Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 267 (2010) 43–57
www.elsevier.com/locate/entcs

View-Augmented Abstractions

Matt Elder[a,1], Denis Gopan[b,2] and Thomas Reps[a,b,3]

[a] *Computer Sciences Department, Univ. of Wisconsin; Madison, WI; USA*

[b] *GrammaTech, Inc.; Ithaca, NY; USA*

Abstract

This paper introduces *view-augmented abstractions*, which specialize an underlying numeric domain to focus on a particular expression or set of expressions. A view-augmented abstraction adds a set of materialized views to the original domain. View augmentation can extend a domain so that it captures information unavailable in the original domain. We show how to use finite differencing to maintain a materialized view in response to a transformation of the program state. Our experiments show that view augmentation can increase precision in useful ways.

Keywords: Numeric abstract domains, view maintenance, abstract-interpretation precision.

1 Introduction

Program analysis involves learning the potential values of a program's variables, together with relationships among the variables' values. A common approach to program analysis is to design abstract domains that can infer whether an arbitrary relation of a given class holds, e.g., polynomial equalities [20] or inequalities [2] of bounded degree. However, adding precision "uniformly" in this manner is usually expensive [20]: typically, the more complex the class is, the more expensive the domain is.

One challenge to maintaining precision is that the analyzer often needs to find information about particular conditions or expressions. For example:

- Reachability analysis benefits from information of the values of conditions.
- Assertion checking requires information about asserted conditions.
- Buffer overrun and underrun analyses require information about array-access expressions.

[1] Email: elder@cs.wisc.edu
[2] Email: gopan@grammatech.com
[3] Email: reps@cs.wisc.edu

doi:10.1016/j.entcs.2010.09.005

The relationships needed to obtain such information take specific, often complex, forms, depending on the actions of the program of interest. Uniformly increasing the precision of the underlying numeric abstraction to capture the *entire* class of more complex expressions is likely to severely encumber the overall analysis. For instance, to compute an index into a packed upper-triangular matrix involves the square of an index variable. To capture that relationship, one needs to use a domain that can handle polynomials; however, no scalable domains do so.

In contrast to a uniform approach to tracking complex relationships, it should be less expensive to augment an abstract domain to track the values of a few relevant complex expressions. Toward this end, the paper focuses on the following problem:

> *How can a given abstract domain be augmented inexpensively to track information that characterizes the value of a given expression?*

By "inexpensively," we mean that the solution should be

- *Parsimonious*: Only a small amount of additional information should be tracked, such as the values of a small number of auxiliary variables.
- *Delegating*: Nothing "fundamental" should change about the abstraction in use. In particular, all operations performed on auxiliary variables should be performed using existing operations of the underlying domain.

We address these issues in the context of numeric abstract domains by introducing a mechanism to create and maintain *abstract views* in numeric abstract domains. Abstract views take advantage of the following principle:

Observation 1.1 (Instrumentation Principle) *Suppose that $S^\sharp$ is an abstract value that represents the set of concrete states S. By explicitly storing in $S^\sharp$ an abstraction of the values that an expression e has in S, it is sometimes possible to extract more precise information from $S^\sharp$ than can be obtained just by the abstract evaluation of e with respect to $S^\sharp$.*

Like a materialized view in a database [11], which provides a precomputed answer to a specific relational query, an abstract view maintains—in a fresh auxiliary variable—a value for a specific numeric expression. An abstract view can track information otherwise unrepresentable in the original abstraction. For instance, consider the abstract state $\{x \mapsto [0,5], y \mapsto [0,5]\}$ in the interval domain. Suppose we augment the domain with the view-variable v_{x+y}, which will track the value of $x+y$. By tracking v_{x+y} along with x and y, we might discover that $x+y$ must lie in the interval $[2,7]$. The augmented abstract state is thus $\{x \mapsto [0,5], y \mapsto [0,5], v_{x+y} \mapsto [2,7]\}$ whose concretization (projected onto the x and y axes) is a hexagon, rather than a square (see Fig. 1).

In contrast to the relationships built up during the course of program execution, individual state transformations are typically simple (e.g., `x = x+1`). When tracking views, the challenge is to incorporate the effect of state transformations on the values of complex view expressions (e.g., x^2y^2). Recomputation based on the underlying domain is generally too imprecise [22]. In this paper, we present a systematic framework that automatically updates view-variables, based on finite differencing

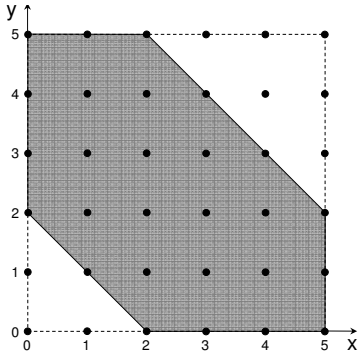

Fig. 1. Shaded hexagon: concretization of $\{x \mapsto [0,5], y \mapsto [0,5], v_{x+y} \mapsto [2,7]\}$ (projected onto the x and y axes). Dotted square: concretization of $\{x \mapsto [0,5], y \mapsto [0,5]\}$.

[9,21].

The idea of augmenting domains with instrumentation values has been used before in predicate-abstraction domains [10], which maintain the values of a given set of Boolean predicates. Numeric domains have also seen extensions by instrumentation variables. Weakly-relational domains [17] extend non-relational domains to maintain information about expressions of the form $x - y$. Octagons [16] extend difference-bounded matrices [8] to maintain information about expressions of the form $\pm x \pm y$. Template Constraint Matrices [25] maintain information about finite sets of linear inequalities using linear programming.

The most closely related work to ours is the work on automatically creating abstract transformers for instrumentation predicates that augment canonical-abstraction domains [22]. However, prior work on augmenting canonical-abstraction domains applies to formulas over a relational vocabulary, and the technique does not apply to expressions over numeric quantities.

The contributions of the paper can be summarized as follows:

- We show how to augment any numeric abstraction with abstract views.
- We give a systematic technique, based on finite differencing, to maintain an over-approximation of a view-variable's value in response to a transformation of the program state. All operations performed on view-variables use existing operations of the underlying abstract domain.
- We report on experiments with a prototype implementation of view-augmented abstract interpretation based on the Apron framework [1].

Organization. Sect. 2 introduces views, and presents the view-maintenance technique that we developed at a semi-formal level. (Some formal development appears in Apps. A and B.) Sect. 3 presents two techniques that are needed to maintain precision during view maintenance. Sect. 4 presents experimental results. Sect. 5 discusses related work. Sect. 6 concludes.

```
if x*x >= 4 then      (a)
    x = x+1           (b)
    assert(x*x > 0)   (c)
```

Fig. 2. An example program

2 Views and View Maintenance

Example 2.1 To demonstrate the utility of view-augmentation, consider an analysis of the program fragment in Fig. 2 using the interval abstract domain. Suppose that the analyzer has discovered that $x \mapsto [-2, 2]$ holds just before line `(a)`. Because of the test on line `(a)`, $x \in \{-2, 2\}$ just before line `(b)`. Consequently, just before line `(c)` we have $x \in \{-1, 3\}$, and hence the assertion at line `(c)` is always true (because $-1 * -1 > 0$ and $3 * 3 > 0$).

Unfortunately, the interval domain cannot express with sufficient precision the information $x \in \{-2, 2\}$, which holds just before line `(b)`: the most precise interval-domain fact that over-approximates $x \in \{-2, 2\}$ is $x \mapsto [-2, 2]$, which represents the set $\{-2, -1, 0, 1, 2\}$. Thus, just before line `(c)` we have $x \mapsto [-1, 3]$, and consequently, because $0 \in \gamma([-1, 3])$, the value of `x*x` can equal 0 according to the interval domain—even if `x*x` is evaluated using the most-precise squaring operation for intervals. Consequently, the analyzer cannot prove the assertion at line `(c)`.

Now suppose that we augment the interval abstraction with the view-variable v_{x*x} to track the value of the expression `x*x`. Augmenting the interval abstraction means that we augment abstract states with an additional variable (i.e., an auxiliary dimension) that tracks the value of v_{x*x}. Again, suppose that the analyzer has discovered that $x \mapsto [-2, 2]$ holds just before line `(a)`.

The view-variable must start with some sound initial value. We can obtain such a value by directly evaluating `x*x` $= [-2, 2] \cdot [-2, 2] = [-4, 4]$. Thus, the initial abstract state is $\{x \mapsto [-2, 2], v_{x*x} \mapsto [-4, 4]\}$. (If `x*x` were interpreted as a squaring operation, we would obtain $\{x \mapsto [-2, 2], v_{x*x} \mapsto [0, 4]\}$.)

Just before line `(b)`, we would like the value of view-variable v_{x*x} to capture the assumption `x*x >= 4`. Moreover, during the abstract interpretation of line `(a)`, the operations on view-variable v_{x*x} should all be standard operations supported by the interval abstract domain. To obtain this effect, we interpret the expression `x*x` as an access on the view-variable v_{x*x}, and express the abstract transformer of **assume** `x*x >= 4` (line `(a)`) as

$$\lambda z.z \sqcap \{x \mapsto \top, v_{x*x} \mapsto [4, \infty]\} = \lambda z.z[v_{x*x} \mapsto (z(v_{x*x}) \sqcap [4, \infty])]\}.$$

The abstract state just before line `(b)` becomes

$$\{x \mapsto [-2, 2], v_{x*x} \mapsto [4, 4]\}, \tag{1}$$

which captures the fact that there is only one possible concrete value for `x*x`, namely, 4.

To obtain the abstract state just before line `(c)`, we must abstractly interpret the statement `x = x+1`. By evaluating the right-hand-side expression $x + 1$ (as an interval-domain expression) in abstract state (1), we obtain $x \mapsto [-1, 3]$. To obtain the abstract value of view-variable v_{x*x}, we use *finite differencing*. We start with the expression that defines the view: $v_{x*x} \stackrel{\text{def}}{=} \texttt{x*x}$. From the defining view-expression, finite differencing creates an appropriate view-maintenance expression for v_{x*x}:

$$v'_{x*x} = v_{x*x} + 2 \cdot x + 1, \tag{2}$$

where v'_{x*x} refers to the post-state value of v_{x*x}. For the moment, we leave aside the details of the algorithm used to derive Eqn. (2); they are explained below and in App. B. By evaluating Eqn. (2) in abstract state (1)—again, all abstract interpretation is performed solely over the interval domain—we obtain $v'_{x*x} = [4, 4] + 2 \cdot [-2, 2] + 1 = [1, 9]$. Thus, the abstract state just before line `(c)` is $\{x \mapsto [-1, 3], v_{x*x} \mapsto [1, 9]\}$.

Because $0 \notin \gamma([1, 9])$, the analyzer can use the value of view-variable v_{x*x} to prove the assertion on line `(c)`. Again, this requires interpreting the occurrence of `x*x` on line `(c)` as an access on the view-variable v_{x*x}. □

In Ex. 2.1, view-augmentation yields results that the unaugmented abstraction could not achieve on its own. While Ex. 2.1 is admittedly small and contrived, it demonstrates the benefit of a view-augmented abstraction: a view-augmented abstraction can capture, maintain, and use information that the unaugmented abstraction cannot represent.

Returning to the criteria given in Sect. 1 for an "inexpensive" method to improve the precision of a given numeric domain, we see that it is:

- *Parsimonious*: It was only necessary to introduce and track a single auxiliary variable, namely view-variable v_{x*x}.
- *Delegating*: All abstract operations, including those for updating v_{x*x} were performed using operations of the original abstract domain (in this case the interval domain); however, it was necessary to interpret occurrences of the expression `x*x` as accesses on view-variable v_{x*x}.

Maintaining views via finite differencing. Given state σ and statement *stmt*, the *future value* of an expression $\eta(x)$, denoted by $F_{stmt}[\eta(x)]$, is the value of $\eta(x)$ in the state $\sigma' = [\![stmt]\!]\sigma$ obtained by executing *stmt* on σ. Our goal is to create view-maintenance expressions that specify how to compute $F_{stmt}[\eta(x)]$. In particular, if we have a view-variable $v_{\eta(x)}$, view-maintenance expressions have the form $v'_{\eta(x)} = F_{stmt}[v_{\eta(x)}]$, where $v'_{\eta(x)}$ denotes the post-state value of $v_{\eta(x)}$.

Example 2.2 Returning to Ex. 2.1, how can we obtain the post-state value of v_{x*x} after the execution of the statement `x = x+1`? It is sound to compute $v'_{x*x} = F_{\texttt{x=x+1}}[v_{x*x}]$ as follows:

$$v'_{x*x} = F_{\texttt{x=x+1}}[v_{x*x}] = F_{\texttt{x=x+1}}[x] * F_{\texttt{x=x+1}}[x] = (x + 1) * (x + 1). \tag{3}$$

However, Eqn. (3) has significant drawbacks. In the concrete collecting semantics, just before line `(b)` we have $x \in \{-2, 2\}$. Consequently, just after line `(b)` we have $x \in \{-1, 3\}$, and thus in the collecting semantics we have $v_{x*x} \in \{1, 9\}$. The most precise interval-domain fact that over-approximates $v_{x*x} \in \{1, 9\}$ is $v_{x*x} \mapsto [1, 9]$.

In contrast, the abstract state just before line `(b)` is $\{x \mapsto [-2, 2], v_{x*x} \mapsto [4, 4]\}$ (see Eqn. (1)). By evaluating the right-hand-side expression $x + 1$ (as an interval-domain expression), we obtain $F_{\texttt{x=x+1}}[x] = [-1, 3]$. If we then use Eqn. (3) to obtain the value of v_{x*x}, we obtain $v_{x*x} \mapsto [-1, 3] * [-1, 3] = [-3, 9]$.

Why did we end up with $v_{x*x} \mapsto [-3, 9]$ instead of $v_{x*x} \mapsto [1, 9]$? One issue is that Eqn. (3) ignores the correlation in the expression `x*x`; i.e., in all concrete executions the *same* value of `x` is used twice in evaluating the expression. Yet, even if we use the most-precise squaring operation for intervals the result, $v_{x*x} \mapsto [0, 9]$, is still not precise enough.

A second issue is that Eqn. (3) forgets any information that was previously kept in view-variable v_{x*x}; v_{x*x} represents the closest approximation that we have in hand for the value of `x*x`, but Eqn. (3) uses only the value of `x`.

To create a view-maintenance expression that produces a more precise result, we employ finite-differencing, which yields a view-maintenance expression that uses the view-variable's pre-state value. Consequently, the finite-differencing approach addresses the second issue mentioned above. In the case of the statement `x=x+1`, it also addresses the first issue, albeit indirectly.

Let $\Delta_{\texttt{x=x+1}}[v_{x*x}]$ denote the additive change in v_{x*x}. Using the fact that across `x = x+1` the additive change in x, denoted by $\Delta_{\texttt{x=x+1}}[x]$, is 1, we can compute $F_{\texttt{x=x+1}}[v_{x*x}]$ as follows, to derive Eqn. (2):

$$\begin{aligned} v'_{x*x} &= F_{\texttt{x=x+1}}[v_{x*x}] \\ &= v_{x*x} + \Delta_{\texttt{x=x+1}}[v_{x*x}] \\ &= v_{x*x} + \Delta_{\texttt{x=x+1}}[x * x] \\ &= v_{x*x} + \Delta_{\texttt{x=x+1}}[x] \cdot x + x \cdot \Delta_{\texttt{x=x+1}}[x] + \Delta_{\texttt{x=x+1}}[x] \cdot \Delta_{\texttt{x=x+1}}[x] \\ &= v_{x*x} + 1 \cdot x + x \cdot 1 + 1 \cdot 1 \\ &= v_{x*x} + 2 \cdot x + 1. \end{aligned} \tag{4}$$

Eqn. (4) gives us a view-maintenance expression for computing the post-state value of v_{x*x} across `x = x+1` that uses the pre-state value of v_{x*x}. As shown earlier, when Eqn. (4) (Eqn. (2)) is evaluated in the abstract state $\{x \mapsto [-2, 2], v_{x*x} \mapsto [4, 4]\}$, we obtain $v'_{x*x} = [4, 4] + 2 \cdot [-2, 2] + 1 = [1, 9]$. □

In general, for an arbitrary view-variable $v_{\eta(x)}$ and statement *stmt*, the same approach can be applied to create a view-maintenance expression:

$$v'_{\eta(x)} = F_{stmt}[v_{\eta(x)}] = v_{\eta(x)} + \Delta_{stmt}[v_{\eta(x)}] = v_{\eta(x)} + \Delta_{stmt}[\eta(x)] \tag{5}$$

Eqn. (5) is depicted in Fig. 3(a) (which is explained in more detail in App. B).

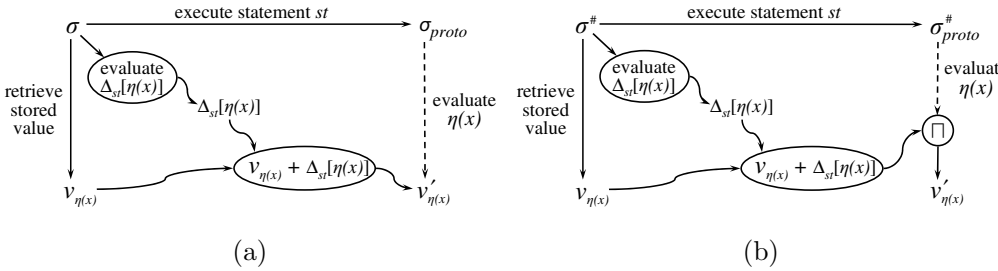

Fig. 3. (a) Method to compute the concrete value of a view-variable in a non-standard way, using finite differencing. (b) A method that can increase the precision of the abstract value of a view-variable using only existing operations of the underlying abstract domain.

```
COERCE(worklist, A)
    v := dequeue(worklist)
    for u ∈ Neighbors[v]:
        for r ∈ Relations[u]:
            A' := ⟦assume(r)⟧A
        if A' ⊏ A:
            then enqueue(worklist, u)
    return A'
```

Fig. 4. The COERCE function.

```
FIND-RELATIONS(views)
    for (v_μ, μ) ∈ views:
        M := REFORMULATE(μ)
        m-rels := ⋀{ISOLATE(v_μ = m) | m ∈ M}
        Rewrite m-rels in disjunctive normal form
        m-disjuncts = the set of disjuncts in m-rels
        for r ∈ m-disjuncts:
            vars = the set of variables occurring in r
            for w ∈ vars:
                Relations[w] := Relations[w] ∪ {r}
                Neighbors[w] := Neighbors[w] ∪ vars
```

Fig. 5. The FIND-RELATIONS function.

Rules for computing $\Delta_{stmt}[\eta(x)]$ according to the form of $\eta(x)$ are given in App. B (Fig. B.1).

In Ex. 2.2, the result $v_{x*x} \mapsto [1, 9]$ computed via the finite-differencing approach equals the result that would be computed by the best transformer. This is not always guaranteed. In fact, the finite-differencing approach is not even guaranteed to produce a result that is better than naïvely re-evaluating the view-variable's defining expression in the post-state (à la Eqn. (3)). However, one can always maintain the view-variable $v_{\eta(x)}$ by evaluating both maintenance expressions and taking their meet, as depicted in Fig. 3(b).

3 Increasing Precision

View-expression reformulation. View-expression reformulation finds occurrences of the defining expression $\eta(x)$ of some view-variable $v_{\eta(x)}$ and replaces them with references to $v_{\eta(x)}$. Reformulation is applied to each assignment expression and assume condition in the program. Reformulation can increase the precision of view-augmented abstract interpretation because the pre-computed value of $v_{\eta(x)}$ is always at least as precise as the value obtained by recomputing $\eta(x)$ (see Fig. 3(b)).

In our prototype, view-expression reformulation is implemented by REFORMULATE, a simple pattern-matching algorithm that searches for occurrences of view-variables' defining expressions. It accounts for commutativity and associativity of addition and multiplication, but not distributivity.

Information propagation among variables. Another important technique for obtaining good results from a view-augmented abstract interpretation is to perform

semantic reductions [7] by propagating information from variable to variable. Consider the following value in the interval domain: $\sigma = \{x \mapsto [0,5], y \mapsto [0,5], v_{x+y} \mapsto [0,2]\}$. Using x or y in a transition from σ will lead to unnecessary imprecision unless v_{x+y} somehow affects their values.

The Coerce operation (see Fig. 4) propagates information among the core and view variables of a single abstract state. The starting point for propagation is that every view variable $v_{\eta(x)}$ entails the equality constraint $v_{\eta(x)} = \eta(x)$. The Isolate operation identifies additional constraints among variables and view expressions using the commutativity and inverse properties of operations; Coerce then asserts those constraints on the abstract state.

In particular, suppose that the expression $\mu(x)$ is equivalent to the variable m, either because $\mu(x)$ is the variable expression m or m is $v_{\mu(x)}$. If $\mu(x)$ is a subexpression of a view expression $\eta(x)$, then Isolate can symbolically manipulate the known constraint $v_{\eta(x)} = \eta(x)$ into the form $m = \kappa$, where $v_{\eta(x)}$ occurs in the expression κ. Information can then be propagated from $v_{\eta(x)}$ to m (via κ) in the state σ as follows: $\sigma := [\![assume(m = \kappa)]\!]\sigma$.

In the example above, Coerce imposes the conditions $x = v_{x+y} - y$ and $y = v_{x+y} - x$ on σ. Using only abstract operations from the underlying domain, Coerce computes an improved σ via

$$\sigma := [\![assume(x = v_{x+y} - y)]\!]\sigma \sqcap [\![assume(y = v_{x+y} - x)]\!]\sigma.$$

This method yields the abstract value $\{x \mapsto [0,2], y \mapsto [0,2], v_{x+y} \mapsto [0,2]\}$.

The other key notion in the algorithm for Coerce is that constraining an abstract value via one constraint may enable other information to be propagated via another constraint. Thus, Coerce performs semi-naive evaluation [27] with respect to the graph over the variables in which two variables are connected if they are related by a constraint. Propagation continues until it quiesces, or until some user-specified number of propagation steps has been performed.

The Coerce function takes a *worklist* of variables and an abstract value A as input. Coerce returns the result of assuming the relevant facts from Isolate using semi-naive evaluation. The global dictionary *Relations* maps each program variable and view variable to the set of relations that contain them. The global dictionary *Neighbors* maps each variable to the set of variables that it shares a relation with. That is, $Neighbors[v] = \bigcup \{\text{variables in } r \mid r \in Relations[v]\}$. Both *Relations* and *Neighbors* hold facts about symbolic relationships among views.

Computing *Relations* and *Neighbors* during each step of abstract interpretation is prohibitively expensive. However, because they represent symbolic information that does not change across CFG nodes, they are precomputed once by Find-Relations (Fig. 5) and thereafter referred to by Coerce. To perform this precomputation, Find-Relations calls Reformulate to get sets of expressions known to equal the view variables, and Isolate to derive general relations from these expression sets.

Isolate takes an input condition of the form "*lhs op rhs*" and returns a larger, logically implied condition. Isolate algebraically isolates the variables on the right-

Program	Analysis	Time	Asserts
Berkeley	intervals	0.024s	2
	octagons	0.117s	2
	intervals + views	14.503s	3
Seesaw	octagons	0.087s	0
	octagons + views	7.736s	2
Sqrt	intervals	0.014s	0
	polyhedra	0.029s	0
	intervals + views	0.621s	2

Fig. 6. Summary of experiments. The column labeled "Asserts" indicates the number of assertions verified by the analysis.

```
real x, y;
y = 4;
assume (0 < x);
assume (0 < y*x*x && y*x*x < 3);
while (y*x*x <= .999
    || y*x*x >= 1.001) {
  x = x*(3 - y*x*x)/2.0;
}
assert (y*y*x*x - y <= 0.001);
assert (y*y*x*x - y >= -0.001);
```

Fig. 7. Code for Sqrt.

hand side of its input condition. As it does so, it accumulates further necessary conditions on the results. For instance, to isolate b in $l < ab$, the resulting condition is different depending on whether $a < 0$, $a = 0$, or $a > 0$. ISOLATE returns a predicate that handles each of these cases:

$$(a > 0 \wedge l/a < b) \vee (a < 0 \wedge l/a > b) \vee (a = 0 \wedge l < 0)\,.$$

4 Experiments

To test the capabilities of view-augmented abstraction, we implemented a prototype analyzer based on the Apron framework [1] and Interproc analyzer [15]. The experiments were run on a machine with a 3.40 GHz Pentium 4 dual processor and 2 GB of memory, running 32-bit Red Hat Linux Enterprise 5.

The experiments were designed to answer the following questions:

- Can a view-augmented abstraction give more precise results for the program variables of the original unaugmented abstraction?
- Can a view-augmented abstraction give more precise values of views than the original unaugmented abstraction?
- How expensive is view-augmented abstraction?

Berkeley. We translated the program Berkeley from the StInG suite [26] into the Interproc modeling language. Berkeley is equipped with three assertions to verify: $e \geq 0$, $u \geq 0$, and $i + u + e + n \geq 1$. In our tests, the interval and octagon domains cannot verify the third assertion, but the interval domain augmented with the views $i + u + e$ and $2i + u + 2e$ does.

Seesaw. We translated the Seesaw program from the StInG suite into the Interproc modeling language. Seesaw is equipped with two assertions to verify: $2y - x \geq 0$ and $3x - y \geq 0$. In our tests, the octagon domain does not verify either assertion, but the octagon domain augmented with the views $3x$ and $3x - y$ does.

Sqrt. The Sqrt algorithm we analyzed is shown in Fig. 6. It computes the reciprocal of the square root of y, avoiding divisions, by forcing x to converge to $1/\sqrt{y}$, and then returns yx [13]. Thus, the two assertions at the end of Sqrt check that $(yx)^2$ is close to the value of y.

In our tests, neither the standard interval domain nor the polyhedral domain can verify either assertion, but the interval domain augmented with the nonlinear views $yyxx$ and yxx verifies both.

Discussion. These examples give positive answers to our first two questions. Each assertion in each test uses only the original program variables. Thus, the fact that view-augmentation can verify otherwise-unverifiable assertions shows that a view-augmented abstraction can increase precision for the set of original program variables, as well as for the view expressions themselves.

The research is not sufficiently mature to conclusively address the question of analysis cost. On the examples in Fig. 6, view-augmentation is quite expensive—about 40 to 600 times more expensive than interpretation over the underlying domain. However, our implementation is an initial prototype, focusing on correctness rather than efficiency. Comparing this implementation against the carefully tuned Apron library is bound to give a poor impression of the performance of view-augmentation. Optimization and scalability are research goals to be addressed in future work.

In summary, the additional cost of view-augmentation is currently substantial, but we believe that the technique shows promise.

5 Related Work

Materialized views and view maintenance. Materialized views [11] are used in databases to cache commonly-requested queries, thereby increasing efficiency by providing answers without having to recompute queries on large data sets. Algorithms for incremental view maintenance are used to update the values of materialized views when there are changes to the base relations.

In databases, view maintenance is solely an optimization; the information can always be obtained by re-evaluating the view's defining formula. In the abstract-interpretation context, re-evaluating a view's defining expression does not usually yield a precise answer (cf. the running example in Sect. 2). Here, the main motivation for using materialized views and a finite-differencing method is to have an effective technique for retaining precision.

The *instrumentation relations* of canonical-abstraction domains [24] are materialized views of formulas over a relational vocabulary. As in our method, the views are expressed in the same language in which the concrete semantics is expressed—i.e., using first-order logic plus transitive closure in the case of canonical-abstraction domains; using numeric expressions in the case of the present paper. A finite-differencing method for logical formulas [22] is used to create abstract transformers automatically that maintain the values of each materialized view that augments a canonical-abstraction domain. The method operates on logical formulas expressed in first-order logic plus transitive closure. The method in the present paper operates on numeric expressions.

Bagnara et al. [2] developed a technique to generate invariant polynomial inequalities of bounded degree. Their technique introduces additional variables to

represent nonlinear terms, and uses convex polyhedra to represent polynomial cones in the extended set of variables. To reduce the loss of precision induced by this over-approximation step, the polyhedra are enriched with additional linear constraints that enforce some (semantically redundant) nonlinear constraints that would otherwise be lost. The rules for maintaining the values of the auxiliary variables are based on repeated substitution [2, Ex. 3 in §3.2].

Charles et al. [4] describe an algorithm for over-approximating the integer solutions of a set of non-linear constraints using an abstract domain based on linear constraints. The technique is not related to views *per se*, and assumes that a symbolic projection algorithm is available for a system of non-linear constraints, which is not always possible.

Abstraction refinement. In the past few years, many researchers have studied ways to refine predicate-abstraction domains. Refinement is central to counterexample-guided abstraction refinement [14,5,3], as used for example in SLAM [3] and BLAST [12]. Our work provides machinery that enables refinement of numeric domains, which can express properties that cannot be expressed using predicate abstraction.

Automatic creation of abstract transformers. The problem of creating view-maintenance formulas is related to the problem of automatically creating abstract transformers. For certain abstract-interpretation frameworks [10,23,28], it is known how to create *best* abstract transformers [7]. That is, the abstract transformers created are the most precise transformers possible, given the abstraction in use. For instance, Graf and Saïdi show how to use theorem provers to generate best transformers for predicate-abstraction domains. In contrast, the abstract transformers created using the algorithm described in Sect. 2 and App. B are not best transformers; however, the algorithm uses only very simple, linear-time, recursive tree-traversal procedures, whereas theorem provers are not guaranteed to terminate.

Miné [18] developed two methods to simplify numeric expressions before passing them to abstract transformers. One technique abstracts arbitrary expressions into affine forms with interval coefficients; the other technique performs constant propagation symbolically. The methods yield more precise abstract transformers, but do not improve the expressiveness of the underlying domain. In contrast, as shown in Sect. 2, view-augmentation can capture, maintain, and use information that the underlying abstraction cannot represent.

Sankaranarayanan et al. [25] describe Template Constraint Matrices (TCMs), a parametrized family of linear inequality domains. They give a parametrized meet, join, and set of abstract transformers for all domains in the family. Monniaux [19] gives algorithms that find best transformers among TCM domains across any straight-line blocks, and good transformers across more complicated control flow.

6 Conclusions

View-augmentation can enable any numeric abstract interpretation to capture more precise information. Our preliminary experiments demonstrate that view-

augmented abstraction is an effective way to improve the precision of a numeric abstraction. Directions for future work include improving performance and methods to automatically select fruitful views.

References

[1] *Apron Project home page*, http://apron.cri.ensmp.fr/.

[2] Bagnara, R., E. Rodríguez-Carbonell and E. Zaffanella, *Generation of basic semi-algebraic invariants using convex polyhedra*, in: *SAS*, 2005.

[3] Ball, T. and S. Rajamani, *Automatically validating temporal safety properties of interfaces*, in: *SPIN*, 2001.

[4] Charles, P., J. Howe and A. King, *Integer polyhedra for program analysis*, in: *AAIM*, 2009.

[5] Clarke, E., O. Grumberg, S. Jha, Y. Lu and H. Veith, *Counterexample-guided abstraction refinement*, in: *CAV*, 2000, pp. 154–169.

[6] Cousot, P. and R. Cousot, *Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points*, in: *POPL*, 1977, pp. 238–252.

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

[8] Dill, D., *Timing assumptions and verification of finite-state concurrent systems*, in: *Automatic Verification Methods for Finite State Systems*, 1989, pp. 197–212.

[9] Goldstine, H., "A History of Numerical Analysis," Springer-Verlag, 1977.

[10] Graf, S. and H. Saïdi, *Construction of abstract state graphs with PVS*, in: *CAV*, 1997.

[11] Gupta, A. and I. Mumick, editors, "Materialized Views: Techniques, Implementations, and Applications," The M.I.T. Press, Cambridge, MA, 1999.

[12] Henzinger, T., R. Jhala, R. Majumdar and K. L. McMillan, *Abstractions from proofs*, in: *POPL*, 2004, pp. 232–244.

[13] Hsieh, P., *Square roots*, http://www.azillionmonkeys.com/qed/sqroot.html.

[14] Kurshan, R., "Computer-Aided Verification of Coordinating Processes," Princeton Univ. Press, 1994.

[15] Lalire, G., M. Argoud and B. Jeannet, *Interproc analyzer*, http://pop-art.inrialpes.fr/people/bjeannet/bjeannet-forge/interproc.

[16] Miné, A., *The octagon abstract domain*, in: *WCRE*, 2001, pp. 310–322.

[17] Miné, A., *A few graph-based relational numerical abstract domains*, in: *SAS*, 2002.

[18] Miné, A., *Symbolic methods to enhance the precision of numerical abstract domains*, in: *VMCAI*, 2006, pp. 348–363.

[19] Monniaux, D., *Automatic modular abstractions for template numerical constraints*, Logical Methods in Comp. Sci. (2010).

[20] Müller-Olm, M. and H. Seidl, *Precise interprocedural analysis through linear algebra*, in: *POPL*, 2004.

[21] Paige, R. and S. Koenig, *Finite differencing of computable expressions*, TOPLAS **4** (1982), pp. 402–454.

[22] Reps, T., M. Sagiv and A. Loginov, *Finite differencing of logical formulas for static analysis*, in: *ESOP*, 2003, pp. 380–398.

[23] Reps, T., M. Sagiv and G. Yorsh, *Symbolic implementation of the best transformer*, in: *VMCAI*, 2004, pp. 252–266.

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

[25] Sankaranarayanan, S., H. Sipma and Z. Manna, *Scalable analysis of linear systems using mathematical programming*, in: *VMCAI*, 2005.

[26] *Stanford Invariant Generator (StInG)*, http://www.cs.colorado.edu/~srirams/Software/sting.html.

[27] Ullman, J. D., "Principles of Database and Knowledge-Base Systems, Volume I: Classical Database Systems," Comp. Sci. Press, Rockville, MD, 1988.

[28] Yorsh, G., T. Reps and M. Sagiv, *Symbolically computing most-precise abstract operations for shape analysis*, in: *TACAS*, 2004, pp. 530–545.

A Concrete and Abstract Semantics

We assume that we have a control flow graph with assignments and assumes on edges; the concrete, collecting, and abstract semantics of the CFG are defined as usual.

Concrete semantics. A program is specified by a *control flow graph* (CFG) $G = (N, E)$, where N is the set of program locations and $E \subseteq N \times N$ is the set of control-flow edges. Variables, in the set *ProgVars*, have values in $\mathbb{V}$, which could be any of $\mathbb{Z}$, $\mathbb{Z}_n$, $\mathbb{Q}$, or $\mathbb{R}$. The set of possible program states is $\Sigma = \mathit{ProgVars} \to \mathbb{V}$. We typically use σ to denote an individual state in Σ.

The function $\Pi_G : E \to (\Sigma \to \Sigma)$ defines the concrete semantics of each edge in the CFG. These program statements may be assertions or assignments; we allow unrestricted, nondeterministic expressions and conditions. When *stmt* is a program statement, its concrete semantic action is denoted by $[\![\mathit{stmt}]\!]$.

In general, we lift semantic functions to operate on sets by the usual point-wise extension. The collecting semantics is the least fixed point of a set of equations (defined in the standard way [6]), which yields a mapping $N \to \mathcal{P}(\Sigma)$.

Abstract semantics. Static analysis sidesteps undecidability by using abstraction: sets of program states are approximated by elements of an abstract domain $\mathbb{D} = (D, \alpha, \gamma, \sqsubseteq, \top, \bot, \vee, \sqcap)$. The function $\Pi^\sharp_G : E \to (D \to D)$ gives the abstract semantics of individual program statements. The abstract semantics is the least fixed point of a set of equations (again, defined in the standard way [6]), which yields a mapping $N \to D$.

B View Maintenance via Finite Differencing

ViewVars denotes the set of view-variables, where the concrete semantics of each view-variable $v_{f(\boldsymbol{u})}$ is specified by $v_{f(\boldsymbol{u})} \stackrel{\text{def}}{=} f(\boldsymbol{u})$. In general, $f(\boldsymbol{u})$ denotes an expression over $\mathit{ProgVars} \cup \mathit{ViewVars}$. View-variables may appear in the defining expressions of other view-variables, provided that there are no circular dependences.

We use $\overline{\Sigma} = (\mathit{ProgVars} \cup \mathit{ViewVars}) \to \mathbb{V}$ to denote the set of all possible augmented program states. When $\overline{\sigma} \in \overline{\Sigma}$ denotes an augmented program state, $\sigma \in \Sigma$ denotes the corresponding unaugmented state in which the values of all view-variables are forgotten.

The *future-value* operator $F_{\mathit{stmt}}[\eta]$ is defined as follows:

$$F_{\mathit{stmt}}[\eta] \stackrel{\text{def}}{=} \eta + \Delta_{\mathit{stmt}}[\eta], \tag{B.1}$$

exp	$\Delta_{stmt}[exp]$
$c \in ConstSyms$	0
$x \in ProgVars$ and transformer component $x := \eta$ is of the form $x := x + \delta, \delta \in ConstSyms$	δ
$x \in ProgVars$ and transformer component $x := \eta$ is not of the form $x := x + \delta, \delta \in ConstSyms$	$\eta - x$
$v_{f(\boldsymbol{u})} \in ViewVars$	$\Delta_{stmt}[f(\boldsymbol{u})]$
$f(\boldsymbol{u}) + g(\boldsymbol{u})$	$\Delta_{stmt}[f(\boldsymbol{u})] + \Delta_{stmt}[g(\boldsymbol{u})]$
$f(\boldsymbol{u}) * g(\boldsymbol{u})$	$f(\boldsymbol{u}) * \Delta_{stmt}[g(\boldsymbol{u})] + \Delta_{stmt}[f(\boldsymbol{u})] * g(\boldsymbol{u}) + \Delta_{stmt}[f(\boldsymbol{u})] * \Delta_{stmt}[g(\boldsymbol{u})]$

Fig. B.1. A finite-differencing scheme for numeric expressions.

where $\Delta_{stmt}[\eta]$ is defined in Fig. B.1. In particular, by Eqn. (B.1) and the fourth case of Fig. B.1,

$$F_{stmt}[v_{f(\boldsymbol{u})}] = v_{f(\boldsymbol{u})} + \Delta_{stmt}[f(\boldsymbol{u})]. \tag{B.2}$$

The semantics of assignment transitions in augmented program states is defined as follows:

$$[\![\boldsymbol{x} := \boldsymbol{\eta}]\!](\overline{\sigma}) = \lambda v. \begin{cases} [\![\boldsymbol{x} := \boldsymbol{\eta}]\!](\sigma) & \text{if } v \in ProgVars \\ \begin{pmatrix} [\![F_{\boldsymbol{x} := \boldsymbol{\eta}}[v_{f(\boldsymbol{u})}]]\!](\overline{\sigma}) \\ \sqcap\ [\![f(\boldsymbol{u})]\!]([\![\boldsymbol{x} := \boldsymbol{\eta}]\!](\sigma)) \end{pmatrix} & \text{if } v \equiv v_{f(\boldsymbol{u})} \in ViewVars \end{cases} \tag{B.3}$$

The view-augmented versions of Π_G and $\Pi_G^\sharp$ map each edge to an augmented-state transformer, and $\sqcap$ in Eqn. (B.3) is $\cap$ in the case of the concrete collecting semantics. The view-augmented concrete collecting semantics and abstract semantics are defined by least fixed points as before, but using the view-augmented versions of Π_G and $\Pi_G^\sharp$, respectively.

The second case of Eqn. (B.3) is illustrated in Fig. 3(b), for a view-augmented abstract semantics. As we will see in Thm. B.1 below, in the case of the concrete collecting semantics, the second case of Eqn. (B.3) can be simplified to either "$[\![F_{\boldsymbol{x} := \boldsymbol{\eta}}[v_{f(\boldsymbol{u})}]]\!](\overline{\sigma})$, if $v \equiv v_{f(\boldsymbol{u})} \in ViewVars$" or "$[\![f(\boldsymbol{u})]\!]([\![\boldsymbol{x} := \boldsymbol{\eta}]\!](\sigma))$, if $v \equiv v_{f(\boldsymbol{u})} \in ViewVars$". The former choice, where $F_{\boldsymbol{x} := \boldsymbol{\eta}}[v_{f(\boldsymbol{u})}]$ has been expanded by Eqn. (B.2), is illustrated in Fig. 3(a).

Soundness of view-augmented abstract interpretation is established by first showing that, in the concrete semantics, for all expressions η, $F_{stmt}[\eta]$ produces an *exact* maintenance expression (Thm. B.1 below). If the underlying abstract interpretation is sound, then soundness follows immediately for the view-augmented abstract interpretation: the latter uses abstract augmented-state transformers, but

with $\Pi^{\sharp}_{G}$ mapping each edge to an augmented-state transformer that incorporates expressions of the form $F_{stmt}[\eta]$ via Eqn. (B.3).

Theorem B.1 *Let stmt be a statement with transformer* $[\![\boldsymbol{x} := \boldsymbol{\eta}]\!]$. *Let* $\overline{\sigma} \in \overline{\Sigma}$ *be an augmented state, and let* σ_{proto} *be the result of evaluating* $[\![\boldsymbol{x} := \boldsymbol{\eta}]\!]$ *on unaugmented state* σ. *Let* $\overline{\sigma}'$ *be the structure obtained using* σ_{proto} *as the first approximation to* $\overline{\sigma}'$ *and then assigning to each view-variable* $v_{f(\boldsymbol{u})} \in ViewVars$, *in a topological ordering of the dependences among the view variables, by successively performing*

$$\overline{\sigma}' := \overline{\sigma}'[v_{f(\boldsymbol{u})} \leftarrow [\![f(\boldsymbol{u})]\!](\overline{\sigma}')].$$

Then for every expression η, $[\![F_{stmt}[\eta]]\!](\overline{\sigma}) = [\![\eta]\!](\overline{\sigma}')$.

Sketch of Proof: The proof is by induction using a size measure for expressions based on a process of normalizing η so that it is defined solely in terms of *ProgVars* (i.e., no *ViewVars*). Such normalization is always possible because of the assumption that view-variables are not circularly defined. The size measure is the size of the normalized η, except that each occurrence of a view-variable $v_{f(\boldsymbol{u})}$ is counted as being 1 larger than the size measure of the expression $f(\boldsymbol{u})$. The proof is thus similar to a standard structural-induction proof, except that in the case for a view-variable $v_{f(\boldsymbol{u})}$, we may assume that the induction hypothesis holds for $f(\boldsymbol{u})$.

In $[\![\eta]\!](\overline{\sigma}')$, each time a view-variable $v_{f(\boldsymbol{u})}$ is encountered, the value $\overline{\sigma}'(v_{f(\boldsymbol{u})})$ is used, which by the definition of $\overline{\sigma}'$ equals $[\![f(\boldsymbol{u})]\!](\overline{\sigma}')$. Because the size measure of $f(\boldsymbol{u})$ is strictly smaller than η, we have, by the induction hypothesis,

$$[\![f(\boldsymbol{u})]\!](\overline{\sigma}') = [\![F_{stmt}[f(\boldsymbol{u})]]\!](\overline{\sigma}).$$

The remaining cases (for +, *, etc.) follow from the definition of $F_{stmt}[\eta]$ (Eqn. (B.1)) and the well-known rules for finite differencing (Fig. B.1).