Lifting Abstract Interpreters to Quantified Logical Domains

Sumit Gulwani
Microsoft Research
Redmond, WA, USA
sumitg@microsoft.com

Bill McCloskey
EECS Department, University of California, Berkeley
Berkeley, CA, USA
billm@cs.berkeley.edu

Ashish Tiwari
CS Lab, SRI International
Menlo Park, CA, USA
tiwari@csl.sri.com

Abstract

We describe a general technique for building abstract interpreters over powerful *universally quantified* abstract domains that leverage existing quantifier-free domains. Our quantified abstract domain can represent universally quantified facts like $\forall i(0 \leq i < n \Rightarrow a[i] = 0)$. The principal challenge in this effort is that, while most domains supply over-approximations of operations like join, meet, and variable elimination, working with the guards of quantified facts requires *under*-approximation. We present an automatic technique to convert the standard over-approximation operations provided with all domains into sound under-approximations. We establish the correctness of our abstract interpreters by identifying two lattices—one that establishes the soundness of the abstract interpreter and another that defines its precision, or completeness. Our experiments on a variety of programs using arrays and pointers (including several sorting algorithms) demonstrate the feasibility of the approach on challenging examples.

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

General Terms Algorithms, Theory, Verification

Keywords Underapproximation Algorithms, Quantified Invariants, Abstract Interpreter, Logical Lattices

1. Introduction

Proving the correctness of software almost always requires the use of universal quantifiers, since program invariants often need to constrain unbounded segments of a data structure. However, abstract interpreters are typically designed to constrain only a bounded set of program variables; they cannot express quantified facts. In this paper, we describe a general method of transforming a quantifier-free abstract domain into a universally quantified domain. Since there already is a huge variety of quantifier-free domains available, our technique can be readily instantiated in many areas. For example, in our experiments we were able to reason about unbounded numbers of array locations and heap objects using some fairly simple base domains (difference constraints and reachability predicates, respectively).

We began our research by considering many potentially useful quantified invariants. Ultimately, we were able to express all of them in the following form.

$$E \wedge \bigwedge_{j=1}^{n} \forall U_j (F_j \Rightarrow e_j) \tag{1}$$

Here, E, F_j and e_j are quantifier-free facts from three potentially different domains, $\mathcal{D}_a$, $\mathcal{D}_b$, and $\mathcal{D}_c$, respectively. These domains are the parameters to our universally quantified domain; we call them the base domains. The abstract element E, called the environment, contains quantifier-free facts about program variables. Without any quantified facts, our domain reduces to $\mathcal{D}_a$. Elements F_j are the guards to the quantified facts e_j, which are quantified over variables U_j.

This universal domain, written $\mathcal{D}_\forall$, represents typical quantified facts quite naturally. To constrain array values, we use the domain of difference constraints for the guards and uninterpreted functions to represent array access in the quantified facts: $\forall i(0 \leq i < n \Rightarrow \mathtt{A}[i] = 0)$. In some cases, more complex domains are required. The invariant $\forall i, j(0 \leq i < n \wedge 0 \leq j < \mathtt{size}[i] \Rightarrow \mathtt{A}[i][j] \leq \mathtt{A}[i][j+1])$ can be represented using the combination of difference constraints and uninterpreted functions for both the guards and the facts [11]. In our experiments section we describe a reachability domain that allows us to represent constraints like $\forall n(n \in R(\mathtt{list}) \wedge n \notin R(\mathtt{p}) \Rightarrow n.\mathtt{data} = 0)$. Here, all elements reachable from the variable `list` up to the element pointed to by `p` are required to have a data field containing zero. Even some existential facts can be represented via Skolemization. If our domain includes a function NLEN defined to be the length of null-terminated C string (and undefined for non-strings), then $\forall i(0 \leq i < \text{NLEN}(\mathtt{s}) \Rightarrow \mathtt{s}[i] \neq \mathtt{'!'})$ forbids the string `s`, regardless of its length, from containing exclamation points (facts like this are useful for analyzing string sanitization routines for security).

To ensure adequate performance, we place some simple restrictions on the base domains. The fact e_j must be a single atomic fact—a predicate of the form $p(t_1, \ldots, p_m)$, such as $t_1 < t_2$ or $t_1 = t_2$. Elements E and F_j must be finite conjunctions of atomic facts. There is no way to represent disjunctions explicitly in any of these elements, which eliminates a possible source of exponential explosion in the size of the element.

The abstract interpreters for the base domains will be supplied with transfer functions for assignments, assume statements, and control-flow joins. All of these functions are over-approximations of the ideal result. However, the crux of this paper is that quantification introduces extra complexity: in order to *over*-approximate a function on a quantified abstract element, we need to *under*-

Research was done while second author was at Microsoft. Research of the third author was supported in part by NSF grant CCR-ITR-0326540.

POPL'08, January 7–12, 2008, San Francisco, California, USA.

approximate its effect on the quantifier guard. Under-approximation has been studied in the context of backward analysis, but it has never been applied in this way before. The following example demonstrates the importance of under-approximation.

Under-approximation operators at work. The following example initializes elements 0 to $n-1$ of array `A` to 0.

```
for (i = 0; i < n; i++) A[i] = 0
```

We would like to prove that $\forall k(0 \leq k < n \Rightarrow \mathtt{A}[k] = 0)$ when the loop terminates. There are many ways to explore all the possible control-flow paths taken by this loop. For explanatory purposes, we focus on the first two unrollings of this loop. We let $\mathcal{D}_a$ be the combined domain of difference constraints and uninterpreted functions [11]. After the first iteration, this domain infers the following quantifier-free fact in the environment E:

$$\mathcal{E}_1 \quad : \quad i = 1 \ \wedge \ A[0] = 0$$

We assume that no join occurs, so that the abstract interpreter continues to execute the second iteration, producing another quantifier-free fact:

$$\mathcal{E}_2 \quad : \quad i = 2 \ \wedge \ A[0] = 0 \ \wedge \ A[1] = 0$$

At this time, we would like to join $\mathcal{E}_1$ and $\mathcal{E}_2$. Using the standard join algorithm for the base domain does not yield a satisfying result; the invariant we desire requires quantifiers, as n can be arbitrarily large. Therefore, the join algorithm for our universally quantified domain $\mathcal{D}_\forall$ begins by introducing quantifiers in both elements. (Throughout this paper, we call anything of the form $\forall U(F \Rightarrow e)$ a *quantifier* or *quantified fact*, while a domain element from $\mathcal{D}_\forall$ is called a *quantified domain element*.) The details of quantifier introduction are explained later; the result is as follows:

$$\mathcal{E}_1' \ : \ i = 1 \ \wedge \ \forall k(k = 0 \Rightarrow A[k] = 0)$$
$$\mathcal{E}_2' \ : \ i = 2 \ \wedge \ \forall k(k = 0 \Rightarrow A[k] = 0) \ \wedge \ \forall k(k = 1 \Rightarrow A[k] = 0)$$

Note that $\mathcal{E}_1'$ and $\mathcal{E}_2'$ are equivalent to $\mathcal{E}_1$ and $\mathcal{E}_2$. Also note that the two quantified facts in $\mathcal{E}_2'$ have the same right-hand side. Therefore, they can be simplified to a single quantifier. We call this simplification *quantifier merging*. The tricky part is merging the guards. In general, given two quantified facts from the same domain element, $\forall U(F_1 \Rightarrow e)$ and $\forall U(F_2 \Rightarrow e)$, it is logically sound to merge them into $\forall U(F_1 \vee F_2 \Rightarrow e)$. However, our domain does not represent disjunctions since they are a major source of inefficiency. The standard way of approximating disjunction in an abstract domain is with the join operation. However, using join will lead to unsoundness here. Consider the facts $\forall i(i = 0 \Rightarrow e)$ and $\forall i(i = 2 \Rightarrow e)$. The join of the guards in a typical numerical domain will yield $0 \leq i \leq 2$. However, the fact $\forall i(0 \leq i \leq 2 \Rightarrow e)$ is not a sound over-approximation of the conjunction of the two original facts.

The source of the problem is that the join algorithm for an abstract domain is required to generate an over-approximation of the disjunction of two elements. Since the guard of a quantifier appears in a negative position (in the antecedent of an implication), we must *under*-approximate the disjunction of guards instead. Unfortunately, most domains are not equipped with under-approximation operators. This paper addresses the problem by constructing sound under-approximations for each domain operation, given the corresponding over-approximations. These under-approximations work in fairly general circumstances, which allows us to parametrize our universal domain with a wide variety of base domains.

Our solution is quite simple. We under-approximate disjunctions by taking the join in the base domain $\mathcal{D}_b$ of the two disjuncts (an over-approximation) and then adding extra constraints to the result until it is a valid under-approximation. Fortunately, the join of the two guards in $\mathcal{E}_1'$ and $\mathcal{E}_2'$, $0 \leq k \leq 1$, is already a valid under-approximation, so there is no need to refine it. Hence, after the simplification of $\mathcal{E}_2'$, we now are faced with joining these two elements in $\mathcal{D}_\forall$:

$$\mathcal{E}_1'' \quad : \quad i = 1 \ \wedge \ \forall k(k = 0 \Rightarrow A[k] = 0)$$
$$\mathcal{E}_2'' \quad : \quad i = 2 \ \wedge \ \forall k(0 \leq k \leq 1 \Rightarrow A[k] = 0)$$

The facts about i in the two environments can be joined using the standard over-approximation operator from $\mathcal{D}_a$. Since the two quantifiers have the same right-hand side, $A[k] = 0$, we choose to join them together. In general, when joining two quantifiers $\forall U(F_1 \Rightarrow e)$ and $\forall U(F_2 \Rightarrow e)$ from different quantified domain elements, a safe answer is $\forall U(F_1 \wedge F_2 \Rightarrow e)$ (meaning it over-approximates the $\vee$ of the two inputs). However, doing such a simple conjunction of guards will almost always lead to imprecision. In this case, it simply yields the new fact $\forall k(k = 0 \Rightarrow A[k] = 0)$, which is no better than if we had joined the quantifier-free facts.

The key to solving this problem is to join quantifiers *in the presence of their environments*. The i constraints in $\mathcal{E}_1''$ and $\mathcal{E}_2''$ offer natural upper bounds for k. So rather than generating $F_1 \wedge F_2$, we would like to produce $F = (E_1 \Rightarrow F_1) \wedge (E_2 \Rightarrow F_2)$, where E_1 and E_2 are the respective environments of the universally quantified elements to be joined. Unfortunately, we are again faced with the problem of under-approximating a logical formula that is essentially disjunctive (since implication is another form of disjunction).

To solve this problem, we use a similar approach to the one before: we use an over-approximation of the join algorithm in $\mathcal{D}_b$ to start with, and then refine this answer until it is a valid under-approximation. The initial join takes the environments into account by joining $E_1 \wedge F_1$ with $E_2 \wedge F_2$. In the array initialization example, we must join $i = 1 \ \wedge \ k = 0$ with $i = 2 \ \wedge \ 0 \leq k \leq 1$. The result, $1 \leq i \leq 2 \ \wedge \ 0 \leq k < i$, is a sound and precise guard for the fact $A[k] = 0$. Thus, there is no need to refine the result, since it is already a valid under-approximation. For simplicity, we can eliminate the constraints on i, since they are already represented in the joined environment. Ultimately, we get the following element in the quantified domain, which is the one we desired:

$$\mathcal{E} \quad : \quad 1 \leq i \leq 2 \ \wedge \ \forall k(0 \leq k < i \Rightarrow A[k] = 0)$$

The steps we took to arrive at this result may appear somewhat ad-hoc. Perhaps if the loop had not been unrolled twice, or the initial quantifiers had been introduced differently, or if the join algorithm had not always produced a valid under-approximation, our approach would have been less successful? In powerful domains like this one, the threat is ever-present that small changes to the inputs can lead to a dramatic loss of precision. To alleviate this fear, we prove that our domain is complete relative to a fairly intuitive partial order in the quantified domain (although not relative to the implication partial order, which is undecidable). Our under-approximation operators are also complete for common domains. Our implementation does perform "tricks" like loop unrolling, but it does so only as a performance optimization. It would still arrive at the desired invariant even without loop unrolling; however, doing so forces it to consider many more possible quantified invariants, most of which are spurious.

Contributions. The contributions of this paper are four-fold. We describe a parametrized framework of universally quantified abstract domains. We describe a general technique to construct under-approximation operations from their respective over-approximations. We provide soundness proofs and restricted completeness proofs for both. Finally, we present benchmarks on complex array- and pointer-based examples demonstrating that it is possible to construct practical abstract interpreters using our methodology.

2. Quantified Abstract Domain

The *quantified domain*, $\mathcal{D}_\forall$, is parametrized by three (not necessarily different) base abstract domains, $\mathcal{D}_a$, $\mathcal{D}_b$, and $\mathcal{D}_c$. We assume these are *conjunctive* domains, meaning that each domain element is a conjunction of *atomic facts* of the form $p(t_1, \ldots, t_k)$ or $\neg p(t_1, \ldots, t_k)$. p can be any predicate, such as $=$, $<$, or reachability. Each t_i is an arbitrary term. We assume that these domains are equipped with a partial order $\preceq$. However, since the domains are conjunctive domains, we can think of domain elements as formulas in some logical theory. As such, the partial order $\preceq$ should always respect the standard logical implication relation $\overset{\mathbb{T}}{\Rightarrow}$ for that theory (i.e., $\preceq \subseteq \overset{\mathbb{T}}{\Rightarrow}$).

An abstract element $\mathcal{E}$ in quantified domain $\mathcal{D}_\forall$ is of the form

$$\mathcal{E} \quad = \quad E \wedge \bigwedge_i \forall U_i (F_i \Rightarrow e_i) \tag{2}$$

where (a) $E \in \mathcal{D}_a$ is the environment, (b) $F_i \in \mathcal{D}_b$ and $e_i \in \mathcal{D}_c$ for each i, and (c) without loss of generality, for each i, U_i is a set of variables disjoint from the program variables.

The semantics of the abstract element $\mathcal{E}$ is described in terms of the semantics of the constituent abstract domains $\mathcal{D}_a$, $\mathcal{D}_b$, and $\mathcal{D}_c$. Let $\gamma_{\mathcal{D}_a}$, $\gamma_{\mathcal{D}_b}$, and $\gamma_{\mathcal{D}_c}$, be the concretization functions of the base abstract domains. The concretization function $\gamma_{\mathcal{D}_\forall}$ of the quantified abstract domain is defined as follows. A program state $\rho \in \gamma_{\mathcal{D}_\forall}(\mathcal{E})$ iff

- $\rho \in \gamma_{\mathcal{D}_a}(E)$, and
- for every i, if ρ' is an extension of state ρ with assignment to variables in U_i such that $\rho' \in \gamma_{\mathcal{D}_b}(F_i)$, then, $\rho' \in \gamma_{\mathcal{D}_c}(e_i)$.

A special and important class of quantified abstract domains is obtained when $\mathcal{D} := \mathcal{D}_a = \mathcal{D}_b = \mathcal{D}_c$. For simplicity and ease of presentation throughout the rest of this paper we will only refer to the base domain $\mathcal{D}$ and the quantified domain $\mathcal{D}_\forall$.

2.1 Quantified Partial Order

The most obvious partial order for the quantified domain is logical implication. However, given that the partial order for the base domain may be weaker than logical implication, we have no hope of deciding implication on the quantified domain.

A more tractable approach is to obtain a quantified partial order whose power is relative to the partial order given for the base domain. However, even this problem is quite tricky. Consider the following two quantified abstract elements.

$$\mathcal{E}_1 \quad : \quad \forall j, k (0 \leq j < n - 1 \wedge k = j + 1 \Rightarrow A[j] \leq A[k])$$
$$\mathcal{E}_2 \quad : \quad \forall j, k (0 \leq j < k < n \Rightarrow A[j] \leq A[k])$$

$\mathcal{E}_1$ implies $\mathcal{E}_2$, but proving the implication requires induction. We would like to avoid solving such hard problems. Therefore, we use a simplified partial order defined as follows. We say

$$E \wedge \bigwedge_{i=1}^{n} \forall U_i (F_i \Rightarrow e_i) \;\; \preceq_\forall \;\; E' \wedge \bigwedge_{i=1}^{n'} \forall U'_i (F'_i \Rightarrow e'_i)$$

when the following criteria are satisfied:

(i) The environments match up in the base domain: $E \preceq E'$.

(ii) For each fact $\forall U (F'_i \Rightarrow e'_i)$ on the right-hand side, there is a fact $\forall U (F_j \Rightarrow e_j)$ on the left-hand side (assumed to have the same set of variables, for simplicity) such that: (a) $E \wedge F'_i \preceq F_j$, and (b) $E \wedge e_j \preceq e'_i$.

We choose this partial order because it is fairly natural and because it is powerful enough that the examples presented later are analyzable. We prove completeness relative to this partial order, but we prove soundness with respect to the standard logical implication partial order, $\overset{\mathbb{T}}{\Rightarrow}$.

One important point to note is that when completeness is defined relative to this partial order of quantified elements an analysis that never introduces any quantified facts is still considered complete. This partial order only guarantees that, once a quantified fact is introduced, it will be preserved as long as possible. Additionally, the partial order treats quantified facts as independent of each other. Thus, for example, it does not require an analysis to transitively close quantified facts to achieve completeness. We consider this property to be beneficial, since it allows us to forgo transitive closure of quantified facts for efficiency while still providing a measure of completeness.

2.2 Templates

A serious problem in implementing a quantified domain is knowing when to introduce quantifiers. For example, consider the quantifier-free element whose environment is $A[0] = 0$. One valid way of turning this fact into a quantified fact results in $\forall k (k = 0 \Rightarrow A[k] = 0)$. However, there are many other ways, such as $\forall k (k = 0 \Rightarrow A[k] = k)$ and even $\forall k (k = 0 \Rightarrow A[A[k]] = A[k])$. According to the quantified partial order given above, these different quantified elements are independent of each other, since their right-hand sides are not related by $\preceq$ in any environment. Introducing a greater number of quantifiers may improve precision, but it incurs a cost in performance.

Therefore, we add extra structure to the $\mathcal{D}_c$ domain by requiring that all atomic facts represented in it match one of a given set of *templates*. A template is a formula such as $a[\alpha] = 0$. An atomic fact e matches a template t if there is a substitution σ over the template's variables such that $e = t\sigma$ via syntactic equality. In the examples above, we would not introduce the fact $A[A[0]] = A[0]$ when using the template $a[\alpha] = 0$. We would introduce $A[0] = 0$, since there is a substitution $\sigma = \{a = A, \alpha = 0\}$.

Templates have three kinds of variables, Greek, Roman, and fixed. Fixed variables are treated the same as constants—they can only be matched with a program variable of the same name; Greek and Roman variables can match arbitrary expressions. When introducing quantifiers, we quantify over the Greek variables but not the Roman ones. Greek variables are written α, Roman variables as a, and fixed variables as `A`. Thus, from the fact $A[0] = 0$ and the template $a[\alpha] = 0$ we would create a single quantifier, $\forall \alpha (\alpha = 0 \Rightarrow A[\alpha] = 0)$.

Templates have an important impact on precision in practice. Without any templates, no quantifiers are ever introduced, making the analysis no more powerful than the base domain. However, we did not find it difficult to produce good templates during our experiments. All the quantified invariants that we found useful have complex guards, but their right-hand sides are simple. Since templates are needed only for the right-hand sides, they are not a significant obstacle in generating useful invariants.

3. Abstract Interpreter for $\mathcal{D}_\forall$

This section presents the principal transfer functions for $\mathcal{D}_\forall$, along with soundness and completeness proofs:

- The join function computes an over-approximation of the logical $\vee$ of two quantified abstract elements (essentially their union), which involves under-approximating the $\wedge$ of their quantifier guards.
- The assignment transfer function eliminates any references to the variable being assigned to. Then it adds a new equality representing the assignment. Variable elimination is handled by over-approximating $\exists$ quantifier elimination, which, in turn, re-

$\lceil\vee\rceil_{\mathcal{D}_\forall}(\mathcal{E},\mathcal{E}')$

1 `Let` $\mathcal{E}$ `be` $E \wedge \bigwedge_{i=1}^{n} \forall U_i(F_i \Rightarrow e_i)$.
2 `Let` $\mathcal{E}'$ `be` $E' \wedge \bigwedge_{j=1}^{m} \forall U'_j(F'_j \Rightarrow e'_j)$.
3 $E'' := \lceil\vee\rceil_{\mathcal{D}}(E, E')$;
4 `result` $:= E''$;
5 `forall` $i \in \{1,..,n\}, j \in \{1,..,m\}$:
6 $\quad e := \lceil\vee\rceil_{\mathcal{D}_c}((e_i, E), (e'_j, E'))$;
7 $\quad$ `if` $(e = \emptyset)$ `continue`;
8 $\quad F''_{ij} := \lfloor\wedge\rfloor((F_i, E), (F'_j, E'))$;
9 $\quad$ `result` $:=$ `result` $\wedge \forall U_i, (F''_{ij} \Rightarrow e)$
10 `forall` $i \in \{1,..,n\}$:
11 $\quad F''_{i0} := \lfloor\wedge\rfloor((F_i, E), (\texttt{false}, E'))$;
12 $\quad$ `result` $:=$ `result` $\wedge \forall U_i(F''_{i0} \Rightarrow e_i)$
13 `forall` $j \in \{1,..,m\}$:
14 $\quad F''_{0j} := \lfloor\wedge\rfloor((\texttt{false}, E), (F'_j, E'))$;
15 $\quad$ `result` $:=$ `result` $\wedge \forall U'_j(F''_{0j} \Rightarrow e'_j)$
16 `return result`;

Figure 1. $\lceil\vee\rceil_{\mathcal{D}_\forall}$. The join algorithm for the quantified domain.

quires under-approximating $\forall$ quantifier elimination for quantifier guards.

- Quantifier introduction uses templates to move facts from the environment into a new quantified fact. Quantifier merging combines the guards of two quantified facts with the same right-hand sides. Guards are merged using an under-approximation of $\vee$.
- A widening step ensures termination of abstract interpretation.

We discuss these basic operators in this section; the next section describes the under-approximation operators that they depend on. A common theme here is that over-approximating an operator for the quantified domain requires us to under-approximate its dual for the guard. Since both under-approximation and over-approximation are so important in this area, we use a special notation. An under-approximation of a formula F is denoted $\lfloor F \rfloor$, while an over-approximation is written $\lceil F \rceil$. Similarly, operators like $\wedge$ are written as either $\lfloor\wedge\rfloor$ or $\lceil\wedge\rceil$.

3.1 Quantified Join Algorithm

Consider two elements of $\mathcal{D}_\forall$, $\mathcal{E} := E \wedge \bigwedge_{i=1}^{n} \forall U(F_i \Rightarrow e_i)$ and $\mathcal{E}' := E' \wedge \bigwedge_{j=1}^{m} \forall U(F'_j \Rightarrow e'_j)$. The join algorithm is described below; pseudo-code is shown in Figure 1.

Let $L := (\mathcal{D}, \preceq)$ denote the base logical lattice. We assume that we are given a procedure, $\lceil\vee\rceil_{\mathcal{D}}$, that implements a sound and complete join operator for the base domain with respect to the lattice L. The first step in joining two quantified domain elements is to join their environments in the base domain, as shown on Line 3 of Figure 1.

Next consider the process of joining quantified facts. When computing the join of, say, $E \wedge \forall U(F_1 \Rightarrow e_1)$ and $E' \wedge \forall U(F'_1 \Rightarrow e'_1)$, we first need to "match" e_1 and e'_1—that is, make them identical. To do so, we take advantage of the fact that all quantifiers are generated from templates, so the right-hand side of each quantifier matches some template. We say two quantifiers match if they match the same template in the same way.

However, this matching can be tricky, as illustrated by an example. Consider the following two quantified domain elements to be joined (facts like these appear in practice when analyzing selection sort).

$$\mathcal{E}_1 \quad : \quad (x = \texttt{min}) \wedge \forall u(F_1(u) \Rightarrow A[x] \leq A[u])$$
$$\mathcal{E}_2 \quad : \quad (y = \texttt{min}) \wedge \forall u(F'_1(u) \Rightarrow A[y] \leq A[u])$$

When joining these elements, we would like to generate a quantifier whose right-hand side is $A[\texttt{min}] \leq A[u]$. The facts in both of these elements were generated from a template $t := a[v] \leq a[\alpha]$. It is easy to see that in $\mathcal{E}_1$, the right-hand side matches the template t with substitution $\sigma_1 = \{a = A, v = x, \alpha = u\}$. In $\mathcal{E}_2$ the substitution is $\sigma_2 = \{a = A, v = y, \alpha = u\}$. Somehow from these two substitutions we must generate $A[\texttt{min}] \leq A[u]$.

To do so, we take advantage of the environments of the two quantified elements using the join algorithm from the base domain. We convert each substitution into a fact in the base domain. For example, σ_1 is converted to $S_1 := a = A \wedge v = x \wedge \alpha = u$ (templates are required to use fresh variables). Then we compute the join of the substitutions: $S := \lceil\vee\rceil_{\mathcal{D}}(E \wedge S_1, E' \wedge S_2)$. In the example, we get $S = (a = A \wedge v = \texttt{min} \wedge \alpha = u)$. If S implies that each template variable is equal to some expression, then S can be converted to a substitution σ, and σ applied to t yields the desired right-hand side of the joined quantifier, $A[\texttt{min}] \leq A[u]$. If S does not imply a value for each template variable, then the quantifiers are said not to match. The function $\lceil\vee\rceil_{\mathcal{D}_c}((e_1, E), (e'_1, E'))$ carries out the algorithm just described. It returns either an atomic fact as its result, or $\emptyset$ if the inputs do not match.

Now, suppose that we have made the right-hand sides of the two quantifiers identical and we are left with the problem of computing the join of $E \wedge \forall U(F_1 \Rightarrow e)$ and $E' \wedge \forall U(F'_1 \Rightarrow e)$. It is easy to see that the result of this join should contain a fact $\forall U(F \Rightarrow e)$, where F satisfies the following property:

$$F \wedge E \;\Rightarrow\; F_1 \qquad \text{and} \qquad F \wedge E' \;\Rightarrow\; F'_1$$

Hence F is an under-approximation of $(E \Rightarrow F_1) \wedge (E' \Rightarrow F'_1)$. We need a function on the base domain that computes such an F. Specifically, we assume that we have the procedure $\lfloor\wedge\rfloor((F^l_i, E^l), (F^r_j, E^r))$ that under-approximates $(E^l \Rightarrow F^l_i) \wedge (E^r \Rightarrow F^r_j)$ in $\mathcal{D}$. Section 4 presents an algorithm to compute this function, which is sound and, in some cases, complete.

Given this algorithm, the pseudo-code in Figure 1 proceeds as one would expect. Lines 6-9 match up quantifiers on either side and relate their guards via $\lfloor\wedge\rfloor$. Lines 10-15 perform a similar function, but their job is to match up a fact $\forall U(F \Rightarrow e)$ on one side with a dummy fact $\forall U(\texttt{false} \Rightarrow e)$ on the other. For example, consider the following quantified elements (similar to the find example in our experiments).

$$\mathcal{E}_1 \quad : \quad \texttt{found} = \texttt{false}$$
$$\mathcal{E}_2 \quad : \quad \texttt{found} = \texttt{true} \wedge \forall i(0 \leq i < 10 \Rightarrow P(i))$$

The join of these two facts should not be simply $\top$. We can arrive at a more precise fact, namely:

$$\mathcal{E}' \quad : \quad \forall i(\texttt{found} = \texttt{true} \wedge 0 \leq i < 10 \Rightarrow P(i))$$

We produce this quantifier because $\mathcal{E}_1$ has an "implicit" quantifier saying $\forall i(\texttt{false} \Rightarrow P(i))$. This implicit quantifier, when matched with the quantifier in $\mathcal{E}_2$, yields the quantifier in $\mathcal{E}'$. This is true because of the following fact:

$$\lfloor\wedge\rfloor((\texttt{false}, \texttt{found} = \texttt{false}), (0 \leq i < 10, \texttt{found} = \texttt{true}))$$
$$= (\texttt{found} = \texttt{true} \wedge 0 \leq i < 10)$$

Therefore, our join algorithm also tries to match each quantifier on either side with `false`, as shown in Lines 10-15.

Correctness. We prove the soundness of the join with respect to the strongest partial order possible: logical implication, $\overset{\mathbb{T}}{\Rightarrow}$. However, we prove completeness using a weaker partial order, $\preceq_\forall$, since a complete algorithm with respect to logical implication is undecidable. Proofs of both these lemmas appear in the full version of this paper [10]. The soundness (completeness) proof requires that the base domain operations, $\preceq$ and $\lfloor\wedge\rfloor$, are themselves sound (respectively complete). The $\lfloor\wedge\rfloor$ implementation we present later is sound, but complete only in some circumstances.

$\lceil\exists\rceil(\ell, \mathcal{E})$

1 `Let` $\mathcal{E}$ `be` $E \wedge \bigwedge_{i=1}^{n} \forall U_i(F_i \Rightarrow e_i)$;

`// Instantiate some quantified facts.`

2 $E' := E \wedge \bigwedge\{e_i\sigma \mid E \preceq_{\mathcal{D}} F_i\sigma; \ell$ `occurs in` $e_i\sigma\}$

`// Fix` F_i `1: Remove affected instances`

3 $T_i := \{t \mid t$ `occurs in` $F_i, e_i; Vars(t) \cap U_i \neq \emptyset\}$

4 $F_i' := F_i \wedge \bigwedge_{t \in T_i}\{$`NotEffect`$(\langle \ell, E' \rangle, t)\}$;

`// Fix` F_i `2: Remove` ℓ `from` F_i

5 $F_i'' := \lfloor\forall\rfloor(\ell, F_i', E')$;

`// Fix` e_i`: Remove` ℓ `from` e_i

6 $e_i' := \lceil\exists\rceil(\ell, e_i, E' \wedge F_i')$;

`// Fix` E`: Remove` ℓ `from` E

7 $E'' := \lceil\exists\rceil(\ell, E')$;

8 `return` $E'' \wedge \bigwedge_{i=1}^{n} \forall U_i(F_i'' \Rightarrow e_i')$;

Figure 3. $\lceil\exists\rceil$. Procedure for existential elimination in the quantified domain.

LEMMA 1 (Soundness of Join wrt $\overset{\mathbb{T}}{\Rightarrow}$). *If $\mathcal{E}''$ is the fact returned by the function $\lceil\vee\rceil_{\mathcal{D}_\forall}(\mathcal{E}, \mathcal{E}')$, then $\mathcal{E} \overset{\mathbb{T}}{\Rightarrow} \mathcal{E}''$ and $\mathcal{E}' \overset{\mathbb{T}}{\Rightarrow} \mathcal{E}''$.*

LEMMA 2 (Completeness of Join wrt $\preceq_\forall$). *Let $\mathcal{E}$ be the fact returned by the function $\lceil\vee\rceil_{\mathcal{D}_\forall}(\mathcal{E}^l, \mathcal{E}^r)$. If $\mathcal{E}'$ is another fact such that $\mathcal{E}^l \preceq_\forall \mathcal{E}'$ and $\mathcal{E}^r \preceq_\forall \mathcal{E}'$, then $\mathcal{E} \preceq_\forall \mathcal{E}'$.*

3.2 Existential Elimination for Assignment

The transfer function for an assignment $\ell := r$ is shown on the left side of Figure 2. Most of the machinery for this function is handled by $\lceil\exists\rceil(\ell, \mathcal{E})$, which eliminates any reference to the lvalue ℓ from the quantified domain element $\mathcal{E}$. Once all references have been eliminated, the equality created by the assignment is assumed. This function assumes that assignments where ℓ appears in r have been decomposed into the form "$t := r;\ \ell := t$", where t is a fresh variable.

The difficult part of implementing assignment is delegated to existential quantifier elimination, $\lceil\exists\rceil(\ell, \mathcal{E})$. This function eliminates all references to ℓ from $\mathcal{E}$. It is described throughout the remainder of this subsection. When the assignment is to a program variable (i.e., when $\ell = v$ for some v) then the goal is to generate an over-approximation of the formula $\exists\ell.\mathcal{E}$, with the complication that $\exists$ cannot be represented directly in our domain.

When a heap location is being updated (say, $\ell = A[0]$) then the situation is more complicated. In this case, ℓ is a term $F(t_1, t_2)$, where F is an operator like array sub-scripting or field access. We define the notion of existential quantification of the term $F(t_1, t_2)$ in a formula ϕ as follows:

$$\begin{aligned}\exists(F(t_1,t_2),\phi) \quad \equiv \quad & \exists F'([\forall x, y(x \neq t_1 \vee y \neq t_2) \Rightarrow \\ & F'(x,y) = F(x,y)] \wedge \phi[F'/F]) \quad (3)\end{aligned}$$

The above definition is in terms of second-order existential quantification (of a function symbol) applied on a specific form of a universally quantified formula.

We require the base domain to supply an operator $\lceil\exists\rceil(\ell, E)$ that returns an abstract element E' that over-approximates the logical operator $\exists(\ell, E)$ and that does not "contain" ℓ. To formally define the $\lceil\exists\rceil$ operator on the lattice, we need to define what it means for a term ℓ to *not occur* in an abstract domain element E'.

DEFINITION 1 (Provably not in, $\notin_E$). *A term t is* provably not in E assuming environment E', *denoted by $t \notin_{E'} E$, if either*
(a) t is a variable and t does not syntactically occur in E, or
(b) t is of the form $F(t_1, t_2)$ and for every term $F(t_1', t_2')$ that (syntactically) occurs in E, it is the case that either $E' \preceq t_1 \neq t_1'$ or $E' \preceq t_2 \neq t_2'$.

$$\begin{aligned}
\texttt{NotEffect}(\langle x, E\rangle, x) &= \texttt{false} \\
\texttt{NotEffect}(\langle x, E\rangle, t) &= \texttt{true} \quad \text{when } t \text{ is not } x \\
\texttt{NotEffect}(\langle t, E\rangle, x) &= \texttt{true} \quad \text{when } t \text{ is not } x \\
\texttt{NotEffect}(\langle A[e], E\rangle, B[e']) &= e \neq e' \quad \text{if } E \preceq A = B \\
\texttt{NotEffect}(\langle A[e], E\rangle, B[e']) &= \texttt{NotEffect}(\langle A[e], E\rangle, e') \\
&\qquad \text{if } E \preceq A \neq B \\
\texttt{NotEffect}(\langle A[e], E\rangle, B[e']) &= \texttt{NotEffect}(\langle A[e], E\rangle, e') \\
&\quad \wedge e \neq e' \qquad \text{otherwise}
\end{aligned}$$

Figure 4. `NotEffect` function for the Array Base Domain.

The basic job of $\lceil\exists\rceil(\ell, E)$ is to return an E' so that $\ell \notin E'$. It is clear from this definition that the more disequalities we know, the fewer terms will have to be eliminated. Thus, we define a more refined operator $\lceil\exists\rceil(\ell, E, E')$ that eliminates ℓ from E using disequalities from the environment E'. This operator can be defined from the simpler one by first computing $\lceil\exists\rceil(\ell, E \wedge E')$ and then eliminating from the result any facts that are implied by the environment E'.

EXAMPLE 4. *In the logical lattice induced by the theory of uninterpreted function symbols, we have*
(a) $\lceil\exists\rceil(F(x), x = y \wedge u = F(x))$ is $x = y$,
(b) $\lceil\exists\rceil(F(x), F(x) = F(y) \wedge u = F(x))$ is `true`*, (note here that soundness requires that $u = F(y)$ not be in the result – because it is not the case that $F(x) \notin_{F(x)=F(y)\wedge u=F(x)} u = F(y)$ since $F(x) = F(y) \wedge u = F(x)$ does not imply $y \neq x$.)*
(c) $\lceil\exists\rceil(F(x), x \neq y \wedge F(x) = F(y) \wedge u = F(x))$ is $x \neq y \wedge u = F(y)$,
(d) $\lceil\exists\rceil(F(F(x)), F(x) = F(y) \wedge u = F(x) \wedge x \neq u)$ is $x \neq u \wedge u = F(x)$.

The $\lceil\exists\rceil(\ell, E, E')$ operation can be used to eliminate ℓ from the environment of a quantified domain element $\mathcal{E}$. Figure 3 shows the complete algorithm for eliminating ℓ from $\mathcal{E}$. Line 7 shows the elimination from the environment. In the remainder of the section, we describe what the other lines do.

Elimination from $\mathcal{D}_c$. A universal quantifier represents an infinite conjunction over all possible instantiations of the quantified variables. Some of these instances may be affected by a change to ℓ and others may not be. We would like to eliminate the affected instances while leaving the others alone. This can be achieved by strengthening the guard. Consider the following fact:

$$\mathcal{E} := \forall i(0 \leq i < 10 \Rightarrow A[i] = 0)$$

If the location $A[0]$ is updated, then we can update the guard of this quantifier, but otherwise leave it alone:

$$\mathcal{E}' := \forall i(0 \leq i < 10 \wedge i \neq 0 \Rightarrow A[i] = 0)$$

The general process of updating the guard is handled by Line 4 of Figure 3. For each term in the quantifier that includes a quantified variable, it computes a condition under which the term will not be affected by a change to the updated lvalue. In the example above, it knows that the term $A[i]$ appearing in $A[i] = 0$ will not be affected by an update to $A[0]$ as long as $i \neq 0$. This constraint is then added to the guard.

Computing these constraints is the responsibility of the `NotEffect` function. This function will be somewhat domain-specific, since it depends on the semantics of the functions used to represent heap access. In general, $\texttt{NotEffect}(\langle \ell, E\rangle, t)$ generates a constraint in $\mathcal{D}$ that holds when a change to ℓ does not affect the value of term

PreCondition: ℓ is not a prefix of r.

`PostAssign`$(\mathcal{E}, \ell := r)$

1 $\mathcal{E}' := \lceil\exists\rceil(\ell, \mathcal{E})$;
`// Let` $\mathcal{E}'$ `be` $E' \wedge \bigwedge_{i=1}^{n} \forall U_i(F_i' \Rightarrow e_i')$
`// Add the new fact to the environment`
2 $E'' :=$ `PostAssume`$(E', \ell = r)$;
3 `return` $E'' \wedge \bigwedge_{i=1}^{n} \forall U_i(F_i' \Rightarrow e_i')$;

PreCondition: e is an atomic formula on the base domain

`PostAssume`$(\mathcal{E}, e)$

`// Let` $\mathcal{E}$ `be` $E \wedge \bigwedge_{i=1}^{n} \forall U_i(F_i \Rightarrow e_i)$
`// Add the new fact to the environment`
1 $E' := E\lceil\wedge\rceil e$;
2 `return` $E' \wedge \bigwedge_{i=1}^{n} \forall U_i(F_i \Rightarrow e_i)$;

Figure 2. `PostAssign` and `PostAssume`. The procedures for computing post-conditions of assignment and assume statements are standard.

t. `NotEffect` may use information from the environment E. An example `NotEffect` function that works for Java-style arrays is shown in Figure 4.

After adding extra constraints to the guard, Line 6 of the algorithm eliminates any terms from the quantifier's right-hand side that may be affected by the update to ℓ. The $\lceil\exists\rceil$ algorithm is allowed to use disequalities from both the environment and from the guard, some of which may have been introduced by `NotEffect`.

Elimination from guard. A common theme of this paper is that performing an over-approximation operation on a quantified domain element requires us to perform an under-approximation of its dual on the guard. Consider the following example.

$$\mathcal{E} := i \geq n \wedge \forall k(0 \leq k < i \Rightarrow P(i))$$

This is a typical situation after exiting a loop that establishes property P. Imagine now that the programmer assigns to variable i so it can be used for another loop. It is wrong to simply eliminate facts from the guard involving i, since that would produce $\forall k(0 \leq k \Rightarrow P(i))$, which is not a sound inference. Under-approximating existential elimination on the guard is also a mistake, since that would produce $\forall k(\texttt{false} \Rightarrow P(i))$, which is sound but useless.

Instead, we will under-approximate *universal* quantification. In environment E, with guard F, we will find the weakest fact F' such that $F' \Rightarrow \forall \ell.(E \Rightarrow F)$. In the example above, our goal is to find F' so that $F' \Rightarrow \forall i.(i \geq n \Rightarrow (0 \leq k < i))$. It is easy to verify that $F' := 0 \leq k < n$ is such a fact.

We use the function $\lfloor\forall\rfloor(\ell, F, E)$ to find the under-approximation of $\forall \ell$ on fact F in environment E. An algorithm to compute $\lfloor\forall\rfloor$ is given later in the paper. Line 5 of Figure 3 relies on this algorithm to eliminate ℓ from the guard of a quantifier.

Quantifier instantiation. One important step in existential elimination is to instantiate some quantifiers and move the instantiations to the environment, especially those that may be invalidated. This may seem pointless, since these facts will be immediately eliminated from the environment as well. However, they may lead to inferences in the environment before they are eliminated. Consider the following example, which is derived from insertion sort.

$$\mathcal{E} := \forall k(0 \leq k < i \Rightarrow A[k] \leq A[k+1])$$

The programmer assigns $A[j+1] := A[j]$, where j is a local variable. Our `NotEffect` algorithm will convert the quantifier to the following:

$$\forall k(0 \leq k < i \wedge k \neq j+1 \wedge k+1 \neq j+1 \Rightarrow A[k] \leq A[k+1])$$

Compared to the old quantifier, two facts are lost: $A[j] \leq A[j+1]$ and $A[j+1] \leq A[j+2]$. However, if we put these two facts into the environment, then even after eliminating $A[j+1]$ from them, we retain $A[j] \leq A[j+2]$ by transitivity. Hence, we get a stronger result.

Line 2 of Figure 3 performs this quantifier instantiation. Note that because of the way our partial order $\preceq_\forall$ is defined, this step is not needed to prove completeness. However, it increases precision in practice, so we include it nonetheless.

Soundness and completeness. These two lemmas prove soundness and completeness for $\lceil\exists\rceil$ on $\mathcal{D}_\forall$. They are similar to the corresponding ones for the join operation. They depend on the soundness and completeness of the base domain operations, $\preceq$, $\lceil\exists\rceil$, and $\lfloor\forall\rfloor$. Both are proved in the full version of this paper [10].

LEMMA 3 (Soundness of $\lceil\exists\rceil$ wrt $\overset{\mathbb{T}}{\Rightarrow}$). *If* $\mathcal{E}' := \lceil\exists\rceil(\ell, \mathcal{E})$, *then* $\mathcal{E} \overset{\mathbb{T}}{\Rightarrow} \mathcal{E}'$ *and* $\ell \notin_{\mathcal{E}} \mathcal{E}'$.

LEMMA 4 (Completeness of $\lceil\exists\rceil$ wrt $\preceq_\forall$). *If* $\mathcal{E}'' := \lceil\exists\rceil(\ell, \mathcal{E})$, *then for all* $\mathcal{E}'''$ *s.t.* $\mathcal{E} \preceq_\forall \mathcal{E}'''$ *and* $\ell \notin_{\mathcal{E}} \mathcal{E}'''$, *it is the case that* $\mathcal{E}'' \preceq_\forall \mathcal{E}'''$.

3.3 Miscellaneous Transfer Functions

Conditionals. We assume that all conditionals in the program are converted to `assume` statements. The right side of Figure 2 shows the transfer function for `assume`. It simply adds the assumed condition to the environment.

Partial order. When computing a fixed point, it is necessary to check if one quantified domain element is below another in the partial order. The definition of $\preceq_\forall$ is purely in terms of $\preceq$. Hence, the check for $\preceq_\forall$ is easily implemented using an implementation of $\preceq$. The soundness and completeness of this implementation follows directly from the soundness and completeness of $\preceq$ and the definition of $\preceq_\forall$.

3.4 Quantifier Introduction and Merging

Quantifiers are introduced by finding facts in the environment that match a given set of templates, as described in Section 2.2. This step takes place before a join, as joins may throw away facts from the environment unless they are quantified first. This step clearly preserves soundness. It is complete according to our lattice $\preceq_\forall$, since the resulting quantified element is actually *below* the original one in the lattice.

A related operation, merging two quantifiers into a single one, is sometimes desirable. Frequently, after introducing new quantifiers from the environment, we get a fact like:

$$\mathcal{E} := \forall k(k = 0 \Rightarrow A[k] = 0) \wedge \forall k(k = 1 \Rightarrow A[k] = 0)$$

Since our lattice $\preceq_\forall$ essentially treats different quantifiers independently, the analysis may become more precise if we merge these facts into a single one:

$$\mathcal{E}' := \forall k(0 \leq k \leq 1 \Rightarrow A[k] = 0)$$

We merge two quantifiers (in a process called `Merge`) when their right-hand sides match the same template with the same substitutions. In this case, we can write these quantifiers as $\forall U(F_1 \Rightarrow e)$ and $\forall U(F_2 \Rightarrow e)$. To merge the guards F_1 and F_2, we compute an under-approximation of their disjunction (essentially unioning them together). In Section 4, we describe how to implement an operator $\lfloor\vee\rfloor(F_1, F_2, E)$ that under-approximates disjunction in an environment E. Given this function, we can eliminate the original quantifiers and replace them with $\forall U(F \Rightarrow e)$, where $F = \lfloor\vee\rfloor(F_1, F_2, E)$. This transformation is sound (complete) assuming $\lfloor\vee\rfloor$ is sound (complete).

3.5 Widening, Termination, and Complexity

Widening is used in abstract interpreters to ensure termination. We define a widening operator, $\triangledown_\forall$, on the quantified domain using a widening operator, $\triangledown$, on the base domain and an operator, $\triangle$, on the base domain that is dual of the widening operator.

DEFINITION 2 (Dual Widening). *An operator $\triangle(F_1, F_2)$ is a dual widening operator if (i) $\triangle(F_1, F_2) \preceq F_1$, (ii) $\triangle(F_1, F_2) \preceq F_2$, and (iii) for every infinite sequence $F_1 \succeq F_2 \succeq \cdots$, the sequence $F'_1, F'_2, F'_3, \ldots$, where $F'_1 := F_1$ and $F'_i := \triangle(F'_{i-1}, F_i)$ (for $i \geq 2$), is not strictly decreasing.*

A trivial $\triangle$ operator is one that always returns `false`.

Given quantified abstract domain elements $\mathcal{E}_1$ and $\mathcal{E}_2$, the *widening operator* $\triangledown_\forall(\mathcal{E}_1, \mathcal{E}_2)$ returns $\mathcal{E}_3$. Let $\mathcal{E}_i$ be written $E_i \wedge \bigwedge_{j=1}^{n_i} \forall U(F_{ij} \Rightarrow e_{ij})$. We assume that the universal variables in e_{1j} and e_{2k} have been appropriately matched up. Then $\mathcal{E}_3$, the widening result, is defined as follows.
(a) E_3 is $\triangledown(E_1, E_2)$,
(b) if $E_2 \not\preceq E_1$ (that is, E_2 is not equivalent to E_1), then n_3 is equal to n_2 and for each $k = 1, \ldots, n_3$, e_{3k} is e_{2k} and F_{3k} is F_{2k}.
(c) if $E_2 \preceq E_1$ (that is, E_2 is equivalent to E_1), then n_3 is equal to n_2 and for each $k = 1, \ldots, n_3$, if, for some $j \in \{1, \ldots, n_1\}$, $e_{1j} \preceq e_{2k}$ and $E_1 \sqcap F_{2k} \preceq E_1 \sqcap F_{1j}$, then e_{3k} is equal to $\triangledown(e_{1j}, e_{2k})$ and F_{3k} is $\triangle(E_1 \wedge F_{1j}, E_1 \wedge F_{2k})$; otherwise, $F_{3k} := F_{2k}$ and $e_{3k} := e_{2k}$.

Termination. We now establish termination of our abstract interpreter. Suppose that it does not terminate. This can happen only if we get an infinite chain of successively weaker facts,

$$\mathcal{E}_1 \preceq_\forall \mathcal{E}_2 \preceq_\forall \mathcal{E}_3 \preceq_\forall \cdots,$$

where $\mathcal{E}_i$ is $E_i \wedge \bigwedge_{j=1}^{n_i} \forall U(F_{ij} \Rightarrow e_{ij})$. This infinite chain and the the definition of $\preceq_\forall$ together imply that, $E_1 \preceq E_2 \preceq E_3 \preceq \cdots$. The widening step (a) guarantees that the base abstract interpreter always terminates, and hence there is a finite m such that all E_i's, for $i \geq m$, are (logically) equivalent.

Now let us consider the quantified facts in the above infinite chain starting from the m-th element. The infinite chain above (and the definition of $\preceq_\forall$) implies that we will have a chain of quantifiers,

$$\forall U(F_{mj} \Rightarrow e_{mj}), \forall U(F_{m+1,j'} \Rightarrow e_{m+1,j'}), \cdots,$$

such that (we assume universal variables have been renamed and made equal)

$$\begin{array}{c|c} E_m \sqcap e_{mj} \preceq e_{m+1,j'} & E_m \sqcap F_{m+1,j'} \preceq F_{m,j} \\ E_{m+1} \sqcap e_{m+1,j'} \preceq e_{m+2,j''} & E_{m+1} \sqcap F_{m+2,j''} \preceq F_{m+1,j'} \\ \vdots & \vdots \end{array}$$

We know that all E's in the above are equivalent to E_m. Hence, it follows that,

$$\begin{array}{ccccccc} E_m \sqcap F_{m,j} & \succeq & E_m \sqcap F_{m+1,j'} & \succeq & E_m \sqcap F_{m+2,j''} & \succeq & \cdots \\ E_m \sqcap e_{m,j} & \preceq & E_m \sqcap e_{m+1,j'} & \preceq & E_m \sqcap e_{m+2,j''} & \preceq & \cdots \end{array}$$

The widening operator, $\triangledown$, on the e's and the dual widening operator, $\triangle$, on the F's guarantee that this chain is not strictly decreasing. This establishes termination of our abstract interpreter.

Complexity. Recall that we use `Merge` to merge multiple quantified facts $\forall U(F_1 \Rightarrow e)$, $\forall U(F_2 \Rightarrow e)$, ..., $\forall U(F_k \Rightarrow e)$ with the same right-hand side, e. In our implementation, we modify the `Merge` rule so that it keeps at most K different quantified facts with the same right-hand side e. Second, by using a finite set of templates to specify $\mathcal{D}_c$ (Section 2.2), we ensure that the number of distinct facts e that can occur on the right-hand side of a quantified fact is bounded, say by M. Assuming M and K are fixed constants, we can show that the complexity of our abstract interpreter is only a polynomial factor over the complexity of the base domain abstract interpreter [10].

4. Under-Approximation Operators

The underapproximation operators, $\lfloor\vee\rfloor$, $\lfloor\wedge\rfloor$, and $\lfloor\forall\rfloor$, are needed by the abstract interpreter for the quantified domain. They are not, in general, reducible to the other standard lattice operators. Hence, we have to develop dedicated algorithms for these operators for each base domain. Here, we present sound *generic* procedures that work for *any* base domain. These procedures are complete only for certain theories, like difference constraints, as will be explained. The following example demonstrates the difficulty of computing under-approximations.

EXAMPLE 5. *In a version of insertion sort without loop unrolling (unlike the one appearing later in the experiments section), the following two quantified domain elements are generated on different paths.*

$$\begin{array}{lll} j = i - 1 & \wedge & \forall u(0 \leq k < i - 1 \Rightarrow A[k] < A[k+1]) \\ j < i - 1 & \wedge & \forall u(0 \leq k < i \Rightarrow A[k] < A[k+1]) \end{array}$$

Both facts say that the array is sorted except at position j, but they say it in different ways. Eventually, these paths are joined together, causing the guards (and their respective environments) to be combined via $\lfloor\wedge\rfloor$.

$$\begin{array}{l} \lfloor\wedge\rfloor((0 \leq k < i - 1,\ j = i - 1),\ (0 \leq k < i,\ j < i - 1)) \\ \quad := \quad \lfloor(j = i - 1 \Rightarrow 0 \leq k < i - 1)\ \wedge\ (j < i - 1 \Rightarrow 0 \leq k < i)\rfloor \end{array}$$

One valid result is $0 \leq k < i \wedge k \neq j$. When used as a guard, we get the following quantifier. It agrees with our intuition that the array is sorted at every position but j.

$$\forall u(0 \leq k < i, k \neq j \Rightarrow A[k] < A[k+1])$$

There are two difficulties in computing the under-approximations. First, the under-approximation result must be an element of a conjunctive domain, so it cannot use any disjunction. Second, each input to the under-approximation must be understood in the context of its environment, and the two inputs may have different environments (as is the case for $\lfloor\wedge\rfloor$).

We solve these difficulties using a technique called *abduction*. Abduction is a process in artificial intelligence that generates an *explanation* for a fact given a set of assumed facts (the environment). We define $\texttt{abduct}(E, F')$ to be the set of all explanations F for a base fact F' in the context of an environment E. Formally,

$$E \wedge F \stackrel{\mathbb{T}}{\Rightarrow} F' \quad \text{and} \quad E \wedge F \text{ is consistent}$$

We require $\texttt{abduct}(E, F')$ to compute the set of all possible explanations F that satisfy the requirement above. An algorithm to compute `abduct` will be given later. Using it, we can now define the two underapproximation functions as follows:

$$\begin{array}{rcl} \lfloor\vee\rfloor(F_1, F_2, E) & := & \texttt{abduct}(E, F_1 \vee F_2) \\ \lfloor\wedge\rfloor((F_1, E_1), (F_2, E_2)) & := & \texttt{abduct}(E_1, F_1) \cap \texttt{abduct}(E_2, F_2) \end{array}$$

We can use these two definitions as algorithms for computing $\lfloor\wedge\rfloor$ and $\lfloor\vee\rfloor$. However, since `abduct` can return many different answers, this algorithm would be fairly inefficient. Instead, we use the standard over-approximations from the base domain to compute $\lceil\wedge\rceil$ and $\lceil\vee\rceil$, and then use `abduct` to add additional constraints until we reach a valid under-approximation. Pseudo-code is given in Figure 5 and Figure 6; the next sections explain these algorithms in detail.

$\lfloor\wedge\rfloor((F_1, E_1), (F_2, E_2))$

1 $F := \lceil\vee\rceil(F_1\lceil\wedge\rceil E_1, F_2\lceil\wedge\rceil E_2)$
2 `Forall` $e_1 \in F_1 - F$:
3 $\quad F := F \wedge \texttt{abduct}(E_1\lceil\wedge\rceil F, e_1)$;
4 `Forall` $e_2 \in F_2 - F$:
5 $\quad F := F \wedge \texttt{abduct}(E_2\lceil\wedge\rceil F, e_2)$;
6 `Return` F;

Figure 5. Underapproximation of conjunction. This operator is used for implementing $\lceil\vee\rceil$ on the quantified domain.

4.1 Under-approximating Conjunction in Two Environments

Under-approximating conjunction is necessary when joining two quantifiers from different domain elements. For a quantified domain element $E_1 \wedge \forall U(F_1 \Rightarrow e)$ and another $E_2 \wedge \forall U(F_2 \Rightarrow e)$, the joined quantified element must have a guard that under-approximates $(E_1 \Rightarrow F_1) \wedge (E_2 \Rightarrow F_2)$ (recall that simply using $F_1 \wedge F_2$ is too imprecise). The function $\lfloor\wedge\rfloor((F_1, E_1), (F_2, E_2))$ computes this under-approximation.

As mentioned in the previous section, $\texttt{abduct}(E_1, F_1) \cap \texttt{abduct}(E_2, F_2)$ is a sound answer for this problem. However, using `abduct` in this way may lead to many different answers, and it would be inefficient. Instead, we start by computing a conjunction of facts that *must* be in the answer, and then add more conjuncts using `abduct` until we have a sound result. Another way to think of this algorithm is that we start with an initial "guess" and then refine it to a valid answer.

An *over*-approximation of the conjunction of guards in their environments is a good place to start, since it is like an under-approximation but less constrained (because the under-approximation implies the over-approximation). Therefore, starting with the over-approximation will not preclude any potential answers; we then add extra constraints by abduction until we find a correct under-approximation.

Fortunately, it is easier to find a good over-approximation of $(E_1 \Rightarrow F_1) \wedge (E_2 \Rightarrow F_2)$ than to find a good under-approximation. We are allowed to assume $E_1 \vee E_2$, since it is known in the join that either one quantified domain element or the other holds. In this case, the following inference is valid.

$$\begin{array}{l|l} (E_1 \Rightarrow F_1) \wedge (E_2 \Rightarrow F_2) & (E_1 \Rightarrow F_1) \wedge (E_2 \Rightarrow F_2) \\ \quad \Rightarrow \quad E_1 \wedge F_1 \quad \text{assuming } E_1 & \quad \Rightarrow \quad E_2 \wedge F_2 \quad \text{assuming } E_2 \\ \hline \multicolumn{2}{c}{\Rightarrow (E_1 \wedge F_1) \vee (E_2 \vee F_2) \quad \text{assuming } E_1 \vee E_2} \end{array}$$

It is easy to over-approximate this formula in the base domain using its join algorithm, $\lceil\vee\rceil$.

In fact, with a bit more examination above, it's clear that when E_1 is disjoint from E_2 (that is, $E_1 \Rightarrow \neg E_2$), then the top formula and the bottom formula are equivalent. In most cases, the two environments are disjoint, usually having forms like $E_1 = E \wedge i = 1$ and $E_2 = E \wedge i = 2$. Due to this fact, we know that $(E_2 \wedge F_2) \vee (E_1 \wedge F_1)$ is not just an over-approximation, it is usually the best over-approximation.

Figure 5 shows the remainder of the algorithm for conjunction in the presence of environments. After computing the over-approximation, it generates more constraints via abduction to ensure that the result is a valid under-approximation. Note that since abduction may generate multiple answers, this algorithm may also have many results. The pseudo-code makes sense if we consider `abduct` as a non-deterministic algorithm, and let the results of $\lfloor\wedge\rfloor$ be the set of all possible results, given the non-determinism of `abduct`.

The algorithm first considers all atomic facts in F_1 that are not already implied by F, the over-approximation. For each such fact, abduction is used to find explanations for why this fact is true in the first environment. For example, if E_1 says $j = 2$ and e is $i = 2$, then one possible abduction result is simply $i = 2$. However, $i = j$ is another possible answer, which may be more precise. The same process is repeated for F_2, using the updated F. Note that the closer the initial over-approximation is to a valid under-approximation, the fewer calls there are to `abduct`, and the quicker the algorithm runs.

EXAMPLE 6. *Consider these inputs, which occur when zero iterations of an array initialization are joined with one iteration.*

$$E_1 := i = 0 \quad F_1 := \mathit{false}$$
$$E_2 := i = 1 \quad F_2 := j = 0$$

The procedure $\lfloor\wedge\rfloor((F_1, E_1), (F_2, E_2))$ will compute $j = 0 \wedge i = 1$. This result is obtained in Line 1, and the other lines do not contribute to the result.

Similarly, the following occurs when the first iteration of an array initialization loop is joined with the second iteration.

$$E_1 := i = 1 \qquad F_1 := j = 0$$
$$E_2 := i = 2 \quad F_2 := 0 \leq j < 2$$

In this case, the procedure will compute $0 \leq j < i$. Again, this result is obtained in Line 1, and the other lines do not contribute to the result.

The following example illustrates the importance of the results returned by abduction.

EXAMPLE 7. *After unrolling the inner loop of bubble sort once, we get three different sets of facts. We consider only two sets here. In set one, the first 3 elements of the array are already sorted and we get*

$$\mathcal{E}_1 \quad : \quad \mathit{change} = 0 \wedge A[0] \leq A[1] \leq A[2].$$

In the second set, the array is not sorted initially, but the largest element moves up, and we get

$$\mathcal{E}_2 \quad : \quad \mathit{change} = 1 \wedge A[0] \leq A[2] \wedge A[1] \leq A[2].$$

These facts are generalized to give the following elements of the quantified domain,

$$\mathcal{E}_1 : \mathit{change} = 0 \wedge \forall U(0 \leq u_1 \leq u_2, 1 \leq u_2 \leq 2 \Rightarrow A[u_1] \leq A[u_2])$$
$$\mathcal{E}_2 : \mathit{change} = 1 \wedge \forall U(\underline{0 \leq u_1 \leq 2, u_2 = 2} \Rightarrow A[u_1] \leq A[u_2])$$

where $U := (u_1, u_2)$, and F_1 and F_2 are the underlined formulas in the two elements respectively. When we join these two facts, we have to compute the following underapproximation,

$$\lfloor\wedge\rfloor((F_1, \mathit{change} = 0), (F_2, \mathit{change} = 1))$$

This returns two mutually incomparable answers: $\mathit{change} = 0 \wedge F_1$ and F_2. The first one leads to the invariant which says that if `change` is zero, then the array is sorted. The second one gives the quantified invariant that the last element is the largest element (which is the correctness statement of the inner loop of bubble sort).

The soundness of the procedure in Figure 5 follows directly from the soundness of `abduct`. The completeness follows from the observation that (i) we do not lose any solutions in Line 1 as it is an over-approximation of any possible answer; and (ii) completeness of `abduct` guarantees that we generate all possible explanations.

4.2 Under-approximating Disjunction in an Environment

Under-approximating disjunctions of base facts is useful when two quantifiers from the same quantified domain element are to be combined into a single quantifier. In this case, their guards are combined via $\lfloor\vee\rfloor(F_1, F_2, E)$, where E is the environment and F_1 and F_2 are the guards.

$\lfloor\vee\rfloor(F_1, F_2, E)$

1 $F := \lceil\vee\rceil(F_1\lceil\wedge\rceil E, F_2\lceil\wedge\rceil E)$

2 `Foreach` $e_1 \in F_1 - F$ `and` $e_2 \in F_2 - F$:

3 $\quad F := F \wedge$ `abduct`$(E\lceil\wedge\rceil F, e_1 \vee e_2)$;

4 `Return` F;

Figure 6. Underapproximation of disjunction. This operator is used for implementing $\lceil\wedge\rceil$ on the quantified domain.

As in the algorithm for conjunction, we start with an initial over-approximation of the result and then refine it with more constraints. In this case, the over-approximation is even easier to compute. We could start with $\lceil\vee\rceil(F_1, F_2)$. However, we wish to take advantage of facts that are known to be true in the environment, so instead it is safe to start with $\lceil\vee\rceil(F_1\lceil\wedge\rceil E, F_2\lceil\wedge\rceil E)$.

Figure 6 shows the remainder of the algorithm, which adds more constraints using abduction to get a sound under-approximation, as was done for conjunctions.

EXAMPLE 8. *For the inputs,*

$$F_1 := (1 \le k \le n) \quad F_2 := (k = 0) \quad E := \mathit{true}$$

the procedure $\lfloor\vee\rfloor(F_1, F_2, E)$ returns $0 \le k \le n$. This result is obtained in Line 1, and the other lines do not contribute to the result. We note here that this particular example is reminiscent of the "range merging" approach of [13].

As another example, consider the inputs

$$F_1 := (j = 0, k = 1) \quad F_2 := (j = k) \quad E := (0 \le k \le 1)$$

In this case, the procedure $\lfloor\vee\rfloor(F_1, F_2, E)$ will compute $0 \le j \le k$. Again, this result is obtained in Line 1 itself.

The next example shows how abduction is useful.

$$F_1 := (0 \le i < 10) \quad F_2 := (11 \le i < 20) \quad E := \mathit{true}$$

The join in Line 1 produces $F = (0 \le i < 20)$. The loop in Line 2 has a single iteration, with $e_1 = (i < 10)$ and $e_2 = (11 \le i)$. These are the facts that are not implied by F. Line 3 calls `abduct`$((0 \le i < 20), (i < 10) \vee (i \ge 11))$. *It returns the answer $i \ne 10$, which is correct because $(0 \le i < 20) \wedge (i \ne 10)$ implies (and thus under-approximates) $(i < 10) \vee (i \ge 11))$.*

The soundness of the procedure in Figure 6 follows directly from the soundness of `abduct`. The completeness follows from the fact that (i) Line 1 generates an over-approximation and does not forbid any solution; while (ii) the completeness of `abduct` guarantees that all solutions are generated.

4.3 Generating Explanations via Abduction

The previous two sections use abduction to generate alternate explanations for a formula in the context of an environment. This section explains how to implement abduction. The simplest sound implementation of `abduct`$(E, e_1 \vee \ldots \vee e_k)$ can return any e_i. However, a more sophisticated abduction procedure that takes advantage of facts from the environment will improve the precision of the rest of the system. The abduction algorithm we describe in this section is even complete for certain base domains.

Our abduction algorithm is based on the following principle.

$$(E \wedge (\bigwedge_i \neg e_i) \stackrel{\mathbb{T}}{\Rightarrow} e) \text{ implies } (E \wedge \neg e \stackrel{\mathbb{T}}{\Rightarrow} \bigvee_i e_i)$$

That is, the negation of any fact e implied by E and the negation of all the e_is is a valid answer for `abduct`. Therefore, we take the set of all such facts that are maximally strong and return their negations. As an optimization, we ignore all facts implied by the environment alone, since they are not useful. Figure 7 shows pseudo-code for this algorithm.

`abduct`(E, ϕ)

1 `let` ϕ `be` $e_1 \vee \cdots \vee e_k$, `where` e_i `is an atomic fact`

2 `ans` $:= \emptyset$;

3 `Foreach maximally strong`

$\quad$ `atom` e `s.t` $E \wedge \bigwedge_i \neg e_i \Rightarrow e$ `and` $E \not\Rightarrow e$

4 $\quad$ `ans` $:=$ `ans` $\cup \{\neg e\}$;

5 `return(ans);`

Figure 7. Abductive reasoning in the base domain using forward reasoning. A literal is an atomic formula or its negation. The `abduct` function is used for implementing all underapproximation operators. It assumes that negated atomic formulas can be represented in the base domain.

EXAMPLE 9. *Consider computing* `abduct`(E, ϕ), *where*

$$E := j = 2 \quad \phi := i = 2$$

We will search for all maximally strong facts e such that $j = 2 \wedge i \ne 2$ implies e (and $j = 2$ alone does not imply e). The most obvious answer is $i \ne 2$, but another potentially useful answer is $i \ne j$. We return the negations of both these facts as possible answers.

Implementing this algorithm requires two non-trivial properties of the base domain. First, it must support negation of atomic facts. All of the domains that we have implemented for our experiments can perform sound reasoning about negated facts. Second, the domain must be able to enumerate the maximally strong facts implied by a set of formulas. It is easy to do this for saturation-based domains like difference constraints or reachability.

If one of these properties is not satisfied by the base domain, it is still possible to implement a simpler but less complete abduction procedure, such as the trivial one that returns its inputs. However, we believe that most domains used in practice do fulfill these requirements.

EXAMPLE 10. *Let us revisit Example 5 wherein we had to compute*

$$\lfloor\wedge\rfloor((0 \le u < i - 1,\ j = i - 1),\ (0 \le u < i,\ j < i - 1)).$$

In Line 1, we compute a join and get

$$F := (0 \le u < i, j \le i - 1)$$

This is not a sound underapproximation. We need to strengthen it as in Line 3 and Line 5 of Figure 5. For example, we will need to strengthen it by `abduct`$(F \wedge j = i-1, u < i-1)$. *We compute this by computing atomic facts implied by $F \wedge j = i - 1 \wedge u \ge i - 1$. Some such atomic facts are $u = i-1, u = j, \ldots$. Using the second atomic fact, $u = j$, and adding its negation to F results in the answer $0 \le u < i, u \ne j$ that is used in Example 5.*

We remark here that the inductive invariant generated in Example 5 is not present in the code in any way. It took 3 hours of manual effort to generate this precise inductive invariant, which is automatically generated in our approach.

The soundness of the abduction procedure (Figure 7) is obvious from its description. There is a potential loss of completeness on Line 3 since we only search for *atomic* (and not arbitrary) e. However, the procedure in Figure 7 can be shown to be complete for certain theories [10].

4.4 Under-approximating $\forall$ in an Environment

The underapproximation operator $\lfloor\forall\rfloor$ requires computing an explanation for a fact in the context of a known fact such that the explanation does not contain the quantified term. Recall that $\lfloor\forall\rfloor(\ell, F, E)$ under-approximates the logical formula $\forall\ell(E \Rightarrow F)$. The function $\lfloor\forall\rfloor(\ell, F, E)$ can be computed using the procedure

```
⌊∀⌋(ℓ, F, E)
1  let F be e_1 ∧ e_2 ∧ ··· ∧ e_k, where e_i is atomic
2  ans := true;
3  for i = 1, ..., k do
4     Nondeterministically choose maximally strong
        atom e in ⌈∃⌉(ℓ, ¬e_i, E)
5     ans := ans ∧ ¬e;
6  return(ans);
```

Figure 8. Underapproximation of $\forall$ Quantification. This operator is used in implementing $\lceil\exists\rceil$ on the quantified domain.

given in Figure 8. The soundness of the procedure follows from noting the following logical equivalences:

$$\texttt{ans} \Rightarrow \forall \ell (E \Rightarrow F) \quad \text{iff} \quad (\exists \ell . (E \wedge \neg F)) \Rightarrow \neg \texttt{ans}$$

Assuming that the abstract domain can precisely represent literals (atomic formulas and negated atomic formulas), it is easy to see that the procedure of Figure 8 always returns an underapproximation of the forall quantified formula.

5. Experiments

In our experiments, we instantiated our quantified domain by two base domains. The first base domain was the logical domain defined by the combination of linear arithmetic and uninterpreted symbols [11]. We extended this domain to include some disequality reasoning. Disequalities permit us to represent disjoint ranges in a single quantifier guard, rather than splitting them into multiple quantifiers. The second base domain we used was a saturation-based domain that has inference rules for reasoning about data structure reachability. One appeal of our approach is that our quantified domain can switch from linear arithmetic/uninterpreted functions to reachability with little effort. This section first gives an overview of the reachability domain, and then describes our experiments over both domains.

5.1 Reachability

We considered a domain in which each element is a conjunction of atomic facts of the form

$$R(e_1, e_2) \quad | \quad \neg R(e_1, e_2) \quad | \quad e_1 = e_2 \quad | \quad e_1 \neq e_2$$

where each expression e is of the form

$$e ::= x \quad | \quad \texttt{null} \quad | \quad e \rightarrow \texttt{next}$$

The reachability predicate, $R(x, y)$, intuitively denotes that there is a path from the object specified by x to the object specified by y, where the path passes through a set of fixed pointer fields in the objects (e.g., the `next` field). A reachability predicate is commonly used in the analysis of pointer data structures [2].

Given a set of known reachability facts, we saturate it by applying a set of inference rules to add new facts (as described in [2]). Examples of these reachability inference rules for lists include:

$$\begin{aligned} e_2 = e_1 \rightarrow \texttt{next} &\Rightarrow R(e_1, e_2) \\ R(e_1, e_2) \wedge R(e_2, e_3) &\Rightarrow R(e_1, e_3) \\ R(e_1, e_2) \wedge e_1 \neq e_2 &\Rightarrow R(e_1 \rightarrow \texttt{next}, e_2) \end{aligned}$$

The algorithms for meet, join, and existential elimination are beyond the scope of this paper.

We used the generic under-approximation algorithms described in Section 4 for this domain. Despite not being complete in this context, they are precise enough to handle the following examples, which we extracted from our linked list manipulation benchmarks.

Underapproximation of disjunction in an environment. When initializing the data fields of an acyclic singly-linked list to zero, we arrive at the following quantified domain element. It represents the case where property P has been established for the first two list elements (recall that we use loop unrolling for better performance). Variable ℓ is the head of the list and the iteration pointer p points two elements beyond it. We would like to merge the two quantifiers.

$$p = \ell \rightarrow \texttt{next}^2 \wedge \forall u (u = \ell \Rightarrow P(u)) \wedge \forall u (u = \ell \rightarrow \texttt{next} \Rightarrow P(u))$$

This requires computing an underapproximation of disjunction of $F_1 = (u = \ell)$ and $F_2 = (u = \ell \rightarrow \texttt{next})$ in presence of the environment $E = (p = \ell \rightarrow \texttt{next}^2)$. The answer that is computed by our generic under-approximation algorithm is as follows. It says that P is true for all nodes between ℓ and $\ell \rightarrow \texttt{next}$, inclusive, which is what we desire.

$$\lfloor\vee\rfloor(F_1, F_2, E) \quad = \quad R(\ell, u) \wedge R(u, \ell \rightarrow \texttt{next})$$

Underapproximation of conjunction in two environments. The preceding example shows the case when the first two elements have been initialized. Now consider joining this case with the one where only the first element has been initialized.

$$\begin{aligned} \mathcal{E}_1 &:= p = \ell \rightarrow \texttt{next} \wedge \forall u (u = \ell \Rightarrow P(u)) \\ \mathcal{E}_2 &:= p = \ell \rightarrow \texttt{next}^2 \wedge \forall u (R(\ell, u) \wedge R(u, \ell \rightarrow \texttt{next}) \Rightarrow P(u)) \end{aligned}$$

This requires computing an underapproximation of conjunction of $F_1 = (u = \ell)$ and $F_2 = R(\ell, u) \wedge R(u, \ell \rightarrow \texttt{next})$ in presence of their environments $E_1 = (p = \ell \rightarrow \texttt{next})$ and $E_2 = (p = \ell \rightarrow \texttt{next}^2)$.

$$\lfloor\wedge\rfloor((F_1, E_1), (F_2, E_2)) \quad = \quad R(\ell, u) \wedge R(u \rightarrow \texttt{next}, p)$$

The above answer gives the guard for the quantified fact that says that the property P is true for all nodes between ℓ and the node immediately before p, inclusive. This is an inductive loop invariant that can now be used to verify that the loop establishes P for all elements.

5.2 Results

We applied our techniques to automatically generate invariants for some standard sorting routines and other similar problems that have been studied as challenge problems for generation of quantified invariants in the literature. We report on some of the examples in detail in Table 1. Benchmarks were run on a 3 GHz Intel processor with 2 GB of RAM. In the array examples, we used fixed variables instead of Roman variables in the templates to increase performance. We also unrolled loops two times to avoid discovering spurious invariants.

The first few examples in Table 1 have been taken from [13]. *ArrayInit* is the example from our Section 1. Our tool discovers the invariant that all array elements are initialized to 0. It uses a template $\texttt{A}[\alpha] = 0$. *VarArg* (from [13]) counts the number of non-null entries in a list, and then scans over that many list entries again, asserting non-nullness. We use a template $\texttt{A}[\alpha] \neq 0$. *ArrayCopy* (from [9]) simply copies the contents of one array into another, distinct array. Our tool discovers the invariant that the two arrays are the same. We use the template $\texttt{A}[\alpha] = \texttt{B}[\alpha]$. *ArrayCopyProp* (from [13]) starts with the pre-condition that a source array has only non-zero elements. It copies this array into a distinct destination array. Our tool checks that the destination array has only non-zero elements. We use the template $\texttt{A}[\alpha] = \texttt{B}[\alpha]$. *Find* (from [8]) searches an input array for a specific value, setting a flag if it is found. After the search, our tool discovers that if the flag is not set, the value is not present in the array. We use the template $\texttt{A}[\alpha] \neq \texttt{v}$. *PartialInit* (from [9]) copies those indices of a source array for which the source array's value is positive into a target array. Our tool discovers that the source array's values are all positive at the indices stored in the target array. We use the template $\texttt{A}[\texttt{B}[\alpha]] = 0$. *Partition* (from [1]) copies the zero and non-zero elements of a

source array into two different arrays. Our tool discovers that the two destination arrays have entirely zero or non-zero entries. We use the two templates $\mathtt{B}[\alpha] = 0$ and $\mathtt{C}[\alpha] \neq 0$.

The next four examples constitute the inner loop of various sorting algorithms. For each of these examples, we were able to discover the inductive invariant required for proving that the final array is sorted. We use three kinds of templates in these examples: $\mathtt{A}[\alpha] \leq e$, $e \leq \mathtt{A}[\alpha]$, and $\mathtt{A}[\alpha] \leq \mathtt{A}[\alpha + 1]$. (Caveat: Because of a bug in our quantified instantiation routine, we had to hand-annotate the results of one quantifier instantiation in both InsertionSort and SelectionSort and the timing results do not take this into account. However, we do not anticipate any significant increase in the timing of these examples with the fix of the quantifier instantiation routine.)

The final four examples are standard acyclic singly-linked list manipulation routines. We were primarily limited by our current implementation of the reachability base domain that can only handle one `next` link. However, our technique theoretically works equally well for other data-structures such as cyclic linked lists, doubly-linked lists, and trees, and we have manually traced our algorithm to successfully discover similar invariants for these data-structures.

All four list examples are designed to establish or preserve the invariant that all list elements have a `data` field set to zero. They all use the template that $\alpha{\rightarrow}\mathtt{data} = 0$. The insert and remove routines start with a pre-condition that ℓ is a list with `data` fields set to zero, and t is a pointer into the list. They insert or remove at position t. The list init example assumes a valid list on entry, and we discover that its `data` fields are zero on exit. The list create example creates a list from scratch, and we discover that the result is a list, whose `data` fields are zero. (Caveat with the list create, insert, and remove examples: a bug in our reachability saturation procedure forced us to "hard-code" the results of some reachability domain operations. As a result, the timings for these benchmarks may under-estimate the true time.)

The total time take by some of these examples seems rather large compared to the size of these examples. However, it must be noted that we have a very naive and inefficient saturation based implementation of the underlying difference constraints domain and reachability domain. This is corroborated by the fact that the time taken to do abstract interpretation over simply the base quantifier-free abstract domain in those examples is also large. An interesting attribute to read from the tables in that case is the ratio of the time taken to do abstract interpretation over quantified domain and the corresponding quantifier-free domain.

Procedure	$\mathcal{D}_\forall$ Time	$\mathcal{D}$ Time	Ratio
ArrayInit	3.2	1.5	2.1
VarArg	4.1	2.0	2.1
ArrayCopy	5.5	2.2	2.5
ArrayCopyProp	11.3	6.8	1.7
PartialInit	12.0	6.1	2.0
Find	24.6	8.3	3.0
Partition	73.0	22.7	3.2
InsertionSort (inner loop)	35.9	2.0	18
QuickSort (inner loop)	42.2	4.5	9.4
SelectionSort (inner loop)	59.2	8.1	7.3
MergeSort (inner loop)	334.1	73.5	4.5
List Remove	20.5	1.4	14.6
List Insert	23.9	1.4	17.1
List Init	24.5	1.9	12.9
List Create	42.0	3.4	12.4

Table 1. Benchmark results. Times are in seconds. The third column reports time for the mode wherein abstract interpretation is performed over the base quantifier-free abstract domain.

6. Related Work

Automatic generation of (universally) quantified invariants has been a topic of extensive research [8, 5, 15, 9, 3, 1, 13]. The various approaches differ in the extent of user guidance assumed. In one class of methods, the user specifies the predicates and the tool searches for the "right" boolean structure that gives an inductive invariant [8, 3, 15]. In the dual approach, the user specifies the boolean structure and the tool searches for the "right" predicates [1]. In our approach, the user is not required to specify either the boolean structure or the predicates.

[8] require that all atomic formulas in the quantified invariants be given. Given such a set of atomic formulas, their approach generates a quantified invariant that can be expressed using only those atomic formulas. For example, to generate the quantified invariant $\forall u(0 \leq u < n \Rightarrow A[u] = 0)$ for the array initialization procedure, their method requires that the three predicates $u \geq 0$, $u < n$, and $A[u] = 0$ be given. Our approach, on the other hand, would generate the quantified invariant with only the template $a[\alpha] = 0$. In our approach, the antecedent, $0 \leq u < n$, is *learned* using join of facts generated in 0, 1, or 2 iterations of the loop.

[13] have described an interpolation based technique to generate quantified invariants. They have instantiated their technique to discover *range predicates* that capture properties of sequences of array properties (e.g., facts such as "the elements of the array M from index i through j are positive"). The range predicate $R(t_1, t_2, p)$ denotes $\forall i(t_1 \leq i < t_2 \Rightarrow p)$, where i is a free variable in p. This can not represent, for example, that the contents of a two-dimensional array is initialized: $\forall i, j(0 \leq i < n, 0 \leq j < m \Rightarrow A[i][j] = 0)$. Representing this fact would require a new and different range predicate with its own set of axioms. It is not clear what is a good choice for a complete set of such range predicates. There will always be interesting examples that are not representable using a predefined range predicate.

The logic flow analysis by [16] also maintains quantified facts about contiguous ranges of array elements. It does not use predicate abstraction, but it still is unable to handle complex guards like those that appear in the sorting examples.

[1] combine invariant generation and predicate abstraction techniques using path invariants. While this technique is independent of any particular invariant generation technique, [1] describe a particular template-based approach for generating quantified invariants. The template-based technique reduces the search of an invariant to constraint solving over a large number of (unknown) variables used to specify the templates. While it is sound in general, it is complete only for a very specific form of invariants, $\forall U(\wedge_i t_i(X) \leq u_i \leq t'_i(X) \Rightarrow r(X, U))$, where X are the program variables and t_i, t'_i, r are linear expressions (with parameterized coefficients) with the provision that array reads of the form $A[u_i]$ can occur in r. Note that correctness of sorting routines can not be stated using the above template since it disallows comparison between quantified variables. Furthermore, the translation from the original templates to the final constraints is not generic and has to be freshly worked out for each base domain. While the high-level approach (choose templates and translate to constraints) is general, the details are not general.

One fundamental difference of our framework for generating quantified invariants, compared to these other works [8, 15, 3, 1, 13], is that it is based on abstract interpretation. This offers a more efficient methodology of discovering useful program invariants, albeit at the cost of merging facts at join points (which may result in loss of some precision). While abstract interpretation based

	Requirements on base domain $\mathcal{D}$	Section(s)
R1	Compute $\lceil\vee\rceil$, $\lceil\wedge\rceil$, $\lceil\exists\rceil$, $\preceq$, $\bigtriangledown$	3.1, 3.3, 3.2, 3.3, 3.5
R2	Compute $\lfloor\wedge\rfloor$, $\lfloor\vee\rfloor$, $\lfloor\forall\rfloor$, $\bigtriangleup$	3.1, 3.4, 3.2, 3.5
R3	Compute `NotEffect`, Matching	3.2
R4	Templates, Represent equality	3.4 (2.2)
R2'	Compute consequences	4

Table 2. Base domain requirements. In addition to a base abstract interpreter (R1), we need to compute under-approximations (R2), compute `NotEffect` and match terms in $\mathcal{D}$ (R3), and represent equality and match terms with templates (R4). We can replace R2 by R2' if we use our under-approximation procedures.

methods have been thoroughly investigated for unquantified domains [6, 14, 7], they have seen only limited exploration for quantified domains [4, 5, 9]. In the work of [5] and [9], quantified facts are encoded as new *unquantified* predicates and lattice operators are defined directly on these *unquantified* predicates. The automation achievable via the under-approximation operators has to be painfully hard-coded in these methods.

[5] has used an abstract domain consisting of elements of the form $\langle lt(t, a, b, c, d), r\rangle$, which informally says that all elements of t between indices a and b are less than any element of t between indices c and d. In our notation, this element is represented as,

$$r(a, b, c, d) \wedge \forall u_1, u_2 (a \leq u_1 \leq b \wedge c \leq u_2 \leq d \Rightarrow t[u_1] \leq t[u_2])$$

(Since the auxiliary variables a, b, c, d always have definitions in terms of program variables, they can be eliminated and we get exactly an element of our quantified abstract domain.) [5] presents procedures to compute the abstract logical operators on this domain, and it can be easily verified that the underapproximation calculation is built into the definitions. Using this abstract domain, [5] generates invariants for a sorting routine. The work of [9] is similar in spirit to the work of [5]. They use an abstract domain $\langle P, \Omega, \Delta\rangle$, where P is a partition of an array's indices, Ω associates each partition with a numerical abstract domain element, and Δ is a valuation of some given abstract predicates on each partition. Again, this abstract domain element can be expressed in our quantified abstract domain as

$$\bigwedge_{\pi \in \Pi} \forall u (u \in \pi \Rightarrow \Omega(\pi)) \wedge \bigwedge_{\pi \in \Pi} \forall u_1, u_2 (u_1 \in \pi \wedge u_2 \in \pi \Rightarrow \Delta(\pi)),$$

where $u \in \pi$ essentially denotes a conjunction of constraints, and $\Omega(\pi)$ and $\Delta(\pi)$ denote some atomic facts. It should again be noted that the antecedent is carefully fixed here and [9] provide dedicated descriptions of the transfer functions.

Our paper uniformly generalizes these specific abstract domains (that represent quantified facts) by explicitly making quantified facts first-class objects. It builds on our earlier work [12] – that developed an abstract quantified domain with *must* and *may* equalities – but goes significantly beyond by developing a precise formal theory for abstract interpretation on generic quantified domains using under-approximation operators. We also present generic procedures for computing transfer functions for the quantified domain – thus automating a lot of the manual effort in building special domains such as in [4, 5, 9].

7. Conclusion

Quantified abstract domains provide the expressive power required to state universally quantified invariants of unbounded data-structures. We formally define quantified abstract domains by using base domains – whose facts are used to build universally quantified facts – as parameters. We provide a general framework for building abstract interpreters over such quantified domains. This is achieved using a rich interface provided by the base domain (Table 2). This interface consists of the standard over-approximation functions along with additional functions that compute under-approximations of logical boolean operators. The under-approximation functions play a foundational role in the process of invariant generation.

We also instantiate this framework to obtain two specific abstract interpreters – one for programs that manipulate arrays and the other for programs that manipulate heap-based linked data-structures. These abstract interpreters are used to successfully generate quantified invariants stating correctness of several procedures that work on arrays and lists.

Acknowledgments. We thank Krishna Mehra and Lakshmisubrahmanyam Velaga for their help implementing the base domains.

References

[1] D. Beyer, T. Henzinger, R. Majumdar, and A. Rybalchenko. Path invariants. In *PLDI*, 2007.

[2] J. D. Bingham and Z. Rakamaric. A logic and decision procedure for predicate abstraction of heap-manipulating programs. In *VMCAI*, pages 207–221, 2006.

[3] A. R. Bradley, Z. Manna, and H. Sipma. What's decidable about arrays? In *VMCAI*, volume 3855 of *LNCS*, pages 427–442. Springer, 2006.

[4] P. Cerny. Verification par interpretation abstraite de predicats parametriques. Master's thesis, Univ. Paris VII & Ecole normale superieure, Paris 20, 2003.

[5] P. Cousot. Verification by abstract interpretation. In *Verification: Theory and Practice*, volume 2772 of *LNCS*, pages 243–268, 2003.

[6] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 234–252, 1977.

[7] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL*, pages 84–97, 1978.

[8] C. Flanagan and S. Qadeer. Predicate abstraction for software verification. In *POPL*, pages 191–202, 2002.

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

[10] S. Gulwani, B. McCloskey, and A. Tiwari. Lifting abstract interpreters to quantified logical domains. Technical Report MSR-TR-2007-87, Microsoft Research, July 2007.

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

[12] S. Gulwani and A. Tiwari. Static analysis of heap manipulating low-level software. In *CAV*, LNCS, 2007.

[13] R. Jhala and K. McMillan. Array abstractions from proofs. In *CAV*, 2007.

[14] M. Karr. Affine relationships among variables of a program. In *Acta Informatica*, pages 133–151. Springer, 1976.

[15] S. K. Lahiri and R. E. Bryant. Indexed predicate discovery for unbounded system verification. In *CAV*, pages 135–147, 2004.

[16] M. Might. Logic-flow analysis of higher-order programs. In *POPL*, pages 185–198, 2007.