathsf{X}, \mathsf{RX}\}\}$$
$$N(S_{01}) = \{\{\mathsf{Y}, \mathsf{RY}\}\}$$
$$N(S_{11}) = \{\{\mathsf{X}, \mathsf{RX}\}, \{\mathsf{Y}, \mathsf{RY}\}\}$$
$$N(S_{20}) = \{\{\mathsf{X}, \mathsf{RX}\}, \{\mathsf{RX}\}\}$$
$$N(S_{02}) = \{\{\mathsf{Y}, \mathsf{RY}\}, \{\mathsf{RY}\}\}$$
$$N(S_{12}) = \{\{\mathsf{X}, \mathsf{RX}\}, \{\mathsf{Y}, \mathsf{RY}\}, \{\mathsf{RY}\}\}$$
$$N(S_{21}) = \{\{\mathsf{X}, \mathsf{RX}\}, \{\mathsf{RX}\}, \{\mathsf{Y}, \mathsf{RY}\}\}$$
$$N(S_{22}) = \{\{\mathsf{X}, \mathsf{RX}\}, \{\mathsf{RX}\}, \{\mathsf{Y}, \mathsf{RY}\}, \{\mathsf{RY}\}\}$$

In general n lists require 3^n disjuncts. This is clearly insupportable.

2.6.1 Present = 1/2

The crux of the problem is that in TVLA a summary node always abstracts *at least one* concrete node. If a summary node could represent zero or more concrete nodes, then the scalability problem would mostly be solved. Only one disjunct would be needed to abstract a single list. Consequently, the number of disjuncts for n lists would fall from 3^n to $1^n = 1$.

In this section, we propose a new property for nodes called *present*. If a node is present, then it behaves as normal. If a node is not present, then it is as if the node does not exist. If present is 1/2, then the node may or may not be present in the structure.

It is tempting to treat present as if it were another predicate, much as summary nodes are formalized using the equality predicate. Unfortunately, this is not possible. To see why, we must make a diversion and consider the logical meaning of a TVLA structure. Consider the following structure.
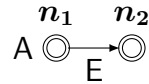
$$\mathbf{n_1} \quad \mathbf{n_2}$$
$$\text{A } \circledcirc \xrightarrow[\text{E}]{} \circledcirc$$

We write the logical denotation of S as follows. (See Yorsh et al. [51] for more information on the logical meaning of heap structures.)

$$\exists n_1.\; \mathsf{A}(n_1)$$
$$\exists n_2.\; \neg\mathsf{A}(n_2)$$
$$\forall n, n'.\; \mathsf{A}(n) \wedge \neg\mathsf{A}(n') \Rightarrow \mathsf{E}(n, n')$$

Each existential quantifier corresponds to a node in the structure, postulating its existence based on the abstraction predicates it satisfies. Each universally quantified fact corresponds to some definite predicate fact (i.e., not 1/2) for a non-abstraction predicate, like E above. Even summary-node-ness can be encoded this way. If n_1 were a singleton, then we would have the fact $\forall n, n'.\; \mathsf{A}(n) \wedge \mathsf{A}(n') \Rightarrow n = n'$.

Notice now that the present property discussed above *cannot* be encoded as a predicate because it is not representable with a universal quantifier. Instead, present-ness affects the initial existential quantifiers. This is a fundamental change to the meaning of structures so it must be handled completely differently.

Semantics. Recall that a structure S is formalized as $\langle U, \iota \rangle$. We update this to include present information as $\langle U, \rho, \iota \rangle$. The new value ρ is a function mapping nodes to $\{0, 1/2, 1\}$, denoting whether they are present. This change requires a few definitions to be updated. First we change the definition of embedding.

Definition 3 Given two TVLA structures $S = \langle U, \rho, \iota \rangle$ and $S' = \langle U', \rho', \iota' \rangle$, we say that S *embeds* into S' if there exists a function f (called the *embedding function*) that satisfies the following conditions.

- For any predicate P and any nodes $n_1, \ldots, n_k \in U$,

$$\iota(\mathsf{P})(n_1, \ldots, n_k) \sqsubseteq \iota'(\mathsf{P})(f(n_1), \ldots, f(n_k)).$$

- For any node $n' \in U'$,

$$\left(\bigvee_{n \in U : f(n) = n'} \rho(n) \right) \sqsubseteq \rho'(n').$$

When these conditions hold we write $S \sqsubseteq S'$. □

The difference between this definition and Definition 2 is that instead of requiring the embedding function to be a surjection, we add the extra condition on ρ. This condition reduces to surjectivity when ρ and ρ' always equal 1.

Note that this definition does not match our intuition in one way. Let S be a structure with a single node where $\rho = 0$ and let S' be an empty structure. Although these structures seem semantically equivalent, they are not according to the embedding relation. However, we can simply remove the $\rho = 0$ node from S or add a $\rho = 0$ node to S' to satisfy the definition.

We also redefine the notion of a concrete structure. Now we say a structure is concrete if all predicate values are definite and if ρ is 1 everywhere. Then we restate the definition of γ:

$$\gamma(S) = \{S_0 : S_0 \text{ is concrete} \wedge S_0 \sqsubseteq S\}.$$

The modified definition of a concrete structure allows us to ignore the $\rho = 0$ issue mentioned above, at least as it related to γ. We can always add new nodes with $\rho = 0$ to any abstract structure without changing its meaning. Any concrete structure that embeds into the original structure will embed into the new one and vice versa.

Example 35 Consider a heap with two singleton nodes, n and n'. Let $\rho(n) = 1$ and $\rho(n') = 1/2$. Additionally, let the binary predicate E hold from n to n' (i.e., $\iota(\mathsf{E})(n, n') = 1$).

There are two concrete structures that embed into this one. One structure has both nodes.

$$S_2 = \langle \{n, n'\}, \rho_2, \iota_2 \rangle \qquad \rho_2(n) = \rho_2(n') = 1 \qquad \iota_2(\mathsf{E})(n, n') = 1$$

The other structure has only n.

$$S_1 = \langle \{n\}, \rho_1, \iota_1 \rangle \qquad \rho_1(n) = 1$$

The interesting point is that S_1 contains no E edges even though the original structure required E to be 1 between n and n'. Nevertheless, the embedding relationship holds. □

Join. We still define the join in essentially the same way. Let m be a bijection mapping nodes of a structure $S = \langle U, \rho, \iota \rangle$ to nodes of a structure $S' = \langle U', \rho', \iota' \rangle$. Then $S \sqcup S' = \langle U, \rho'', \iota'' \rangle$.

$$\iota''(\mathsf{P})(u_1, \ldots, u_k) = \begin{cases} \iota'(\mathsf{P})(m(u_1), \ldots, m(u_k)) & \text{if } \exists i.\ \rho(u_i) = 0 \\ \iota(\mathsf{P})(u_1, \ldots, u_k) & \text{if } \exists i.\ \rho(m(u_i)) = 0 \\ \iota(\mathsf{P})(u_1, \ldots, u_k) \sqcup \iota'(\mathsf{P})(m(u_1), \ldots, m(u_k)) & \text{otherwise} \end{cases}$$

$$\rho''(u) = \rho(u) \sqcup \rho'(m(u))$$

As before, both S and S' embed into the joined structure—S via the identity embedding function and S' via m^{-1}.

Query evaluation. The new *present* property forces us to modify some important parts of the system. The most important is query evaluation. Imagine that the query $\exists x.\ \varphi(x)$ is being evaluated and that, for some node u, $\varphi(u)$ is true. Normally we would return true in this case. However, if $\rho(u) = 0$, we must find another u or else return false. If $\rho(u) = 1/2$, we must find another u or return $1/2$. Essentially, we must convert every query of the form $\exists x.\ \varphi$ into $\exists x.\ \rho(x) \wedge \varphi$. Similarly, we convert every $\forall x.\ \varphi$ query into $\forall x.\ \rho(x) \Rightarrow \varphi$.

Transitive closure queries can be similarly transformed, although there are two possible ways to do so. We could transform the query $\mathrm{TC}(s, t; x, y).\ \varphi$ as follows: $\mathrm{TC}(s, t; x, y).\ \rho(x) \wedge \varphi$ or $\mathrm{TC}(s, t; x, y).\ \rho(y) \wedge \varphi$ (or we could use ρ on both x and y but this is redundant).

Unfortunately, neither way is perfectly precise. To see why, we can consider the transitive closure query as an infinite disjunction $P_0 \vee P_1 \vee \cdots$, where P_i holds when there is a path of length i from s to t. Then we write P_i as follows:

$$\exists x_1, x_3, \ldots, x_{i-1}.\ \varphi(s, x_1) \wedge \varphi(x_1, x_2) \wedge \cdots \wedge \varphi(x_{i-2}, x_{i-1}) \wedge \varphi(x_{i-1}, t).$$

This formula has the advantage that we can transform it as we do the existential formulas. It becomes the following.

$$\exists x_1, x_3, \ldots, x_{i-1}.\ \rho(x_1) \wedge \cdots \rho(x_{i-1}) \wedge \varphi(s, x_1) \wedge \varphi(x_1, x_2) \wedge \cdots \wedge \varphi(x_{i-1}, t).$$

This is a *different* formula than the one obtained by adding $\rho(x)$ or $\rho(y)$ to φ. That formula would include an extra ρ term s or t respectively. The difference turns out to be significant.

Example 36 Consider a structure S with three singleton nodes: n_1, n_2, and n_3, with predicates A, B, and E as follows.
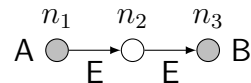
The gray coloration means that $\rho(n_1) = \rho(n_3) = 1/2$.

Suppose we want to evaluate the query $\mathsf{A}(s) \wedge \mathsf{B}(t) \wedge \mathrm{TC}(s, t; x, y).\ \mathsf{E}(x, y)$. It is valid to return the answer 1 at $s = n_1, t = n_3$. However, we will get the answer $1/2$ using the transformed version of the formula. □

The only way to fix the problem is to change the evaluation algorithm to account for ρ. The algorithm must weight results by ρ only for the inner nodes. Recall the previous definition for evaluation of $[\![\mathrm{TC}(s,t;x,y).\ \varphi']\!]_{S,S',A}$ from §2.4.1; we update it as follows.

$$\iota(\mathsf{Eq})(A(s),A(t)) \vee \bigvee_{P\in\mathrm{Paths}(S,A(s),A(t))} \left(\bigwedge_{n\in\mathrm{In}(P,A(s),A(t))} \rho(n) \right) \wedge \left(\bigwedge_{\langle n_1,n_2\rangle\in P} [\![\varphi']\!]_{S,S',A[x\to n_1,y\to n_2]} \right)$$

We define the function $\mathrm{In}(P,n_s,n_t)$ on a path P to return the "inner" nodes, as follows.

$$\mathrm{In}(P,n_s,n_t) = \{n : \exists n'.\ (\langle n,n'\rangle \in P \wedge n \neq n_s) \vee (\langle n',n\rangle \in P \wedge n \neq n_t)\}$$

We cannot always exclude n_s and n_t as inner nodes because the path may loop around through s or t. We detect this by noticing that the path *enters* s (or *leaves* t); in such cases, s (or t) is considered an inner node.

Rather than transforming universal and existential quantifiers, we can change their evaluation rules as well, giving us a more uniform treatment of the subject. The results are shown in Table 2.10.

φ	$[\![\varphi]\!]_{S,S',A}$ (assume $S = \langle U,\iota\rangle$)
$\forall v.\ \varphi'$	$\bigwedge_{n\in U} \neg\rho(n) \vee [\![\varphi']\!]_{S,S',A[v\to n]}$
$\exists v.\ \varphi'$	$\bigvee_{n\in U} \rho(n) \wedge [\![\varphi']\!]_{S,S',A[v\to n]}$

Table 2.10: Semantics of formulas.

Sharpening.   There are two changes needed in the sharpening algorithm. First, we saw in the previous paragraph that queries must be transformed to deal with the *present* property. We perform the same transformations on sharpening queries. Consequently, sharpening can now return two kinds of results: the normal kind of the form $p^v(u_1,\ldots,u_k)$, and a new kind of the form $\rho^v(u)$. The new kind says that $\rho(u)$ should be sharpened to v. Here is an example of how this might come about.

Example 37 Let S be a structure containing a single node n where $\rho(n) = 1/2$. Let A be a predicate and let $\mathsf{A}(n) = 1/2$. Suppose that we have the integrity constraint $\exists x.\ \mathsf{A}(x)$.

$$\overset{n}{\mathsf{A}?\ \bigcirc}$$

We transform this constraint into $\exists x.\ \rho(x) \wedge \mathsf{A}(x)$. Then we use our standard sharpening algorithm. On the subquery $\mathsf{A}(x)$ it returns $\mathsf{A}^1(n)$. On the subquery $\rho(x)$ it returns $\rho^1(n)$. The $\wedge$ operator unions these sharpenings so that the final result is $\{\mathsf{A}^1(n), \rho^1(n)\}$, meaning that n must be present and must satisfy A. □

The second modification is more subtle. To understand it, we change the above example somewhat.

Example 38 We start with the same structure S but suppose that we have the integrity constraint $\forall x.\ \mathsf{A}(x)$ instead.

$$n$$
$$\mathsf{A}?\ \bullet$$

We transform this constraint into $\forall x.\ \neg\rho(x) \vee \mathsf{A}(x)$. In this case the standard sharpening algorithm returns no results: at the $\vee$ operator, it intersects $\{\rho^0(n)\}$ with $\{\mathsf{A}^1(n)\}$, producing nothing.

In fact, it is legal to sharpen A to 1 at n. (We cannot sharpen $\rho(n)$ at all.) The reason is that if n exists then the integrity constraint requires it to satisfy A. If it doesn't exist then there is no harm in setting $\mathsf{A}(n) = 1$ since it will have no effect on the meaning of the element. □

How do we make our sharpening algorithm to generate this result? We modify its handling of disjunctions when both sides evaluate to $1/2$. The algorithm used to return $M(\varphi_1, S, A) \cap M(\varphi_2, S, A)$. Now we allow a result $p^v(u_1, \ldots, u_k)$ to be returned if it is returned by one side of the disjunction as long as the other side returns $\rho^0(u_i)$ for some i. Formally, we return the following additional results.

$$\{p^v(u_1, \ldots, u_k) : \exists i.\ p^v(u_1, \ldots, u_k) \in M(\varphi_i, S, A) \wedge \exists j.\ \rho^0(u_j) \in M(\varphi_{1-i}, S, A)\}$$

**Focus.** Having ρ available to us, we can simplify the focus operation as follows.

Example 39 Recall the focus that took place in Example 25. We started with the following structure.

$$u_1 \quad \boldsymbol{u_2}$$
$$\mathsf{X}\ \bigcirc\text{-}\text{-}\!\rightarrow\!\circledcirc$$
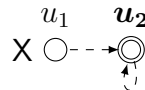
Then we focused $next[x]$, yielding two structures.

S_1		S_2		
u_1	u_2	u_1	u_2	$\boldsymbol{u_2'}$
$\mathsf{X}\ \bigcirc\!\longrightarrow\!\bigcirc$		$\mathsf{X}\ \bigcirc\!\longrightarrow\!\bigcirc\text{-}\text{-}\!\rightarrow\!\circledcirc$		

In some sense, S_2 is the more general structure. We need S_1 only because the summary node $\boldsymbol{u_2'}$ in S_2 cannot represent zero concrete nodes. But using ρ, we can generate a single structure into which both S_1 and S_2 embed.

$$u_1 \quad u_2 \quad \boldsymbol{u_2'}$$

X
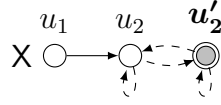
This structure differs from S_2 only in that $\rho(\boldsymbol{u_2'}) = 1/2$. □

Using $\rho = 1/2$ allows us to return only one result for most focus operations.

2.6.2 Disjunctive Abstraction

The reason for introducing $\rho = 1/2$ nodes is to solve TVLA's scalability issues. An obvious way to do this is to eliminate disjunctions entirely from domain elements. Given a domain element $S_1 \vee \cdots \vee S_n$, we want to form a single structure S that overapproximates every S_i.

We cannot form $S_1 \sqcup \cdots \sqcup S_n$ directly because the join operation still requires a bijection between nodes. Instead, we collect all the canonical node names, $A = N(S_1) \cup \cdots \cup N(S_n)$. Then we generate new structures S_i' by adding nodes from A to S_i if they are not already present; these nodes are added with $\rho = 0$. Finally, we use the join algorithm to form $S_1' \sqcup \cdots \sqcup S_n' = S$. We call this abstraction "*single*" because every domain element contains a single structure.

Unfortunately, the single abstraction tends to be imprecise. By merging so much, it throws away important information. Instead, we have found a compromise between TVLA's disjunctive abstraction and the single abstraction.

When joining a structure S with a structure S', TVLA represents the result as $S \sqcup S'$ if $N(S) = N(S')$ (i.e., if the canonical names match), and as $S \vee S'$ otherwise. We change the condition to $N(S) \subseteq N(S')$ or $N(S) \supseteq N(S')$. Doing so requires some tweaking, of course. Since computing $S \sqcup S'$ requires a bijection between the nodes, we introduce new nodes with $\rho = 0$ to the "smaller" structure (just as we do in the single abstraction).

Example 40 Consider the following three structures, which make up the normal linked list abstraction in traditional TVLA. We abbreviate the head pointer predicate as H and the reachability predicate as R.

S_0	S_1	S_2
	u_1	$u_1 \quad \boldsymbol{u_2}$
(empty)	H,R ○	H,R ○- - →○ R

Our goal is to join these three structures together. Their canonical names are as follows:

$$N(S_0) = \emptyset$$
$$N(S_1) = \{\{H, R\}\}$$
$$N(S_2) = \{\{H, R\}, \{R\}\}$$

TVLA would normally leave these three structures separate, leading to an exponential number of disjuncts where multiple lists are involved.

The modified heuristic above joins these three structures together into $S_0 \sqcup S_1 \sqcup S_2$, yielding the following structure.

One unfortunate property of this join algorithm is that it is not associative. We have not found this to be a problem in practice. Nevertheless, we give an example that exposes the problem.

Example 41 Consider the following three structures, with the unary predicates X, Y, Z, and P. The first three are abstraction predicates and the last one is not.

Consider trying to join these three structures together. We will not join S_a with S_b because there is no subset relationship between their node names. However, we can join S to either S_a or S_b. Which one we choose affects the precision of the result.

First consider doing the join $(S_a \sqcup S) \sqcup S_b$. This causes S to be joined with S_a, leading to a disjunction of the following two structures.

On the other hand, $S_a \sqcup (S \sqcup S_b)$ yields the following disjunction.

The second version is more precise because it contains definite information about P everywhere.

2.6.3 Control Flow-Sensitive Abstraction Predicates

In TVLA, a predicate is either an abstraction predicate or it is not. This section introduces the freedom for the user to temporarily change whether a predicate is an abstraction predicate. This allows him to use a coarse abstraction most of the time while switching to a finer abstraction when invariants need to be temporarily broken. We give an example now.

Example 42 We use an example similar to the linked list one earlier. However, now we suppose that the list is terminated by a *null* object that satisfies the predicate N, which is an abstraction predicate. If the head predicate, H, is also an abstraction predicate, then we need two disjuncts to represent lists of arbitrary size.
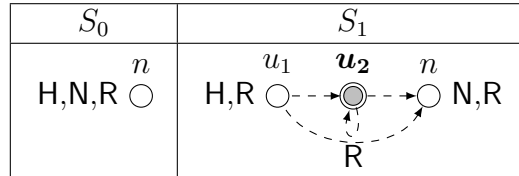
These two structures will not be merged because the node names are incomparable. Sometimes it is beneficial for these two structures to be kept separate. For example, when traversing a list, it is useful to keep the "null" case separate because the loop test will select it out anyway. But throughout most of the program we want to merge these cases to avoid state-space explosion. At those times we remove the H predicate from $\mathcal{A}$, resulting in the following structure.

The implementation of this feature is straightforward. The annotations `@enable(P)` and `@disable(P)` add and remove predicate P from the set of abstraction predicates $\mathcal{A}$. When P is removed, we re-apply the canonical abstraction to coarsen the abstraction. Adding P to $\mathcal{A}$ performs case splitting everywhere $P = 1/2$ and then sharpens the results.

2.6.4 Experiments

In this section we try to quantify the effects of the Deskcheck abstraction (the one presented here) versus the normal TVLA abstraction. We use our `thttpd` benchmark. Our first experiment compares the running time when using the TVLA abstraction versus the Deskcheck abstraction. We also tested the benefit of using flow-sensitive abstraction predicates. We measured both the analysis time and the sizes of the inferred loop invariants, in terms of the number of disjuncts. For each loop that appears in the program, we added up the number of disjuncts that were inferred at that program point. Since there are many loops, we present the minimum, median, and maximum loop invariant size.

The Deskcheck abstraction runs 8.6 times faster than the TVLA abstraction with fixed abstraction predicates. It also uses many fewer disjuncts. Besides the performance difference, presenting fewer disjuncts to the user makes the analysis results much easier to understand. Figure 2.3 presents histograms of the loop invariant sizes. Most loops in the Deskcheck abstraction have only two disjuncts in their inferred loop invariants. This is the minimum number: we record the loop invariant before testing the loop condition, so it includes the loop exit case. (Loops with only one disjunct are typically non-terminating.)

Abstraction	Abstraction predicates	Running time	# disjuncts
TVLA	Fixed	574.91	6/18/70
TVLA	Flow-sensitive	460.71	4/12/47
Deskcheck	Flow-sensitive	66.87	1/2/6

Table 2.11: Performance comparison of abstractions. The number of disjuncts is shown in the form of min/median/max.

Besides its performance advantages, we found that the DESKCHECK abstraction makes analysis results easier to understand. Analyzing a program with the TVLA abstraction generates a huge number of disjuncts at each program point. If the user finds an error in the program, or if the analysis results are imprecise, the user must pore over each disjunct to understand the problem. In TVLA, this takes the form of paging through a hundred-page PostScript file of analysis results. The DESKCHECK abstraction collapses the disjuncts, often into a single structure, making the results much easier to read and understand.

## 2.7 Related Work

2.7.1 TVLA

The TVLA system of Sagiv, Reps, and Wilhelm [46] forms the basis of the heap domain. The most complete description of TVLA is in the TVLA journal paper [46]. However, a number of innovations have been made since then that we describe.

Abstractions. The disjunctive abstraction used by TVLA, in which two structures with the same canonical name are merged, is described by Manevich et al. [34]. §2.6 describes our modifications to this abstraction to make it more concise and efficient by collapsing structures if one's canonical name is a subset of the other's.

§2.6 also describes $\rho = 1/2$ nodes. Our use of these nodes is not a new innovation. In an early paper, Sagiv, Reps, and Wilhelm [45] describe a shape analysis system using a single structure per program point. A command-line option causes TVLA to run in this mode, called *single mode*. TVLA uses a property similar to ρ (called *active*) to represent summary nodes that may not be present. It also uses a join algorithm similar to ours.[3] However, its sharpening algorithm is not able to make inferences about ρ as ours does. The single mode does not appear to be used much because of the imprecision of having one structure per program point. Our subset-based abstraction, along with the improved sharpening, seems to solve this problem.

Nodes with $\rho = 1/2$ are also addressed by Arnold [1]. That paper presents an analysis based on TVLA where all summary nodes have $\rho = 1/2$. Unlike TVLA's single abstraction,
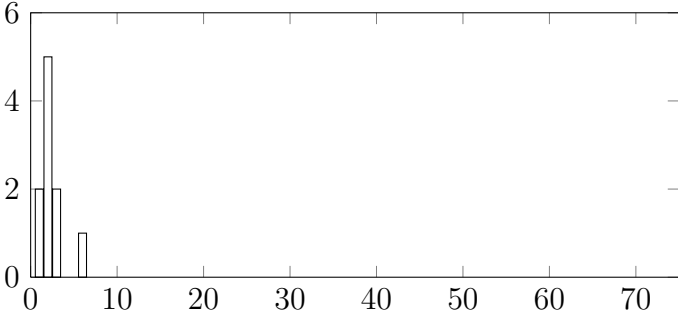
---

[3]See `tvla.core.generic.GenericSingleTVSSet`.
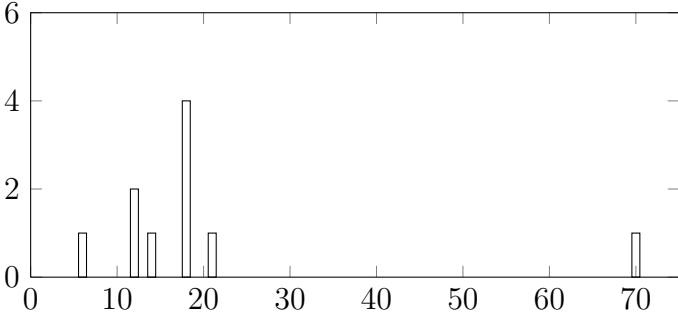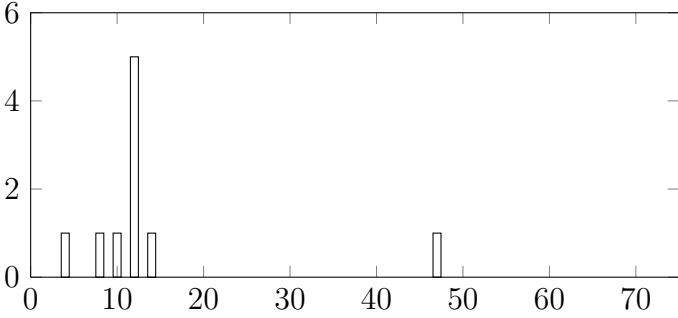
Figure 2.3: Histograms of the size of loop invariants under different abstractions. (a) TVLA abstraction with fixed abstraction predicates. (b) TVLA abstraction with flow-sensitive abstraction predicates. (c) Deskcheck abstraction with flow-sensitive abstraction predicates.

that work allows multiple structures per program point. The extra flexibility in summary nodes allows it to represent most invariants using a single structure. "Degenerate" structures are then eliminated since they embed into the more general structure. It is unclear how this strategy compares to our subset-based abstraction. However, their experiments suggest that it uses more structures per program point on similar programs (singly linked list manipulation).

A different approach to the state-space explosion problem is taken by Manevich et al. [33]. This work allows a domain element to contain conjunctions of structures as well a disjunctions. Thus, the abstraction for two linked lists, L and L', might look as follows: $(S_0 \vee S_1 \vee S_2) \wedge (S_0' \vee S_1' \vee S_2')$. Each of the structures S_i describes one case of list L while using a very imprecise abstraction for list L'. The structures S_j' use a precise abstraction of L' and an imprecise abstraction of L. Together they describe both lists precisely as long as no correlations exist between the lengths of the lists. The advantage over the usual approach is that n lists require $3n$ structures rather than 3^n structures. Compared to our approach, their decomposition technique is more powerful but also more complicated, both in theory and in implementation. Additionally, the user is required to describe how the heap should be decomposed into separate conjunctions. Implementing transformers is also more complex.

Finite differencing. Using finite differencing to generate transfer functions was first described by Reps, Sagiv, and Loginov [40, 41]. We improve on this technique by introducing the definite operator, $\mathbf{1}[\cdot]$, and by refining the finite differencing of transitive closure. A related piece of work by Yorsh et al. [52] uses theorem provers to generate transfer functions. This technique may be more precise than finite differencing, but it runs much more slowly.

Sharpening. We have already compared TVLA's sharpening algorithm to our own. The best descriptions of TVLA's algorithm are in Sagiv et al. [46] and in Bogudlov's work on improving the performance of sharpening [5].

2.7.2 Automated Separation Logic Analyses

Analyses based on separation logic form the other main branch of shape analysis research. The first paper describing a program analysis using separation logic presents the Smallfoot system of Berdine et al. [4]. Other tools include Space Invader [19], SLAyer [49], and Xisa [9]. Separation logic analyses tend to be more efficient than TVLA but less powerful. Almost all separation logic analyses focus on programs that manipulate linked lists (Xisa [9] is the sole exception).

An important speed advantage that separation logic analyses have is that they do not require sharpening. A separation logic analysis uses summarization and materialization much like TVLA, but no sharpening is needed after materialization. Given that sharpening accounts for most of the work in a TVLA-based analysis, this is an important advantage. Note, though, that sharpening imposes only a constant-factor performance cost on TVLA; it

may cause the analysis to run 2x or even 10x slower, but not polynomially or exponentially slower. Thus, avoiding sharpening improves performance but not scalability.

The main *scalability* advantage of separation logic analyses is their sophisticated join algorithms, which allow them to use only a single disjunct per program point (much as we try to do in §2.6). Separation logic analyses have been shown to scale to fairly large programs. The largest program is a Windows Firewire driver that is over 10,000 lines of code [49]. However, this result should be treated with some care. Device drivers are often fairly stylized, using only a few data structure patterns that are easy to analyze. Additionally, the authors of that paper modified the driver by eliminating arrays and by converting some doubly linked lists into singly linked lists. Nevertheless, the result is very impressive. We hope that our techniques from §2.6 would be as successful as theirs, but their is no evidence either way.

A third advantage of separation logic analyses is that interprocedural analysis is somewhat easier. Since the heap is broken into disjoint pieces, the portion to be used by a callee can be disconnected from the caller's heap. The call is then analyzed in isolation and the resulting heap is stitched back into the caller's original heap without much effort. In TVLA this process is made more difficult because instrumentation predicate values can depend on nodes arbitrarily far away. Although some progress has been made on interprocedural analysis in TVLA [28, 42, 43], the goal has typically been to improve precision rather than scalability. This thesis does not address the problem of interprocedural analysis at all.

The downside of analyses based on separation logic is that they are more restricted than TVLA. The *language* of separation logic is a powerful formalism for reasoning about data structures, but the automated analyses based on it use only simple fragments. Most of the analyses mentioned above are limited to analyzing linked list-like structures. Berdine et al. [3] describe a technique to handle hierarchical lists of lists.

Chang and Rival [9] present a much more flexible technique in which the user describes a data structure via a *checker*. A checker is a function that traverses a data structure and ensures that it satisfies some desired invariants. The language used for checkers is somewhat restricted; in particular, a node may only be traversed once. Nevertheless, checkers allow for a much more expressive analysis. They have been shown to work for binary trees, including red-black trees.

Despite these advances, analyses based on separation logic are still limited by their fairly simplistic materialization heuristics. To see how, consider the following data structure:

```
1  struct node {
2    struct node *left, *right;
3    struct node *next;
4  };
5  struct node *tree_root;
6  struct node *list_head;
```

The `left` and `right` pointers form a binary tree rooted at `tree_root`. The `next` pointers form a link list, headed by `list_head`, *through the same set of nodes as the tree* (or possibly

a subset). The following structure is an example, where *next* pointers are dotted and solid edges show *left* and *right* pointers.
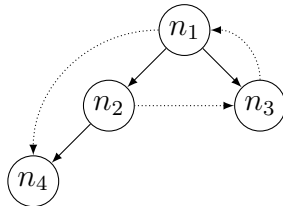
$$\texttt{tree\_root} = n_1 \quad \texttt{list\_head} = n_2$$

This structure might appear in an operating system, where the tree of nodes is a process tree and the linked list forms a ready queue of processes.

The only clear way to represent such a structure is to use two recursive predicates: one to describe the tree fields and one to describe the list fields, as follows:

$$T(\alpha) := (\alpha = \textit{null}) \ \vee \ (\exists \beta, \gamma. \ \textit{left}[\alpha] \mapsto \beta \ * \ \textit{right}[\alpha] \mapsto \gamma \ * \ T(\beta) \ * \ T(\gamma))$$
$$L(\alpha) := (\alpha = \textit{null}) \ \vee \ (\exists \beta. \ \textit{next}[\alpha] \mapsto \beta \ * \ L(\beta))$$

Then we can describe the state via $T(\texttt{tree\_root}) * L(\texttt{list\_head})$. Technically, since the fields that each predicate describes are distinct, the separating conjunction here is valid (at least in Xisa).

Unfortunately, unfolding then becomes a problem. If, in the middle of traversing the list, the programmer begins traversing down the tree, there is a problem: we must unfold the tree predicate "in the middle," at the place where the list iteration has stopped. No separation logic analysis is currently capable of doing this. Xisa uses an unfolding heuristic where two fields of the same object must be a finite number of checker unfoldings apart. No other analysis even supports such predicates.

This is not to say that the problem is insolvable. It simply requires more complex heuristics. Nevertheless, finding a heuristic that is powerful, elegant, and fast is quite a challenge. TVLA's approach, based on focus and sharpening, is powerful, elegant, and slow.

Ultimately, practitioners are faced with a choice between two imperfect techniques. TVLA has been proven to work on some very complex algorithms (garbage collectors, concurrent wait-free data structures, the `thttpd` cache), but it does not yet scale to large programs. The abstraction techniques of §2.6 should improve TVLA's scalability, but the lack of a good interprocedural analysis makes it impossible for us to test this hypothesis. On the other hand, separation logic analyses are reasonably expressive and already have been tested on much larger programs.

Aside from the trade-offs just mentioned, we have another reason for using TVLA as the basis of our heap analysis. TVLA predicates are more *compositional* than separation logic predicates. In TVLA, each property of a data structure—for example, cyclicity, sharing, reachability, or a numerical property—is characterized by a separate predicate. Separation

logic typically captures all of these properties with a single predicate. In TVLA, to analyze a data structure that combines properties of a list and a tree, we use existing list predicates togerther with existing tree predicates. Separation logic requires a new, combined predicate. Hence, we say that TVLA is more composable. We believe that composability makes our analysis easier to use.

In the next chapter, we describe a combined domain for reasoning about heap and integer properties. We use predicates to share information between the heap domain and the integer domain. Individually, TVLA predicates are less constraining than separation logic predicates, so they allow us to share information with the combined domain at a finer granularity. This feature makes for a more precise combined analysis.

2.7.3   Hoare-style Verification

This verification methodology uses user-specified loop invariants and machine-generated verification conditions to check program correctness. This work can be split into two pieces, depending on whether the proving is done manually or automatically.

Automated approaches.   McPeak and Necula [35] describe an automatic technique based on loop invariants. Their logic does not support transitive closure reasoning. Instead, data structure specifications are entirely local, such as $n.\mathtt{next.prev} = n$. When a data structure cannot be described this way, the code must be changed. In practice, this means that back pointers must be added to singly linked lists and parent pointers to trees. They use a modified version of the Simplify theorem prover for deduction. For the local specifications described, they have a complete decision procedure. Verification time is low in practice, although annotation overhead varies between 15% and 153% of the original program size.

The Pointer Assertion Logic Engine [37] is another automatic tool using loop invariants. Specifications are written in monadic second-order logic and checked via the MONA tool [27]. Monadic second-order logic can express a form of reachability, but it has limited support for integer reasoning. MONA checks these specifications via automata conversion. Although MONA is heavily optimized, its complexity is non-elementary (it is a tower of exponents whose height is the number of quantifier alternations in the specification). In our limited experience, this worst-case complexity is realized alarmingly often. However, expert users are probably able to avoid cases that lead to bad performance.

A third automated technique is the HAVOC tool by Lahiri and Qadeer [29]. These authors use a decidable fragment of first-order logic, including a form of reachability. In this case, the decision procedure is NP-complete. Verification times in practice are relatively low for the experiments that are described. However, their technique seems to depend fairly strongly on the very advanced Z3 theorem prover; Simplify is unable to prove some of their examples. No information on annotation burden is provided.

Manual approaches. The Jahob system uses more powerful logics to express loop invariants [53, 54]. These formulas are checked using a variety of systems, including MONA, Coq, and Isabelle. The latter two, being interactive theorem provers, require user guidance. Programs are written in a language that combines loop invariants with proofs of verification conditions. These annotations include typical proving hints like case splits and quantifier instantiation.

Another example is the Ynot system [10]. Programs and specifications are written in Coq, allowing an ML program to be extracted. The specification language uses separation logic. Unlike the separation logic analyses above, the proof process is interactive, so users can make use of the full power of Coq. The programs that are verified are small but sophisticated. The annotation burden is relatively light compared to other Hoare-style approaches (less than 100% in all cases). The main drawback is the difficulty of using Coq.

2.8 Conclusion

In this section we have presented four improvements to TVLA.

Finite differencing. In §2.3 we introduced the definite operator, $\mathbf{1}[\cdot]$, which improves the precision of finite differencing. This operator allows us to characterize the completeness of finite differencing in a way that was not previously possible. It also means that we no longer need to run sharpening after every assignment, as TVLA does. Given the expense of sharpening, this is a significant benefit.

We also described a new formula for computing finite differences of transitive closure formula in the presence of arbitrary changes to core predicates. TVLA uses a more restricted formula with many special cases and correctness checks. Our formula allows us to avoid this complexity.

Formula evaluation. In §2.4 we described an efficient algorithm for evaluating formulas in first-order logic with transitive closure. Our algorithm borrows optimization techniques from databases to make queries run three orders of magnitude faster than the naive evaluation algorithm.

Sharpening. §2.5 presented a new algorithm for improving the precision of a heap structure. We proved the soundness of the algorithm and characterized its completeness. The completeness of our algorithm means that programmers typically do not have to supply customized integrity constraints to our system, as they do in TVLA. Our approach also allows us to use our fast formula evaluation algorithms from §2.4 for sharpening.

Abstraction. Finally, §2.6 describes several techniques to make our heap abstractions smaller and to make our analysis more scalable. We introduced $\rho = 1/2$ nodes and we presented a new join algorithm for heap domain elements. Taken together, these improvements

increase absolute performance by almost an order of magnitude. For larger programs with more complex invariants, we expect the improvements would be even greater.

2.9 Proofs

2.9.1 Soundness of Sharpening

The following theorems prove the soundness and completeness of the revised sharpening algorithm. The algorithm is proved complete only in the sense that it returns answers that are at least as precise as the sharpening algorithms in the previous section based on case analysis.

We begin by presenting an augmented version of the sharpening algorithm, which we will shortly prove is equivalent to the original algorithm. The augmented algorithm returns tuples of the form $p^v(u : \{x, y\})$ instead of just $p^v(u)$; x and y are $\forall$-quantified variables in A. This change simplifies the soundness proof.

First we define some auxiliary functions.

function $\mathrm{Var}(v, A)$
    $\langle u, e \rangle := A(v)$
    if $e = \exists$ **then** $u : \emptyset$ **else** $u : \{v\}$

function $\mathrm{Filter}'(p^v(v_1, \ldots, v_k), A)$
    if for all i, $A(v_i) = \langle u_i, \exists \rangle \Rightarrow u_i$ a non-summary node
    then $\{p^v(\mathrm{Var}(v_1, A), \ldots, \mathrm{Var}(v_k, A))\}$ **else** $\emptyset$

function $\mathrm{Elim}(v, T)$
    foreach $p^v(u_1 : V_1, \ldots, u_k : V_k) \in T$, remove v from each V_i

function $\mathrm{Repl}(x, y, T)$
    foreach $p^v(u_1 : V_1, \ldots, u_k : V_k) \in T$, replace x with y in each V_i

function $\cap (T, T')$
    $T'' := \{p^v(u_1 : (V_1 \cup V_1'), \ldots, u_k : (V_k \cup V_k')) :$
        $p^v(u_1 : V_1, \ldots, u_k : V_k) \in T \wedge p^v(u_1 : V_1', \ldots, u_k : V_k') \in T'\}$
    return T''

Next we make a few assumptions about formulas. We assume throughout that all quantifiers use variables with distinct names to avoid scoping problems. We also assume all formulas are in negated normal form. Rather than writing $p(v_1, \ldots, v_n)$ or $\neg p(v_1, \ldots, v_n)$ for an atomic literal, we write $p^v(v_1, \ldots, v_n)$ where $v = 1$ means a positive literal and $v = 0$ means a negative literal. The modified algorithm is as follows.

φ	$M'(\varphi, S, A)$
$p^v(v_1, \ldots, v_k)$	$\text{Filter}'(p^v(v_1, \ldots, v_k), A)$

$$\varphi_1 \wedge \varphi_2 \quad \begin{cases} M'(\varphi_1, S, A) & \text{if } [\![\varphi_2]\!]_{S,A} \neq 1/2 \\ M'(\varphi_2, S, A) & \text{if } [\![\varphi_1]\!]_{S,A} \neq 1/2 \\ M'(\varphi_1, S, A) \cup M'(\varphi_2, S, A) & \text{otherwise} \end{cases}$$

$$\varphi_1 \vee \varphi_2 \quad \begin{cases} M'(\varphi_1, S, A) & \text{if } [\![\varphi_2]\!]_{S,A} \neq 1/2 \\ M'(\varphi_2, S, A) & \text{if } [\![\varphi_1]\!]_{S,A} \neq 1/2 \\ M'(\varphi_1, S, A) \cap M'(\varphi_2, S, A) & \text{otherwise} \end{cases}$$

$$\forall v. \, \varphi' \quad \bigcup_{u \in U}\{\text{Elim}(v, M'(\varphi', S, A[v \rightarrow \langle u, \forall \rangle])) : [\![\varphi']\!]_{S,A[v \rightarrow u]} = 1/2\}$$

$$\exists v. \, \varphi' \quad \bigcap_{u \in U}\{M'(\varphi', S, A[v \rightarrow \langle u, \exists \rangle]) : [\![\varphi']\!]_{S,A[v \rightarrow u]} = 1/2\}$$

$$\text{TC}(s, t; x, y). \, \varphi' \quad \bigcap_{P \in \text{Paths}(S, A(s), A(t))} \bigcup_{(a_1, a_2) \in P}$$
$$\{\text{Repl}(x, s, \text{Repl}(y, t, M'(\varphi', S, A[x \rightarrow a_1, y \rightarrow a_2]))) :$$
$$[\![\varphi']\!]_{S,A[x \rightarrow a_1, y \rightarrow a_2]} = 1/2\}$$

First we prove that this algorithm returns the same results as the original algorithm, but augmented with more information. Specifically, since we want the soundness of M' to imply soundness of M, we need to ensure that if M returns a sharpening, then M' will return some augmentation of it.

Theorem 2 *If $p^v(u_1, \ldots, u_k) \in M(\varphi, S, A)$, then $p^v(u_1 : V_1, \ldots, u_k : V_k) \in M'(\varphi, S, A)$, for some V_is.* □

PROOF The proof is by induction on the structure of φ. Rather than write out each case, we simply note that Filter' behaves the same as Filter aside from the addition of the V_i components. Similarly, Elim and Repl only affect the V_i components of their argument. And $\cap$ is just standard intersection aside from the changes to the V_is. All the induction cases simply apply the induction hypothesis and then take advantage of these observations.∎

Before giving the formal soundness proof, we describe it at a high level. We are given a sharpening formula φ, which may have free variables. Our goal is to connect the abstract and the concrete world. The abstract world is composed of an abstract state S, an augmented assignment A from the free variables of φ to nodes of S, and a sharpening result $p^v(u_1 : V_1, \ldots, u_k : V_k)$ returned by M'.

The concrete world is composed of a concrete state $S^\sharp$ that embeds into S, an assignment $A^\sharp$ to nodes of $S^\sharp$, and the guarantee that φ holds in $S^\sharp$ under $A^\sharp$. We wish to prove that the actual value of predicate p in $S^\sharp$ at the given location is v, thus ensuring that M' returned a sound result.

The crux of the proof is that while the results of M' are phrased in terms of abstract nodes, we need to prove facts about p in the concrete state $S^\sharp$. To solve this problem, we use the embedding function f that maps nodes of $S^\sharp$ to nodes of S. Specifically, if M' returns $p^v(u_1 : V_1, \ldots, u_k : V_k)$, then p should have the value v in $S^\sharp$ at any tuple of nodes $\langle u'_1, \ldots, u'_k \rangle$ where $f(u'_i) = u_i$ for all i.

There is one exception. Namely, if $\varphi = R(x)$, then x is a free variable that is mapped to a concrete node by $A^\sharp$. When M' returns $R^1(u : V)$ for φ, we are only interested in proving that $R = 1$ at the node $A^\sharp(x)$—*not* at every node in $S^\sharp$ that f maps to u. Only higher up in the induction, when x is finally quantified universally, are we concerned with every node mapping to u. To summarize, we want to prove that $p = v$ in $S^\sharp$ at all nodes u'_i that map to u_i, *except* when $A^\sharp$ picks out a specific concrete node, in which case we only care about that one.

To formalize the ideas in the previous paragraph, we need to define an auxiliary function F, which returns the set of places where we have proved that $p = v$ in $S^\sharp$. As described above, it picks out the concrete nodes u'_i that map to abstract nodes u_i and that match the mapping in $A^\sharp$ when one is present.

$$F(\langle u_1 : V_1, \ldots, u_k : V_k \rangle, A^\sharp) =$$
$$\{\langle u'_1, \ldots, u'_k \rangle : f(u'_i) = u_i \wedge \forall 1 \le i \le k. \; \forall v \in V_i. \; A^\sharp(v) = u'_i\}$$

Notice that tracking the V_i variables is necessary in order to know which variable names were responsible for an M' result, which allows us to check $A^\sharp$ for a mapping for that variable.

We also define the following notation. Assume A is an augmented assignment of variables to nodes in some structure S and $A^\sharp$ is an assignment from the same variables to nodes in a concrete structure $S^\sharp$ (which embeds into S via f). We say that $A^\sharp$ embeds into A if for every variable v, $A(v) = \langle f(A^\sharp(v)), e \rangle$ where e can be either $\forall$ or $\exists$.

Now we can state the main soundness theorem. Aside from the issues mentioned above, it is a straightforward induction.

Theorem 3 *Let φ be a formula. Let S be a structure and $S^\sharp$ a concrete structure that embeds into S via f. Let A be an augmented assignment from the free variables of φ to S nodes. Let $A^\sharp$ be an assignment, from the free variables of φ to $S^\sharp$ nodes, which embeds into A. Assume $[\![\varphi]\!]_{S,A} = 1/2$ and $[\![\varphi]\!]_{S^\sharp,A^\sharp} = 1$. Assume that $p^v(u_1 : V_1, \ldots, u_k : V_k) \in M(\varphi, S, A)$. Then $\iota^\sharp(p)(u'_1, \ldots, u'_k) = v$ for all $\langle u'_1, \ldots, u'_k \rangle \in F(\langle u_1 : V_1, \ldots, u_k : V_k \rangle, A^\sharp)$.* □

PROOF The proof is by induction on the structure of φ.

• *Case $\varphi = p^v(v_1, \ldots, v_k)$.* Let $\langle u'_1, \ldots, u'_k \rangle \in F(\langle u_1 : V_1, \ldots, u_k : V_k \rangle, A^\sharp)$. Since we know that $p^v(u_1 : V_1, \ldots, u_k : V_k) \in M(\varphi, S, A)$, it must be that each u_i is either a non-summary node or else is universally quantified in S. If u_i is a non-summary node, then it must be that $A^\sharp(v_i) = u'_i$, since there is only one possible node that can embed into u_i. If u_i is a summary node, then v_i must be universally quantified in A, so $V_i = \{v_i\}$. Therefore, $A^\sharp(v_i) = u'_i$. Now we know that $A^\sharp(v_i) = u'_i$ for all i. From $[\![\varphi]\!]_{S^\sharp,A^\sharp} = 1$, we get that $\iota^\sharp(p)(u'_1, \ldots, u'_k) = v$.

• *Case $\varphi = \varphi_1 \wedge \varphi_2$.* The M' algorithm has three cases. In the first case, $p^v(u_1 : V_1, \ldots, u_k : V_k)$ comes from φ_1 alone. Also, $[\![\varphi_1]\!]_{S,A} = 1/2$. And since $[\![\varphi]\!]_{S^\sharp,A^\sharp} = 1$, it must be that $[\![\varphi_1]\!]_{S^\sharp,A^\sharp} = 1$ as well. With these facts, we can apply the induction hypothesis to φ_1 using the same A and $A^\sharp$. This gives us precisely what we need.

The second case in M is symmetric to the first, so we consider the third case. We know that $p^v(u_1 : V_1, \ldots, u_k : V_k)$ comes from either $M'(\varphi_1, S, A)$ or $M'(\varphi_2, S, A)$. Assume

the first, without loss of generality. $[\![\varphi_1]\!]_{S,A} = 1/2$, and the other facts from the previous paragraph still hold, so we can induct again over φ_1, obtaining the desired result.

• *Case* $\varphi = \varphi_1 \vee \varphi_2$. Again, M' has three cases. We consider the first. In this case, $p^v(u_1 : V_1, \ldots, u_k : V_k) \in M'(\varphi_1, S, A)$ and $[\![\varphi_1]\!]_{S,A} = 1/2$. We also know that $[\![\varphi_2]\!]_{S,A} = 0$ (it cannot be 1/2, and if it were 1, then φ would be 1). By the embedding theorem [46], any formula with a definite value in S has that same value in $S^\sharp$. Thus, $[\![\varphi_2]\!]_{S^\sharp, A^\sharp} = 0$. And since $[\![\varphi]\!]_{S^\sharp, A^\sharp} = 1$, it must be that $[\![\varphi_1]\!]_{S^\sharp, A^\sharp} = 1$. Now we have everything we need to apply the induction hypothesis to φ_1 and obtain the desired result.

The second M' case is symmetric to the first, so we jump to the third case. It must be that $p^v(u_1 : V_1', \ldots, u_k : V_k') \in M'(\varphi_1, S, A)$ and $p^v(u_1 : V_1'', \ldots, u_k : V_k'') \in M'(\varphi_2, S, A)$, where $V_i = V_i' \cup V_i''$ for each i. We know that $[\![\varphi_i]\!]_{S,A} = 1/2$ for $i \in \{1, 2\}$. From the assumption that $[\![\varphi]\!]_{S^\sharp, A^\sharp} = 1$ we know that either $[\![\varphi_1]\!]_{S^\sharp, A^\sharp} = 1$ or $[\![\varphi_2]\!]_{S^\sharp, A^\sharp} = 1$. Without loss of generality, assume the first case. We apply the induction hypothesis on φ_1 using variables $V_1', \ldots, V_k'$. Now our goal is to prove that

$$F(\langle u_1 : (V_1' \cup V_1''), \ldots, u_k : (V_k' \cup V_k'')\rangle, A^\sharp) \subseteq F(\langle u_1 : V_1', \ldots, u_k : V_k'\rangle, A^\sharp).$$

Let $\langle u_1', \ldots, u_k'\rangle$ be some tuple in the left-hand set. We already know that $f(u_i') = u_i$ for all i. And since we know for each i that $(\forall v \in (V_i' \cup V_i''). A^\sharp(v) = u_i')$, it must be that $(\forall v \in V_i'. A^\sharp(v) = u_i')$. Thus, the subset relationship holds and we obtain the desired result.

• *Case* $\varphi = \forall v. \varphi'$. Since $p^v(u_1 : V_1, \ldots, u_k : V_k) \in M'(\varphi, S, A)$, there must be some u such that $p^v(u_1 : V_1, \ldots, u_k : V_k) \in \text{Elim}(v, M'(\varphi', S, A[v \to \langle u, \forall\rangle]))$. Therefore, $p^v(u_1 : V_1', \ldots, u_k : V_k') \in M'(\varphi', S, A[v \to \langle u, \forall\rangle])$, where $V_i = V_i' - \{v\}$ for each i. Additionally, we know that $[\![\varphi']\!]_{S,A[v \to u]} = 1/2$. For each u' such that $f(u') = u$, we can define a new $A_{u'}^\sharp = A^\sharp[v \to u']$. We can extend $A' = A[v \to \langle u, \forall\rangle]$. And since $[\![\varphi]\!]_{S^\sharp, A^\sharp} = 1$, it must be that $[\![\varphi']\!]_{S^\sharp, A_{u'}^\sharp} = 1$. Then we can apply the induction hypothesis, for each u' that maps to u, using φ', with $A_{u'}^\sharp$, A', and the V_i's. To complete this case, we need to show that

$$F(\langle u_1 : V_1, \ldots, u_k : V_k\rangle, A^\sharp) \subseteq \bigcup_{u':f(u')=u} F(\langle u_1 : V_1', \ldots, u_k : V_k'\rangle, A_{u'}^\sharp).$$

We have some additional information based on the syntactic restriction that applies to sharpening formulas. The restriction says that a universally quantified variable can appear in only one position for a given p^v. This implies that there is at most one i such that $V_i \neq V_i'$, and at this i, $V_i \cup \{v\} = V_i'$. Now let's say that $\langle u_1', \ldots, u_k'\rangle$ is in the left-hand F. If $V_i = V_i'$ for all i, then we can extend $A^\sharp$ with $[v \to u']$ for any u' where $f(u') = u$, and we have shown that $\langle u_1', \ldots, u_k'\rangle$ is in some right-hand F. So then let i be the place where $V_i \cup \{v\} = V_i'$. In that case we can extend $A^\sharp$ with $[v \to u_i']$ and again we are done. Note that without the syntactic restriction, we potentially would have had to extend $A^\sharp(v)$ to be both u_i' and u_j', for some i and j, and these two values might not be equal.

• *Case* $\varphi = \exists v. \varphi'$. We know that $[\![\varphi]\!]_{S^\sharp, A^\sharp} = 1$. Thus, there is some node u' such that $[\![\varphi']\!]_{S^\sharp, A^\sharp[v \to u']} = 1$. Let $A_{u'}^\sharp = A^\sharp[v \to u']$ and let $A' = A[v \to \langle f(u'), \exists\rangle]$. Our intention is

to use the induction hypothesis over φ' with $A^\sharp_{u'}$ and A'. Since $[\![\varphi]\!]_{S,A} = 1/2$, it must be that $[\![\varphi']\!]_{S,A'} = 1/2$—it cannot be 1, since that would make φ be 1, and it cannot be zero, because then φ' would be zero in $S^\sharp$ by the embedding theorem. Finally, since we know that $p^v(u_1 : V_1, \ldots, u_k : V_k) \in M'(\varphi, S, A)$, it must be that $p^v(u_1 : V'_1, \ldots, u_k : V'_k) \in M'(\varphi', S, A')$, where $V'_i \subseteq V_i$ for all i. Now we can induct on φ' with A', $A^\sharp_{u'}$, and the V'_is. To complete the case, we need to show that

$$F(\langle u_1 : V_1, \ldots, u_k : V_k \rangle, A^\sharp) \subseteq F(\langle u_1 : V'_1, \ldots, u_k : V'_k \rangle, A^\sharp_{u'}).$$

Choose some $\langle u'_1, \ldots, u'_k \rangle$ in the left-hand F. We will show that $\langle u'_1, \ldots, u'_k \rangle$ is in $F(\langle u_1 : V'_1, \ldots, u_k : V'_k \rangle, A^\sharp_{u'})$. We know that $V'_i \subseteq V_i$ for all i. We also know that v is not in any V_i, since existential variables never appear in these places. By examining the definition of F, we obtain the desired result.

- *Case $\varphi = \mathrm{TC}(s, t; x, y)$. φ'.* Since $[\![\varphi]\!]_{S^\sharp, A^\sharp} = 1$, there must be a path from $A^\sharp(s)$ to $A^\sharp(t)$ in $S^\sharp$. This path induces a corresponding path in S from $A(s)$ to $A(t)$ (though it may not be a simple path). Let these paths be $P^\sharp$ and P. P is one of the paths in $\mathrm{Paths}(S, A(s), A(t))$. We know that $p^v(u_1 : V_1, \ldots, u_k : V_k) \in M'(\varphi, S, A)$, so therefore there is some edge $(a_1, a_2) \in P$ such that $p^v(u_1 : V'_1, \ldots, u_k : V'_k) \in M'(\varphi', S, A[x \to a_1, y \to a_2])$. The V'_i are such that $V'_i[x \mapsto s, y \mapsto t] \subseteq V_i$ (due to Repl).

Let (w_1, w_2) be the edge that corresponds to (a_1, a_2) in $S^\sharp$. Let $A' = A[x \to a_1, y \to a_2]$ and let $(A^\sharp)' = A^\sharp[x \to w_1, y \to w_2]$. Then we know that $[\![\varphi']\!]_{S,A'} = 1/2$ and $[\![\varphi']\!]S^\sharp, (A^\sharp)' = 1$. Now we can apply the induction hypothesis with φ', A' and $(A^\sharp)'$. The final step is to prove that

$$F(\langle u_1 : V_1, \ldots, u_k : V_k \rangle, A^\sharp) \subseteq F(\langle u_1 : V'_1, \ldots, u_k : V'_k \rangle, (A^\sharp)').$$

There are now several cases. The first case is that $a_1 = \langle f(w_1), \exists \rangle$ and $a_2 = \langle f(w_2), \exists \rangle$. In this case, neither x nor y occur in the V'_i, meaning that $V'_i \subseteq V_i$ for each i. Thus, the subset relationship is satisfied.

Now let a_1 and a_2 be universally quantified. For this to happen, (a_1, a_2) must be both the first and the last edge on the path. Therefore $A^\sharp(s) = w_1$ and $A^\sharp(t) = w_2$. x and y may occur in the V'_i sets but that they are replaced with s and t in V_i. If we consider the restrictions on $A^\sharp$ imposed by F, the differences are that on the right side we have $(A^\sharp)'(x) = u'_i$ and on the left side we have the corresponding $A^\sharp(s) = u'_i$ (as well as similar constraints on y and t). However, $(A^\sharp)'(x) = w_1$ by the definition of $(A^\sharp)'$. We already know that $A^\sharp(s) = w_1$, so the corresponding constraints on either side are equivalent and we are done.

The cases where only one of a_1 or a_2 is universally quantified are similar to the two above. ∎

A corollary of this theorem is that M' is sound. In the text below, we write the empty assignment as ϵ.

Corollary 1 *Let φ be a sharpening rule. Let S be a structure, and $S^\sharp$ a concrete structure that embeds into S via f. Assume $[\![\varphi]\!]_{S,\epsilon} = 1/2$. Assume that $[\![\varphi]\!]_{S^\sharp,\epsilon} = 1$. Assume that*

$p^v(u_1 : V_1, \ldots, u_k : V_k) \in M(\varphi, S, \epsilon)$. *Then* $\iota^\sharp(p)(u'_1, \ldots, u'_k) = v$ *for all* $\langle u'_1, \ldots, u'_k \rangle$ *such that* $f(u'_i) = u_i$ *for each* i. □

PROOF We simply apply the soundness theorem with $A = \epsilon, A^\sharp = \epsilon$. ∎

2.9.2   Completeness of Sharpening

In this section we prove a limited completeness result. One limitation of the result is that it does not hold when the equality predicate occurs in φ. Equality makes sharpening much more difficult because it allows the sharpening formula to constrain the size of the model. In general, dealing with cardinality is a very difficult problem, one which is not solved by our sharpening algorithm.

We can see the difficulty caused by equalities in φ in the following example. Let $\varphi = (\exists n.\ \neg A(n)) \wedge (\forall n, n'.\ A(n') \Rightarrow n = n')$. Then consider the structure It is safe to sharpen A to zero here, but algorithm M will not discover this fact.

Now for the proof. The overall goal is to prove that if $p^v(u_1, \ldots, u_k) \notin M(\varphi, S, A)$, then there is some structure S' that embeds into S where p does not equal v and where φ holds. More precisely, we show that there is an assignment A' that embeds into A and a set of concrete nodes $u'_1, \ldots, u'_k$ that embed into $u_1, \ldots, u_k$ where $\iota'(p)(u'_1, \ldots, u'_k) = \neg v$ and $[\![\varphi]\!]_{S',A'} \neq 0$.

Ideally, we would like S' to be a concrete structure (meaning no 1/2 predicate values) and we would like to show that $[\![\varphi]\!]_{S',A'} = 1$. However, this is a much more difficult problem, since it requires us to determine if the model S is satisfiable under the constraint φ. Since φ is an arbitrary formula in first order logic with transitive closure, this problem is undecidable. Instead, we allow S' to be an abstract structure—one that itself may be unsatisfiable as well.

A simple way to explain our completeness result is as follows: "If some sharpening result p^v is not returned by M, then by materializing some nodes in S and changing some predicate values from 1/2 to 1 we can obtain a structure S' where p^v is violated and where φ is not falsified." This is a much weaker result, but it is still quite useful for understanding whether M will generate a given sharpening.

To make the proof work, we need to strengthen the induction hypothesis. Instead of proving that there exist *some* S' and A' for a given S and A, we prove that *all* S' and A' of a particular form violate p^v while still satisfying φ. After the theorem we prove a corollary that some such S' is guaranteed to exist.

Theorem 4 *Let S be a structure and let φ be a formula whose free variables are mapped to nodes of S by A. Assume there are no occurrences of equality in φ. Let $[\![\varphi]\!]_{S,A} = 1/2$. Let $p^v(u_1, \ldots, u_k) \notin M(\varphi, S, A)$, where p is not the equality predicate. Then let S' be any structure and A' be any assignment that satisfy the following.*

  1. S' embeds into S via some embedding function f.

  2. A' embeds into A via f.

3. *There are $u'_1, \ldots, u'_k$ such that each u'_i satisfies $f(u'_i) = u_i$.*

4. *$\iota'(p)(u'_1, \ldots, u'_k) = \neg v$ (where ι' defines the predicate values for S').*

5. *For any other $\langle u''_1, \ldots, u''_k \rangle \neq \langle u'_1, \ldots, u'_k \rangle$, $\iota'(p)(u''_1, \ldots, u''_k) = \iota(p)(f(u''_1), \ldots, f(u''_k))$.*

6. *For any predicate $q \neq p$, $\iota'(q)(u''_1, \ldots, u''_k) = \iota(q)(f(u''_1), \ldots, f(u''_k))$ (if q is not the equality predicate).*

7. *For every u_i that is a summary node, there is some $u^s_i \neq u'_i$ where $f(u^s_i) = u_i$.*

8. *There is no v, i so that $A(v) = \langle u_i, \exists \rangle$, where u_i is a summary node and $A'(v) = u'_i$.*

Then $[\![\varphi]\!]_{S',A'} \neq 0$. □

PROOF The proof is by induction over the structure of φ.

• *Case $\varphi = q^{v'}(v_1, \ldots, v_k)$.* If $q \neq p$, then $[\![\varphi]\!]_{S',A'} = [\![\varphi]\!]_{S,A} = 1/2 \neq 0$. So assume $q = p$. In the case where there is some i so that $A'(v_i) \neq u'_i$, then $[\![\varphi]\!]_{S',A'} = [\![\varphi]\!]_{S,A} = 1/2 \neq 0$. So consider now the case where, for all i, $A'(v_i) = u'_i$. If $v' \neq v$, then $[\![\varphi]\!]_{S',A'} = 1$ because (by the assumptions above) $\iota'(p)(u'_1, \ldots, u'_k) = \neg v = v'$. If $v' = v$, then consider the following. Since $p^v(u_1, \ldots, u_k) \notin M(\varphi, S, A)$, it must be that some u_i is a summary node and $A(v_i) = \langle u_i, \exists \rangle$. However, this possibility is contradicted by the assumption on A and A'.

• *Case $\varphi = \varphi_1 \wedge \varphi_2$.* We consider the first case in M, where $[\![\varphi_2]\!]_{S,A} = 1$. By the embedding theorem, $[\![\varphi_2]\!]_{S',A'} = 1$. We know that $p^v(u_1, \ldots, u_k) \notin M(\varphi_1, S, A)$, so we can induct on φ_1 to get $[\![\varphi_1]\!]_{S',A'} \neq 0$. Thus, $[\![\varphi_1 \wedge \varphi_2]\!]_{S',A'} \neq 0$.

The second M case is symmetric to the first, so consider the third case. $[\![\varphi_i]\!]_{S,A} = 1/2$ for $i \in \{1, 2\}$. Also, $p^v(u_1, \ldots, u_k) \notin M(\varphi_i, S, A)$ for $i \in \{1, 2\}$. Thus, we can apply the induction hypothesis to both φ_1 and φ_2 to obtain $[\![\varphi_i]\!]_{S',A'} \neq 0$ for $i \in \{1, 2\}$. Thus, $[\![\varphi]\!]_{S',A'} \neq 0$.

• *Case $\varphi = \varphi_1 \vee \varphi_2$.* We again have three cases. In the first, $[\![\varphi_2]\!]_{S,A} = 0$ and $[\![\varphi_1]\!]_{S,A} = 1/2$. We also know that $p^v(u_1, \ldots, u_k) \notin M(\varphi_1, S, A)$, so we can use the induction hypothesis on φ_1. This tells us that $[\![\varphi_1]\!]_{S',A'} \neq 0$, which means that $[\![\varphi]\!]_{S',A'} \neq 0$.

The second case is symmetric to the first. In the third case, we know that either $p^v(u_1, \ldots, u_k) \notin M(\varphi_1, S, A)$ or $p^v(u_1, \ldots, u_k) \notin M(\varphi_2, S, A)$ (or possibly both), so assume $p^v(u_1, \ldots, u_k) \notin M(\varphi_i, S, A)$. We use the induction hypothesis on φ_i to obtain $[\![\varphi_i]\!]_{S',A'} \neq 0$, which tells us that $[\![\varphi]\!]_{S',A'} \neq 0$.

• *Case $\varphi = \forall v. \varphi'$.* The goal is to prove that $[\![\varphi']\!]_{S',A'[v \rightarrow u']} \neq 0$ for all u' in S'. Since $[\![\varphi]\!]_{S,A} = 1/2$, we know that φ' is either $1/2$ or 1 everywhere in S. So if u' is a place where φ' is 1 at $f(u')$, then the embedding theorem does the work for us. On the other hand, if u' is such that φ' is $1/2$ at $f(u')$, then we know that $p^v(u_1, \ldots, u_k) \notin M(\varphi', S, A[v \rightarrow \langle f(u'), \forall \rangle])$. Thus, we can apply the induction hypothesis there with $A[v \rightarrow \langle f(u'), \forall \rangle]$ and with $A'[v \rightarrow u']$, which gives $[\![\varphi']\!]_{S',A'[v \rightarrow u']} \neq 0$. Combining these facts together, we get $[\![\varphi]\!]_{S',A'} \neq 0$.

- *Case* $\varphi = \exists v.\ \varphi'$. There is some u where $[\![\varphi']\!]_{S,A[v\to u]} = 1/2$ and where $p^v(u_1,\ldots,u_k) \notin M(\varphi', S, A[v \to \langle f(u'), \exists \rangle])$. Let u' be some node such that $f(u') = u$. If $u' = u'_i$ for some i, and u is a summary node, then pick $u' = u_i^s$ instead. Apply the induction hypothesis on φ' with $A[v \to \langle u, \exists \rangle]$ and $A'[v \to u']$. It is valid to do this, since we know that if u is a summary node then $u' \neq u'_i$ for any i, as required by the induction hypothesis. This gives $[\![\varphi']\!]_{S',A'[v\to u']} \neq 0$, which implies the result we desire.

- *Case* $\varphi = \mathrm{TC}(s,t;x,y).\ \varphi'$. We know that there is some path P in S from $A(s)$ to $A(t)$. Our goal is to find a path P' in S' from $A'(s)$ to $A'(t)$ such that φ' is non-zero along all the edges of P'. We select P' as follows. Each edge corresponds to an edge in P. If there is a P edge from u to w, then we select a u' and w' in S' as follows and P' has an edge from u' to w'. For the first edge, $u = A(s)$ and we select $u' = A'(s)$. Similarly, for the last edge, $w = A(t)$ and we choose $w' = A'(t)$. For all the intermediate nodes, we select any u' such that $f(u') = u$, except that if $u = u_i$ for some i and u is a summary node then we choose $u = u_i^s$. We use the same method for selecting w.

    Now consider each edge in P. Let it be from (a_1, a_2), where $a_1 = \langle u, Q_1 \rangle$ and $a_2 = \langle w, Q_2 \rangle$. If $[\![\varphi']\!]_{S,A[x\to u, y\to w]} = 1$, then we know that $[\![\varphi']\!]_{S',A'[x\to u', y\to w']} = 1$ by the embedding theorem. If $[\![\varphi']\!]_{S,A[x\to u, y\to w]} = 1/2$, then we use induction on φ', with the assignments $A[x \to a_1, y \to a_2]$ and $A'[x \to u', y \to w']$. The induction shows that $[\![\varphi']\!]_{S',A'[x\to u', y\to w']} \neq 0$, as desired.

    The induction is safe according to assumption 8 above because we have chosen the intermediate nodes in the same way as in the previous case of existential quantification. At the endpoints, we are also guaranteed to be safe because the given condition is assumed on s and t, so in the induction it also holds for x and y. $\blacksquare$

Corollary 2 *Let S be a structure and let φ be a sharpening rule (a sentence). Assume there are no occurrences of equality in φ. Let $[\![\varphi]\!]_{S,\epsilon} = 1/2$. Let $p^v(u_1,\ldots,u_k) \notin M(\varphi, S, \epsilon)$, where p is not the equality predicate. Assume that $\iota(p)(u_1,\ldots,u_k) = 1/2$. Then there exists a structure S' that embeds into S via f such that $[\![\varphi]\!]_{S',\epsilon} \neq 0$.* $\square$

PROOF We apply the completeness theorem above with $A = A' = \epsilon$. We need to find an S' that satisfies the conditions of the proof. First we consider the nodes $u_1,\ldots,u_k$. For each one that is a summary node, we materialize it into a summary node u_i^s and a concrete node u'_i. The predicate values for these nodes are copied from those over u_i. If u_i is not a summary node, then we let $u'_i = u_i$. The predicate values are unchanged except that we set $\iota'(p)(u'_1,\ldots,u'_k)$ to be $\neg v$. (Note that the equality predicate will also differ because of materialization.) It should now be easy to check that this S' satisfies the conditions of the completeness theorem. $\blacksquare$

Note that we can prove a slightly more restricted theorem in the case where the p in $p^1(u,u)$ is the equality predicate. We require that the only changes between S and S' be a single materialization of the node u. Proving completeness when φ is allowed to contain equality is more difficult. In the case where φ contains only negative occurrences of equality, the proof goes through essentially unchanged. When φ contains positive equality but no

existential quantification, we can use an S' where no materialization has taken place to show $[\![\varphi]\!]_{S',A'} \neq 0$. Finding a more general treatment of equality is a subject for future work.

Chapter 3

Combination Domain

The previous chapter described our heap domain based on TVLA. Unfortunately, this domain is not capable of reasoning about integers at all. We could augment the heap domain to support integer reasoning, but in this chapter we choose a more general approach. We describe a framework where two arbitrary domains can be combined. This way, our heap domain can be combined with any one of the many existing integer domains (intervals, difference constraints, octagon, polyhedra).

Techniques for combining domains have appeared in the past. The novel aspect of our technique is that it deals with quantification. Quantification is particularly important when reasoning about the heap, since most invariants of interest are quantified over all elements of a data structure. Another important property of our technique is that the domains are treated symmetrically. Most techniques for combining heap and integer reasoning treat the integer reasoning as subservient. We treat both domains equally, which makes our framework more powerful.

The introduction of this chapter presents some examples that show the power of our combination framework:

- We show how to maintain a correlation between the size of a data structure (the number of elements it contains) and an integer variable.

- We show how the integer domain exposes information about array index ranges so that the heap domain can reason about arrays and quantify over array elements.

- We show how the heap domain exposes information about its nodes so that the integer domain can reason about properties of heap objects and the values of their fields.

- We show how the heap and integer domains can expose instrumentation predicate information to each other so that complex invariants involving a mixture of heap and integer properties are provable.

- We show how the integer domain can define "instrumentation functions" that are defined as the cardinality of some set. This allows the combined domain to reason

about complex invariants of manually reference-counted data structures.

In later sections, we explain the transfer functions for the domain and prove that it is sound. We also describe some requirements that we demand of the domains being combined beyond what is needed for abstract interpretation. Without these requirements, we would be unable to handle quantification as precisely.

3.1   Introduction

The combined domain is built on top of two other domains, called the *base domains*. An abstract element from the combined domain is an ordered pair of abstract elements, $\langle e_1, e_2 \rangle$, from the base domains. In our examples, e_1 is a heap domain element and e_2 is an integer domain element. However, we make some modifications to the way disjunctions are handled. Previously, we allowed disjunctions to appear in heap domain elements. A heap domain element was defined as a disjunction of three-valued structures. If we were to continue to use this scheme, then a combined domain element would look as follows.

$$E = \langle S_1 \vee S_2 \vee \cdots \vee S_n, Z \rangle$$

Each S_i is a three-valued structure and Z is an integer domain element.

The problem with this structure is that all the S_is must share a single integer domain element. We would prefer to move disjunctions higher so that a domain element has the following form.

$$E = \langle S_1, Z_1 \rangle \vee \cdots \vee \langle S_n, Z_n \rangle$$

Now each S_i has its own integer element Z_i. This is potentially more precise.

To make this work, we remove disjunctions from the heap domain so that a heap domain element is a single three-valued structure. Then a combined domain element has the form $\langle S, Z \rangle$. Disjunctions are still necessary, of course. We wrap elements of the combined domain in a special *disjunctive domain*, as shown in this diagram.
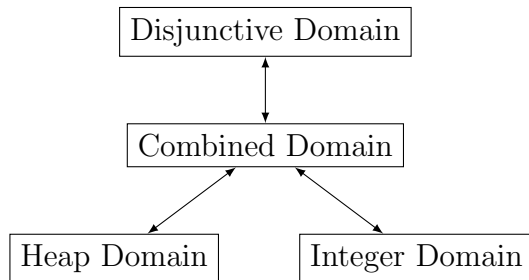
The disjunctive domain uses the same heuristics to merge disjuncts as we did in the heap domain (based on canonical names of nodes in the heap domain element).

Having presented the structure of a combined domain element, we now show some examples of how it works. Later sections will flesh out the details. All of these examples are analyzed using the disjunctive domain applied to the combined heap/integer domain.

3.1.1   Cardinality Invariants

In this example we would like to infer that an integer variable `n` holds the size of a linked list. Neither a heap analysis nor an integer analysis would be able to infer this invariant by itself.

```
1  type ListNode;
2  global next[ListNode]:ListNode;
3
4  procedure test()
5    n:int;
6    list, node:ListNode;
7  {
8    n := 0;
9    list := null;
10   while (*) {
11     node := new ListNode;
12     next[node] := list;
13     list := node;
14     label loop_mid;
15     n := n+1;
16   }
17 }
```

When we analyze the program above, we split up the variables between the two domains. The integer domain is responsible for a variable x if its signature is "$x[t_1, \ldots, t_n] : \text{int}$". Otherwise the heap domain is responsible. In the example, the integer domain manages the n variable and the heap domain manages the *next*, *list*, and *node* maps. Assignments to a given variable are handled by the domain responsible for that variable. So the assignment $n := 0$ is handled by the integer domain. That means that, given a combined domain element $\langle e_1, e_2 \rangle$, the integer domain updates the e_2 component of the domain element using its transfer function for assignments.

Assume for now that no communication takes place between domains. Consider the domain elements that are inferred at the program point labeled `loop_mid` (line 14). In the first loop iteration, $n = 0$ and there is only one node, pointed to by *node*. The second iteration yields the same state as the previous one as well as a state where $n = 1$ and *node* points to a two-element list. We depict this as follows.

E_1	E_2
Node,List $\overset{n_1}{\bigcirc}\!\!\longrightarrow\!\!\overset{z}{\bigcirc}$ List,Null	Node,List $\overset{n_1}{\bigcirc}\!\!\longrightarrow\!\!\overset{n_2}{\bigcirc}\!\!\longrightarrow\!\!\overset{z}{\bigcirc}$ Null
$n = 0$	$n = 1$

Up until now we have managed to preserve, via the disjunctive abstraction, the correlation between n and the size of the list. Unfortunately, we are now forced to join E_1 and E_2
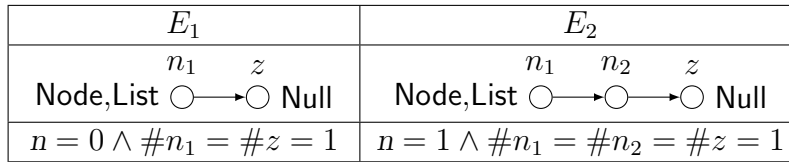
because the canonical names of nodes in E_1 are a subset of canonical names in E_2. The join is done component-wise on the combined domain element: the heap elements are joined via the heap domain's join operation and the same for the integer domain elements. After the join, we get the following domain element.
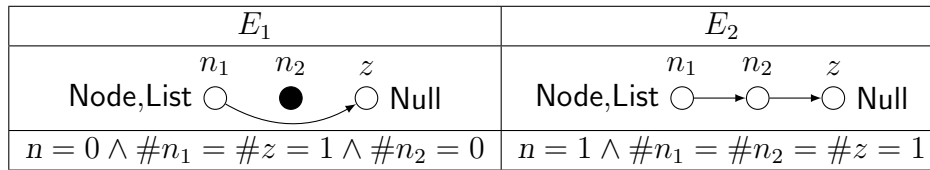
$E_1 \sqcup E_2$
Node,List $\underset{n_1}{\bigcirc} \dashrightarrow \underset{n_2}{\bullet} \dashrightarrow \underset{z}{\bigcirc}$ Null
$0 \leq n \leq 1$
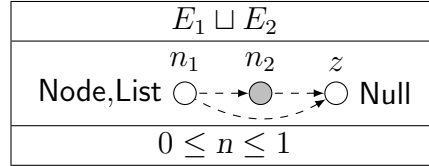
Now we have lost the correlation between n and the list size.

To solve this problem, we share some information between the heap domain and the integer domain. For each node in the heap domain, we create a variable in the integer domain to denote the number of concrete nodes it represents. We call this a cardinality variables. For a node u, the variable is written $\#u$. This variable is always equal to 1 for singleton nodes and 0 for nodes where $\rho = 0$.
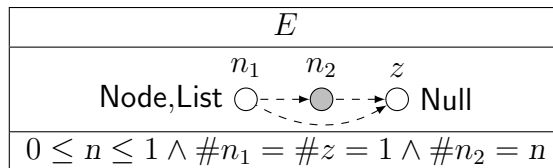
To see how cardinality variables are useful, consider the same domain elements, now augmented with cardinality variables.

E_1	E_2
Node,List $\underset{n_1}{\bigcirc} \longrightarrow \underset{z}{\bigcirc}$ Null	Node,List $\underset{n_1}{\bigcirc} \longrightarrow \underset{n_2}{\bigcirc} \longrightarrow \underset{z}{\bigcirc}$ Null
$n = 0 \wedge \#n_1 = \#z = 1$	$n = 1 \wedge \#n_1 = \#n_2 = \#z = 1$

Recall how the join works. Since the set of node names in E_1 is a subset of the node names in E_2, we add nodes to E_1 to make up the difference. These nodes, shown in black, are added with $\rho = 0$.

E_1	E_2
Node,List $\underset{n_1}{\bigcirc} \quad \underset{n_2}{\bullet} \curvearrowright \underset{z}{\bigcirc}$ Null	Node,List $\underset{n_1}{\bigcirc} \longrightarrow \underset{n_2}{\bigcirc} \longrightarrow \underset{z}{\bigcirc}$ Null
$n = 0 \wedge \#n_1 = \#z = 1 \wedge \#n_2 = 0$	$n = 1 \wedge \#n_1 = \#n_2 = \#z = 1$

Now when we join these two structures, we get the following result.

E
Node,List $\underset{n_1}{\bigcirc} \dashrightarrow \underset{n_2}{\bullet} \dashrightarrow \underset{z}{\bigcirc}$ Null
$0 \leq n \leq 1 \wedge \#n_1 = \#z = 1 \wedge \#n_2 = n$

The integer domain now preserves a correlation between n and $\#n_2$, which is precisely what we need to prove our desired invariant! We need to perform more iterations to complete the analysis; however, we continually are able to preserve our invariant in the integer domain using cardinality variables.

Classes. The foregoing trick of introducing cardinality variables may have seemed fairly specific to the heap/integer combination. However, we can generalize the trick into a useful technique for arbitrary domains. To do so, we define the concept of a *class*. A class is a collection of individuals from some type (such as `int` or `ListNode`). Each domain is responsible for collecting all the individuals it manages into a finite number of classes. This grouping can change over time as individuals are moved between classes or as classes come into and out of existence.

The heap domain creates one class for every node in a structure. The members of this class are the set of concrete objects abstracted by the node, so the class for a singleton node has only one member. We give names to classes and, like in the heap domain, we use bold text to denote summary classes (those with an arbitrary size). The integer domain is also required to group its individuals, the integers, into classes. In this example, there is no need for integer classes, so we lump them all the integers into a single class called $\boldsymbol{Z}$.

Whenever a base domain changes the membership of a class (by moving an individual from one class to another, say) it must inform the combined domain. The combined domain, in turn, informs the other base domain of the change. This is called *repartitioning*. Domains also share information about the size and membership of their classes. They may declare that a class is empty, a singleton, or that it has one or more elements. They also may declare that two classes are disjoint, or that one class is a subset of another.

Cardinality variables themselves are not a feature of the combined domain. They are maintained entirely by the integer domain. When the integer domain is informed of a new, necessarily empty, class, it generates a new cardinality variable equal to zero. When two classes are merged, the size of the new class is the sum of the sizes of the originals. When a class is split in two, we maintain the constraint that the sizes of the new classes sum to the old class size. Thus, every repartitioning causes the integer domain to update the cardinality variables. The next chapter provides more detail about how this works.

3.1.2 Class Sharing

We now consider an example with arrays.

```
1  type T;
2  global table[int]:T;
3
4  procedure init(n:int)
5    i:int;
6  {
7    i := 0;
8    while (i < n) {
9      label loop_inv;
10     table[i] := null;
11     i := i+1;
```

```
12    }
13  }
```

This code initializes an array of pointers to null. Our goal is to ensure that when the loop terminates all the array entries are null. What makes it a challenge is that while the *table* map is managed by the heap domain, its key is an integer. How can the heap domain reason about the values stored in *table* when it knows nothing about integers?

The solution is to use integer classes. In the previous example, the integer domain lumped every integer into a single class $\boldsymbol{Z}$. In this example, we split up the integers more precisely. We maintain five classes.

$$\boldsymbol{I}_{\ll} = \{x : x < 0\}$$
$$\boldsymbol{I}_{<} = \{x : 0 \leq x < i\}$$
$$I_i = \{i\}$$
$$\boldsymbol{I}_{>} = \{x : i < x < n\}$$
$$\boldsymbol{I}_{\gg} = \{x : n \leq x\}$$

Just as we shared the heap classes with the integer domain, we share these classes with the heap domain. Every time the grouping changes, such as in the statement $i := i + 1$, we notify the heap domain of the change.

The heap domain stores these integer classes as nodes in its three-valued structure, as if they were no different than nodes for heap objects. This permits it to treat the *table* map as a predicate, as it does for all the maps it manages. If, for example, we know that *table*[i] points to a node n, then we record $\mathsf{Table}(I_i, n) = 1$.

Returning to the example, we infer the following invariant at the program point labeled `loop_inv`. The edges show the values of the Table predicate.
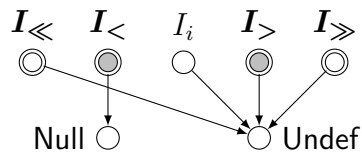
We have introduced a new node in this diagram, labeled with the Undef predicate. When a value has not been assigned to some entry in a map, the heap node assigns it the special value *undef*. This is to avoid the complexity of reasoning about partial functions.

The crucial feature of the invariant is that *table*[$\boldsymbol{I}_{<}$] is known to be null. When the loop terminates, $i = n$ and so all the desired array entries have been initialized to null.

3.1.3 Numerical Fields

The previous example illustrated how integer classes can be useful to reason about a map managed by the heap domain. In this example, we will use heap classes to reason about a map managed by the integer domain.

```
 1  type ListNode;
 2  global data[ListNode]:int;
 3  global next[ListNode]:ListNode;
 4
 5  procedure test()
 6    node, list:T;
 7  {
 8    list := null;
 9    while (*) {
10      node := new ListNode;
11      data[node] := 0;
12      next[node] := list;
13      list := node;
14      label loop_end;
15    }
16    return list;
17  }
```

Suppose we want to verify that the *data* field of every list element generated here is zero. The problem here is similar to the one with *table* before: while the *data* field is managed by the integer domain, its argument (of type `ListNode`) is managed by the heap domain. The solution is the same. We have already explained how the integer domain must be aware of heap domain classes. Besides maintaining cardinality variables about them, it uses them to track the values of maps as well.

Here is a domain element that occurs at the program point labeled `loop_end`. (We have elided cardinality variables.)
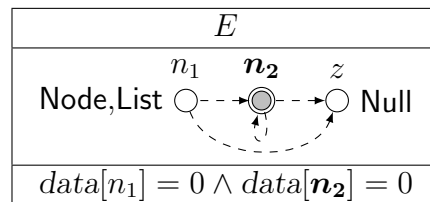
The integer domain is able to track facts about the *data* field in terms of the heap domain classes. This is how we infer the $data = 0$ invariant.

There is some subtlety in the meaning of the fact $data[\boldsymbol{n_2}] = 0$. It is a quantified fact, but the quantifiers are implicit. Writing them out,

$$\forall n \in \boldsymbol{n_2}.\ data[n] = 0.$$

Every class that appears inside such a fact is implicitly quantified. For example, consider the fact $a[\boldsymbol{C}] = b[\boldsymbol{C}]$. Intuitively, we might expect it to mean that every node abstracted by $\boldsymbol{C}$ has equal a and b fields. In fact, it means something much stronger:

$$\forall n \in \boldsymbol{C}.\ \forall n' \in \boldsymbol{C}.\ a[n] = b[n'].$$

Since separate quantified variables are used, it means that every node abstracted by C has an a field equal to the b field of any other node abstracted by C.

Another tricky aspect here is that the heap domain may rearrange its classes, possibly affecting facts in the integer domain. Suppose that the integer domain is notified that the classes n_1 and $\boldsymbol{n_2}$ have been merged into a class $\boldsymbol{n}$. In this case, it is easy to see that we can replace the *data* facts above with the new fact $data[\boldsymbol{n}] = 0$. However, if we had started with the facts $data[n_1] = 10$ and $data[\boldsymbol{n_2}] = 20$, then the merge would force us to replace these with the weaker fact $10 \leq data[\boldsymbol{n}] \leq 20$. We describe later how this works.

3.1.4 Predicate Sharing

Sometimes it is necessary to share more than class information between the domains. Consider these definitions.

```
1  type T;
2  global table[int]:T, index[T]:int;
```

The intention here is that *table* is an array of pointers to T objects. Each such T object has an *index* field that holds its index in the array. Based on their signatures, the *table* array is managed by the heap domain and the *index* field is managed by the integer domain. We want to preserve the invariant that whenever $table[j] = e$, $index[e] = j$. We use the following code as an example.

```
1  procedure add(i:int):T
2    obj:T;
3  {
4    obj := new T;
5    table[i] := obj;
6    index[obj] := i;
7    return obj;
8  }
```

It should be clear that this code preserves the given invariant. We partition the integers in the same way as the previous example. For simplicity, we omit the $\boldsymbol{I_{\ll}}$ and $\boldsymbol{I_{\gg}}$ classes from diagrams. For now, let us assume that all entries in *table* are null except for the one at i. If we analyze the function, we end up with the following state, which implies the given invariant.
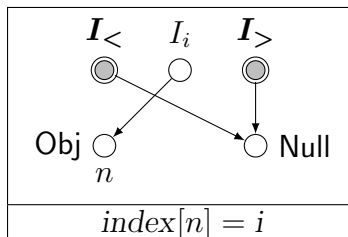
Unfortunately, the assumption that all entries of *table* are initially null is not true in general. Usually, the other entries will point to objects where the invariant is satisfied. We can try to use summary nodes to represent this initial situation.
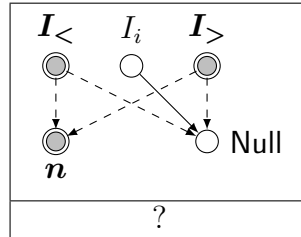
However, it is not clear what fact to put in the integer domain. We would like to constrain $index[\boldsymbol{n}]$, but the value it must take on is different for each concrete node abstracted by $\boldsymbol{n}$. There simply is no way to represent this invariant without improvements to the integer domain.

Integer domain predicates. In TVLA, the solution to such problems is to introduce instrumentation predicates that preserve extra information about summary nodes. We use the same technique here, but the predicate is an integer predicate. Predicates in the integer domain have a much simpler syntax than heap domain predicates, which use first-order logic with transitive closure. Integer predicates are simple atomic formulas of the following form:

$$\text{formula} ::= \text{term} \leq \text{term} \mid \text{term} = \text{term} \mid \text{term} \geq \text{term}$$
$$\text{term} ::= \text{constant} \mid x \mid map[x_1, \ldots, x_n] \mid \#x$$

Each of the x values used here must be an argument to the given predicate. For example, the following is a valid predicate definition.

$$\mathsf{HasIndex}(e{:}\texttt{T}, j{:}\texttt{int}) := index[e] = j$$

This defines a binary predicate that links an object to an integer if the object's *index* field equals the integer. The following diagram shows the same initial state as before, but now the edges represent the $\mathsf{HasIndex}$ predicate.
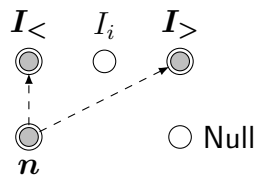

The integer domain is responsible for updating the value of this predicate whenever an assignment takes place. We can also implement an analog of the sharpening operation for integer predicates. The big leap is to expose this predicate to the heap domain. That is, we

tell the heap domain where the HasIndex predicate holds. Every time an assignment takes place, the heap domain is informed of any changes to HasIndex.

Once the heap domain is aware of the HasIndex predicate, it can treat it like any other predicate. In particular, it can define new predicates of its own that use HasIndex.

$$\mathsf{Inv}(j{:}\mathtt{int}) := \forall e{:}\mathtt{T}.\ \mathsf{Table}(j, e) \Rightarrow (\mathsf{HasIndex}(e, j) \vee \mathsf{Null}(e))$$

The following diagram enforces our invariant (that $table[j] = e$ implies $index[e] = j$) using the Inv predicate. The Table and HasIndex predicates are the same as what has already been shown; we omit them.
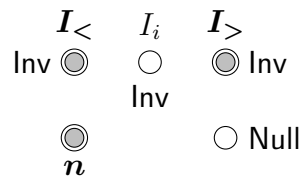
All that matters here is that the $I_<$ and $I_>$ classes satisfy Inv. Consequently, if $table[j] = e$ for any j in $I_<$ or $I_>$, then either $\mathsf{HasIndex}(e, j)$ or $\mathsf{Null}(e)$. If $table[j]$ is null, then we don't care about its index. Otherwise, $\mathsf{HasIndex}(e, j)$ tells us that $index[e] = j$ as desired.

Notice that I_i satisfies Inv. This is because we assumed that $table[i]$ is initially null.

Now we show how to analyze the example code using the HasIndex and Inv predicates. After the allocation of `obj`, we get the following state. The edges show the Table predicate.
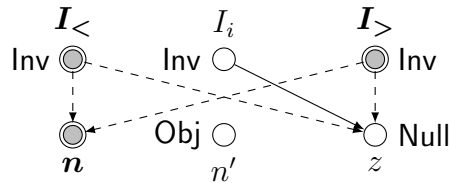
This next diagram shows the HasIndex predicate for the same state. Notice that $index[obj]$ is unknown here.
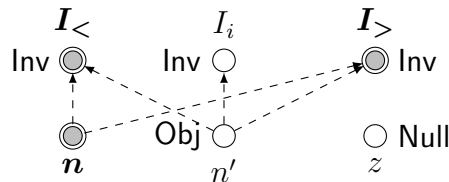
When we analyze the update to *table*, we swing the Table edge from *null* to *obj*. In doing so, we potentially violate $\mathsf{Inv}(I_i)$ because $\mathsf{HasIndex}(n', I_i)$ is not known to be true. No communication with the integer domain is needed here. We get the following state, where edges show the Table predicate.
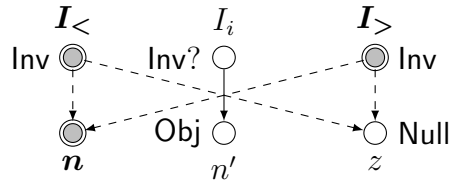
Next, we analyze the $index[obj] := i$ assignment, which is handled by the integer domain. The integer domain recognizes that $\mathsf{HasIndex}(n', I_i)$ must hold after the assignment; it sends this fact to the heap domain. The heap domain realizes that Inv now holds at I_i and we get the following result (we show the $\mathsf{HasIndex}$ edges this time).
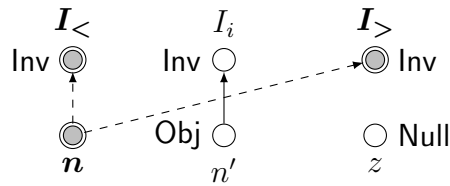
Since Inv holds at all the integer nodes, we have established that our invariant is preserved by the procedure. *It is important to realize that establishing this invariant required us to share class information and share predicate information throughout the analysis.*

3.1.5 Cardinality Functions

Consider the following code, which implements manual reference counting.

```
1  type T, C;
2  global rc[T]:int, contains[C]:T;
3
4  procedure incref(c:C, obj:T)
5  {
6    assert(contains[c] = null);
7    rc[obj] := rc[obj] + 1;
8    contains[c] := obj;
9  }
10
11 procedure decref(c:C)
12   obj:T;
13 {
14   obj := contains[c];
15   contains[c] := null;
16   rc[obj] := rc[obj]-1;
17   if (rc[obj] = 0)
18     delete obj;
19 }
```

We assume there is a set of objects of type T and they are pointed to by objects of type C (via the *contains* map). Many C objects can point to the same T object. The number of incoming pointers to a T object is stored in its *rc* field (its "reference count"). The `incref` procedure adds a *contains* pointer to a T object and increments its reference count. The `decref` procedure nulls out a *contains* pointer; if the T object has no more incoming pointers, it is deleted for good.

Our goal is to prove that, when the T object is deleted, it has no incoming *contains* pointers that might later be illegally dereferenced. To prove this, we prove a stronger invariant: that the number of incoming *contains* pointers to a T object o is equal to $rc[o]$. Since the deletion only happens only if $rc[obj] = 0$, this stronger fact implies the goal.

For a single T object, we can use cardinality variables to imply the stronger invariant. The edges show the *contains* pointer.
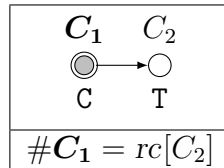
However, there typically will be an arbitrary number of T objects, forcing us to merge them into a summary node.
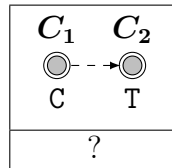
Now there is no clear way to describe the invariant.

To solve the problem, we introduce a new feature to the integer domain: *cardinality functions*. A cardinality function might be defined as follows. (We assume that Contains is a predicate defined by the heap domain to model the *contains* map.)

$$RealRC[n\text{:}\mathtt{T}] := |\{c\text{:}\mathtt{C} : \mathsf{Contains}(c, n)\}|$$

We emphasize that this feature does not require any special support from the combined domain. It can be implemented entirely within the integer domain. The predicate Contains is defined by the heap domain but shared with the integer domain (just as HasIndex was shared with the heap domain in the last section).

Once we have this function, we define an integer domain predicate.

$$\mathsf{RCCorrect}(n\text{:}\mathtt{T}) := rc[n] = RealRC[n]$$

To enforce the invariant, we declare that RCCorrect holds of all T nodes.
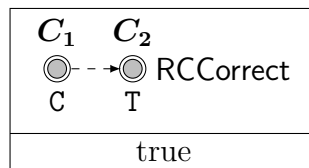
Verifying increment. We assume the following configuration on entering the function. It resembles the state above, except that the `c` and `obj` objects are in their own classes. The `c` object has a *contains* pointer to null.
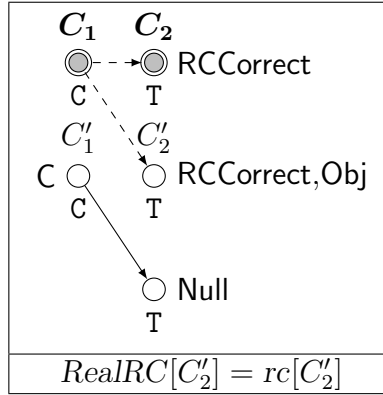
$$RealRC[C_2'] = rc[C_2']$$

When `incref` increments the rc field, the numerical domain handles it. It infers that $RealRC[C_2'] = rc[C_2'] - 1$. It also sets $\mathsf{RCCorrect}(C_2') = 0$ and informs the heap domain.

The heap domain handles the update to the *contains* pointer. It swings the pointer over to the C_2' object and informs the numerical domain that $\mathsf{Contains}(C_1', C_2') = 1$. This causes the numerical domain to update its *RealRC* function at C_2', increasing it by one. Consequently, $RealRC[C_2'] = rc[C_2']$ is reestablished and so $\mathsf{RCCorrect}(C_2')$ is restored. The heap domain is informed, but it performs no further action.

The verification of `decref` is handled similarly.

3.2 Pre-Order and Join

The previous section explained the combined domain at a very high level by example. The next few sections of this chapter present the implementation details of the combined domain. The next chapter explains how the base domains implement the features needed by the examples (particularly how the integer domains handles cardinality reasoning).

In this section, we present the partial order and join operations for the combined domain. We begin with an example to show the challenges inherent in defining an ordering on combined domain elements. Let there be an integer field defined called *data*. In the integer domain we define $\mathsf{P}(x) := data[x] \geq 0$. Consider the elements below.

The heap portion of E^A is called E_1^A and the integer portion is called E_2^A. We use similar notation for E^B.

We are interested in knowing whether $E^A \sqsubseteq E^B$ in the partial order of the combined domain. Stated another way, we want to know if every concrete state abstracted by E^A is also abstracted by E^B. Stated yet another way, does E^A, when interpreted as a constraint on states, imply E^B?

Consider the meanings of the elements. The list in E^A has length $n+1$; every element has a non-negative *data*. In E^B, the list length must be greater than n and the data fields must be non-negative. So E^B is weaker than E^A, meaning that we should be able to establish $E^A \sqsubseteq E^B$. The next few steps show how to do so.

❶ **Saturate shared facts in** E^A. Proving the $\sqsubseteq$ relationship is difficult because the constraints are split across the heap and integer domains in different ways in E^A and E^B. In E^B, the *data* constraint is stored entirely in the heap domain via P. In E^A, it is partially in the heap domain (via $\mathsf{P}(\boldsymbol{C_2})$) and partially in the integer domain (via $data[C_1] = 1$). To remedy the problem, we share facts about P between E_1^A and E_2^A. How this works is described later in the section. In this case, the heap domain informs the integer domain that $\mathsf{P}(\boldsymbol{C_2}) = 1$, so the integer domain recognizes that $data[\boldsymbol{C_2}] \geq 0$. The integer domain knows $data[C_1] = 1$, which implies $\mathsf{P}(C_1) = 1$, so it informs the heap domain of this fact. The heap domain also informs the integer domain that $\#C_1 = 1$, although that fact is not relevant to the discussion.

E^A		E^B
$C_1 \quad \boldsymbol{C_2}$ X,P $\bigcirc$--→◉ P		$\boldsymbol{C_3}$ ◉ X?,P
$\#C_1 = 1 \wedge \#\boldsymbol{C_2} = n \wedge data[C_1] = 1 \wedge data[\boldsymbol{C_2}] \geq 0$		$\#\boldsymbol{C_3} > n$

❷ **Repartition classes.** Next, we notice that E^A and E^B use different class names for some of the same individuals. More precisely, $C_1 \cup \boldsymbol{C_2} = \boldsymbol{C_3}$. To solve the problem, we rewrite E^A to use the same class names as E^B. Note that *both* E_1^A and E_2^A must be re-written, since they both refer to C_1 and $\boldsymbol{C_2}$. In this case, the rewriting process merges C_1 and $\boldsymbol{C_2}$ in E^A. Here is the result.

E^A	E^B
$\boldsymbol{C_3}$ ◉ X?,P	$\boldsymbol{C_3}$ ◉ X?,P
$\#\boldsymbol{C_3} = n + 1 \wedge data[\boldsymbol{C_3}] \geq 0$	$\#\boldsymbol{C_3} > n$

❸ **Apply each subdomain's partial order.** We expect each subdomain D_i to supply its own partial order, which is invoked as $E_i^A \sqsubseteq_i E_i^B$ to test if it holds. After saturation and repartitioning, it should be clear that $E_1^A \sqsubseteq_1 E_1^B$ (since they are identical) and $E_2^A \sqsubseteq_2 E_2^B$ (since E_2^A is stronger). Thus, the relationship $E^A \sqsubseteq E^B$ holds in the combined domain.

This example gives a rough overview of how our partial order is defined. Now we describe each of these steps in depth.

Saturation. We call the process of sharing facts between domains *saturation*. Saturation is a semantic reduction (defined in [14]) 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. Saturation propagates shared facts between the domains until a fixed point is reached.

The shared information passed between domains must be expressed in a language that both domains understand. We describe the common language using a simple grammar.

$$F ::= \forall x \in C.\ F \mid \exists x \in C.\ F \mid \mathsf{P}(x, y, \ldots) \mid \neg \mathsf{P}(x, y, \ldots) \mid t : [\ C_1, C_2, \ldots, C_n\ ] \qquad (3.1)$$

C is a class from either domain. P is a shared predicate from either domain. It may be the equality predicate. All variables appearing in P must be bound by quantifiers.

The equality predicate $=$ is interpreted by both domains. We use it to express cardinality information about classes. For example, $C = \emptyset$ is written as $\forall n \in C.\ n \neq n$. To say $|C| \leq 1$, we write $\forall n, n' \in C.\ n = n'$.

The special form $t : [\ C_1, C_2, \ldots, C_n\ ]$ expresses a few constraints simultaneously. First, it says that the classes C_i are all mutually disjoint. Second, it tells us that every individual of type t belongs to one of the classes C_i. We call the set $\{C_1, \ldots, C_n\}$ a *partitioning*. A domain may expose more than one partitioning of its individuals. Classes from different partitionings may overlap in arbitrary ways. It would be possible to extend the language to allow for even more expressive ways of comparing classes, but we have found partitionings to be sufficient for our purposes.

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

The pseudocode in Figure 3.1 shows how facts are propagated. They are accumulated via $Consequences_i$ and then passed to the domains with $Assume_i$. In the example above, at step ❶, the fact $\forall n \in C_1.\ \mathsf{P}(n)$ was returned by $Consequences_2(E_2^A)$ and then propagated to E_1^A by $Assume_1$. Because the number of predicates and classes in any element is bounded, this process is guaranteed to terminate.

Repartitioning. Classes are the mechanism by which a domain describes to the other domain its individuals and the predicates that hold over them. 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. Repartitioning rewrites class names and may also change how individuals are grouped into classes. This permits the partial order check to normalize the class names between its two arguments.

Step ❷ in the example above was actually made up of two phases. The first phase matched up the classes C_1 and $\boldsymbol{C_2}$ with $\boldsymbol{C_3}$. The second phase used this matching to rewrite

function Saturate(E_1, E_2):
   $F := \emptyset$
   repeat:
      $F_0 := F$
      $F := F \cup Consequences_1(E_1) \cup Consequences_2(E_2)$
      $E_1 := Assume_1(E_1, F)$
      $E_2 := Assume_2(E_2, F)$
   until $F_0 = F$
   return $\langle E_1, E_2 \rangle$

Figure 3.1: Implementation of combined domain saturation.

both E_1^A and E_2^A. Both phases are handled entirely by the subdomains. However, the combined domain defines the interface through which they communicate.

Each domain exposes a function $MatchClasses_i$ that matches the classes of E_i^A to E_i^B. The heap domain matches two classes if they have the same canonical name; we describe how the integer domain handles this operation later. The result of matching is a relation R, where $(C, C') \in R$ if class C in E^A matches with class C' in E^B. In the example, $R = \{(C_1, \boldsymbol{C_3}), (\boldsymbol{C_2}, \boldsymbol{C_3})\}$. In the relation R, we say the the "original" classes are being mapped to "new" classes. In the example, C_1 and $\boldsymbol{C_2}$ are original classes and $\boldsymbol{C_3}$ is the new class.

$MatchClasses_i$ also returns a set F of cardinality and disjointness facts about the new classes. Knowing size and disjointness information about the new classes can make the rewriting more precise.

The rewriting is implemented by the subdomain through the $Repartition_i(E_i, R, F)$ operation. Its job is to rewrite the classes according to R, knowing that the facts in F hold about the new classes.

Pre-Order. The pseudocode in Figure 3.2 shows in detail how to test $E^A \sqsubseteq E^B$. It is a three-step process of saturation, repartitioning, and subdomain ordering. We use a simple utility function, Repartition, whose job is to call $Repartition_i$ for each base domain.

First, note that we only saturate E^A. The purpose of saturation is to strengthen individual subdomain components, and there is no point in strengthening E^B when we are trying to prove that $E^A \Rightarrow E^B$.

Next we use the subdomains' $MatchClasses_i$ functions. The relation R_i maps the classes of E_i^A to those of E_i^B. Both R_1 and R_2 are used to repartition E_1^A and E_2^A. This is because each subdomain can make reference to classes defined by the other subdomain. We do not repartition E^B at all; since Repartition may weaken a domain element, it would be unsound to do so.

Note that repartition is called only once on each subdomain. This is unlike satura-

function Repartition($\langle E_1, E_2 \rangle, R, F$):
   $E_1 := Repartition_1(E_1, R, F)$
   $E_2 := Repartition_2(E_2, R, F)$
   return $\langle E_1, E_2 \rangle$

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

   $\langle R_1, F_1 \rangle := MatchClasses_1(E_1^A, E_1^B)$
   $\langle R_2, F_2 \rangle := MatchClasses_2(E_2^A, E_2^B)$

   $\langle E_1^{A'}, E_2^{A'} \rangle := \text{Repartition}(\langle E_1^A, E_2^A \rangle, R_1 \cup R_2, F_1 \cup F_2)$
   return $(E_1^{A'} \sqsubseteq_1 E_1^B) \wedge (E_2^{A'} \sqsubseteq_2 E_2^B)$

Figure 3.2: Pseudocode for combined domain's partial order.

tion, which iterates to a fixed point. As a consequence, the subdomains must make their repartitioning decisions independent of the other domain's partitioning.

Join. Join is similar to implies checking, but is symmetric in the two arguments. Figure 3.3 shows the code. Instead of matching the classes of E^A to the classes of E^B, we allow both inputs to be repartitioned into a new set of classes that may be more precise than either of the originals. Thus, we require that domains to expose a *MergeClasses_i* operation that returns a mapping from either element's original classes to new classes. Also note in the code that we saturate both inputs, rather than only E^A.

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

    The typical use of widening is as follows. Suppose we have a sequence of domain element $E_1, E_2, E_3, \ldots$ such that $E_1 \sqsubseteq E_2 \sqsubseteq E_3 \sqsubseteq \cdots$. We would like to find a new sequence E_i' such that $E_i \sqsubseteq E_i'$ and such that the E_i' converges after a finite number of steps. We get the

function $\langle E_1^A, E_2^A \rangle \sqcup \langle E_1^B, E_2^B \rangle$:
$\qquad \langle E_1^A, E_2^A \rangle := \text{Saturate}(E_1^A, E_2^B)$
$\qquad \langle E_1^B, E_2^B \rangle := \text{Saturate}(E_1^B, E_2^B)$

$\qquad \langle R_1^A, R_1^B, F_1 \rangle := MergeClasses_1(E_1^A, E_1^B)$
$\qquad \langle R_2^A, R_2^B, F_2 \rangle := MergeClasses_2(E_2^A, E_2^B)$

$\qquad \langle E_1^{A'}, E_2^{A'} \rangle := \text{Repartition}(\langle E_1^A, E_2^A \rangle, R_1^A \cup R_2^A, F_1 \cup F_2)$
$\qquad \langle E_1^{B'}, E_2^{B'} \rangle := \text{Repartition}(\langle E_1^B, E_2^B \rangle, R_1^B \cup R_2^B, F_1 \cup F_2)$

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

Figure 3.3: Pseudocode for combined domain's join algorithm.

desired result if we use widening:

$$E_1' := E_1$$
$$E_2' := E_1' \nabla E_2$$
$$E_3' := E_2' \nabla E_3$$
$$\vdots$$
$$E_{i+1}' := E_i' \nabla E_{i+1}$$

We assume that the base domains already come equipped with widening operations ∇_i. In practice, the heap domain does not require widening since the canonical abstraction is finite. So widening for the heap domain is a join. In the integer domain, widening is important and several widening operators are available.

The challenging part of widening is that some widenings that are "obviously correct" may fail to terminate. Miné [36] describes how this can occur in an integer domain. Widening typically works by throwing away facts, producing a less precise element, in order to reach a fixed point more quickly. The problem occurs if we try to saturate the left-hand operand. Saturation will put back facts that we might have thrown away, thereby defeating the purpose of widening. So to ensure that a widened sequence terminates, we never saturate the left-hand operand. The code is in Figure 3.4.

This code is very similar to the code for the join. Besides avoiding saturate on E^A, we also avoid repartitioning E^A. Our goal is to avoid any changes to E^A that might cause the widening to fail to terminate. Since we do not repartition E^A, we use $MatchClasses_i$ instead of $MergeClasses_i$.

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

  $\langle R_1, F_1 \rangle := MatchClasses_1(E_1^B, E_1^A)$
  $\langle R_2, F_2 \rangle := MatchClasses_2(E_2^B, E_2^A)$

  $\langle E_1^{B'}, E_2^{B'} \rangle := \text{Repartition}(\langle E_1^B, E_2^B \rangle, R_1 \cup R_2, F_1 \cup F_2)$

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

Figure 3.4: Combined domain's widening algorithm.

3.3 Assignment

To see how assignment works, consider a new example.

```
1 type T;
2 global data[T]:int;
3 global next[T]:T;
4 global head:T;
5 global count:int;
```

We assume that T nodes are linked together in a list via *next* pointers starting at *head*. They store integers in their *data* fields. Our goal is to ensure that *count* is equal to the number of T objects reachable from *head* whose *data* fields are zero.

To accomplish this task, we define the following instrumentation predicates and functions, shown along with the domain in which they are defined.

$$\text{Reach}(n{:}\mathsf{T}) := \exists h{:}\mathsf{T}.\ \text{Head}(h) \wedge \text{TC}(h, n; x, y).\ \text{Next}(x, y) \qquad \text{(heap)} \qquad (3.2)$$
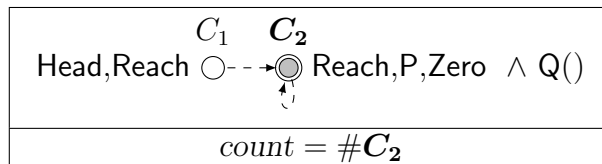$$\text{Zero}(n{:}\mathsf{T}) := data[n] = 0 \qquad \text{(integer)} \qquad (3.3)$$
$$\text{P}(n{:}\mathsf{T}) := \text{Reach}(n) \wedge \text{Zero}(n) \qquad \text{(heap)} \qquad (3.4)$$
$$NumZero := |\{n{:}\mathsf{T} : \text{P}(n)\}| \qquad \text{(integer)} \qquad (3.5)$$
$$\text{Q}() := count = NumZero \qquad \text{(integer)} \qquad (3.6)$$

Now consider the following combined domain element.

This domain element represents a linked list where all nodes except the head have an *data* field equal to zero. We assume that $\mathsf{Q}()$ is known to hold.

Now we perform the assignment $data[head] := 0$. We proceed in four steps.

❶ **Saturate shared facts.** Saturation may increase the precision of the final result. In this example, it doesn't, so we ignore the saturation step.

❷ **Translate assignment terms.** The first problem we face is that *data* is a function defined in the integer domain while *head* is defined in the heap domain. Thus, neither domain is able to understand this assignment by itself. To solve the problem, we translate "foreign" terms into classes. So *head* is translated to C_1 and then the integer domain is asked to assign *data* to 0 at some heap individual in class C_1. Translation is performed recursively to allow for arbitrary nesting of mixed functions.

❸ **Perform initial assignment.** The integer domain performs the assignment to *data* at C_1. It returns a new element that records the fact that $data[C_1] = 0$ (we can say this only because C_1 is a singleton class; if it were a summary class, the assignment would only apply to some nodes abstracted by the class). Assignment also returns two sets, U and C. U (for "updates") is a set of shared facts that hold after the assignment and may not have held before. C (for "change") is a set of shared facts that used to hold and may no longer hold. In this case, U contains the fact $\forall o \in C_1.\ \mathsf{Zero}(o)$.

❹ **Propagate updates to shared predicates.** The U and C information is now passed to the heap domain. The P predicate in the heap domain depends on Zero (even though it doesn't know what Zero means). When it sees that Zero has gone up at C_1, it checks that elements of C_1 also satisfy Reach—which they do—and recognizes that P holds at C_1 after the assignment. So it adds $\forall o \in C_1.\ \mathsf{P}(o)$ to U.

The new U set is now sent back to the integer domain. Since P has gone up at C_1, the value of *NumZero* increases by one. Therefore, $count = NumZero - 1$, and so $\mathsf{Q}()$ no longer holds. Thus, $\neg\mathsf{Q}()$ is added to the set U. The new U set is sent back to the heap domain. It records the new value of Q. At this point no more changes take place and we are done.

So far we have not described C. It contains facts that may have been changed by the assignment and should no longer be trusted. Elements of C have a simpler form than elements of U. When we added $\forall o \in C_1.\ \mathsf{P}(o)$ to U, we add $\mathsf{P}(C_1)$ to C, saying that this fact was affected by the assignment. When $\neg\mathsf{Q}()$ was added to U, we also add $\mathsf{Q}()$ to C because the old value $\mathsf{Q}() = 1$ stored in the heap domain is no longer correct. If a domain sees a fact in C, it checks if updated information is available in U. If not, it must "forget" that fact (or set it to $1/2$ in the heap domain).

This example gives a rough overview of how assignment works. Note the importance of predicates whose definitions depend on other predicates. In the remainder of this section we describe how terms are translated to classes and then we show the pseudocode for assignment.

function TranslateFull$_1$(E_1, E_2, env, $f(e_1, \ldots, e_k)$):
    if $f \in D_1$:
        for $i \in [1..k]$: $\langle e_i', \text{env} \rangle := $ TranslateFull$_1$($E_1, E_2, \text{env}, e_i$)
        return $\langle f(e_1', \ldots, e_k'), \text{env} \rangle$
    else:
        for $i \in [1..k]$: $\langle e_i', \text{env} \rangle := $ TranslateFull$_2$($E_1, E_2, \text{env}, e_i$)
        $x := $ some var not in env
        $\text{env} := \text{env}[x \mapsto \mathit{Translate}_2(E_2, \text{env}, f(e_1', \ldots, e_k'))]$
        return (x, env)

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

Figure 3.5: Pseudocode for term translation.

Term translation. The term translator recursively traverses a term. Its goal is to make a term understandable to one of the subdomains, say D_1. Each time it sees the application of a "foreign" function from D_2, it recursively translates the foreign term to one that D_2 understands (since there may be arbitrary nesting of D_1 and D_2 terms) and then asks D_2 to translate that term to a class. The last step happens via a function $\mathit{Translate}_i$ that the subdomain must provide.

As an example, suppose we translate the term $f[c]$ to a D_1 term, where $f \in D_1$ and c is a nullary function from D_2. Since c is not understood by D_1, we replace it with a fresh variable, say x. This yields the result $f[x]$. To make sense of x, we ask D_2 to translate c to a class. It should return a class containing the individual that c equals. Suppose it is returns C. Then we create an environment $[x \mapsto C]$ along with $f[x]$. This is a form that D_1 can understand.

We can also translate the $f[c]$ term to D_2. We first convert c to a class as before, yielding the term x and the environment $[x \mapsto C]$. Then we convert $f[x]$ to a term that D_2 understands. We apply $\mathit{Translate}_1$ to $f[x]$, giving it the environment $[x \mapsto C]$ as input. Suppose it returns the class C'. Then we make a fresh variable y. Our resulting term is simply y with the environment $[y \mapsto C', x \mapsto C]$.

The function TranslateFull$_1$ implements the procedure above. It converts a given term to one understood by D_1. The related function TranslateFull$_2$ translates to a term understood by D_2. We give the code in Figure 3.5.

Assignment. The assignment operation for the combined domain follows the steps ❶ to ❹ listed above. It saturates the domain element first. Then, if the assignment is to a function f that belongs to D_i, it translates both sides of the assignment to terms D_i understands. Next, it asks D_i to perform the assignment via the subdomain's $Assign_i$ function and return initial

function $\text{Assign}(\langle E_1, E_2 \rangle, f(e_1, \ldots, e_k), e)$:
   $\langle E_1, E_2 \rangle := \text{Saturate}(E_1, E_2)$

   if $f \in D_1$:
      $\langle l, \text{env} \rangle := \text{TranslateFull}_1(E_1, E_2, [\,], f(e_1, \ldots, e_k))$
      $\langle r, \text{env} \rangle := \text{TranslateFull}_1(E_1, E_2, \text{env}, e)$
      $\langle E_1', U, C \rangle := Assign_1(E_1, \text{env}, l, r)$
      $E_2' := E_2$
   else:
      $\langle l, \text{env} \rangle := \text{TranslateFull}_2(E_1, E_2, [\,], f(e_1, \ldots, e_k))$
      $\langle r, \text{env} \rangle := \text{TranslateFull}_2(E_1, E_2, \text{env}, e)$
      $\langle E_2', U, C \rangle := Assign_1(E_2, \text{env}, l, r)$
      $E_1' := E_1$

   $j := 1$
   repeat:
      $\langle E_1', U, C \rangle = PostAssign_1(E_1, E_1', j, U, C)$
      $\langle E_2', U, C \rangle = PostAssign_2(E_2, E_2', j, U, C)$
      $j := j + 1$
   until $j = \text{num\_strata}$

   $\langle R_1, F_1 \rangle := EliminateClasses_1(E_1')$
   $\langle R_2, F_2 \rangle := EliminateClasses_2(E_2')$
   return $\text{Repartition}(\langle E_1', E_2' \rangle, R_1 \cup R_2, F_1 \cup F_2)$

Figure 3.6: Pseudocode for assignment transfer function. num_strata is the total number of shared predicates.

U and C sets. Information in U and C is propagated between D_1 and D_2 via a subdomain function $PostAssign_i$ that must be provided. The full code is shown in Figure 3.6.

First, we take note of the form of U and C. The set U contains standard facts, like the ones returned by $Consequences_i$. The set C is simpler: its elements are tuples of the form $\mathsf{P}(C_1, \ldots, C_k)$. If such a tuple belongs to C, it means that the truth of $\mathsf{P}(a_1, \ldots, a_k)$ may change during the assignment if each $a_i \in C_i$. If $\mathsf{P}(C_1, \ldots, C_k) \in C$, then a domain must drop any information it has about P at those classes However, there may be new facts in U to supercede this information, so the change need not lead to imprecision.

We now discuss the last three lines of the code, which call $EliminateClasses_i$, in the context of the heap abstraction. Since assignment can affect predicate values, two nodes may end up with the same canonical name after the assignment. Neither the $Assign_i$ or $PostAssign_i$ functions are allowed to change partitioning at all, but we still want to give the

domain an opportunity to merge nodes. Partitioning changes must be delayed until after all *PostAssign$_i$* calls are done. Then the *EliminateClasses$_i$* function, provided by the base domains, returns a relation from old classes to new classes and Repartition is called. In the heap domain, *EliminateClasses$_i$* computes canonical names and returns a relation that merges nodes with the same canonical name.

To ensure that the propagation of updates through *PostAssign$_i$* works, we require that shared predicates be ordered. For example, the predicates define in Equation 3.2 through Equation 3.6 are listed in order. A predicate can only depend on a previously listed predicate. This ensures that there are no cycles in the predicate definitions. We let num_strata be the total number of shared predicates. We pass j, the index of the predicate to be updated, to *PostAssign$_i$* so that it knows which predicate to recompute.

3.4   Branching

This section explains how branches are handled in the combination domain. First we explain how branches are handled in general by abstract interpreters. Consider the following control-flow graph fragment.
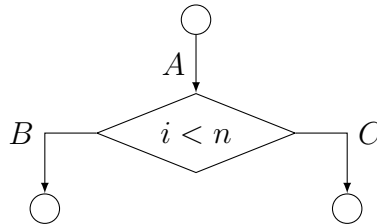
A very conservative way to handle branches would be to ignore them. The domain element at A would then flow directly to B and C. However, we would like to model the branch precisely. Somehow, we want $\gamma(C)$ to be $\gamma(A)$ but with only states where $i < n$. And $\gamma(B)$ should be $\gamma(A)$ but excluding states where $i < n$. We can get this effect by transforming the graph as follows.
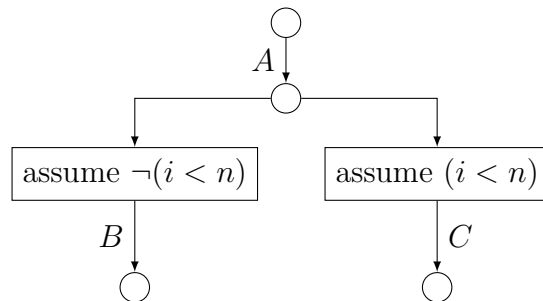
We then define a transfer function for the assume statement that filters out states that do not match the given condition. We already require the base domain to supply a function

Assume$_i$, which is used for saturation. We reuse this function in defining a transfer function Assume for the combined domain.

The inputs to Assume are a domain element and an atomic formula $\mathsf{P}(e_1, \ldots, e_k)$, where each e_i is a closed term, such as *next*[*next*[*head*]]. The predicate P can be a predicate defined by either of the domains or it can be equality. We demonstrate how Assume works with an example. Suppose we want to Assume($\langle E_1, E_2 \rangle, \mathsf{P}(f[c])$), where $f \in D_1$ and $c \in D_2$. The steps are as follows.

❶ **Translate the term to D_1.** The first step is to transform $f[c]$ to a form understood by D_1. We already discussed this example in the section on assignment. Assuming that *Translate$_2$* applied to c yields class C, we get back the term $f[x]$ and the environment $[x \mapsto C]$.

❷ **Translate the term to D_2.** We will also translate the term to D_2 using TranslateFull$_2$. We have also seen this example before. The result is the term y in environment $[y \mapsto C', x \mapsto C]$.

❸ **Existentially quantify the terms.** In D_1, we want to assume the fact $\mathsf{P}(f[x])$ in an environment $[x \mapsto C]$. We translate this to the following quantified fact.

$$\exists x \in C.\ \mathsf{P}(f[x])$$

An existential quantifier is used because the term c corresponds to only one individual in C, so we can assume the fact at only that place in C. Ideally, the class C returned by *Translate$_i$* would be a singleton class, but this is not always possible.

It is easy to convert a term and an environment into this form. For each entry in the environment, we add an existential quantifier. We can do the same thing for the D_2 term, yielding the following.

$$\exists y \in C'.\ \exists x \in C.\ \mathsf{P}(y)$$

❹ **Call Assume in base domains.** We call the *Assume$_i$* function for both base domains. Typically only one domain will understand the assumption and the other will ignore it. However, in the case of equalities, both domains may be able to understand the fact in some way.

In the example, we would make two calls. (The set notation is used because *Assume$_i$*, as used in saturation, takes a set of facts as input.)

$$Assume_1(E_1, \{\exists x \in C.\ \mathsf{P}(f[x])\})$$
$$Assume_2(E_2, \{\exists y \in C'.\ \exists x \in C.\ \mathsf{P}(y)\})$$

The code for all this is shown in Figure 3.7.

3.5   Base Domain Requirements

This section reviews the requirements that must be satisfied by a base domain. First, a base domain must be equipped with the usual partial order, join, and widening operations.

function AddExistentials(env, f):
    for $[x \mapsto C] \in$ env: $f := (\exists x \in C.\ f)$
    return f

function Assume($\langle E_1, E_2 \rangle, \mathsf{P}(e_1, \ldots, e_k)$):
    $\text{env}_1 := [\ ]$
    for $i \in [1..k]$: $\langle e_{1i}, \text{env}_1 \rangle := \text{TranslateFull}_1(E_1, E_2, \text{env}_1, e_i)$
    $f_1 := \text{AddExistentials}(\text{env}_1, \mathsf{P}(e_{11}, \ldots, e_{1k}))$

    $\text{env}_2 := [\ ]$
    for $i \in [1..k]$: $\langle e_{2i}, \text{env}_2 \rangle := \text{TranslateFull}_2(E_1, E_2, \text{env}_2, e_i)$
    $f_2 := \text{AddExistentials}(\text{env}_2, \mathsf{P}(e_{21}, \ldots, e_{2k}))$

    return $\langle Assume_1(E_1, \{f_1\}), Assume_2(E_2, \{f_2\}) \rangle$

Figure 3.7: Pseudocode for assume transfer function.

It also must supply other functions that expose the classes and predicates of a domain element to the combined domain. We describe these functions here. All these requirements were stated in the previous section; we consolidate them here as a service to the reader.

    We use a few type definitions in the descriptions below. A "fact" is defined in Equation 3.1; a set of facts is written Set(fact). A "term" has the following form.

$$t ::= x \mid f[t_1, \ldots, t_k]$$

That is, it is either a variable or a function applied to terms. We list a few other type abbreviations as well.

$$\text{repartitioning} := \text{Set}(\text{class} \times \text{class})$$
$$\text{environment} := \text{Map}(\text{variable} \to \text{class})$$

---

function $Assume_i(E_i{:}D_i,\ F{:}\text{Set(fact)}) : D_i$

    Given a domain element and a set of facts, this function returns a new domain element where the facts in F are constrainted to be true. As an example, if E_i is the integer domain element $x = 0$ and $F = \{y = x + 1\}$, then the result might be $x = 0 \wedge y = 1$.

---

function $Consequences_i(E_i{:}D_i) : \text{Set(fact)}$

This function computes all the possible facts of the form given in Equation 3.1 that are implied by E_i. This set will include facts about classes (their size and disjointness, phrased using the equality predicate) and facts about predicates exposed by the domain.

---

function $Repartition_i(E_i{:}D_i,\ R{:}\text{repartitioning},\ F{:}\text{Set(fact)}) : D_i$

This function rewrites the element E_i according to the repartitioning relation R. An element of R has the form $\langle C, C' \rangle$, meaning that some elements of class C may be elements of class C' in the rewritten element. The set F gives facts about the sizes and disjointness of the new classes.

---

function $MatchClasses_i(E_i{:}D_i,\ E'_i{:}D_i) : \text{repartitioning} \times \text{Set(fact)}$

This function generates a relation that matches the classes of E_i to the classes of E'_i. It also returns a set of facts F about the classes of E'_i. This information can be used to rewrite E_i so that it uses the same set of classes as E'_i.

---

function $MergeClasses_i(E_i{:}D_i,\ E'_i{:}D_i) :$
    $\text{repartitioning} \times \text{Set(fact)} \times \text{repartitioning} \times \text{Set(fact)}$

This function is similar to $MatchClasses_i$. However, it generates a new set of classes rather than using the classes of E'_i. It returns a way to map the classes of both E_i and E'_i to these new classes.

---

function $Translate_i(E_i{:}D_i,\ \text{env:environment},\ t{:}\text{term}) : \text{class}$

This function translates the term t into a class. The individual denoted by the term should belong to the class that is returned. The term may contain free variables; these variables are bound in the environment env. The individual denoted by a variable x belongs to the class env$[x]$.

---

function $Assign_i(E_i{:}D_i,\ \text{env:environment},\ l{:}\text{term},\ r{:}\text{term}) : D_i \times \text{Set(fact)} \times \text{changes}$

This is the transfer function for the assignment $l := r$. Both l and r are terms whose free variables are interpreted according to env. That is, each x corresponds to an individual belonging to the class env$[x]$. The function returns a new domain element reflecting the assignment. It also returns sets U and C, where U is a set of facts that are known to hold after the assignment and C is a set of things whose truth may have been affected by the assignment. The elements of U are standard facts. The elements of C have the form

$\mathsf{P}(C_1, \ldots, C_k)$, meaning that P may have changed at some $(a_1, \ldots, a_k)$ where $a_i \in C_i$ for each i. The work to process an assignment is finished by $PostAssign_i$.

---

function $PostAssign_i(E_i{:}D_i,\ E_i'{:}D_i,\ j{:}\text{int},\ U{:}\text{Set(fact)},\ C{:}\text{changes})$:
    $D_i \times \text{Set(fact)} \times \text{changes}$

This function propagates predicate updates after the initial transfer function for an assignment runs. The element E_i is the initial element before the assignment transfer function ran. The E_i' element is the "current" element reflecting the ongoing progress of processing the assignment. The integer j tells the domain which predicates have already been processed in the ordered list of predicates. The sets U and C list the changes to predicate values. A new element E_i' and new U and C sets are returned after the predicate with index j from domain i has been processed.

---

function $EliminateClasses_i(E_i{:}D_i)$: repartitioning $\times$ Set(fact)

This function may rearrange the class structure of the element E_i due to the changes made by an assignment operation. It returns a repartitioning to be applied to the element and a set of facts about the new classes (size, disjointness, etc.).

3.6 Related Work

There are several approaches to combining numerical reasoning with TVLA. The first one, by Gopan et al. [23], proceeded in two parts. In one paper, the authors defined the concept of a *summarizing numerical domain*, which can express quantified numerical properties. We use many of the ideas from the work in our integer domain, as described in the next chapter. Next, the authors show how a summarizing numerical domain can be integrated into TVLA to handle arrays [22]. Our approach is more general than this one. First, their domain does not support any form of cardinality reasoning. Second, it does not support integer predicates. Finally, it relies on a hard-coded array abstraction; we use canonical abstraction to partition arrays, which is much more flexible.

Gulwani et al. [25] introduce a general method for tracking cardinality. Our combined domain is a generalization of their framework in several ways. Besides cardinality, we support quantified numerical facts, integer predicates, and cardinality instrumentation functions.

Halbwachs and Péron [26] present a technique to reason about the contents of arrays, and the relationship between array elements. This technique infers many useful array properties, but it performs no cardinality reasoning and it cannot reason about recursive data structures.

Many authors [6, 20, 32] have described systems in which pointer analysis is used to convert a program that manipulates heap and integer values into an integer-only program.

This "reduction-based approach" uses a variety of existing integer analyzers on the resulting program. For proving simple properties of singly linked lists it was shown by Bouajjani et al. [6] that there is no loss of precision. However, the technique may lose precision in cases where the heap and integers interact in complicated ways, including many of the examples shown in this chapter. Our approach, in which heap and integer reasoning is arbitrarily interleaved, is more expressive.

Chang et al. [9] describe a separation logic-based analysis in which numerical constraints are supported inside recursive predicates. Their system supports both cardinality reasoning and quantified integer facts, which distinguishes it from the other systems above. However, it does not support instrumentation functions for cardinality, as our system does. Also, the integer portion of the analysis was never implemented.

Emmi et al. [21] handle reference counting using auxiliary functions and predicates similar to the ones we describe. As long as only a finite number of sources and targets are updated in a single transition, they automatically generate the corresponding updates to their auxiliary functions. For abstraction, they use Skolem variables to name single, but arbitrary, objects. Their combination of techniques is specifically directed at reference counting; it supports a form of universal quantification (via Skolem variables) to track the cardinality of reference predicates. In contrast, we have a parametric framework for combining domains, as well as a specific instantiation that supports universal and existential quantification, transitive closure, and cardinality. Their analyzer supports concurrency and ours does not. Because their method is unable to reason about reachability, their method could not verify our examples (or `thttpd`).

The idea to combine numeric and pointer analysis for establishing properties of memory was pioneered by Deutsch [17, 18]. 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. Venet [48] elaborates on Deutsch's work.

Yavuz-Kahveci and Bultan [50] present an algorithm for inferring sizes of singly linked lists. 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 the work of Bouajjani et al. [6] and Cook et al. [32].

Rugina [44] 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.

Calcagno et al. [8] describe a method 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.

The seminal paper by Cousot and Cousot in [14] introduces different methods for combining abstract domains and Deutsch [16] elaborates on domain constructors. There are several methods for implementing or approximating the reduced product [14], which is the

most precise refinement of the direct product. Granger's method of *local descending itera-tions* [24] uses a decreasing sequence of reduction steps to approximate the reduced product. The method provides a way to refine abstract *states*; in abstract *transformers*, domain elements can only interact either before or after transformer application. The *open-product* method [12] allows domain elements to interact *during* transformer application. However, the use of classes in our technique expands set of facts that can be shared between domains beyond what can be expressed in these frameworks. Our technique is consequently more precise, at least when combining quantified domains.

3.7   Conclusion

In this chapter 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 presented examples showing the power of the domain. No other automatic analysis that we know of is able to prove all the invariants that we do in these examples. Proofs of these invariants are necessary for establishing the memory safety of typical systems code. In the final chapter of the thesis, we describe the verification of the cache data structure of a real-world web server. *Every single one of the example invariants must be proved, in some form, in this server.*

3.8 Proofs

This section presents a formal semantics for the combined domain and a proof that it is sound. It also contains formal correctness requirements for all of the base domain functions.

3.8.1 Meaning of Domain Elements

We formalize the semantics of the base domains in an unusual way. Each domain D_i should supply a meaning predicate $\gamma_i(E_i, S, M, P)$, where S, M, and P are as follows. S is a tuple $\langle A_1, A_2, F \rangle$, where A_1 is the universe of individuals managed by D_1, A_2 is the same for D_2, and F gives an interpretation to functions: $F(f)(a_1, a_2) = a_3$ means that $f(a_1, a_2) = a_3$. All functions are expected to be total. Functions can have mixed signatures. M is a mapping from class names to individuals. P is an interpretation of predicates: if $P(\mathsf{P})(a_1, a_2) = \text{true}$ then $\mathsf{P}(a_1, a_2)$ holds. If the predicate $\gamma_i(E_i, S, M, P)$ holds, then the state S satisfies the constraints of element E_i given the interpretations M and P of classes and predicates.

We define a meaning predicate for the combined domain:

$$\gamma'(\langle E_1, E_2 \rangle, S, M, P) := \gamma_1(E_1, S, M, P) \wedge \gamma_2(E_2, S, M, P).$$

Example 43 To see how these functions work, consider the following domain element, which appeared in §3.3.
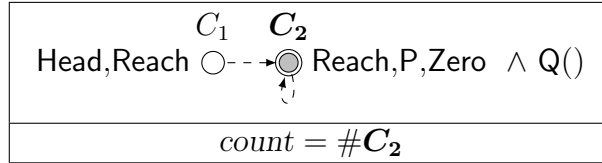
We want to construct values for S, M, and P so that γ' is satisfied for this element.

We let $S = \langle A_1, A_2, F \rangle$. We let the set of heap objects $A_1 = \{o_1, o_2\}$. We let A_2 contain the integers ($\mathbb{Z}$). We define the functions as follows.

$$F(head)() = o_1$$
$$F(count)() = 1$$
$$F(next)(o_1) = o_2$$
$$F(next)(o_2) = o_2$$
$$F(data)(o_1) = 1$$
$$F(data)(o_2) = 0$$

We map the classes so that $M(C_1) = \{o_1\}$ and $M(C_2) = \{o_2\}$. Finally, we give assignments to predicates. We only show the true predicates; everything else is false.

$$P(\mathsf{Head})(o_1), P(\mathsf{Next})(o_1, o_2), P(\mathsf{Next})(o_2, o_2), P(\mathsf{Reach})(o_1), P(\mathsf{Reach})(o_2),$$
$$P(\mathsf{Zero})(o_2), P(\mathsf{P})(o_2), P(\mathsf{Q})()$$

Given these values, $\gamma'(\langle E_1, E_2 \rangle, S, M, P)$ holds. □

We define the predicate $\widehat{M_i}$ to ensure some well-formedness conditions.

$$\widehat{M_i} := (\forall C \in E_i.\ M(C) \subseteq A_i) \wedge \left(\bigcup_{C \in E_i} M(C) \right) = A_i$$

This predicate tells us that, for each class C defined by E_i, $M(C)$ contains only individuals managed by E_i. Additionally, every individual belongs to some class. We require that γ_i hold only if $\widehat{M_i}$ hold. Also, we define $\widehat{M'}$ to hold if $\widehat{M_1}$ and $\widehat{M_2}$ both hold.

3.8.2   Meaning of Facts

The facts exchanged between domain elements have the form shown in Equation 3.1. Here we formalize the meaning of such facts via the function γ_f. We use the syntax $[\cdot]_S$ to interpret terms containing function applications according to F.

$$[x]_S = x$$
$$[f(e_1, \ldots, e_k)]_S = S(f)([e_1]_S, \ldots, [e_k]_S)$$
$$\gamma_f(F, S, M, P) = \forall f \in F.\ \gamma_f(f, S, M, P)$$
$$\gamma_f((\forall x \in C.\ f), S, M, P) = \forall a \in M(C).\ \gamma_f(f, S, M, P)$$
$$\gamma_f((\exists x \in C.\ f), S, M, P) = \exists a \in M(C).\ \gamma_f(f, S, M, P)$$
$$\gamma_f(\mathsf{P}(e_1, \ldots, e_k), S, M, P) = P(\mathsf{P})([e_1]_S, \ldots, [e_k]_S)$$
$$\gamma_f(\neg \mathsf{P}(e_1, \ldots, e_k), S, M, P) = \neg P(\mathsf{P})([e_1]_S, \ldots, [e_k]_S)$$
$$\gamma_f(t : [\ C_1, \ldots, C_n\ ], S, M, P) = \left(\bigwedge_{i,j} M(C_i) \cap M(C_j) = \emptyset \right) \wedge \left(\bigcup_i M(C_i) = A_t \right)$$

In the last equation, A_t is the universe A_i, where D_i is the domain responsible for type t.

3.8.3   Soundness

Saturation

The conditions for the *Consequences$_i$* function are that it return true facts.

$$\forall S, M, P.\ \gamma_i(E_i, S, M, P) \Rightarrow \gamma_f(Consequences_i(E_i), S, M, P)$$

The conditions for *Assume$_i$* say that if a state satisfies the initial domain element and the branch predicate, then it should also satisfy the domain element returned by *Assume$_i$*. That is, for any set of facts F, the following must hold.

$$\forall S, M, P.\ \gamma_i(E_i, S, M, P) \wedge \gamma_f(F, S, M, P) \Rightarrow \gamma_i(Assume_i(E_i, F), S, M, P)$$

Lemma 1 *Let Saturate be applied to input $\langle E_1, E_2 \rangle$, producing output $\langle E'_1, E'_2 \rangle$. Then,*

$$\forall S, M, P.\ \gamma'(E_1, E_2, S, M, P) \Rightarrow \gamma'(E'_1, E'_2, S, M, P). \qquad \square$$

The proof is immediate from the properties above.

Repartition

The correctness condition for a subdomain's $Repartition_i$ 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 $Repartition_i(E_i, R, F) = E_i'$. 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 in $\widehat{R}$ below) must satisfy E_i'. We can write this logically as follows.

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

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 $\widehat{R}$: M' conforms to R if $M'(C)$ contains only individuals that come from classes C where $R(C, C')$. We also check $\widehat{M}$, because we want to ensure that M' does not lose any individuals.

$$\widehat{R}_i(E_i, R, S, M, M') := \widehat{M}_i(E_i, S, M') \wedge \left(\forall C'. M'(C') \subseteq \bigcup_{C:R(C,C')} M(C) \right)$$

As before, we assume that $\widehat{R'}$ checks $\widehat{R_1}$ and $\widehat{R_2}$.

Lemma 2 *Let S, M, P be an arbitrary state and let M' be any class interpretation. Let $\langle E_1', E_2' \rangle$ be the result of applying Repartition to tuple E_1, E_2 over R and F. Then the following holds.*

$$(\gamma'(E_1, E_2, S, M, P) \wedge \widehat{R'}(E_1, E_2, R, S, M, M') \wedge \gamma_f(F, S, M', P)) \Rightarrow \gamma'(E_1', E_2', S, M', P) \quad \square$$

The proof follows directly from the restrictions on $Repartition_i$.

Class Matching

The soundness requirements on $MatchClasses_i$ mirror the assumptions needed to apply $Repartition_i$. They are fairly lax, meaning that $MatchClasses_i$ can choose almost any R as long as every class maps to something. Let $MatchClasses_i(E_i^A, E_i^B) = (R, F)$. Then we require the following.

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

Note that E_i^B does not even appear!

We put similar restrictions on $MergeClasses_i$. We write the restriction for E_i^A but a similar one applies for E_i^B.

$$\forall S, M, P. \ \gamma_i(E_i^A, S, M, P) \Rightarrow \exists M'. \ \widehat{R}_i(E_i^A, R_i^A, S, M, M') \wedge \gamma_f(F, S, M', P)$$

Pre-order

First we state correctness restriction on the subdomains' partial orders.

$$\forall S, M, P.\ E_i^A \sqsubseteq_i E_i^B \Rightarrow (\gamma_i(E_i^A, S, M, P) \Rightarrow \gamma_i(E_i^B, S, M, P))$$

Now we can prove that the combined domain partial order is correct according to γ'.

Theorem 5 *If $\langle E_1^A, E_2^A \rangle \sqsubseteq \langle E_1^B, E_2^B \rangle$ is true and S, M, P satisfy $\gamma'(E_1^A, E_2^A, S, M, P)$, then $\gamma'(E_1^B, E_2^B, S, M', P)$ holds for some M'.*   □

PROOF After saturation, S, M, P continue to satisfy $\langle E_1^A, E_2^A \rangle$. We then call $MatchClasses_i$, which guarantee that R and F are such that a call to Repartition is permitted. The postcondition for Repartition tells us that S and P now satisfy $\langle E_1^{A'}, E_2^{A'} \rangle$ for some M'. Then if $\sqsubseteq_1$ and $\sqsubseteq_2$ return true, then S, M', P satisfy $\langle E_1^B, E_2^B \rangle$.   ■

Additional properties of $\sqsubseteq$ are desirable. We prove that, given some monotonicity and transitivity properties on operations $MatchClasses_i$, $Repartition_i$, $Consequences_i$, and $Assume_i$, our implies relation is both transitive and reflexive. It is not anti-symmetric because there are different abstract elements that have the same denotation. Thus, it is a pre-order.

Transitivity.   To ensure that $\sqsubseteq$ is transitive, we need some restrictions. Assume that A, B, and C are elements of the combined domain. Let $A = \langle A_1, A_2 \rangle$ and the same for B and C. Let $A' = \text{Saturate}(A_1, A_2)$ and the same for the others. In the ensuing paragraphs, we use the notation that $MatchClasses_*(A, B) = \langle MatchClasses_1(A_1, B_1), MatchClasses_2(A_2, B_2) \rangle$. We use a similar convention for other domain functions.

To begin, we need to restrict $MatchClasses_*$. Assume that

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

We define the notation that, when $\mathcal{M} = \langle \langle R_1, F_1 \rangle, \langle R_2, F_2 \rangle \rangle$,

$$\mathcal{M}(E) = \text{Repartition}(\langle E_1, E_2 \rangle, R_1 \cup R_2, F_1 \cup F_2).$$

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,

$$\mathcal{M}(Assume_*(E, F)) = Assume_*(\mathcal{M}(E), F).$$

Finally, we need a monotonicity condition on $Assume_*$ and $Consequences_*$.

$$A_i \sqsubseteq_i B_i \Rightarrow Consequences_i(B_i) \subseteq Consequences_i(A_i)$$
$$A_i \sqsubseteq_i B_i \wedge F \subseteq F' \Rightarrow Assume_i(A_i, F') \sqsubseteq_i Assume_i(B_i, F)$$

Using these conditions, we prove transitivity. We begin by proving a lemma.

Lemma 3 *If $A' \sqsubseteq_* B$ then $A' \sqsubseteq_* B'$.* □

PROOF We first note the following.

$$A_1' = Assume_1(A_1', Consequences_2(A_2'))$$

By the monotonicity of $Consequences_2$, $Consequences_2(B_2) \subseteq Consequences_2(A_2')$. The by the monotonicity of `Assume`,

$$Assume_1(A_1', Consequences_2(A_2')) \sqsubseteq_1 Assume_1(B_1, Consequences_2(B_2)).$$

Therefore,
$$A_1' \sqsubseteq_1 Assume_1(B_1, Consequences_2(B_2)).$$

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.

Theorem 6 *If $A \sqsubseteq B$ and $B \sqsubseteq C$ then $A \sqsubseteq C$.* □

PROOF Using the definition of the combined domain's order, $\sqsubseteq$, we can rewrite $A \sqsubseteq B$ as $\mathcal{M}_{A'B}(A') \sqsubseteq_* B$. We can also rewrite $B \sqsubseteq C$ as $\mathcal{M}_{B'C}(B') \sqsubseteq_* C$. We know that $B' \sqsubseteq_* B$ by monotonicity of $Assume_i$. 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 $\mathcal{M}_{B'C}(\mathcal{M}_{A'B}(A')) \sqsubseteq_* C$. From above, this means that $\mathcal{M}_{A'C}(A') \sqsubseteq_* C$, which is the definition of $A \sqsubseteq C$. ∎

Reflexivity. We need only one condition on $MatchClasses_i$ to guarantee reflexivity: that $\mathcal{M}_{AA}(A) = A$. 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.

Join

First we state the correctness conditions for the subdomains' $\sqcup_i$ operations. (We write the restriction for E_i^A but a similar one applies for E_i^B.)

$$\forall S, M, P. \; \gamma_i(E_i^A, S, M, P) \Rightarrow \gamma_i(E_i^A \sqcup_i E_i^B, S, M, P)$$

Now we prove soundness for the combined domain.

Theorem 7 *If a state S satisfies $\langle E_1^A, E_2^A \rangle$ (or $\langle E_1^B, E_2^B \rangle$) then S satisfies $\langle E_1^A, E_2^A \rangle \sqcup \langle E_1^B, E_2^B \rangle$.* □

PROOF Assume we are given a state S that satisfies $\langle E_1^A, E_2^A \rangle$. S also satisfies the saturated element. The calls to $MergeClasses_i$ guarantee the invariants needed to satisfy the precondition for Repartition; its post-condition guarantees that S satisfies $\langle E_1^{A'}, E_2^{A'} \rangle$. Using the property of $\sqcup_i$ above for both domains, we realize that S satisfies $\langle (E_1^{A'} \sqcup_1 E_1^{B'}), (E_2^{A'} \sqcup_2 E_2^{B'}) \rangle$. ∎

Widening

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 $\langle E_1^A, E_2^A \rangle$ is satisfied. Since E_1^A and E_2^A are passed to the subdomains' widening operators unchanged, it is clear that we S also satisfies the result of our widening.

What if S satisfies $\langle E_1^B, E_2^B \rangle$ instead? After calling $MatchClasses_i$, R and F satisfy the pre-conditions for Repartition; its post-condition tells us that S satisfies $\langle E_1^{B'}, E_2^{B'} \rangle$. Since $E_1^{B'}$ and $E_2^{B'}$ are passed to the subdomains' widening operators, S also satisfies the result of our combined widening.

Assume

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

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

Now we can put conditions on $Translate_i$. Assume $C = Translate_i(E_i, \mathrm{env}, f(e'_1, \ldots, e'_k))$. Let env be an environment and x any variable not in env.

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

Lemma 4 *Assume that $TranslateFull_i(E_1, E_2, \mathrm{env}, e)$ returns (e', env'). Then*

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

Additionally, if $\mathrm{FV}(e') = V_1$ *and* $\mathrm{dom}(\mathrm{env}) = V_2$, *then* $V_1 \cap V_2 = \emptyset$ *and* $\mathrm{dom}(\mathrm{env}') = V_1 \cup V_2.$ $\square$

PROOF 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 $\mathrm{TranslateFull}_i$ here, inductively. It tells us that for all arguments j, $T_=(\mathrm{env}, M, [e_j]_S, [e'_j]_S)$, 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

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

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

$$T_=(\mathrm{env}[x \mapsto C], M, [f(e_1, \ldots, e_k)]_S, x),$$

assuming C is the result of `Translate`. Since the variable properties are also satisfied, we have proved the goal for $\mathrm{TranslateFull}_i$. ∎

Now we prove the soundness of Assume.

Theorem 8 *Let $\langle E_1, E_2 \rangle$ be the inputs to Assume, along with $P(P)([e_1]_S, \ldots, [e_k]_S)$. Let $\langle E'_1, E'_2 \rangle$ be the output. Then the following holds.*

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

PROOF The calls to TranslateFull$_i$ generate an environment and set of expressions such that $T_=(\text{env}_i, M, [e_{ij}]_S, [e'_{ij}]_S)$. TranslateFull$_i$ 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 $\gamma_f(f_i, S, M, P)$. Then the soundness condition on $Assume_i$ shows that the goal holds. ∎

Assign

We prove that if a state initially satisfies $\langle E_1, E_2 \rangle$, 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 $PostAssign_i(E_i, E'_i, j, U, C)$. 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.

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

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 $PostAssign_i$. 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 $T_j \cap \mathrm{Preds}(D_i)$, where $\mathrm{Preds}(D_i)$ 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 $\mathrm{Similar}(P, P', C)$ to mean that P and P' are equal except at places where

a change has occurred according to C. More formally, $\mathrm{Similar}(P, P', C)$ holds when

$$\forall \mathsf{P}. \; \forall a_1, \ldots, a_k. \; \left(\neg \exists C_1, \ldots, C_k. \; \bigwedge_i a_i \in M(C_i) \wedge \mathsf{P}(C_1, \ldots, C_k) \in C \right) \Rightarrow$$
$$P(\mathsf{P})(a_1, \ldots, a_k) \Leftrightarrow P'(\mathsf{P})(a_1, \ldots, a_k)$$

We make some assumptions about $Assign_i$. We use the notation $A(S)$ to model how the assignment changes the state. If $S = \langle A_1, A_2, F \rangle$, then $A(S) = \langle A_1, A_2, F' \rangle$ where

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

We require the subdomain to provide an $Assign_i$ operation satisfying the following. If the result of $Assign_i(E_i, \mathrm{env}, l, r)$ is $\langle E_i', U, C \rangle$, then the following should hold.

$$\forall S, M, P. \; \gamma_i(E_i, S, M, P) \wedge T_=(\mathrm{env}, M, [e]_S, [r]_S) \wedge T_=(\mathrm{env}, M, [f(e_1, \ldots, e_k)]_S, [l]_S) \Rightarrow$$
$$\exists P'. \; \gamma_i(E_i', A(S), M, P') \wedge I_i(E_i, E_i', 1)$$
$$\wedge \mathrm{Similar}(P, P', C) \wedge \gamma_f(U, A(S), M, P') \wedge T_1 \subseteq \mathrm{dom}(P')$$

We also place conditions on $PostAssign_i$. Let $(E_i'', U', C') = PostAssign_i(E_i, E_i', j, U, C)$. Then,

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

With these conditions in place, soundness follows directly.

Chapter 4

Domain Adaptations

The previous chapter described a framework for combining together two abstract domains, such as a heap domain and an integer domain. Our combination framework places additional requirements on the base domains beyond what is normally needed (see §3.5). It is very easy to satisfy these requirements in a trivial way: a domain need not expose any classes or predicates and it can ignore information from the other base domain. However, such a domain will get no benefit from our combination framework; the analysis results would be the same if the domain were used independently.

Consequently, this chapter describes adaptations to the heap and integer domains so that they work well together. These adaptations are required in order to be able to analyze the examples from §3.1. We start with the heap domain from Chapter 2 and make some simple changes. Then we describe the domain of difference-bound matrices, which reasons about integer variables. We make more significant additions to this domain so that it supports classes, predicates, and cardinality reasoning.

4.1 Heap Domain Modifications

This section deals with two fundamental issues regarding the heap domain that we have not yet addressed.

- Our description of the heap domain in Chapter 2 did not discuss how the domain handles all the features of the PBJ language. The next section describes how objects with different types are handled, how the domain reasons about null and undefined values, and how PBJ map variables are interpreted as TVLA predicates.

- The heap domain assumes that any two nodes in a three-valued structure represent different objects. In the combined domain, we allow base domains to expose classes that are not mutually disjoint. Since we model these classes as nodes in the heap domain, we must now deal with the fact that nodes may not represent disjoint individuals.

4.1.1   Types and Functions

First we discuss how we deal with types and uninterpreted functions (i.e., map variables) in the heap domain. §1.1 provides some background on PBJ map variables.

Types.   When we first described the heap domain, we treated all nodes homogenously. However, the individuals modeled by these nodes have distinct types in a PBJ program. The first change we make to the heap domain is to annotate each heap node with the type of the individuals it represents. Formally, if a heap element $E = \langle U, \rho, \iota \rangle$, we now say that U is a mapping from classes to their types. We include the type in the canonical name of a node so that nodes with different types are never merged together.

We also add type annotations to bound variables in predicate definitions, as in the following.

$$\mathsf{BufferShared}(b\text{:}\texttt{Buffer}) := \exists n_1\text{:}\texttt{Map}.\ \exists n_2\text{:}\texttt{Map}.\ \mathsf{Next}(n_1, n) \wedge \mathsf{Next}(n_2, n) \wedge n_1 \neq n_2$$

In some places in the thesis we already included types for explanatory purposes. But these types also have semantic significance. When we evaluate a quantifier (universal, existential, or transitive closure) we only consider nodes of the correct type.

Partial functions and special nodes.   Every PBJ type except `int` contains a special *null* value. We allow this value because it is such a common feature of imperative languages. TVLA handles *null* by modeling a node whose *next* field is *null* as having no outgoing Next edge. This corresponds to treating functions as partial.

However, our PBJ semantics in §1.1 already uses partial functions to model undefined variables. We found dual uses of partial functions to be confusing. Our solution is to treat all functions as total and to introduce special *null* and *undef* individuals to each type to model null and undefined variables. These individuals satisfy the abstraction predicates Null and Undef respectively. Each one has its own node in the three-valued structure. When a new variable *map* comes into scope, its value is *undef* at every point in its domain. Assigning *null* to it causes it to point to the null node.

We also use a special "ur" node to handle allocation. The ur node is a summary node whose cardinality is unbounded. When a new node of a given type is allocated, we split an arbitrary singleton node from the ur node. This interpretation of allocation means that our universe of individuals is fixed, which simplifies the semantics of formula evaluation (concretely, it means that individuals cannot appear and disappear over time).

Function translation.   Consider the following code.

```
1  type T;
2  global map[T]:T;
3  global term:T;
```

The heap domain stores the *map* variable using a core predicate Map. Our convention is that $\mathsf{Map}(x, y) = 1$ if $map[x] = y$. We require that all instrumentation predicates refer to variables like *map* using their core predicates. However, it is easier for the user to use the function syntax directly, as in the following.

$$\mathsf{P}(x{:}\mathsf{T}) := map[x] = term$$

We would like to translate the uses of *map* and *term* into uses of core predicates Map and Term automatically.

One way of performing the translation is to introduce existential quantifiers, as in the following.

$$\mathsf{P}(x \;{:}\mathsf{T}) := \exists y{:}\mathsf{T}.\; \exists x{:}\mathsf{T}.\; \mathsf{Map}(x, y) \wedge \mathsf{Term}(z) \wedge y = z$$

This translation can be done very mechanically. Suppose e is an expression containing a function application somewhere. We write this as $e(f(a, b, c))$. We translate it to $\exists d.\; \mathsf{F}(a, b, c, d) \wedge e(d)$. The type annotation we put on d is determined by the signature of the map f. We also require that the new quantifier be placed deep enough that a, b, and c are already in scope.

However, an alternate translation using universal quantifiers is possible. We could translate $e(f(a, b, c))$ to $\forall d.\; \mathsf{F}(a, b, c, d) \Rightarrow e(d)$. Both translations are valid because all of our functions are total: for any tuple (a, b, c), there will be exactly one d satisfying $\mathsf{F}(a, b, c, d)$. From this perspective, it would seem that we can choose arbitrarily between the universal and existential interpretations.

However, it turns out that the distinction does matter. Consider the following structure, where edges denote a predicate E and X and Y represent the variables x and y.
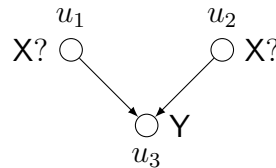
Now suppose we have the formula $\mathsf{E}(x, y)$ to evaluate. If we use the existential interpretation, we will evaluate

$$\exists x.\; \exists y.\; \mathsf{X}(x) \wedge \mathsf{Y}(y) \wedge \mathsf{E}(x, y).$$

This will evaluate to $1/2$. However, this value is imprecise. No matter whether $x = u_1$ or $x = u_2$, x has an edge to y. So the answer 1 is correct. And indeed, the universal interpretation below will evaluate to 1.

$$\forall x.\; \forall y.\; \mathsf{X}(x) \Rightarrow \mathsf{Y}(y) \Rightarrow \mathsf{E}(x, y)$$

The problem here is that the existential interpretation does not encode the fact that x is a total function. It considers the possibility that x is undefined, in which case there may be no node satisfying X.

There are also situations where the universal interpretation is imprecise. Consider the following similar structure.

$$u_1 \qquad u_2$$
$$\mathsf{X?}\ \bigcirc \qquad\qquad \bigcirc\ \mathsf{X?}$$

$$\bigcirc\ \mathsf{Y}$$
$$u_3$$

When we make the same query, $\mathsf{E}(x, y)$, the existential interpretation yields 0, the correct value. The universal interpretation evaluates to $1/2$.

We can solve this problem by using both interpretations simultaneously. That is, we convert to the following query.

$$(\exists x.\ \exists y.\ \mathsf{X}(x) \wedge \mathsf{Y}(y) \wedge \mathsf{E}(x, y)) \sqcap (\forall x.\ \forall y.\ \mathsf{X}(x) \Rightarrow \mathsf{Y}(y) \Rightarrow \mathsf{E}(x, y))$$

Here, $\sqcap$ is the meet operator over the lattice of truth values. Given two truth values, it picks the most precise one. It should never be applied to values that disagree (like 0 and 1). We can modify our derivative algorithm to handle such formulas. Given a meet formula, returns the meet of the derivatives of the operands.

Our current implementation uses the existential translation. We have seen cases in practice where the universal translation would be more precise. Eventually we would like to switch to the translation that uses meet, but we have not yet done so.

4.1.2   Class Representation

As we mentioned earlier, there is a one-to-one relationship between combined domain classes and nodes in the heap structure. Integer classes are treated no differently than the heap domain's own classes. This approach allows us to effortlessly handle functions and predicates with mixed signatures, like `x[T]:int`, since the X predicate can link a heap class to an integer class.

However, classes as defined in the combined domain are more general than heap domain classes. Classes in the heap domain are mutually disjoint. The combined domain, on the other hand, allows classes to overlap.[1] Information about how classes overlap is given in the form of partitionings, which were explained in §3.2. A partitioning is a set of classes. Two classes from the same partitioning must be mutually disjoint. Classes from different partitionings may overlap. Additionally, every individual must be abstracted by some class in a given partitioning; consequently, we say that the partitioning is *exhaustive*. Thus, a partitioning is an exhaustive collection of mutually disjoint classes. We write $t : [\, C_1 \mid \cdots \mid C_n \,]$ to describe a partitioning of individuals, of type t, containing the classes $C_1, \ldots, C_n$.

Since we model classes as heap nodes, we must eliminate our assumption that heap nodes are mutually disjoint. Doing so requires a few changes.

---

[1]This flexibility is important for the integer domain. As we show later, the user is allowed to divide up the integers in multiple ways simultaneously. For example, when reasoning about several arrays, it can be useful to use a different partitioning of the integers for each one.

- First, we must change how equality is handled.

- Second, we change the way that quantifiers are handled when evaluating formulas.

Equality. Chapter 2 described how we use the Eq predicate to track whether a node is a singleton or a summary node. If $\mathsf{Eq}(n, n) = 1$, it is a singleton. If $\mathsf{Eq}(n, n) = 1/2$, then it is a summary node. And for two distinct nodes n and n', we always have $\mathsf{Eq}(n, n') = 0$. This last fact follows from the property that all heap classes are mutually disjoint.

Since integer classes may not be mutually disjoint, it is possible that $\mathsf{Eq}(n, n') \neq 0$ for two distinct integer classes n and n'. We set up the equality predicates according to partitionings. Given two distinct integer classes C and C', we say the following.

$$\mathsf{Eq}(C, C') = \begin{cases} 1/2 & \text{if } C \text{ and } C' \text{ are in different partitionings} \\ 0 & \text{if } C \text{ and } C' \text{ are in the same partitioning} \end{cases}$$

We define $\mathsf{Eq}(C, C)$ based on whether C is a singleton class or a summary class.

Quantifiers. Consider the following example. Suppose that there are two integer partitionings, $\mathtt{int} : [\, C \,]$ and $\mathtt{int} : [\, C' \,]$. This means that C contains all the integers and so does C'. Suppose that $\mathsf{P}(C) = 1/2$ and $\mathsf{P}(C') = 1$. Imagine that we wish to evaluate the query $\forall x{:}\mathtt{int}.\ \mathsf{P}(x)$.

Using our existing query evaluation algorithm, we will consider all integer classes, computing the answer as $\mathsf{P}(C) \wedge \mathsf{P}(C')$. This will give us $1/2$. However, since we know that C' covers the entire set of integers, we could simply evaluate $\mathsf{P}(C')$, which gives the more precise answer 1.

In general, let there be k partitionings, $P_1, \ldots, P_k$, where $P_i = \{C_1, \ldots, C_{|P_i|}\}$. Each partitioning's classes, taken together, cover the entire universe of individuals. Therefore, it is sound to quantify over only the classes in any one partitioning. But which one do we choose? When we evaluate the query, we separately evaluate the quantifier for each partitioning and then take the most precise answer among them all. Writing this formally, for a given formula φ and a universal quantifier,

$$\left(\bigwedge_{C \in P_1} \varphi(C) \right) \sqcap \cdots \sqcap \left(\bigwedge_{C \in P_k} \varphi(C) \right).$$

We evaluate existentials in the same way.

$$\left(\bigvee_{C \in P_1} \varphi(C) \right) \sqcap \cdots \sqcap \left(\bigvee_{C \in P_k} \varphi(C) \right)$$

We have ignored the effect of ρ here, but it must of course be factored in.

4.1.3   Matching, Merging, and Repartitioning

The functions $MatchClasses_1$ and $MergeClasses_1$ are fairly simple in the heap domain. They rely entirely on the canonical names of nodes. When applied to elements E^A and E^B, the $MatchClasses_1$ function is supposed to find a repartitioning R that rewrites the class names of E^A to the class names of E^B. It iterates through every heap class C in E^A. For each one, it looks for a node in E^B with the same canonical name. If it finds a match C', it adds the pair (C, C') to the repartitioning R. Otherwise, it generates a new node name C'' and adds (C, C'') to R. The second case is necessary if E^A has more nodes than E^B.

The $MergeClasses_1$ function is very simple. For arguments E^A and E^B, its job is to find repartitionings R^A and R^B that, when applied to their respective elements, generate two resulting elements that use the same class names. For R^A, we simply call $MatchClasses_1$. For R^B, we generate the identity repartitioning: $R^B = \{(C, C) : C \in E^B\}$. The result is that the rewritten elements will use the same class names as E^B. Using a richer set of class names might produce a more precise result, but so far we have not seen a need to do so.

The $Repartition_1$ function is also straightforward. Suppose we are given a structure $S = \langle U, \rho, \iota \rangle$ and a repartitioning R. We return the following new structure. The notation $C : (C, C') \in R$ means to consider every C such that $(C, C') \in R$.

$$U' = \{C' : \exists C.\ (C, C') \in R\}$$

$$\rho'(C') = \bigvee_{C:(C,C')\in R} \rho(C)$$

$$\iota'(\mathsf{P})(C'_1, \ldots, C'_k) = \bigsqcup_{C_1:(C_1,C'_1)\in R} \cdots \bigsqcup_{C_k:(C_k,C'_k)\in R} \iota(\mathsf{P})(C_1, \ldots, C_k)$$

The $EliminateClasses_1$ function is very simple. It computes the canonical name of every node. All nodes with the same canonical name are mapped to the same node in the repartitioning that is returned.

4.1.4   Assignment

This section describes the implementation of the assignment $f[e_1, \ldots, e_k] := e$. Recall that in the combined domain, the work is split between $Assign_1$ and $PostAssign_1$. In the heap domain, $Assign_1$ is responsible for updating core predicates and $PostAssign_1$ updates instrumentation predicates via finite differencing. We first describe $Assign_1$ through several examples, since there are a number of complications that can occur due to overlapping classes.

Example 44 The simplest case is an assignment such as $f[x] := y$ in the following structure. The edge shows the existing F predicate and X and Y model the x and y variables.

$$n_x$$
X ○
|
○   ○ Y
n_z n_y
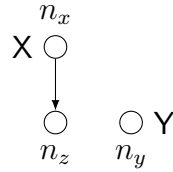
In this case, we set $\mathsf{F}(n_x, n_z) = 0$ and set $\mathsf{F}(n_x, n_y) = 1$. □

Example 45 A problem arises if e (in this case y) cannot be resolved to a single class. Consider the following.

$$n_x$$
X ○
|
○   ○ Y?   ○ Y?
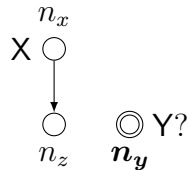n_z n_y n'_y
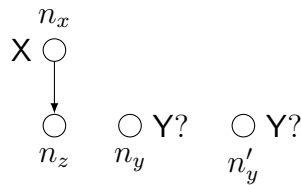
In this case, we set $\mathsf{F}(n_x, n_z) = 0$ and set $\mathsf{F}(n_x, n_y) = \mathsf{F}(n_x, n'_y) = 1/2$.

The same problem occurs in the following example.

$$n_x$$
X ○
|
○   ◎ Y?
n_z $\boldsymbol{n_y}$

Again, we are forced to set $\mathsf{F}(n_x, \boldsymbol{n_y}) = 1/2$. □

The situation shown above, where we were forced to set a predicate value to $1/2$ instead of 1, is called a *weak update*. When we are allowed to set a predicate value to 1, we call the update a *strong update*.

Example 46 A worse problem happens when a function argument e_i cannot be unambiguously determined, as in the following with x.

$$n_x \quad n'_x$$
X? ○   ○ X?
|
○   ○ Y
n_z n_y
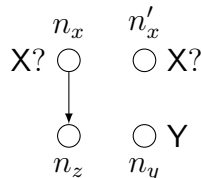
In this case, we are unable even to set the old value of f to zero. Consequently, we end up with the following state after the assignment.
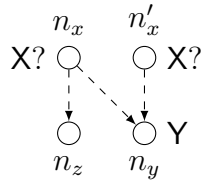
Typically the focus operation avoids such problems. □

Example 47 The addition of integer classes makes assignment more difficult. The problem is that, while the integer domain may translate a term to a singleton class, there may be other classes that contain the same individual as that class. This is possible because integer classes are allowed to overlap. We rely on the equality predicate to tell us where there is overlap

As an example, suppose we want to assign $f[i] := y$, where i is an integer variable. Consider the following heap structure.

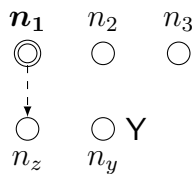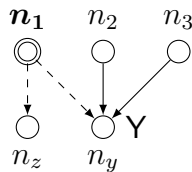Assume that the term i is translated by the integer domain to class n_2. Additionally, suppose that $\mathsf{Eq}(n_2, n_3) = 1$ and that $\mathsf{Eq}(n_2, \boldsymbol{n_1}) = 1/2$.

In this situation, we can make strong updates to n_2 and n_3. We must make a weak update to $\boldsymbol{n_1}$.

Situations like this are fairly common in the integer domain when there are multiple partitionings of the integers. □

The examples above show how we change the predicate F in response to an assignment to f. Besides updating the abstract element, $Assign_1$ is also required to return sets U and C, detailing changes to shared predicates. Since the heap domain shares all its predicates, we return changes to the F predicate in these sets.

More precisely, if we update the value of $\mathsf{F}(C_1, C_2, C_3)$, then we add $\mathsf{F}(C_1, C_2, C_3)$ to C. This registers the fact that a change to F has taken place there. Additionally, if we set $\mathsf{F}(C_1, C_2, C_3) = 1$, then we add the following to U.

$$\forall x_1 \in C_1.\ \forall x_2 \in C_2.\ \forall x_3 \in C_3.\ \mathsf{P}(x_1, x_2, x_3)$$

Similarly, if we set $\mathsf{F}(C_1, C_2, C_3) = 0$, we add the following.

$$\forall x_1 \in C_1.\ \forall x_2 \in C_2.\ \forall x_3 \in C_3.\ \neg\mathsf{P}(x_1, x_2, x_3)$$

PostAssign. The operation of the *PostAssign*$_1$ function is similar. Given a set of values in the U and C sets, it converts them to derivatives. For each entry $\mathsf{P}(C_1, \ldots, C_k) \in C$, we check if there is a corresponding positive entry in U. If there is, we let $\mathsf{P}^+(C_1, \ldots, C_k) = 1$ and $\mathsf{P}^-(C_1, \ldots, C_k) = 0$. If there is a negative entry, we let $\mathsf{P}^-(C_1, \ldots, C_k) = 1$ and $\mathsf{P}^+(C_1, \ldots, C_k) = 0$. If there are no corresponding U entries, we let $\mathsf{P}^+(C_1, \ldots, C_k) = \mathsf{P}^-(C_1, \ldots, C_k) = 1/2$.

After setting up the initial derivates based on U and C, we evaluate the derivative formulas for the predicate who update is requested (via the j parameter). We add the computed derivatives to U and C using the same rules as above and return these new sets.

4.2 Integer Domain

So far we have discussed our integer domain only at a very high level. This section fills in the details. The domain is based on the existing domain of difference-bound matrices [36], which is a compromise between the imprecision of interval arithmetic and the exponential complexity of polyhedra [2]. §1.4 gives a more detailed overview of why we chose this domain.

We begin with an overview of the domain, which was first described by Miné [36]. Next, we add new kinds of dimensions to the domain, such as the cardinality dimensions $\#C$. We then add integer predicates, which behave in much the same way as instrumentation predicates in the heap domain. Integer predicates are the basis of our integer abstraction, which divides up the integers into a finite set of classes. Then we describe cardinality functions, which were used to reason about reference counting in §3.1.5. Finally, we explain functions like *Consequences*$_2$, which allow us to integrate the integer domain with the combined domain.

4.2.1 Overview

The domain of difference-bound matrices is designed to analyze programs with simple, integer-valued variables. To avoid confusing these variables with the more complex ones used in PBJ programs, we call them *dimensions*. We give the dimensions names like x and y. The purpose of the domain is to infer constraints on the dimensions like $x \leq 3$ or $y \geq 7$ or $x - y \leq 12$. We write the constraints in the form of a matrix, which we denote as M. Consider the following matrix.

$$\begin{array}{c}
\begin{array}{ccc} 0 & x & y \end{array} \\
\begin{array}{c} 0 \\ x \\ y \end{array}
\begin{pmatrix}
0 & 0 & 5 \\
10 & 0 & 5 \\
10 & -5 & 0
\end{pmatrix}
\end{array}$$

We read the constraints out of this matrix. If the entry at the row labeled v_i and the column v_j is c_{ij}, then we have the constraint $v_i - v_j \leq c_{ij}$. For example, we have the following.

$$0 - y \leq 5 \qquad\qquad x - 0 \leq 10$$
$$x - y \leq 5 \qquad\qquad y - x \leq -5$$

The special 0 dimension allows us to express constraints of the form $v - 0 \leq c$ and $0 - v \leq c$, which means we can bound the range of a dimension. Using more convenient notation, the matrix above represents the constraints $x \in [0, 10] \wedge y \in [-5, 10] \wedge x = y + 5$.

Saturation. The most important operation in the integer domain is called *saturation*. This form of saturation should not be confused with the saturation performed by the combined domain (although they are similar). We saturate a matrix by adding constraints to it that are implied by existing constraints. For example, in the matrix above, we have the following two constraints.

$$x - 0 \leq 10 \qquad\qquad y - x \leq -5$$

When we sum these constraints together, we get the implied constraint $y - 0 \leq 5$. However, the entry in the matrix says only that $y - 0 \leq 10$. Whenever the implied constraint is stronger than the constraint in the matrix, we enter the implied constraint into the matrix, getting the following.

$$\begin{array}{c@{}c}
& \begin{array}{ccc} 0 & x & y \end{array} \\
\begin{array}{c} 0 \\ x \\ y \end{array} &
\left(\begin{array}{ccc}
0 & 0 & 5 \\
10 & 0 & 5 \\
5 & -5 & 0
\end{array}\right)
\end{array}$$

In general, the saturation procedure looks for entries $u - v \leq c$ and $v - w \leq c'$. It adds the implied entry $u - w \leq c + c'$ if it is stronger than the one already in the matrix.

This procedure very similar to the all-pairs shortest path problem in a graph. We treat the dimensions as nodes in a graph and the constraints as edges between them. For example, the original matrix has the following graph representation.
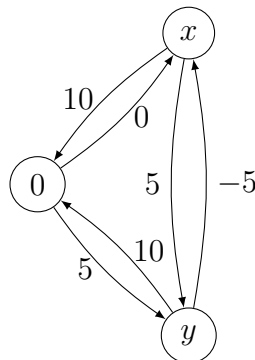
It's fairly clear that the shortest path from y to 0 in this graph has weight 5, which justifies putting that value in the matrix.

To fully saturate the graph, we use the Floyd-Warshall algorithm. It is a very simple cubic algorithm for computing all-pairs shortest paths.

function Floyd-Warshall(V, E):
    for $v \in V$:
        for $u \in V$:
            for $w \in V$:
                $E[u, w] := \min(E[u, w], E[u, v] + E[v, w])$

A nice benefit of this algorithm is that it can be used to detect negative-weight cycles. After the algorithm runs, we look for negative values along the diagonal entries of the matrix, which signal the presence of negative-weight cycles. In the context of our domain, a negative-weight cycle means that the set of constraints is unsatisfiable. In other words, an abstract element with a negative cycle is equivalent to $\perp$ because it does not abstract any concrete states.

Assignment. Saturation forms the basis of the assignment transfer function. Given a domain element, we want to approximate the effect of an assignment $x := e$. We do so in three steps. First, we saturate the element. Then we remove any existing information about the variable x. Any constraint of the form $x - v \leq c$ (or $v - x \leq c$) is replaced by $x - v \leq \infty$ (or $x - v \leq \infty$). The one exception is $x - x \leq 0$, which we leave alone. Then we add the new constraint $x = e$. We may have to perform some syntactic manipulation to get the new constraint in the right form. For example, if $e = y + 1$, then we convert $x = e$ into $x - y \leq 1 \wedge y - x \leq -1$.

Example 48 Suppose we have the original matrix above, before saturation. We perform the assignment $x := 20$. The first step is to saturate, which we have already explained. Next we remove x constraints, giving us the following matrix.

$$
\begin{array}{c}
\begin{array}{ccc}
 & 0 \quad\;\; x \quad\;\; y
\end{array} \\
\begin{array}{c} 0 \\ x \\ y \end{array}
\left(
\begin{array}{ccc}
0 & \infty & 5 \\
\infty & 0 & \infty \\
5 & \infty & 0
\end{array}
\right)
\end{array}
$$

Finally, we add the constraints $x - 0 \leq 20$ and $0 - x \leq -20$.

$$
\begin{array}{c}
\begin{array}{ccc}
 & 0 \quad\;\; x \quad\;\; y
\end{array} \\
\begin{array}{c} 0 \\ x \\ y \end{array}
\left(
\begin{array}{ccc}
0 & -20 & 5 \\
20 & 0 & \infty \\
5 & \infty & 0
\end{array}
\right)
\end{array}
$$

Note that if we had failed to saturate the matrix beforehand, we would have gotten the following matrix instead.

$$
\begin{array}{c}
 \\
0 \\
x \\
y
\end{array}
\begin{array}{ccc}
0 & x & y \\
\left(\begin{array}{ccc}
0 & -20 & 5 \\
20 & 0 & \infty \\
10 & \infty & 0
\end{array} \right)
\end{array}
$$

At this point, the fact that $y \leq 5$ has been irretrievably lost. No amount of saturation will recover it. As a consequence, we must always do our best to saturate the matrix before assignment. □

Partial order. The partial order for this domain is easy to compute. Given two domain elements E and E', our job is to see if the constraints in E imply the constraints in E' (because in that case, every state abstracted by E is abstracted by E', so $\gamma(E) \subseteq \gamma(E')$).

We begin by saturating E. The rationale is that to prove $E \Rightarrow E'$, we first want to make E as strong as possible. Next, we check to see if the constraints in E imply the constraints in E', component-wise. For example, if there is a constraint $x - y \leq c$ in E and $x - y \leq c'$ in E', we check that $c \leq c'$. Treating the domain elements as matrices, this amounts to checking that $E \leq E'$. If this holds, the partial order relationship is satisfied.

Join. To join elements E and E', we first saturate them both. Then we generate a new matrix E''. If E contains the constraint $x - y \leq c$ and E' contains $x - y \leq c'$, then we add $x - y \leq \max(c, c')$ to E''. Based on our partial order, this ensures that E'' is the strongest element implied by both E and E'.

4.2.2 Dimensions

A typical integer domain creates one dimension for every program variable. Since our variables are maps, we may need to use many dimensions. Consider the following code.

```
1 type T;
2 global val[T]:int;
```

The variable *val* is managed by the integer domain. Suppose the heap domain contains two nodes, n and $\boldsymbol{n'}$, of type T. Then we store the value of *val* across two dimensions: *val*[n] and *val*[$\boldsymbol{n'}$].

If we have a constraint *val*[n]$-0 \leq 10$, it is fairly easy to understand what this means. But for the summary node, the interpretation is more complex. The constraint *val*[$\boldsymbol{n'}$]$- 0 \leq 10$ is implicitly quantified. Logically, it means

$$\forall u \in \boldsymbol{n'}.\ val[u] - 0 \leq 10.$$

Each occurrence of a summary class (even the same class appearing more than once) introduces a new quantifier.

Besides dimensions for maps, we introduce some other dimensions. We have already described cardinality variables: for any class C, the dimension $\#C$ holds the cardinality of C. Note that cardinality dimensions do not introduce any implicit quantifiers.

For integer classes, we also create *value dimensions* written $[C]$. A value dimension describes the values of the integers contained in the class. For example, suppose class $C = \{5, 6, \ldots, 20\}$. Then the constraints $[C] - 0 \leq 20$ and $0 - [C] \leq -5$ are valid. These constraints are interpreted logically as follows.

$$\forall u \in C. \ u - 0 \leq 20 \qquad\qquad \forall u \in C. \ 0 - u \leq 5$$

Clearly these dimensions introduce implicit quantifiers.

We call dimensions of the form $f[C_1, \ldots, C_k]$ and of the form $[C]$ *quantified dimensions* since they introduce implicit quantifiers in constraints. They can be constrasted with cardinality dimensions.

Saturation. The use of implicit quantifiers in an integer domain is not new. It was first presented by Gopan et al. [22]. The crux of their work is that reasoning about quantified dimensions is no different than reasoning about program variables; they behave exactly the same. However, they did not consider the problem of classes that are potentially empty. Their analysis was designed to operate alongside TVLA, which normally does not support empty classes.

Unfortunately, classes that are potentially empty introduce problems in the saturation algorithm. As an example, suppose we have three functions, f, g, and h. Suppose that f and h take no arguments while g takes one argument. Let E be a domain element with a class $\boldsymbol{C}$ and the following constraints.

$$f - g[\boldsymbol{C}] \leq 0 \qquad\qquad g[\boldsymbol{C}] - h \leq 0$$

If we saturate this element, then we will infer a new constraint, $f - h \leq 0$. Unfortunately, this inference is invalid. To see why, consider the logical interpretation of these constraints.

$$\forall u \in \boldsymbol{C}. \ f - g[u] \leq 0 \qquad\qquad \forall u \in \boldsymbol{C}. \ g[u] - h \leq 0$$

The inferred constraint has no implicit quantifiers, so it is interpreted logically as $f - h \leq 0$. The problem occurs if $\boldsymbol{C}$ is empty. Then the two quantified constraints simplify to true, which does not imply $f - h \leq 0$.

The solution to this problem is to infer a fact only when it has more implicit quantifiers than the facts it was derived from. Given two constraints θ and π, suppose the Floyd-Warshall algorithm infers the (supposedly) implied constraint ω. For a given constraint, let the classes that we implicitly quantifier over be given by the function classdeps($\cdot$). Then it is safe to infer ω if

$$\text{classdeps}(\theta) \cup \text{classdeps}(\pi) \subseteq \text{classdeps}(\omega).$$

The classdeps function finds all classes occurring in the constraint except in dimensions of the form $\#C$, since these do not create implicit quantifiers.

$$\text{classdeps}(f[C_1, \ldots, C_k]) := \{C_1, \ldots, C_k\}$$
$$\text{classdeps}([C]) := \{C\}$$
$$\text{classdeps}(\#C) := \emptyset$$

We overload the function to operate over constraints by unioning the classdeps of each dimension in the constraint.

This solution above is sound (i.e., all inferred constraints are correct) but incomplete. A better solution would be to explicitly tag every constraint with a set of quantifiers. When inferring ω, as above, we would tag it with the union of the tags of the θ and π. Unfortunately, this technique has a problems. The existing integer domain has the useful property that given two constraints, $x - y \leq c$ and $x - y \leq c'$, one always dominates over the other. Thus, for each matrix entry, we only need to store a single number. If we tag constraints with quantifiers, we lose this property. Given the constraints $\forall u \in C. \ x - y \leq 0$ and $x - y \leq 10$, which do we keep? The two are incomparable: if C is empty, then the second is stronger; otherwise the first is stronger. Due to this problem, we do not take this approach. Although we are sacrificing precision for simplicity, we have never witnessed any imprecision in practice.

4.2.3 Predicates

The heap domain shares information about where its predicates hold with the integer domain. As we will see later, the integer domain uses this information for reasoning about the number of objects satisfying a given predicate. All the heap information comes through as facts, of the form $\forall x_1 \in C_1. \ \cdots \forall x_n \in C_n. \ \mathsf{P}(x_1, \ldots, x_n)$, to $Assume_2$. The integer domain records this information in much the same way as does in the heap domain: for each predicate, it stores an interpretation $\iota(\mathsf{P})(C_1, \ldots, C_k)$ that can be 0, 1/2, or 1.

We write an integer domain element as a tuple, $E = \langle U, \iota, M \rangle$. Like in the heap domain, U is a mapping from classes to their types and ι gives predicate interpretations. M is the difference matrix. We have no need for the ρ component of the heap domain because that information is stored via cardinality dimensions in M.

Integer predicates. As mentioned in Chapter 3, the integer domain can define its own predicates. They have the following form.

$$\text{formula} ::= \text{term} \leq \text{term} \mid \text{term} = \text{term} \mid \text{term} \geq \text{term}$$
$$\text{term} ::= \text{constant} \mid x \mid map[x_1, \ldots, x_n] \mid \#x$$

The x variables must be bound as arguments to the predicate. One purpose of integer predicates is to share information with the heap domain, as we saw in §3.1. Another purpose,

to be described soon, is to allow us to distinguish between integer classes using canonical abstraction.

Given a domain element, we can evaluate an integer predicate $\mathsf{P}(x_1, \ldots, x_k)$ somewhere by substituting classes for the variables x_i and then checking if the given constraint is implied by the element. Since predicates definitions do not contain quantifiers, evaluating them is easy and fast.

Example 49 Let $\mathsf{P}(x\text{:int}, y\text{:T}) := x < f[y]$. Suppose we want to evaluate $\mathsf{P}(C, C')$. Then we substitute the classes into the formula. Although we did not mention it above, a bare variable appearing in a formula is converted to a value dimension. So we get the constraint $[C] < f[C]$. Then we simply see if this constraint is implied by the difference matrix. □

Assignment. When an assignment takes place, integer predicates must be updated to account for the change. In the heap domain, we used finite differencing to update predicates. Since integer predicates do not contain quantifiers, updating them is much easier. We do a simple test to determine which predicates may have changed and where they might have changed; we re-evaluate them at these places.

Example 50 Consider the assignment $f[\mathsf{x}] := 0$, where x is translated by the heap domain to class C. In the $Assign_2$ function, we perform the assignment to the dimension $f[C]$ as described in the overview. Suppose that $\mathsf{P}(x) := f[x] \leq 0$. In $PostAssign_2$, we will need to update P. To do so, we look inside the predicate definition and see that it depends on $f[x]$, which is updated by the assignment when $x = C$. Therefore, we need to re-evaluate $\mathsf{P}(C)$ after the assignment. We do so by checking if $f[C] \leq 0$ is implied by the new difference matrix. It is, so we set $\iota(\mathsf{P})(C) = 1$.

Note that if $f[C] \leq 0$ had not been implied, we would have checked if its negation, $f[C] > 0$, was implied. If it had been, we would have set $\iota(\mathsf{P})(C) = 0$. Otherwise we would have set $\iota(\mathsf{P})(C) = 1/2$ (since we know that $f[C]$ changes somehow). Based on the changes to $\iota(\mathsf{P})$, $PostAssign_2$ would update the U and C sets to reflect the change to P. □

Sharpening. Just as the heap domain has a sharpening operation, the integer domain has one as well. Sharpening runs at the same time as heap domain sharpening. We illustrate sharpening with an example.

Example 51 Suppose that $\mathsf{P}(x) := f[x] \leq 0$. To sharpen P, we iterate over every class C in the predicate's domain. Suppose that $\iota(\mathsf{P})(C) = 1$. Then we add the constraint $f[C] \leq 0$ to the difference M. On the other hand, if $\iota(\mathsf{P})(C) = 0$, then we add the negation, $f[C] > 0$.

We can also go in the other direction. We check if $f[C] \leq 0$ is implied by the difference matrix. If it is, then we set $\iota(\mathsf{P})(C) = 1$. If $f[C] > 0$ is implied, then we set $\iota(\mathsf{P})(C) = 0$. In both these cases, we can share the new information about P with the heap domain (via the $Consequences_2$ function). □

There is one problem with the discussion above. Consider the following example.

Example 52 Suppose now that $\mathsf{P}(x) := f[x] \leq g[x]$. Let $\iota(\mathsf{P})(\boldsymbol{C}) = 1$ for some summary class $\boldsymbol{C}$. Following the development above, we should add $f[\boldsymbol{C}] \leq g[\boldsymbol{C}]$ to the difference matrix M. However, this is incorrect. The logical interpretation of $\iota(\mathsf{P})(\boldsymbol{C}) = 1$ is

$$\forall x \in \boldsymbol{C}.\ \mathsf{P}(x) \quad \equiv \quad \forall x \in \boldsymbol{C}.\ f[x] \leq g[x].$$

On the other hand, the logical interpretation of $f[\boldsymbol{C}] \leq g[\boldsymbol{C}]$ is

$$\forall x \in \boldsymbol{C}.\ \forall y \in \boldsymbol{C}.\ f[x] \leq g[y].$$

The second fact is stronger than the first. Therefore, it is incorrect to add the second fact based on the first fact. □

To fix this problem, we only perform the first kind of sharpening (adding a constraint to M based on a fact in ι) if all the classes involved are singletons. Technically, we could use a weaker restriction: that all the summary classes involved appear exactly once in each constraint in the definition of P. However, we have not encountered any imprecision resulting from the simpler restriction.

Note that the second kind of sharpening (adding a fact to ι based on information in M) is always valid because the information in M is "stronger" than the information in ι, as the example shows.

4.2.4 Class Abstraction

We described integer classes in §3.1. This section explains how they work. Unlike the heap domain, integer classes need not be disjoint. We group them into partitionings. Classes in the same partitioning are necessarily disjoint, while classes from different partitionings may overlap.

The default partitioning contains one class that holds all the integers. We always keep this partitioning around to ensure that that every integer belongs to some class. The user is responsible for defining other partitioning via program annotations.

Within a given partitioning, we use the canonical abstraction to distinguish integer classes. Two classes are merged if they satisfy the same abstraction predicates. Otherwise they are kept separate. The feature that distinguishes the integer domain abstraction from the heap abstraction is that each partitioning has its own set of abstraction predicates. When the user defines a new partitioning, he specifies the set of abstraction predicates for it.

Integer classes are typically used for reasoning about arrays. An integer class represents a range of array indexes. Consider a loop iterating over an array. The purpose of the loop is usually to establish some property at each element. It is typically beneficial to break up the elements into three groups: the elements before the current one, the current element, and the elements after the current element. Then the first group of elements all satisfy

the property and the last group all do not. The current element is kept materialized in a singleton class to guarantee strong updates.

Suppose that we use the variable i to loop over array elements. To organize the classes according to the intuition above, we define three integer predicates.

$$\mathsf{P}_<(x{:}\mathtt{int}) := x < i$$
$$\mathsf{P}_=(x{:}\mathtt{int}) := x = i$$
$$\mathsf{P}_>(x{:}\mathtt{int}) := x > i$$

These three predicates will keep the three groups of integers above in separate classes. We can also use additional predicates to segregate integers less than zero and greater than the size of the array.

We typically use one partitioning for each index variable. Suppose that a program contains two arrays. The first array is accessed via the index variable i and the other array by the variable j. We would define two partitionings. The first one would use the three predicates above as abstraction predicates. The other partitioning would use similar predicates, but with j in place of i. That way, an analysis would use six classes: three for the first array and three for the second.

Note that without support for overlapping classes, we would have to use a single abstraction for both i and j. Often we won't know whether i or j is larger, so we would have to introduce many spurious disjunctions to deal with all the possible orderings. Although overlapping classes add a lot of complexity to our analysis, analyzing arrays without them would be impractical.

Splitting classes. Just as the heap domain has a focus operation to materialize a summary node, the integer domain has an operation called *split*. Given an integer class C and a specific integer x, its job is to split C into three classes: $C_<$, $C_=$, and $C_>$. The first class contains the integers from C that are less than x. The second contains those equal to x, and the third contains those greater. Any one of these classes may be empty.

The user is required to annotate the places in the program where a split should happen. Here is an example.

```
1  procedure init(n:int)
2    array[int]:int;
3    i:int;
4
5    predicate Ppos(x:int) := x >= 0;
6    predicate Plt(x:int) := x < i;
7    predicate Peq(x:int) := x = i;
8    predicate Pgt(x:int) := x > i;
9    partitioning PartitionI = Plt, Peq, Pgt;
10 {
11   i := 0;
```

```
12    while (i < n) {
13      a[i] := 0;
14      split(PartitionI, i+1);
15      i := i+1;
16    }
17  }
```

Before the split on line 14, there are four integer classes in the `PartitionI` partition. We call them C_1, C_3, C_3, and C_4. We use value dimensions to track the integers contained in these classes (e.g., $[C_3] = i$). We also record facts about the array values (e.g., $a[C_2] = 0$). Finally, we store predicate values for Ppos and the like in ι. The following diagram shows everything.

$[C_1] < 0$	$0 \leq [C_2] < i$ Ppos, Plt $a[C_2] = 0$	$[C_3] = i$ Ppos, Peq $a[C_3] = 0$	$i < [C_4]$ Ppos, Pgt

The split annotation causes us to split each class C in `PartitionI` into three separate classes, $(C)_<$, $(C)_=$, and $(C)_>$. One class contains the elements of the original class less than $i+1$, another contains the elements equal to $i+1$, and the last contains the elements greater than $i+1$. Theoretically, we would get 4×3 classes total. However, all the elements of C_1 are less than $i+1$. Therefore $(C_1)_< = C_1$ while $(C_1)_=$ and $(C_1)_>$ are empty. We don't bother to create the empty classes, so essentially no splitting of C_1 takes place. The same thing holds true for C_2 and C_3. At C_4, no elements are less than $i+1$, but there is one element equal to $i+1$ and the others are greater. So out of the 12 possible classes, only 5 are non-empty. After the split we get the following.

$[C_1] < 0$	$0 \leq [C_2] < i$ Ppos, Plt $a[C_2] = 0$	$[C_3] = i$ Ppos, Peq $a[C_3] = 0$	$[C_4'] = i+1$ Ppos, Pgt	$i+1 < [C_4'']$ Ppos, Pgt

The last two classes, C_4' and C_4'', have the same canonical name, but we do not apply the canonical abstraction after the split operation. When we analyze the $i := i+1$ statement, we must update the constraints involving i and the predicates as follows.

$[C_1] < 0$	$0 \leq [C_2] < i-1$ Ppos, Plt $a[C_2] = 0$	$[C_3] = i-1$ Ppos, Plt $a[C_3] = 0$	$[C_4'] = i$ Ppos, Peq	$i < [C_4'']$ Ppos, Pgt

At this point we apply the canonical abstraction. Since C_2 and C_3 now have the same canonical name, we merge them into a single node, getting the following.

$[C_1] < 0$	$0 \le [C_2'] < i$ $\mathsf{Ppos}, \mathsf{Plt}$ $a[C_2] = 0$	$[C_4'] = i$ $\mathsf{Ppos}, \mathsf{Peq}$	$i < [C_4'']$ $\mathsf{Ppos}, \mathsf{Pgt}$

If we execute the assignment $a[i] := 0$ on this structure, we get back the one we started with. Thus, we have found a loop invariant.

Future work. The split operation is quite similar to the focus operation used in the heap domain. Its job is to materialize a summary node before it is used by the analysis. However, the focus operation is simpler because there is no need to worry about multiple partitionings. Focus operations can almost always be inferred syntactically; in the `thttpd` code, no focus annotations are necessary. Currently we do not infer split operations, so they must be specified manually by the user.

Eventually, we would like to move away from the canonical abstraction for the integer domain. All of the abstraction predicates we use currently are of the form "$x < i$," or some variation. It would be much easier to define a partitioning by specifying the "dividing lines" between classes. For example, the partitioning above would be written as "$0 \mid i \mid i + 1$." Then the analysis would automatically perform a split operation on a partitioning whenever one of its variables was updated. This would eliminate the need for writing the boilerplate predicates like Peq above as well as the split annotations.

Translate. We are required to supply a *Translate₂* function to the combined domain so that it can translate integer terms into classes that are understood by the heap domain. Our implementation is very simple. Given an integer term t, we search through all integer classes C, looking for a class where $[C] = t$ is implied by the difference matrix. If we find one, we return it. Otherwise we return the default class containing all integers.

4.2.5 Cardinality Functions

This section describes cardinality instrumentation functions. These are like instrumentation predicates except that they evaluate to an integer instead of a boolean. Here is an example.

$$F[x{:}\mathsf{S}] := |\{y{:}\mathsf{T} : \mathsf{E}(x, y)\}|$$

Given an S object, this function counts up the number of outgoing E edges to T objects. In general, a cardinality function can take any number of parameters $x_1, \ldots, x_k$. The summation body is fairly restricted. It must have the form $|\{x_{k+1}{:}\mathsf{t} : \mathsf{P}(\cdots)\}|$, where each argument to P must be drawn from $x_1, \ldots, x_{k+1}$.

We use integer dimensions to record the values of an instrumentation function. For the predicate above, we might have dimensions $F[C_1]$ and $F[C_2]$, where C_1 and C_2 are classes of type S. Instrumentation functions play a role in the analysis in two ways. We define a

new form of sharpening that links their values with class sizes and predicate values. We also augment the assignment function to update instrumentation functions when the predicates they depend on have changed.

Sharpening. There are two forms of sharpening for instrumentation functions. The following example illustrates the first form.

Example 53 Suppose we have the function above, $F[x\!:\!\mathsf{S}] := |\{y\!:\!\mathsf{T} : \mathsf{E}(x,y)\}|$. Consider the following heap structure. Classes of type S are on top and classes of type T are on the bottom. Edges show the E predicate.
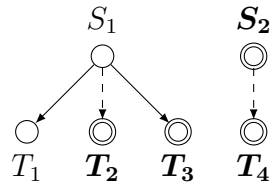
If we are asked to sharpen the F instrumentation function, we can add several constraints to the difference matrix. At node S_1, observe the classes satisfying $\mathsf{E}(S_1, \cdot)$. Classes T_1 and $\boldsymbol{T_3}$ definitely satisfy E while $\boldsymbol{T_2}$ maybe satisfies E. Therefore, the number of edges out of S_1 is $\#T_1$ plus $\#\boldsymbol{T_3}$ plus, possibly, $\#\boldsymbol{T_2}$. We state this as follows.

$$\#T_1 + \#\boldsymbol{T_3} \leq F[S_1] \leq \#T_1 + \#\boldsymbol{T_2} + \#\boldsymbol{T_3}.$$

We would like to add these constraints to the difference matrix. Unfortunately, we cannot represent them directly. However, we know that $\#T_1 = 1$. Therefore, we can at least add $\#\boldsymbol{T_3} - F[S_1] \leq -1$.

We also sharpen at $\boldsymbol{S_2}$. We add the following constraints.

$$0 \leq F[\boldsymbol{S_2}] \leq \#\boldsymbol{T_4}$$

We can also sharpen in the other direction. Suppose the difference matrix implies the constraint $F[S_1] \leq \#T_1 + \#\boldsymbol{T_3}$. In that case, we can sharpen $\mathsf{E}(S_1, \boldsymbol{T_2})$ to zero. □

The general technique is as follows. We iterate over classes that match the signature of F (S_1 and $\boldsymbol{S_2}$ above). For each one, we compute two sums: the number of individuals that definitely satisfy the predicate and the number that may satisfy the predicate. One complication here is that we must compute one pair of sums for each partitioning, since classes from different partitionings may not be disjoint and so we cannot add their sizes. Once we have the two sums, for any given partitioning, we add constraints as above.

Assignment. All of the work of updating instrumentation functions is handled by *PostAssign$_2$*. When it is time to update a function like F, it checks U and C, counting up the number of places where E has changed, and increments or decrements F by that amount. The following example illustrates the process best.

Example 54 Consider same situation as the previous example, with the same heap structure and $F[x{:}\mathsf{S}] := |\{y{:}\mathsf{T} : \mathsf{E}(x,y)\}|$.
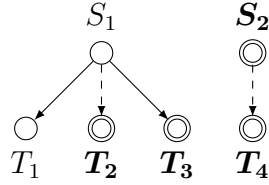
Suppose that $\mathsf{E}(S_1, T_1)$ goes down. This is transmitted to the integer domain as follows.

$$U = \{(\forall s \in S_1.\ \forall t \in T_1.\ \neg\mathsf{E}(s,t))\}$$
$$C = \{\mathsf{E}(S_1, T_1)\}$$

We compute the changes to $F[S_1]$ and $F[\boldsymbol{S_2}]$ separately. For each one, we add up the "size" of the change, called Σ. We do so by iterating over all T classes C. If E goes up at that class, we add $\#C$ to the change size. If it goes down, we add $-\#C$. In the case of $F[S_1]$, $\Sigma = -\#T_1$. Then we update the function as if the assignment $F[S_1] := F[S_1] + \Sigma$ had taken place. In this case, $F[S_1] := F[S_1] - \#T_1$.

In practice, we can only handle the change precisely if Σ is a constant. Luckily, since the change typically happens at a materialized node, the change size is always 1 or -1, as it is here. □

As in sharpening, when we add up the size of the change, we only consider the classes in a given partitioning. Out of all the partitionings, we choose the one that gives us the most precise estimate of the change size.

4.2.6 Consequences

The *Consequences$_2$* function exposes a number of pieces of information to the combined domain.

- For each class C, it tries to find bounds on $\#C$. If $\#C \leq 1$, then it returns the fact $\forall n \in C.\ \forall n' \in C.\ n = n'$. If $\#C \geq 1$, it returns $\exists n \in C.\ n = n$. $\#C = 0$, then it returns $\forall n \in C.\ n \neq n$.

- For each partitioning, it returns the set of classes in that partitioning as the fact $\mathtt{int} : [\ C_1 \ | \ \cdots \ | \ C_n\ ]$.

- For any two classes in different partitionings, it uses value dimensions to see if they might be disjoint. If either $[C_1] < [C_2]$ or $[C_2] < [C_1]$ is implied by the difference matrix, then we return the fact $\forall n \in C_1.\ \forall n' \in C_2.\ n \neq n'$.

4.2.7 Partial Order and Join

Partial order. The partial order for the integer domain needs a few modifications in the context of the combined domain. First, since domain elements now carry around predicate information, we use the same partial order check on ι as is used in the heap domain. Second, we make a few modifications to the partial order check on the difference matrix.

Recall that the standard difference-bound matrices domain, on inputs M^A and M^B, saturates M^A and then checks if its entries are $\leq$ the entries of M^B. This check remains correct, but we make one change to deal with classes of size 0. In pseudocode, we write the following. Recall the classdeps function defined in §4.2.2.

for $x, y \in$ dimensions:
    $c^A := M^A[x, y]$
    $c^B := M^B[x, y]$
    $D := \text{classdeps}(x) \cup \text{classdeps}(y)$
    if $(c^A \leq c^B) \vee (\exists C \in D.\ M^A \Rightarrow \#C = 0)$:
        continue
    else:
        return false
return true

The second disjunct is the crucial one. If any classes mentioned in the constraint are empty in E^A, then the constraint simplifies to true in E^A because of the implicit quantifiers. Therefore, we can assume in E^A that $x - y \leq c$ for any c, and so certainly it is true that $x - y \leq c^B$, which is what we require for the partial order to hold.

Example 55 Suppose we have an element E^A where $\#C = 0$. We also have an element E^B where $f[C] - 0 \leq 100$. According to the algorithm above $E^A \sqsubseteq E^B$. Even though E^A does not directly imply $f[C] - 0 \leq 100$, the fact that class C is empty in E^A means that any fact about elements of C is indirectly implied by E^A. In particular, the fact $f[C] - 0 \leq 100$ from E^B is implied. □

Join. The join algorithm uses a similar test except that it is more symmetric. We show the pseudocode below; the output matrix is M.

for $x, y \in$ dimensions:
    $c^A := M^A[x, y]$
    $c^B := M^B[x, y]$
    $D := \text{classdeps}(x) \cup \text{classdeps}(y)$
    if $\exists C \in D.\ M^A \Rightarrow \#C = 0$ **then** $c := c^B$
    else if $\exists C \in D.\ M^B \Rightarrow \#C = 0$ **then** $c := c^A$
    else $c := \max(c^A, c^B)$
    $M[x, y] := c$
return M

4.2.8   Repartitioning

This operation is the most complicated of the integer domain. When we repartition classes, we need to deal with two issues:

- Dimensions of the form $\#C$ must be updated to deal with the new class sizes.

- Dimensions of the form $f[C_1, \ldots, C_n]$ and $[C]$ must be rewritten to use the new class names.

Since both steps are complicated, we handle them independently. The first repartitioning step fixes up the cardinality dimensions and the second step rewrites class names in quantified dimensions. An additional step is to rewrite the predicate values in ι using the same technique as we did for the heap domain.

Recall that the $Repartition_2$ function is called with two arguments. The first, a relation R, describes how "original classes" are rewritten to "new classes." It is a set of tuples of the form (C, C'), where C is an original class and C' is a new class. The other argument is a set of facts F. It gives cardinality and disjointness information about the new classes.

In the rest of the section, we assume that R preserves partitionings. That is, if C_1 and C_2 are original classes belonging to the same partitioning, and if $(C_1, C_1') \in R$ and $(C_2, C_2') \in R$, then C_1' and C_2' must be in the same partitioning. We make the same requirement in the other direction: if C_1' and C_2' are new classes from the same partitioning, and if $(C_1, C_1') \in R$ and $(C_2, C_2') \in R$, then C_1 and C_2 must be in the same partitioning. Both the heap domain and the integer domain preserve these properties.

Cardinality dimensions.   The first step is to rewrite cardinality dimensions based on the repartitioning R and the facts F. Consider the following example.

Example 56 Suppose that $R = \{(C_1, C_1'), (C_2, C_1'), (C_3, C_2'), (C_3, C_3')\}$. We draw this below.
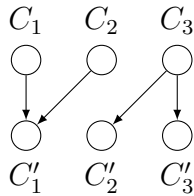
Suppose that all the original classes are in the same partitioning. Also suppose that $\#C_2 = 1$. Let $F = \{(\forall x \in C_2'.\ \forall y \in C_2'.\ x = y), (\exists x \in C_2'.\ x = x), \mathtt{int} : [\ C_1' \mid C_2' \mid C_3'\ ]\}$.

Using F and R, we can generate a set of constraints that describe the new cardinality dimensions. From F, we get that $\#C_2' = 1$. From R, we get a more complex set of constraints. First we note that every individual in C_1 or C_2 (which are disjoint) must flow into C_1'. Therefore, $\#C_1 + \#C_2 \leq \#C_1'$. By the same reasoning, $\#C_3 \leq \#C_2' + \#C_3'$.

We can go in the other direction as well. Every individual in C_1' must come from C_1 or C_2, so $\#C_1' \leq \#C_1 + \#C_2$. And every individual in C_2' and C_3' must come from C_3, so $\#C_2' + \#C_3' \leq \#C_3$.

Putting these four constraints together, we get:

$$\#C_1' = \#C_1 + \#C_2$$
$$\#C_2' + \#C_3' = \#C_3$$
$$\#C_2 = \#C_2' = 1$$

We add these constraints to M, saturate, and then eliminate all constraints involving the original cardinality dimensions. In the example, this amounts to replacing occurrences of $\#C_1$ with $\#C_1' - 1$ and replacing $\#C_3$ with $\#C_3' + 1$. Now we are done repartitioning the cardinality dimensions. □

The general technique presented in the example is to add constraints based on R and F, saturate, and then eliminate occurrences of the old dimensions.

The difficult part is adding constraints based on R. We proceed in two steps. First, for any subset S' of new classes, we find all the original classes S that they might derive from. For example, if $S = \{C_1'\}$, we get $S' = \{C_1, C_2\}$. If $S' = \{C_2'\}$, we get $S = \{C_3\}$. If all the classes in S' are mutually disjoint, we add the following constraint:

$$\sum_{C' \in S'} \#C' \leq \sum_{C \in S} \#C.$$

This is because every individual in any S' class must come from an S class.

Similarly, if we let S be any subset of original classes, and let S' be all the classes that their elements flow into, and if the classes in S are mutually disjoint, then we add the following inequality.

$$\sum_{C \in S} \#C \leq \sum_{C' \in S'} \#C'$$

This is because every individual from an original class in S must flow into some class in S'.

Once these constraints are added, we can saturate and eliminate the old dimensions.

Quantified dimensions. Now we have an element where cardinality dimensions have been rewritten. We must deal with the other dimensions.

Example 57 Suppose we have an element with two classes, where $f[C_x] \leq 10$ and $f[C_y] \leq 20$. That is, $M[f[C_x], 0] = 10$ and $M[f[C_y], 0] = 20$. We repartition this element with $R = \{(C_x, C'), (C_y, C')\}$, thereby merging the two classes into C. Then we want to compute $M'[f[C'], 0]$.

The basic process is to find all the original constraints of the form $f[C] - 0 \leq c$, where $(C, C') \in R$. Out of all the bounds c, we take the maximum to be conservative. In this case, since $M[f[C_x], 0] = 10$ and $M[f[C_y], 0] = 20$, we get the values 10 and 20 for c. Their max is 20, so we set $M'[f[C'], 0] = 20$. □

The following equation generalizes this process. When computing $M'[x', y']$, we use classdeps to find the quantified classes in x' and y' and then we search for all the original classes that might flow into them. We form original dimensions using these classes and take the max over all the matrix bounds there.

$$M'[x', y'] := \max_{C_1:(C_1, C_1') \in R} \cdots \max_{C_n:(C_n, C_n') \in R}$$
$$M[x'[C_1' \mapsto C_1, \ldots, C_n' \mapsto C_n], y'[C_1' \mapsto C_1, \ldots, C_n' \mapsto C_n]]$$
$$(\text{where classdeps}(x') \cup \text{classdeps}(y') = \{C_1', \ldots, C_n'\})$$

Notice the similarity between this equation and the one for repartitioning quantified facts in the heap domain.

4.2.9 Related Work

The work by Gopan et al. [22] shows how to handle quantified dimensions in a sound way. However, it omits discussion of empty classes, which introduces additional complexity as we have described. It describes the repartitioning operation for quantified dimensions, although its explanation is different. It considers only a few special cases for the relation R; however, it explains the correct semantics for many possible integer domains, while we only explain difference-bound matrices.

Chapter 5

Experiments

We have applied DESKCHECK to the cache module of the `thttpd` web server [39]. We chose this data structure because it relies on several invariants that require combined integer and heap reasoning. We believe this data structure is representative of many that appear in systems code, where arrays, lists, and trees are all used in a single composite data structure, sometimes with reference counting used to manage deallocation.

The `thttpd` cache maps files on disk to their contents in memory. Figure 5.1 displays an example of the structure. It is a composite between a hash table and a linked list. The linked list of cache entries starts at the *maps* variable and continues through *next* pointers. These same cache entries are also pointed to by elements of the *table* array. The *rc* field records the number of incoming pointers from external objects, represented by circles. This reference count is allowed to be zero.

Invariants

Figure 5.2 shows excerpts of the C code to add an entry to the cache. Besides the data structures already discussed, the variable *free_maps* is used to track unused cache entries (to
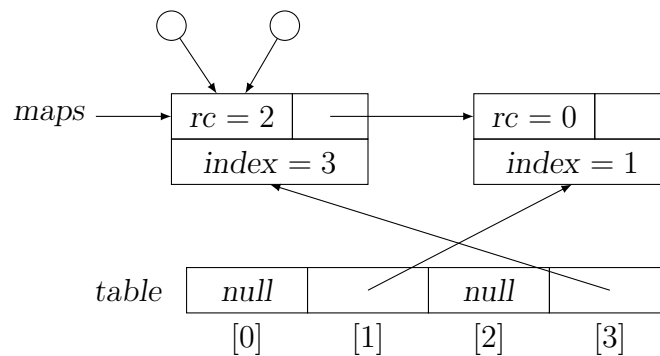

Figure 5.1: `thttpd`'s cache data structure.

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

Figure 5.2: Excerpts of the `thttpd` `map` and `add_hash` functions.

avoid calling `malloc` and `free`). Our goal is to verify that this code, as well as the related code for releasing and freeing cache entries, is memory-safe. One obvious data-structure invariant is that *maps* and *free_maps* should point to acyclic singly-linked lists of cache entries. However, there are two other invariants that are more complex but required for memory safety.

Inv1: When a cache entry e is freed, `thttpd` nulls out its table entry via `table[e.index] = null`. If the wrong element were overwritten, then a pointer to the freed entry would remain in *table*, later leading to a segfault when accessed. **Inv1** guarantees that if $table[i] = e$, where e is the element being freed, then $e.index = i$, so the correct entry will be set to null. §3.1.4 shows a small example of how this invariant is proved.

Inv2: This invariant relates to reference counting. The two main entry points to the cache module are called `map` and `unmap`. The `map` call creates a cache entry if it does not already exist and returns it to the caller. The caller can use the entry until it calls `unmap`. The cache keeps a reference count of the number of outstanding uses of each entry; when the count reaches zero, it is legal (although not necessary) to free the entry. Outstanding references are shown as circles in Figure 5.1. The cache must maintain the invariant that the number of outstanding references is equal to the value of an entry's reference count (rc) field—otherwise an entry could be freed while still in use. §3.1.5 shows an example of how this invariant is proved.

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

Table 5.1: Analysis times of `thttpd` analysis.

Verification

The cache library has four entry points. The `map` procedure checks if a file is already in the cache. If not, it loads the file from disk into a new cache entry. It returns a reference to the entry, incrementing its reference count. The `unmap` procedure is called when the server is finished with the cache entry. The entry's reference count is decremented but it remains in the cache. The `cleanup` procedure frees the data associated with entries whose reference count is zero and adds the entries to the free list. It may leave allow some recently used entries to remain if there is sufficient memory. In low memory situations, it frees entries on the free list. Finally, the `destroy` procedure is called when the server is shut down. It frees all reachable data.

This functionality corresponds to 531 lines of C code, or 387 lines of PBJ. The translation from C to PBJ was done manually. The PBJ code is shorter because it elides the system calls for opening files and reading them into memory; instead, it simply allocates a buffer to hold the data. It also omits logging code and comments. We show the PBJ code on 160.

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

We verify the cache in the context of a client test harness (see the `client` procedure starting at line 73). This client keeps a linked list of simulated HTTP connections. Each connection stores a pointer to data retrieved from the cache. In a loop, the client calls either `map`, `unmap`, or `cleanup`. When the loop terminates, it calls `destroy`. At any time, many connections may share the same data.

Table 5.1 shows the performance of the analysis. The total at the bottom is slightly larger than the sum of the entry-point times because it includes analysis of the client code as well. Information about the number of disjunctions used at each program was already presented in §2.6.4.

We analyze procedure calls by inlining them. This is possible only because there is no recursion. Since some procedures are called in multiple contexts, we analyze some code many times. Finding a more efficient way to handle procedures is the key to finding a scalable analysis. As the program becomes larger, inlining ceases to be a practical strategy.

Invariant type	Number of predicates
Linked list properties	11
Buffer reachability and sharing	2
Integer partitioning	3
Inv1	2
Inv2	2
Total	20

Table 5.2: Breakdown of predicates used in `thttpd` analysis.

We review the code, which starts on page 160. First we define maps corresponding to the fields and global variables for the cache. The linked list headed by `connections` contains the simulated HTTP connections; it is used by the test harness and is not part of the cache data structure itself.

Next we define a series of global predicates starting on line 25. Each one is annotated to say whether it is a heap domain predicate or a numeric domain predicate. Some predicates are annotated as being abstraction predicates. We use `all`, `ex`, and `tc` to expression ∀, ∃, and transitive closure. We summarize the predicates in Table 5.2.

Most of the global predicates (11 of 20) are used to define typical linked list properties for the `maps`, `free_maps`, and `connections` lists. These predicates are essentially "boilerplate;" their definitions could be shortened significantly or even eliminated. The buffer predicates (`BufferReach` and `BufferShared`) essentially say that each buffer object is accessible from exactly one map object; these two are boilerplate. The `Nlo`, `Nok`, and `Nhi` define a default integer abstraction for the `hash_table` array; these three could be eliminated by moving away from the canonical abstraction in the integer domain.

The interesting predicates are the ones for Inv1 and Inv2. We have already described them both in §3.1.4 and §3.1.5. We need two predicates to express Inv1. For Inv2, we use two predicates and an instrumentation function in the integer domain.

Most of the code, which starts in line 133, is straightforward. In a few procedures, we define local predicates and local partitionings. Most of these are to maintain the array abstraction. As we discussed in §4.2.4, moving away from canonical abstraction would eliminate the need for most of these. We also define a few local reachability predicates in functions that traverse linked lists. We suspect these could be inferred by searching for data structure traversals syntactically.

There are two kinds of annotations in the code itself. We have already discussed `split` annotations (§4.2.4), of which there are 14. Moving away from canonical abstraction would eliminate the need for these. The `enable` and `disable` annotations, of which there are 10, add or remove a predicate from the set of abstraction predicates, as discussed in §2.6.3. It would not be difficult to infer these annotations syntactically (adding them around blocks of code that traverse a given data structure).

We also note two places where we had to change the code. The `check_hash_size` proce-

dure is responsible for resizing the hash table if it is too small. The C code expands the hash table by doubling its size. Our integer analysis cannot reason precisely about the assignment `hash_size := hash_size*2`, so we changed it to the simpler `hash_size := hash_size + 100`.

Additionally, the hash table uses a bit mask operation to compute its hash function. Our integer domain does not understand bit masking, so we changed the operation to a modulus, of which it has limited understand.

Conclusion

To sum up, we are able to prove all the invariants about `thttpd` listed in Chapter 1. The time to analyze the program is not unreasonable and the annotation burden is fairly low. Given the expressive power of the analysis—its ability to prove the memory safety of a complex data structure—we feel that the costs are worth the benefit.

5.1 The Code

```
1  type Map;
2  type Buffer;
3  type Conn;
4
5  /* Field declarations */
6  global Map_key[Map]:int;
7  global Map_refcount[Map]:int;
8  global Map_addr[Map]:Buffer;
9  global Map_hash_idx[Map]:int;
10 global Map_next[Map]:Map;
11
12 /* Global variables */
13 global maps:Map;
14 global free_maps:Map;
15 global hash_table[int]:Map;
16 global hash_size:int;
17
18 /* Fields and local variables for client */
19 global Conn_next[Conn]:Conn;
20 global Conn_key[Conn]:int;
21 global Conn_addr[Conn]:Buffer;
22
23 global connections:Conn;
24
25 /*** Beginning of predicate declarations ***/
26
27 /* Used for array abstraction */
28 predicate(numeric, abstraction) Nlo(x:int) = x < 0;
29 predicate(numeric, abstraction) Nok(x:int) = x >= 0 && x < hash_size;
30 predicate(numeric, abstraction) Nhi(x:int) = x >= hash_size;
31
32 /* These constrain the linked list of connections */
33 predicate(heap) ConnTC(n1:Conn, n2:Conn) = tc(n1, n2) Conn_next;
34 predicate(heap, abstraction) ConnReach(n:Conn) =
35   n = connections || ConnTC(connections, n);
36 predicate(heap) ConnShared1(n:Conn) =
37   ex(n1:Conn) ex(n2:Conn) Conn_next[n1] = n && Conn_next[n2] = n && n1 != n2;
38 predicate(heap) ConnShared2(n:Conn) =
39   connections = n && ex(n2:Conn) Conn_next[n2] = n;
40 predicate(numeric) ConnPosKey(n:Conn) = Conn_key[n] >= 0;
41
42 /* These constrain the linked lists of maps (free_maps and maps) */
```

```
43  predicate(heap) MapTC(n1:Map, n2:Map) = tc(n1, n2) Map_next;
44  predicate(heap, abstraction) MapReach(n:Map) = n = maps || MapTC(maps, n);
45  predicate(heap, abstraction) MapReachFree(n:Map) =
46    n = free_maps || MapTC(free_maps, n);
47  predicate(heap) MapShared1(n:Map) =
48    ex(n1:Map) ex(n2:Map) Map_next[n1] = n && Map_next[n2] = n && n1 != n2;
49  predicate(heap) MapShared2(n:Map) = maps = n && ex(n2:Map) Map_next[n2] = n;
50  predicate(heap) MapShared3(n:Map) =
51    free_maps = n && ex(n2:Map) Map_next[n2] = n;
52
53  /* These constrain the buffers of data that maps point to */
54  predicate(heap) BufferReach(b:Buffer) =
55    ex(n:Map) Map_addr[n] = b;
56  predicate(heap) BufferShared(b:Buffer) =
57    ex(n1:Map) ex(n2:Map) Map_addr[n1] = b && Map_addr[n2] = b && n1 != n2;
58
59  /* Predicates for Inv1 */
60  predicate(numeric) MapHasIdx(n:Map, i:int) = Map_hash_idx[n] = i;
61  predicate(heap) MapIndexGood(i:int) =
62    all(n:Map) hash_table[i] != n || MapHasIdx(n, i) || n = null;
63
64  /* Predicates for Inv2 */
65  predicate(heap) MapConnMatch(c:Conn, m:Map) =
66    Map_addr[m] = Conn_addr[c] && Map_addr[m] != null;
67  function(numeric) MapRC(n:Map) = card(c:Conn) MapConnMatch(c, n);
68  predicate(numeric, abstraction) MapRCGood(n:Map) = MapRC[n] = Map_refcount[n];
69
70  /* Partitioning for array abstraction */
71  partitioning(numeric) Phash = Nlo, Nok, Nhi;
72
73  /*** Beginning of client test harness ***/
74
75  procedure client():int
76    key:int;
77  {
78    hash_size := 0;
79    maps := null;
80    free_maps := null;
81    connections := null;
82
83    while (*) {
84      if (*) {
85        havoc key : key >= 0;
86        open_connection(key);
```

```
 87      }
 88      if (*) {
 89        close_connection();
 90      }
 91      if (*) {
 92        mmc_cleanup();
 93      }
 94    }
 95    mmc_destroy();
 96    return 0;
 97  }
 98
 99  procedure open_connection(key:int):int
100    c:Conn;
101    addr:Buffer;
102  {
103    addr := mmc_map(key);
104    c := new Conn;
105    Conn_key[c] := key;
106    Conn_addr[c] := addr;
107    Conn_next[c] := connections;
108    connections := c;
109    return 0;
110  }
111
112  procedure close_connection():int
113    b:Buffer;
114    c:Conn;
115    tmp:Conn;
116  {
117    c := connections;
118    if (c != null) {
119      tmp := Conn_next[c];
120      connections := Conn_next[c];
121      Conn_next[c] := null;
122      tmp := null;
123
124      b := Conn_addr[c];
125      mmc_unmap(b, Conn_key[c]);
126      Conn_addr[c] := null;
127      delete c;
128      c := null;
129    }
130    return 0;
```

```
131 }
132
133 /*** Beginning of mmc.c code ***/
134
135 procedure mmc_map(key:int):Buffer
136   m:Map;
137   b:Buffer;
138 {
139   check_hash_size();
140
141   m := find_hash(key);
142   if (m != null) {
143     Map_refcount[m] := Map_refcount[m]+1;
144     b := Map_addr[m];
145     return b;
146   }
147
148   @enable(free_maps);
149   if (free_maps != null) {
150     m := free_maps;
151     free_maps := Map_next[m];
152     Map_next[m] := null;
153   } else {
154     m := new Map;
155     Map_next[m] := null;
156   }
157   @disable(free_maps);
158
159   Map_key[m] := key;
160   Map_refcount[m] := 1;
161   b := new Buffer;
162   Map_addr[m] := b;
163
164   add_hash(m);
165
166   Map_next[m] := maps;
167   maps := m;
168
169   return b;
170 }
171
172 procedure mmc_unmap(addr:Buffer, key:int):int
173   m:Map;
174
```

```
175   predicate(heap) Reach_m(n:Map) = MapTC(m, n);
176 {
177   m := find_hash(key);
178   if (m != null) {
179     if (Map_addr[m] != addr) {
180       m := null;
181     }
182   }
183
184   if (m = null) {
185     m := maps;
186     while (m != null) {
187       [<loop>]
188
189       if (Map_addr[m] = addr) goto loop_exit;
190
191       m := Map_next[m];
192     }
193     [<loop_exit>]
194   }
195
196   @assert(m != null);
197   Map_refcount[m] := Map_refcount[m]-1;
198
199   return 0;
200 }
201
202 procedure check_hash_size():int
203   m:Map;
204   i:int;
205
206   predicate(heap) Reach_m(n:Map) = MapTC(m, n);
207   predicate(numeric) Ni(x:int) = x = i;
208   predicate(numeric) Nlti(x:int) = x < i;
209   partitioning(numeric) Phash_i = Nlo, Nok, Nhi, Ni, Nlti;
210 {
211   @split(Phash_i, 0);
212   @split(Phash_i, hash_size + 100);
213   @split(Phash, hash_size + 100);
214
215   hash_size := hash_size + 100; // FIXME
216
217   i := 0;
218   @split(Phash_i, i);
```

```
219    while (i < hash_size) {
220      [<null_loop>]
221      hash_table[i] := null;
222      @split(Phash_i, i+1);
223      i := i+1;
224    }
225
226    m := maps;
227    while (m != null) {
228      [<add_loop>]
229      add_hash(m);
230      m := Map_next[m];
231    }
232
233    return 0;
234  }
235
236  procedure add_hash(m:Map):int
237    h:int; he:int; i:int;
238
239    predicate(numeric) Ni(x:int) = x = i;
240    partitioning(numeric) Phash_i = Nlo, Nok, Nhi, Ni;
241  {
242    @split(Phash_i, 0);
243    @split(Phash_i, hash_size);
244
245    h := Map_key[m] % hash_size;
246    he := (h + hash_size - 1) % hash_size;
247    i := h;
248    while (0 = 0) {
249      [<loop>]
250
251      @split(Phash_i, i);
252      if (hash_table[i] = null) {
253        hash_table[i] := m;
254        Map_hash_idx[m] := i;
255        return 0;
256      }
257      if (i = he) goto loop_exit;
258      i := (i+1) % hash_size;
259    }
260    [<loop_exit>]
261    Map_hash_idx[m] := 0;
262    return -1;
```

```
263  }
264
265  procedure find_hash(key:int):Map
266    m:Map;
267    h:int; he:int; i:int;
268
269    predicate(numeric) Ni(x:int) = x = i;
270    partitioning(numeric) Phash_i = Nlo, Nok, Nhi, Ni;
271  {
272    @split(Phash_i, 0);
273    @split(Phash_i, hash_size);
274
275    h := key % hash_size;
276    he := (h + hash_size - 1) % hash_size;
277    i := h;
278    m := null;
279    while (0 = 0) {
280      [<loop>]
281
282      @assert(0 <= i && i < hash_size);
283      @split(Phash_i, i);
284      m := hash_table[i];
285      if (m = null) goto loop_exit;
286      if (Map_key[m] = key) return m;
287      if (i = he) goto loop_exit;
288      i := (i+1) % hash_size;
289
290      m := null;
291    }
292    [<loop_exit>]
293    return null;
294  }
295
296  procedure really_unmap(m:Map, prev:Map):Map
297    rm:Map;
298    b:Buffer;
299    n:int;
300
301    predicate(heap) Reach_rm(n:Map) = MapTC(rm, n);
302    predicate(numeric) Nnlo(i:int) = i < n;
303    predicate(numeric) Nn(i:int) = i = n;
304    partitioning(numeric) Ptable = Nlo, Nok, Nhi, Nn, Nnlo;
305  {
306    b := Map_addr[m];
```

```
307    delete b;
308    b := null;
309    Map_addr[m] := null;
310
311    @split(Ptable, 0);
312    @split(Ptable, hash_size);
313    n := Map_hash_idx[m];
314    @split(Ptable, n);
315    hash_table[n] := null;
316
317    rm := Map_next[m];
318
319    if (prev = null) maps := Map_next[m];
320    else Map_next[prev] := Map_next[m];
321
322    @enable(free_maps);
323    Map_next[m] := free_maps;
324    free_maps := m;
325    @disable(free_maps);
326
327    return rm;
328  }
329
330  procedure mmc_destroy():int
331    m:Map;
332  {
333    m := maps;
334    while (m != null) {
335      m := really_unmap(m, null);
336    }
337
338    @enable(free_maps);
339    while (free_maps != null) {
340      m := free_maps;
341      free_maps := Map_next[m];
342      delete m;
343      m := null;
344    }
345    @disable(free_maps);
346
347    return 0;
348  }
349
350  procedure mmc_cleanup():int
```

```
351    head:Map;
352    m:Map;
353    prev:Map;
354
355    predicate(heap) Reach_m(n:Map) = MapTC(m, n);
356  {
357    @disable(MapRCGood);
358    m := maps;
359    head := m;
360    prev := null;
361    while (m != null) {
362      if (Map_refcount[m] = 0) {
363        m := really_unmap(m, prev);
364        if (prev = null) head := m;
365      } else {
366        prev := m;
367        m := Map_next[m];
368      }
369    }
370
371    prev := null;
372    head := null;
373
374    @enable(free_maps);
375    while (free_maps != null) {
376      m := free_maps;
377      free_maps := Map_next[m];
378      delete m;
379      m := null;
380      if (*) goto loop_exit;
381    }
382    [<loop_exit>]
383    @disable(free_maps);
384    @enable(MapRCGood);
385
386    return 0;
387  }
```

Chapter 6

Conclusion

In §1.5 of the introduction, we summarized the goals of this thesis. One goal is to design a heap domain based on TVLA that is capable of analyzing the `thttpd` code easily and efficiently. Another goal is to make a working combined domain to reason effectively about the invariants of interest to us. The final goal is to augment the integer domain to the point that it can reason about complex cardinality and reference counting invariants. This final chapter reviews our progress on these goals.

In Chapter 2, we presented several major changes to the TVLA heap analysis to make it faster and easier to use. To speed up the analysis, we built a query optimizer and an efficient query execution engine. These two pieces increased query performance by three orders of magnitude. We also developed an improved abstraction that reduces the number of disjuncts that appear in analysis results from 18 to 2 in the median. Eliminating disjuncts increases scalability by an exponential factor. It also makes the analysis results easier for users to understand. Our new sharpening algorithm also improves on TVLA's ease-of-use. Since it is more precise than TVLA's algorithm, users do not have to manually specify integrity constraints for the predicates they define. This reduces the overall annotation burden.

We chose TVLA over analyses based on separation logic because TVLA is a better fit for our combined domain. However, we believe that the changes we have made in our heap domain make it competitive with separation logic in almost every respect—performance, scalability, and ease of use. However, we did not address the important issue of interprocedural analysis.

Chapter 3 presented the combined domain, which allows us to infer invariants requiring mixed heap and integer reasoning. The most salient feature of our combined domain is its generality. We use it to reason about indexing, cardinality, and reference counting, all of which require communication between the heap and the integer domain. However, the combined domain is oblivious to these specific properties because they can all be expressed in the language of predicates and classes. The generality of the combined domain is useful for implementers because new features can be added to the base domains without affecting the combined domain at all. Despite the wide range of invariants we support, the combined domain is expressible in fewer than 500 lines of ML code.

Finally, Chapter 4 presents an integer domain augmented with additional features for quantification, predicates, and cardinality. We are not aware of any other automated analysis that supports these features with the same level of generality that we do. For example, several tools perform cardinality reasoning [25, 31], but we know of no tool that can do *quantified* cardinality reasoning (i.e., reasoning about the cardinality of a set that is parametrized by a quantified variable).

To conclude, we have developed a precise, efficient analysis for real-world systems code and implemented it in the DESKCHECK system. Since the invariants needed for these systems are beyond the power of existing analyses, we developed an abstract domain that combines the power of a heap domain and an integer domain while still allowing the reasoning in these domains to be compartmentalized. We improved the state of the art for both heap domains and integer domains to make them more precise, more efficient, and easier to use. Finally, we tested DESKCHECK on the cache module of the `thttpd` web server and were able to verify its memory safety in two minutes.

Bibliography

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

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

[3] Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, Peter W. O'Hearn, Thomas Wies, and Hongseok Yang. Shape analysis for composite data structures. In *CAV*, 2007.

[4] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Symbolic execution with separation logic. In *In APLAS*, pages 52–68. Springer, 2005.

[5] Igor Bogudlov, Tal Lev-Ami, Thomas W. Reps, and Mooly Sagiv. Revamping TVLA: Making parametric shape analysis competitive. In Werner Damm and Holger Hermanns, editors, *CAV*, volume 4590 of *Lecture Notes in Computer Science*, pages 221–225. Springer, 2007.

[6] 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.

[7] Franois Bourdoncle. Efficient chaotic iteration strategies with widenings. In *In Proceedings of the International Conference on Formal Methods in Programming and their Applications*, pages 128–141. Springer-Verlag, 1993.

[8] 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.

[9] Bor-Yuh Evan Chang and Xavier Rival. Relational inductive shape analysis. In *POPL '08: Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 247–260, New York, NY, USA, 2008. ACM.

[10] Adam Chlipala, Gregory Malecha, Greg Morrisett, Avraham Shinnar, and Ryan Wisnesky. Effective interactive proofs for higher-order imperative programs. In *ICFP '09: Proceedings of the 14th ACM SIGPLAN international conference on Functional programming*, pages 79–90, New York, NY, USA, 2009. ACM.

[11] Jeremy Condit, Matthew Harren, Zachary R. Anderson, David Gay, and George C. Necula. Dependent types for low-level programming. In Rocco De Nicola, editor, *ESOP*, volume 4421 of *Lecture Notes in Computer Science*, pages 520–535. Springer, 2007.

[12] A. Cortesi, B. Le Charlier, and P. Van Hentenryck. Combinations of abstract domains for logic programming. *SCP*, 38(1–3):27–71, 2000.

[13] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM Press, New York, NY.

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

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

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

[17] 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.

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

[19] Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. A local shape analysis based on separation logic. In *In TACAS*, pages 287–302. Springer, 2006.

[20] 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.

[21] M. Emmi, R. Jhala, E. Kohler, and R. Majumdar. Verifying reference counting implementations. In *TACAS*, 2009.

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

[23] Denis Gopan, Thomas W. Reps, and Mooly Sagiv. A framework for numeric analysis of array operations. In *POPL*, pages 338–350, 2005.

[24] P. Granger. Improving the results of static analyses programs by local decreasing iteration. In *FSTTCS*, 1992.

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

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

[27] Jesper G. Henriksen, Ole J.L. Jensen, Michael E. Jrgensen, Nils Klarlund, Robert Paige, Theis Rauhe, and Anders B. Sandholm. MONA: Monadic second-order logic in practice. In *TACAS '95, LNCS 1019*. Springer-Verlag, 1995.

[28] Bertrand Jeannet, Alexey Loginov, Thomas Reps, and Mooly Sagiv. A relational approach to interprocedural shape analysis. *ACM Trans. Program. Lang. Syst.*, 32(2):1–52, 2010.

[29] Shuvendu Lahiri and Shaz Qadeer. Back to the future: Revisiting precise program verification using SMT solvers. In *POPL '08: Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 171–182, New York, NY, USA, 2008. ACM.

[30] K. Rustan M. Leino. This is boogie 2. Available at `http://research.microsoft.com/en-us/um/people/leino/papers/krml178.pdf`.

[31] S. Magill, M.-H. Tsai, P. Lee, and Y.-K. Tsay. THOR: A tool for reasoning about shape and arithmetic. In *CAV*, 2008.

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

[33] R. Manevich, J. Berdine, B. Cook, G. Ramalingam, and M. Sagiv. Shape analysis by graph decomposition. In *In 13th TACAS*, pages 3–18. Springer, 2007.

[34] Roman Manevich, Mooly Sagiv, G. Ramalingam, and John Field. Partially disjunctive heap abstraction. In Roberto Giacobazzi, editor, *Proceedings of the 11th International Symposium, SAS 2004*, volume 3148 of *Lecture Notes in Computer Science*, pages 265–279. Springer, August 2004. Available at http://www.cs.tau.ac.il/∼rumster/sas04.pdf.

[35] Scott McPeak and George C. Necula. Data structure specifications via local equality axioms. In Kousha Etessami and Sriram K. Rajamani, editors, *CAV*, volume 3576 of *Lecture Notes in Computer Science*, pages 476–490. Springer, 2005.

[36] Antoine Miné. A new numerical abstract domain based on difference-bound matrices. In *PADO '01: Proceedings of the Second Symposium on Programs as Data Objects*, pages 155–172, London, UK, 2001. Springer-Verlag.

[37] Anders Moller and Michael I. Schwartzbach. The pointer assertion logic engine. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 221–231, New York, NY, USA, 2001. ACM Press.

[38] George C. Necula, Scott McPeak, and Westley Weimer. Ccured: type-safe retrofitting of legacy code. In *POPL '02: Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 128–139, New York, NY, USA, 2002. ACM.

[39] J. Poskanzer. thttpd - tiny/turbo/throttling http server. `http://acme.com/software/thttpd/`.

[40] Thomas W. Reps, Mooly Sagiv, and Alexey Loginov. Finite differencing of logical formulas for static analysis. In *ESOP*, pages 380–398, 2003.

[41] Thomas W. Reps, Mooly Sagiv, and Alexey Logonov. Finite differencing of logical formulas for static analysis. *ACM TOPLAS*, 32(4), 2010.

[42] Noam Rinetzky, Jörg Bauer, Thomas W. Reps, Mooly Sagiv, and Reinhard Wilhelm. A semantics for procedure local heaps and its abstractions. In *In POPL*, pages 296–309, 2005.

[43] Noam Rinetzky, Mooly Sagiv, and Eran Yahav. Interprocedural shape analysis for cutpoint-free programs. In *12th International Static Analysis Symposium (SAS)*, 2005.

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

[45] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Solving shape-analysis problems in languages with destructive updating. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 16–31, New York, NY, USA, 1996. ACM.

[46] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.*, 24(3):217–298, 2002.

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

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

[49] Hongseok Yang, Oukseh Lee, Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, and Peter O'Hearn. Scalable shape analysis for systems code. In *CAV '08: Proceedings of the 20th international conference on Computer Aided Verification*, pages 385–398, Berlin, Heidelberg, 2008. Springer-Verlag.

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

[51] Greta Yorsh, Thomas Reps, Mooly Sagiv, and Reinhard Wilhelm. Logical characterizations of heap abstractions. *ACM Trans. Comput. Logic*, 8(1):5, 2007.

[52] Greta Yorsh, Thomas W. Reps, and Shmuel Sagiv. Symbolically computing most-precise abstract operations for shape analysis. In Kurt Jensen and Andreas Podelski, editors, *TACAS*, volume 2988 of *Lecture Notes in Computer Science*, pages 530–545. Springer, 2004.

[53] Karen Zee, Viktor Kuncak, and Martin Rinard. Full functional verification of linked data structures. In *ACM Conf. Programming Language Design and Implementation (PLDI)*, 2008.

[54] Karen Zee, Viktor Kuncak, and Martin Rinard. An integrated proof language for imperative programs. In *ACM Conf. Programming Language Design and Implementation (PLDI)*, 2009.