Variance Analyses from Invariance Analyses

Josh Berdine
Microsoft Research
jjb@microsoft.com

Aziem Chawdhary
Queen Mary, University of London
aziem@dcs.qmul.ac.uk

Byron Cook
Microsoft Research
bycook@microsoft.com

Dino Distefano
Queen Mary, University of London
ddino@dcs.qmul.ac.uk

Peter O'Hearn
Queen Mary, University of London
ohearn@dcs.qmul.ac.uk

Abstract

An invariance assertion for a program location ℓ is a statement that always holds at ℓ during execution of the program. Program invariance analyses infer invariance assertions that can be useful when trying to prove safety properties. We use the term *variance assertion* to mean a statement that holds between any state at ℓ and any previous state that was also at ℓ. This paper is concerned with the development of analyses for variance assertions and their application to proving termination and liveness properties. We describe a method of constructing program variance analyses from invariance analyses. If we change the underlying invariance analysis, we get a different variance analysis. We describe several applications of the method, including variance analyses using linear arithmetic and shape analysis. Using experimental results we demonstrate that these variance analyses give rise to a new breed of termination provers which are competitive with and sometimes better than today's state-of-the-art termination provers.

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

General Terms Verification, Reliability, Languages

Keywords Formal Verification, Software Model Checking, Program Analysis, Liveness, Termination

1. Introduction

An *invariance analysis* takes in a program as input and infers a set of possibly disjunctive invariance assertions (*a.k.a.*, invariants) that is indexed by program locations. Each location ℓ in the program has an invariant that always holds during any execution at ℓ. These invariants can serve many purposes. They might be used directly to prove safety properties of programs. Or they might be used indirectly, for example, to aid the construction of abstract transition relations during symbolic software model checking [29]. If a desired safety property is not directly provable from a given invariant, the user (or algorithm calling the invariance analysis) might try to refine the abstraction. For example, if the tool is based on abstract interpretation they may choose to improve the abstraction by delaying the widening operation [28], using dynamic partitioning [33], employing a different abstract domain, etc.

The aim of this paper is to develop an analogous set of tools for program termination and liveness: we introduce a class of tools called *variance analyses* which infer assertions, called *variance assertions*, that hold between any state at a location ℓ and any previous state that was also at location ℓ. Note that a single variance assertion may itself be a disjunction. We present a generic method of constructing variance analyses from invariance analyses. For each invariance analysis, we can construct what we call its *induced variance analysis*.

This paper also introduces a condition on variance assertions called the *local termination predicate*. In this work, we show how the variance assertions inferred during our analysis can be used to establish local termination predicates. If this predicate can be established for each variance assertion inferred for a program, whole program termination has been proved; the correctness of this step relies on a result from [37] on *disjunctively well-founded overapproximations*. Analogously to invariance analysis, even if the induced variance analysis fails to prove whole program termination, it can still produce useful information. If the predicate can be established only for some subset of the variance assertions, this induces a different liveness property that holds of the program. Moreover, the information inferred can be used by other termination provers based on disjunctive well-foundedness, such as TERMINATOR [14]. If the underlying invariance analysis is based on abstract interpretation, the user or algorithm could use the same abstraction refinement techniques that are available for invariance analyses.

In this paper we illustrate the utility of our approach with three induced variance analyses. We construct a variance analysis for arithmetic programs based on the Octagon abstract domain [34]. The invariance analysis used as input to our algorithm is composed of a standard analysis based on Octagon, and a post-analysis phase that recovers some disjunctive information. This gives rise to a fast and yet surprisingly accurate termination prover. We similarly construct an induced variance analysis based on the domain of Polyhedra [23]. Finally, we show that an induced variance analysis based on the separation domain [24] is an improvement on a termination prover that was recently described in the literature [3]. These three abstract domains were chosen because of their relative position on the spectrum of domains: Octagon is designed to be extremely fast, at the expense of accuracy, whereas Polyhedra and the separation domain are more powerful at the cost of speed.

POPL'07 January 17–19, 2007, Nice, France.

```
01 VarianceAnalysis(P, L, I♯) {
02   IAs := InvarianceAnalysis(P, I♯)
03   foreach ℓ ∈ L {
04     LTPreds[ℓ] := true
05     O := Isolate(P, L, ℓ)
06     foreach q ∈ IAs such that pc(q) = ℓ {
07       VAs := InvarianceAnalysis(O, Step(O, {Seed(q)}))
08       foreach r ∈ VAs {
09         if pc(r) = ℓ ∧ ¬WellFounded(r) {
10           LTPreds[ℓ] := false
11         }
12       }
13     }
14   }
15   return LTPreds
16 }
```

Figure 1. Parameterized variance analysis algorithm. P is the program to be analyzed, the set of program locations L is a set of cutpoints, and $I^\sharp$ is the set of initial states. To instantiate the variance analysis one must fix the implementations of InvarianceAnalysis, Step, Seed and WellFounded.

In their own right each of these induced variance analyses is on the leading edge in the area of automatic termination proving. For example, in some cases the Octagon-based tool is the fastest known termination prover. But the more important point is that these variance analyses are not specially-designed: their power is determined almost exclusively by the power of the underlying invariance analysis.

2. Inducing invariance analyses

In this section we informally introduce the basic ideas behind our method. Later, in Sections 3 and 4, we will formally define the components in the algorithm, and prove its soundness.

Fig. 1 contains our analysis algorithm. To instantiate the analysis to a particular domain, we must provide implementations for the following components:

- InvarianceAnalysis: The underlying invariance analysis.
- Step: A single-step function over InvarianceAnalysis's abstract domain.
- Seed: An additional operation on elements of the abstract domain (Definition 15 in Section 4).
- WellFounded: An additional operation on elements of the abstract domain (Definition 13 in Section 4).

The implementations of InvarianceAnalysis and Step are given by the underlying invariance analysis, whereas the implementations of Seed and WellFounded must usually be defined (though they are not difficult to do so in practice).

When instantiated with the implementations of Seed, WellFounded, etc. this algorithm performs the following steps:

1. It first runs the invariance analysis, computing a set of invariance assertions, IAs.
2. Each element q (from IAs) is converted into a binary relation via the Seed operation.
3. The algorithm then re-runs the invariance analysis from the seeded state after a single step of execution to compute a fixed point over variance assertions, VAs. That is, during this step the invariance analysis computes an approximation (represented as a binary relation on states) of the behavior of the loop.
4. The analysis then takes each element of VAs and uses the WellFounded operation in order to establish the validity of a set of local termination predicates, stored in an array $LTPreds$. A location ℓ's local termination predicate holds if $LTPreds[\ell] = \texttt{true}$.

The reason we take a single step before re-running the invariance calculation is that we are going to leverage the result of [37] on disjunctive well-foundedness, which connects well-foundedness of a relation to over-approximation of its non-reflexive transitive closure. Without Step we would get the reflexive transitive closure instead.

In general, VAs, IAs and $I^\sharp$ in this algorithm might be (finite) sets of abstract elements, rather than singletons. We regard these sets as disjunctions and, in particular, if a variance assertion at ℓ is the disjunction of multiple elements of VAs, then ℓ's local termination lemma holds only in the case that WellFounded returns true for each disjunct.

Although we regard each set as a disjunction, we are not insisting that our abstract domains are closed under disjunctive completion [19]. InvarianceAnalysis might even return just a single abstract element, or it might return several without computing the entire disjunctive completion; we might employ techniques such as in [33, 41] to efficiently approximate disjunctive completion. But, the decision of how much disjunction is present is represented in the inputs Step and InvarianceAnalysis, and is not part of the VarianceAnalysis algorithm.

For our experiments with numerical domains, we fitted them with a post-analysis to extract disjunctive information from otherwise conjunctive domains. That is, the invariance analyses used by the VarianceAnalysis algorithm are composed of the standard numerical domain analysis together with a method of disjunction extraction. On the other hand, for our shape analysis instantiation no pre-fitting is required because the abstract domain explicitly uses disjunction (Section 6).

2.1 Illustrative example

Consider the small program fragment in Fig. 2, where `nondet()` represents non-deterministic choice. In this section we will use this program while stepping through the VarianceAnalysis algorithm. We will assume that our underlying invariance analysis is based on the Octagon domain, which can express conjunctions of inequalities of the form $\pm x + \pm y \leq c$ for variables x and y and constant c.

Note that during this example we will associate invariance assertions and variance assertions with line numbers. We will say that an assertion holds at line ℓ if and only if it is always valid at *the beginning* of the line, before executing the code contained at that line. Furthermore, we will choose a set of program location cutpoints to be the first basic block of a loop's body: $L = \{82, 83, 85\}$. Location 82 is the cutpoint for the loop contained in lines 81–91, location 83 is the cutpoint for the loop contained in lines 82–90, and 85 is the cutpoint for the loop within lines 84–86.

Given L, our parameterized variance analysis attempts to establish the validity of a local termination predicate for each location $\ell \in L$, when the program P is run from starting states satisfying input condition $I^\sharp$.

Note that while the outermost loop in Fig. 2 does not guarantee termination, so long as execution remains within the loop starting at location 82, it is not possible for the loop in lines 82–90 to visit location 83 infinitely often. In this example we will show how VarianceAnalysis is able to prove a more local property at location 83:

```
81      while (nondet()) {
82          while (x>a && y>b) {
83              if (nondet()) {
84                  do {
85                      x = x - 1;
86                  } while (x>10);
87              } else {
88                  y = y - 1;
89              }
90          }
91      }
```

Figure 2. Example program fragment.

$\mathcal{LT}(P, L, 83, I^\sharp)$: *Line* 83 *is visited infinitely often only in the case that the program's execution exits the loop contained in lines* 82 *to* 90 *infinitely often.*

The formal definition of $\mathcal{LT}(P, L, \ell, I^\sharp)$, the local termination predicate at ℓ, will be given later (Definition 8 in Section 3).

Although we will not do so in this example, VARIANCEANALYSIS would also attempt to establish local termination predicates for the remaining cutpoints:

$\mathcal{LT}(P, L, 82, I^\sharp)$: *Line* 82 *is visited infinitely often only in the case that the program's execution exits the loop contained in lines* 81 *to* 91 *infinitely often.*

$\mathcal{LT}(P, L, 85, I^\sharp)$: *Line* 85 *is visited infinitely often only in the case that the program's execution exits the loop contained in lines* 84 *to* 86 *infinitely often.*

Because the outer loop is not terminating, VARIANCEANALYSIS would fail to prove $\mathcal{LT}(P, L, 82, I^\sharp)$. As for 85, it would succeed to prove $\mathcal{LT}(P, L, 85, I^\sharp)$.

We are using a program with nested loops here to illustrate the modularity afforded by our local termination predicates: even if the inner loops and outer context are diverging, this will not stop us from proving the local termination predicate at location 83. That is to say: the termination of the innermost loop beginning at line 84 does not affect our predicate. We could replace line 86

```
86              } while (x>10);
```

with

```
86              } while (nondet());
```

and still establish $\mathcal{LT}(P, L, 83, I^\sharp)$. However, $\mathcal{LT}(P, L, 85, I^\sharp)$ would not hold in this case.

Invariance analysis (Line 2 of Fig. 1). We start by running an invariance analysis using the Octagon domain (possibly with a disjunction-recovering post-analysis). In this example, if we had the text of the entire program, we could start with an initial state of $I^\sharp = (pc = 0)$. Note that we will assume that the program counter is represented with an additional equality $pc = c$ in each abstract program state where c is a numerical constant. Instead of starting at location 0, assume that at location 81 we have $I^\sharp = (pc = 81 \wedge x \geq a + 1 \wedge y \geq b + 1)$. From this starting state the invariance analysis could compute an invariant $IA_{83} \in IAs$:

$$IA_{83} \triangleq pc = 83 \wedge x \geq a + 1 \wedge y \geq b + 1$$

An abstract state, of course, denotes a set of concrete states. IA_{83}, for example, represents the set of states:

$$\{s \mid s(\mathtt{pc}) = 83 \wedge s(\mathtt{x}) \geq s(\mathtt{a}) + 1 \wedge s(\mathtt{y}) \geq s(\mathtt{b}) + 1\}$$

Isolation (Line 5 of Fig. 1). The next thing we do, for location 83, is "isolate" the smallest strongly-connected subgraph of P's control-flow graph containing location 83, subject to some conditions involving the set of locations $L = \{82, 83, 85\}$, defined formally in Section 3. Concretely, from the overall program P we construct a new program O, which is the same as P with the exception that the statement at line 90 is now:

```
90      }; assume(false);
```

Because of this `assume` statement, executions that exit the loop are not considered. Furthermore, pc in the isolated program's initial state will be 83. Together, these two changes restrict execution to stay within the loop.

This isolation step gives us modularity for analyzing inner loops. It allows us to establish a local termination predicate for O even when it is nested within another loop P that diverges. Concretely, isolation will eliminate executions which exit or enter the loop.

Inferring variance assertions (Lines 6 and 7 of Fig. 1). From this point on we will use our invariance analysis to reason about the isolated subprogram rather than the original loop. Let $\longrightarrow_O$ denote the transition relation for the isolated subprogram O. We then take all of the disjuncts in the invariance assertion at location 83 (in this case there is only one, IA_{83}) and convert them into binary relations from states to states:

$$\begin{aligned}\textsc{Seed}(IA_{83}) = (&pc = 83 \wedge pc_s = 83 \wedge x \geq a + 1 \wedge y \geq b + 1 \\ &\wedge x_s = x \wedge y_s = y \wedge a_s = a \wedge b_s = b)\end{aligned}$$

$\textsc{Seed}(IA_{83})$ is, of course, just a state that references variables not used in the program—these variables can be thought of as logical constants. However, in another sense, $\textsc{Seed}(IA_{83})$ can be thought of as a binary relation on program states:

$$\begin{aligned}\{(s,t) \quad \mid \quad & s(\mathtt{pc}) = t(\mathtt{pc}) = 83 \\ \wedge \quad & s(\mathtt{x}) = t(\mathtt{x}) \\ \wedge \quad & s(\mathtt{y}) = t(\mathtt{y}) \\ \wedge \quad & s(\mathtt{a}) = t(\mathtt{a}) \\ \wedge \quad & s(\mathtt{b}) = t(\mathtt{b}) \\ \wedge \quad & t(\mathtt{x}) \geq t(\mathtt{a}) + 1 \\ \wedge \quad & t(\mathtt{y}) \geq t(\mathtt{b}) + 1 \qquad \}\end{aligned}$$

Notice that we're using x_s to represent the value of $\mathtt{x}$ in s, and x to represent the value of $\mathtt{x}$ in t. That is, s gives values to the variables $\{pc_s, x_s, y_s, a_s, b_s\}$ while t gives values to $\{pc, x, y, a, b\}$.

We call this operation seeding because it plants a starting (diagonal) relation in the abstract state. Later in the algorithm this relation will grow into one which indicates how progress is made.

We then step the program once from $\textsc{Seed}(IA_{83})$ with STEP, approximating one step of the program's semantics, giving us:

$$\begin{aligned}pc_s = 83 \wedge pc = 84 \wedge x \geq a + 1 \wedge y \geq b + 1 \\ \wedge x_s = x \wedge y_s = y \wedge a_s = a \wedge b_s = b\end{aligned}$$

Finally, we run INVARIANCEANALYSIS again with this new state as the starting state, and the isolated subprogram O as the program, which gives us a set of invariants at locations 82, 83, etc. that corresponds to the set VAs in the VARIANCEANALYSIS algorithm

of Fig. 1.

$$VA^A_{83} \triangleq (pc_s = 83 \wedge pc = 83 \wedge x \geq a+1 \wedge y \geq b+1 \wedge x_s \geq x+1 \wedge y_s \geq y \wedge a_s = a \wedge b_s = b)$$

$$VA^B_{83} \triangleq (pc_s = 83 \wedge pc = 83 \wedge x \geq a+1 \wedge y \geq b+1 \wedge x_s \geq x \wedge y_s \geq y+1 \wedge a_s = a \wedge b_s = b)$$

$$VA^C_{83} \triangleq (pc_s = 83 \wedge pc = 83 \wedge x \geq a-1 \wedge y \geq b+1 \wedge x_s \geq x+1 \wedge y_s \geq y+1 \wedge a_s = a \wedge b_s = b)$$

$$\{VA^A_{83}, VA^B_{83}, VA^C_{83}\} \subseteq VAs$$

The union of these three relations

$$VA^A_{83} \vee VA^B_{83} \vee VA^C_{83}$$

forms the variance assertion for line 83 in P, which is to say a superset of the possible transitions from states at 83 to states also at line 83 reachable in 1 or more steps of the program's execution. (Note that in this case INVARIANCEANALYSIS is extracting disjunctive information implicit in the fixed point computed.) The disjunction $VA^A_{83} \vee VA^B_{83} \vee VA^C_{83}$ is a superset of the transitive closure of the program's transition relation restricted to pairs of reachable states both at location 83.

One important aspect of this technique is that the analysis is not aware of our intended meaning of variables like x_s and y_s: it simply treats them as symbolic constants. It does not know that the states are representing relations. (See Definition 12 and the further remarks on relational analyses at the end of Section 4.) However, as it was for SEED(IA_{83}), it is appropriate for us to interpret the meaning of VA^A_{83} as a relation on pairs of states.

The variance assertion $VA^A_{83} \vee VA^B_{83} \vee VA^C_{83}$ shows us different ways in which the subprogram can make progress. Because $VA^A_{83} \vee VA^B_{83} \vee VA^C_{83}$ is a variance assertion, this measure of progress holds between *any two* states s and t at location 83 where s is reachable and t is reachable in 1 or more steps from s. Notice that VA^A_{83} contains an inequality between x and x_s, whereas SEED(IA_{83}) contained an equality. This means that, in the first of the three disjuncts in the variance assertion at line 83, x is at least 1 less than x_s: In its relational meaning, because it is a variance assertion the formula says "in the current state, `x` is less than it was before".

Finally, when we "run the analysis again" on subprogram O the inner loop containing location 85 must be analyzed. Literally, then, to determine the local termination property for locations 83 and 85 involves some repetition of work. However, if we analyzed an inner loop first an optimization would be to construct a summary, which we could reuse when analyzing an outer loop. The exact form of these summaries is delicate, and we won't consider them explicitly in this paper. But, we remark that the summary would not have to show the inner loop terminating: When an inner loop fails to terminate this does not stop the local termination predicate from holding for the outer loop, as the example in this section demonstrates.

Proving local termination predicates (Lines 8–11 of Fig. 1). We now attempt to use the variance assertion at line 83 in O to establish the local termination predicate at line 83 in P. Consider the relation

$$Tr_{83} = \{(s,t) \mid s \models IA_{83} \wedge s \longrightarrow_O^+ t \wedge t(pc) = 83\}$$

Showing that Tr_{83} is well-founded allows us to conclude the local termination predicate:

Location 83 *is not visited infinitely often in executions of the isolated subprogram* O.

The reason is due to the over-approximation computed at line 2 in Fig. 1. The abstract state VA_{83} over-approximates all of the states that can be reached at line 83, even as parts of ultimately divergent executions, so we now do not need to consider other states to understand the behavior of this subprogram.

We stress that the "not visited infinitely often" property here does *not* imply in general that the isolated subprogram O terminates. In the example of this section the inner loop does terminate, but a trivial example otherwise is

```
1       while (x>a) {
2           x=x-1;
3           while (nondet()) {}
4       }
```

Here we can show that location 2 is not visited infinitely often when the program is started in a state where $x > a$.

Continuing with the running example, due to the result of [37], a relation Rel is well founded if and only if its transitive closure Rel^+ is a subset of a finite union $T_1 \cup \cdots \cup T_n$ and each relation T_i is well-founded. Notice that we have computed such a finite set: VA^A_{83}, VA^B_{83}, and VA^C_{83}. We know that the union of these three relations over-approximates the transitive closure of the transition relation of the program P limited to states at location 83. Furthermore, each of the relations are, in fact, well-founded. Thus, we can reason that the program will not visit location 83 infinitely often unless it exits the subprogram infinitely often. We will make this connection formal in Section 3.

The last step to automate is proving that each of the relations VA^A_{83}, VA^B_{83}, and VA^C_{83} are well-founded. Because these relations are represented as a conjunction of linear inequalities, they can be automatically proved well-founded by rank function synthesis engines such as RANKFINDER [36] or POLYRANK [6, 7].

Benefits of the approach

- The technique above is fast. Using the Octagon-based program invariance analysis packaged with [34] together with RANKFINDER, this example is proved terminating in 0.07 seconds. TERMINATOR, in contrast, requires 8.3 seconds to establish the same local termination predicate.
- Like TERMINATOR [14] or POLYRANK [6, 7], the technique is completely automatic. No ranking functions need to be given by the user. Simply *checking* termination arguments is easy, and has been done automatically since the 1970s. In contrast, both automatically *finding and checking* termination arguments for programs is a much more recent step. This will be discussed further in Section 7.
- As in TERMINATOR, the technique that we have described makes many little well-foundedness checks instead of one big one. If we can find the right decomposition, this makes for a strong termination analysis. In the proposed technique, we let the invariance generator choose a decomposition for us (*e.g.* VA^A_{83}, VA^B_{83}, VA^C_{83}). Furthermore, we let the invariance engine approximate all of the choices that a program could make during execution with a finite set of relations.
- As is true in TERMINATOR, because this analysis uses a disjunctive termination argument rather than a single ranking function, our termination argument can be expressed in a simpler domain. In our setting this allows us to use domains such as Octagon [34] which is one of the most efficient and well-behaved numerical abstract domains.

  For example, consider a traditional ranking function for the loop contained in lines 82–90:

  $$f(s) = s(\mathtt{x}) + s(\mathtt{y})$$

  Checking termination in the traditional way requires support for four-variable inequalities in the termination prover, as we must

prove $R \subseteq T_f$, where R is the loop's transition relation and

$$T_f = \{(s,t) \mid \ f(s) \geq f(t) - 1 \wedge \ f(t) \geq 0\}$$

i.e.

$$T_f = \{(s,t) \mid \ s(\mathtt{x}){+}s(\mathtt{y}) \geq t(\mathtt{x}){+}t(\mathtt{y}){-}1 \wedge t(\mathtt{x}){+}t(\mathtt{y}) \geq 0\}$$

Notice the four-variable inequality (where $s(\mathtt{x})$ and $t(\mathtt{x})$ will be treated with different arithmetic variables):

$$s(\mathtt{x}) + s(\mathtt{y}) \geq t(\mathtt{x}) + t(\mathtt{y}) - 1$$

Thus, we cannot use the Octagon domain in this setting. We can in our setting because VA^A_{83}, VA^B_{83}, and VA^C_{83} are simpler than T_f: they are all conjunctions of two-variable inequalities, such as $x \leq x_s$ but not $x + y > x_s$.

- Although tools like RANKFINDER synthesize ranking functions, we do not need them—we simply need a Boolean result. This is in contrast to TERMINATOR, which uses the synthesized ranking functions to create new abstractions from counterexamples. As a consequence, any sound tool that proves well-foundedness will suffice for our purposes.
- Our technique is robust with respect to arbitrarily nested loops, as we're simply using the standard program analysis techniques to prove relationships between visits to location 83. Even if the innermost loop did not terminate, we would still be able to establish the local termination predicate at location 83. For this reason our new analysis fits in well with termination decomposition techniques based on cutpoints [25].
- If the termination proof does not succeed due to the discovery of a non-well-founded disjunct, the remaining well-founded disjuncts are now in a form that can be passed to a tool like TERMINATOR—TERMINATOR can then use this as a better initial termination argument than its default one from which it will refine based on false counterexamples as described in [12].
- In contrast to TERMINATOR, VARIANCEANALYSIS seeds in a dynamic fashion. This means that *abstract* states are seeded after some disjunction has been introduced by the invariance analysis, which can improve precision and allows us to dynamically choose which variables to include in the seeding. In fact, an alternative method of approximating our core idea would be to first use the source-to-source transformation described in [13] on the input program and then apply an invariance analysis on the resulting program. We have found, though, that taking this approach results in a loss of precision.
- We do not need to check that the disjunction of the variance assertions forms a transition invariant—it simply holds by construction. In TERMINATOR this inclusion check is the performance bottleneck.

3. Concrete semantics and variance assertions

In this section we give a precise account of the local termination predicates, their relation to well-foundedness for isolated programs, and the relation to variance assertions. These properties can be formulated exclusively in terms of concrete semantics.

3.1 Programs and loops

DEFINITION 1 (Locations). *We assume a fixed finite set $\mathbb{L}$ of* program locations.

DEFINITION 2 (Programs). *A* program $P \in \mathbb{P}$ *is a rooted, edge-labeled, directed graph over vertex set $\mathbb{L}$.*

Programs are thought of as a form of control-flow graphs where the edges are labeled with commands which denote relations $2^{(D_C \times D_C)}$ on program states D_C.[1] This formulation represents programs in a form where all control flow is achieved by nondeterministic jumps, and branch guards are represented with assumptions. For example, Fig. 3 shows a representation of the program from Fig. 2 in this form.

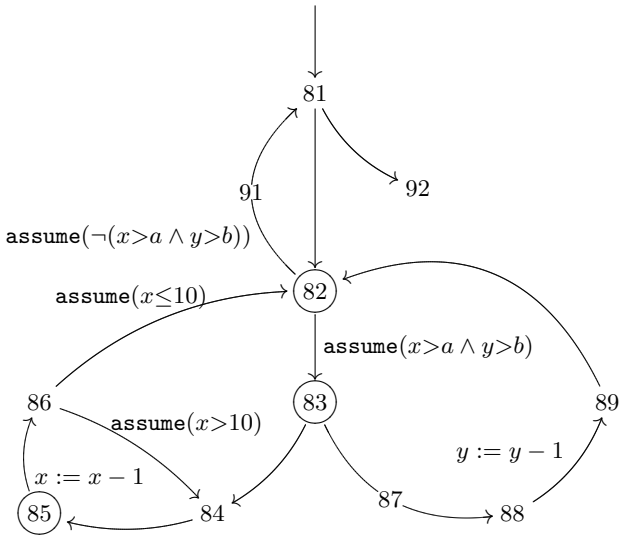

Figure 3. Graph representation of the program from Fig. 2, where we have circled a set of cutpoints. Note that assumptions involving `nondet` have been elided.

We use the following notation: We write $P(\sigma)$ to indicate that there is a directed path in graph P through the ordered sequence of vertices σ. We write $\cdot$ for sequence concatenation.

The control-flow graph structure of programs is used to define the notion of a set of cutpoints [25] in the usual way.

DEFINITION 3 (Cutpoints). *For a program P, a set L of* cutpoints *is a subset of $\mathbb{L}$ such that every (directed) cycle in P contains at least one element of L.*

3.2 Isolation

In order to formally describe the ISOLATE procedure from Fig. 1, we first must define several constructs over program control-flow graphs.

DEFINITION 4 (SCSG). *For a program P and set of cutpoints L, we define a set* $\text{SCSG}(P, L)$ *of* strongly-connected subgraphs *of P:*

$$\text{SCSG}(P, L) \triangleq \bigcup_{\ell \in \mathbb{L}} mscsgd(\ell)$$

where $O \in mscsgd(\ell)$ iff

1. *O is a non-empty, strongly-connected subgraph of P;*
2. *all vertices in O are dominated by ℓ, where for vertices m and n, n is* dominated by m *iff $P(r \cdot \sigma \cdot n)$ implies $m \in \sigma$ where r is the root vertex;*
3. *every cycle in P (that is, a cycle in the control-flow graph, not in the executions of the program) is either contained in O or contains a cutpoint in L that is not in O; and*
4. *there does not exist a strict supergraph of O that satisfies these conditions.*

[1] The invariance analysis algorithm relies (via the ISOLATE operation) on being able to identify loops in programs. This led us to be explicit about control-flow graphs, rather than use the usual, syntax-free, formulation in terms of functions over concrete or abstract domain elements.

For a well-structured program, and the set of cutpoints consisting of all locations just inside of loop bodies and recursive function call-sites, Definition 4 identifies the innermost natural loop containing ℓ. This also handles non-well-structured but reducible loops, but does not allow isolation of non-reducible subgraphs (such as loops formed by `goto`s from one branch of a conditional and back). The subgraphs of P identified by $\text{SCSG}(P, L)$ are the strongly-connected components of P, plus some which are not maximal. Condition 2 limits the admitted non-maximal subgraphs to only those that, intuitively, are inner loops of a strongly-connected component. Condition 3 ensures that the allowed subgraphs are not at odds with the given set of cutpoints, which may force merging multiple loops together into one subgraph. Condition 4 ensures that the subgraph for a loop includes its inner loops. These sorts of issues are familiar from compilation [1, 2].

Note that the elements of $\text{SCSG}(P, L)$, being a superset of the strongly-connected components of P, cover every cycle in (the control-flow graph of) P. Another point to note is that two elements of $\text{SCSG}(P, L)$ are either disjoint or one is a subset of the other.

DEFINITION 5 (LP). *For a program P, set of cutpoints L, and location ℓ, $\text{LP}(P, L, \ell)$ is the set of vertices of the smallest element of $\text{SCSG}(P, L)$ which contains ℓ.*

As an example, if P is the program in Fig. 3, and $L = \{82, 83, 85\}$, $\text{SCSG}(P, L) = \{\{84..86\}, \{82..90\}, \{81..91\}\}$, and we have:

$$\begin{aligned} \text{LP}(P, L, 82) &= \{81..91\} \\ \text{LP}(P, L, 83) &= \{82..90\} \\ \text{LP}(P, L, 85) &= \{84..86\} \end{aligned}$$

DEFINITION 6 (ISOLATE). *For program P, set of cutpoints L, and program location ℓ, $\text{ISOLATE}(P, L, \ell)$ is the induced subgraph based on $\text{LP}(P, L, \ell)$. That is, the subgraph of P containing only the edges between elements of $\text{LP}(P, L, \ell)$. The root of $\text{ISOLATE}(P, L, \ell)$ is ℓ.*

Informally, $\text{ISOLATE}(P, L, \ell)$ constructs a subprogram of P such that execution always remains within $\text{LP}(P, L, \ell)$.

Note that we have given mathematical specifications of, but not algorithms for computing, sets of cutpoints, SCSG, LP, etc. In practice efficient algorithms are available.

3.3 Local termination predicates

We now develop the definition of a local termination predicate. To do so we must also develop notation for several fundamental concepts, such as concrete semantics.

DEFINITION 7 (Concrete semantics). *The concrete semantics of a program is given by:*

- *a set D_C of* program states, *and*
- *a function $\longrightarrow_{(\cdot)} : \mathbb{P} \to 2^{(D_C \times D_C)}$ from programs to* transition relations.

We use a presentation where program states include program locations, which we express with

- *a function $\mathsf{pc}_P : D_C \to \mathbb{L}$ from program states to values of the* program counter.

The transition relations are constrained to only relate pairs of states for which there is a corresponding edge in the program, that is, $s \longrightarrow_P t$ implies $P(\mathsf{pc}_P(s) \cdot \mathsf{pc}_P(t))$.

When we associate a program P with a set $I_P \subseteq D_C$ of *initial states* we will require that $\mathsf{pc}(s)$ is the root of the control-flow graph for each $s \in I_P$.

Recall from Section 2 that the local termination predicate at line 82 was informally stated as

> *Line 83 is visited infinitely often only in the case that the program's execution exits the loop contained in lines 82 to 90 infinitely often.*

That is, the local termination predicate is a liveness property about location 83, which could be expressed in linear temporal logic [35] as:

$$\Box\big(\Box\Diamond\, pc = 83 \implies \Diamond\, pc \notin \text{LP}(P, L, 83)\big)$$

Next we formally define the notion of local termination predicate.

DEFINITION 8 (Local termination predicate ($\mathcal{LT}$)). *For program P, cutpoint set L, program location ℓ, and set of initial states I_P,*

$$\mathcal{LT}(P, L, \ell, I_P) \text{ holds}$$

if and only if for any infinite execution sequence

$$s_0, s_1, ..., s_i, ... \qquad \text{with } s_0 \in I_P \text{ and } \forall i.\ s_i \longrightarrow_P s_{i+1}$$

for all $j \geq 0$

if $\mathsf{pc}(s_k) = \ell$ for infinitely many $k > j$
then $\mathsf{pc}(s_{k'}) \notin \text{LP}(P, L, \ell)$ for some $k' > j$.

We now define a variant of well-foundedness (of the concrete semantics) in which the domain and range of the relation is specialized to a given program location ℓ.

DEFINITION 9 ($\mathcal{WF}$). *For program O, program location ℓ, and set of initial states I_O, we say that $\mathcal{WF}(O, \ell, I_O)$ holds iff for any infinite execution sequence*

$$s_0, s_1, ..., s_i, ... \qquad \text{with } s_0 \in I_O \text{ and } \forall i.\ s_i \longrightarrow_P s_{i+1}$$

there are only finitely many $j > 0$ such that $\mathsf{pc}(s_j) = \ell$.

The key lemma is the following, which links well-foundedness for an isolated loop to the $\mathcal{LT}(P, L, \ell, I_P)$ property.

PROPOSITION 1 (Isolation). *Let $O = \text{ISOLATE}(P, L, \ell)$ and suppose*

- *I_P is a set of initial states for program P, and*
- *$I_O = \{t \mid \exists s \in I_P.\ s \longrightarrow_P^* t \ \wedge\ \mathsf{pc}(t) = \ell\}$.*

If $\mathcal{WF}(O, \ell, I_O)$ holds, then $\mathcal{LT}(P, L, \ell, I_P)$ holds.

Proof: Removing a finite prefix ending just before a state at ℓ from a counterexample to $\mathcal{LT}(P, L, \ell, I_P)$ yields a counterexample to $\mathcal{WF}(O, \ell, I_O)$. That is: Suppose by way of contradiction that $\mathcal{WF}(O, \ell, I_O)$ and that $\neg\mathcal{LT}(P, L, \ell, I_P)$, that is, there exists an infinite execution sequence $s_0, s_1, ..., s_i, ...$ with $s_0 \in I_P$ and $\forall i.\ s_i \longrightarrow_P s_{i+1}$ where there exists a $j \geq 0$ such that $\mathsf{pc}(s_k) = \ell$ for infinitely many $k > j$ and $\mathsf{pc}(s_{k'}) \in \text{LP}(P, L, \ell)$ for all $k' > j$. Consider the suffix $s_{j'}, s_{j'+1}, ..., s_{j'+i}, ...$ of the infinite execution sequence for some $j' \geq j$ such that $\mathsf{pc}(s_{j'}) = \ell$. Since $\mathsf{pc}(s_{j'+i}) \in \text{LP}(P, L, \ell)$ for all $i \geq 0$, and $O = \text{ISOLATE}(P, L, \ell)$, we have an execution sequence in O that visits ℓ infinitely often. That is, $s_{j'+0}, s_{j'+1}, ..., s_{j'+i}, ...$ with $s_{j'} \in I_O$ and $\forall i.\ s_i \longrightarrow_O s_{j'+i+1}$ such that $\mathsf{pc}(s_{j'+k}) = \ell$ for infinitely many $k > j$. This contradicts $\mathcal{WF}(O, \ell, I_O)$. □

Finally, if an analysis can establish the validity of a complete set of local termination predicates, then this is sufficient to prove whole program termination.

PROPOSITION 2. *Let L be a set of cutpoints for P and I_P be a set of initial states. If, for each $\ell \in L$, $\mathcal{LT}(P, L, \ell, I_P)$, then there is no infinite execution sequence starting from any state in I_P.*

Proof: Suppose for each $\ell \in L$, $\mathcal{LT}(P, L, \ell, I_P)$. Suppose by way of contradiction that there is an infinite execution sequence: $s_0, s_1, ..., s_i, ...$ with $s_0 \in I_P$ and $\forall i.\, s_i \longrightarrow_P s_{i+1}$. Therefore at least one location is visited infinitely often. Each of the infinitely-often visited locations ℓ has an associated $\text{LP}(P, L, \ell)$. Let ℓ' be an infinitely-often visited location whose set of locations $\text{LP}(P, L, \ell')$ has cardinality not smaller than that of $\text{LP}(P, L, \ell''')$ for any other infinitely-often visited location ℓ'''. A consequence of the definition of $\mathcal{LT}(P, L, \ell', I_P)$ is that execution must leave, and return to, the set (of control-flow graph locations) $\text{LP}(P, L, \ell')$ infinitely often. Therefore there is a cycle $C \subseteq \mathbb{L}$ in P which is not contained in $\text{LP}(P, L, \ell')$ and, by Definition 4, contains an infinitely-often visited cutpoint ℓ'' not in $\text{LP}(P, L, \ell')$. [Definition 4 does not directly guarantee that ℓ'' is visited infinitely often, but since execution leaves and returns to $\text{LP}(P, L, \ell')$ infinitely often, by a pigeonhole argument, at least one of the choices of cycle C must include an infinitely-often visited cutpoint. Without loss of generality we choose ℓ''.] Therefore, since the elements of $\text{SCSG}(P, L)$ cover every cycle of P, there must exist $\text{LP}(P, L, \ell'')$ that contains C. Since C is not disjoint from $\text{LP}(P, L, \ell')$ and contains $\ell'' \notin \text{LP}(P, L, \ell')$, $\text{LP}(P, L, \ell') \subset \text{LP}(P, L, \ell'')$. In particular, $\text{LP}(P, L, \ell'')$ is larger than $\text{LP}(P, L, \ell')$. Now since it contains an infinitely-often visited cutpoint not in $\text{LP}(P, L, \ell')$, this contradicts the proof's assumption that $\text{LP}(P, L, \ell')$ is maximal. □

4. From Invariance Abstraction to Termination

In this section we use abstract interpretation to formally define the items in the VARIANCEANALYSIS algorithm. We then link local termination predicates and well-foundedness for isolated programs to abstraction to prove soundness of VARIANCEANALYSIS.

4.1 Abstract interpretations

We will assume that an abstract interpretation [18, 19] of a program is given by two pieces of information. The first is an over-approximation of the individual transitions in programs, such as by a function $\text{STEP} : \mathbb{P} \to 2^{D^\sharp} \to 2^{D^\sharp}$ that works on abstract states $D^\sharp$. $\text{STEP}(P, X)$ typically propagates each state in X forward one step, in a way that over-approximates the concrete transitions of program P. The second is the net effect of what one gets from the overall analysis results, which may be a function $\text{INVARIANCEANALYSIS} : \mathbb{P} \to 2^{D^\sharp} \to 2^{D^\sharp}$ that for program P, over-approximates the reflexive, transitive closure $\longrightarrow_P^*$ of the concrete transition relation of P. INVARIANCEANALYSIS is typically defined in terms of STEP. However the details as to how they are connected is not important in this context. Widening or other methods of accelerating fixed-point calculations might be used [18]. In this paper we are only concerned with the net effect, rather than the way that INVARIANCEANALYSIS is obtained, and our formulation of over-approximation below reflects this. We do, however, presume that STEP and INVARIANCEANALYSIS are functions from programs to abstractions. This assumption allows the local variance analysis using ISOLATE.

If R is a binary relation then we use $\text{IMAGE}(R, X)$ to denote its post-image $\{y \mid \exists x \in X.\, xRy\}$.

DEFINITION 10 (Over-Approximation). *An* over-approximation A *of a concrete semantics* $\longrightarrow_{(\cdot)}$ *over concrete states* D_C *is*

- *a set* $D^\sharp$ *of abstract states*
- *a function* $[\![\cdot]\!] : D^\sharp \to 2^{D_C}$
- *a function* $\mathsf{pc}^\sharp : D^\sharp \to \mathbb{L}$
- *a function* $\text{STEP}_{(\cdot)} : \mathbb{P} \to 2^{D^\sharp} \to 2^{D^\sharp}$
- *a function* $\text{INVARIANCEANALYSIS}_{(\cdot)} : \mathbb{P} \to 2^{D^\sharp} \to 2^{D^\sharp}$

such that

- *for all* $X \subseteq D^\sharp$. $\text{IMAGE}(\longrightarrow_P, [\![X]\!]) \subseteq [\![\text{STEP}(P, X)]\!]$ *and* $\text{IMAGE}(\longrightarrow_P^*, [\![X]\!]) \subseteq [\![\text{INVARIANCEANALYSIS}(P, X)]\!]$
- *if* $s \in [\![a]\!]$ *then* $\mathsf{pc}_P(s) = \mathsf{pc}^\sharp(a)$.

where we use the point-wise lifting of $[\![\cdot]\!]$ *to sets of abstract states:* $[\![\cdot]\!] : 2^{D^\sharp} \to 2^{D_C}$.

We use a powerset domain due to the fact that most successful termination proofs must be path sensitive and thus we would like to have explicit access to disjunctive information.[2] Since we have lifted the meaning function $[\![\cdot]\!]$ pointwise, it is distributive (preserving unions) as a map from $2^{D^\sharp}$ to 2^{D_C}. But, we are not requiring the analysis to be a (full) disjunctive completion.

In particular, note that we do not require distributivity, or even monotonicity, of INVARIANCEANALYSIS or STEP; thus, we can allow for acceleration methods that violate these properties [8, 20, 22]. Furthermore, we do not require that union be used at join-points in the analysis, such as at the end of `if` statements; our definition is robust enough to allow an over-approximation of union to be used. We have used the powerset representation simply so the result VAs in the VARIANCEANALYSIS algorithm gives us a collection of well-foundedness queries, allowing us to apply the result of [37]. If the invariance analysis is not disjunctive, then the VAs result set will be a singleton. In this case the variance analysis will still be sound, but will give us imprecise results.

Notice that this definition does not presume any relation between STEP and INVARIANCEANALYSIS, even though the latter is usually defined in terms of the former; the definition just presumes that the former over-approximates $\longrightarrow_P$, and the latter $\longrightarrow_P^*$. We have just put in minimal conditions that are needed for the soundness of our variance analysis.[3]

We do not assume that $\text{INVARIANCEANALYSIS}(P, X)$ always returns a finite set, even when X is finite. However, if the returned set is not finite when $\text{VARIANCEANALYSIS}(P, L, I^\sharp)$ calls $\text{INVARIANCEANALYSIS}(P, X)$, our variance analysis algorithm will itself not terminate.

4.2 Seeding, well-foundedness, and ghost state

We now specify seeding (SEED) and the well-foundedness check (WELLFOUNDED) used in the VARIANCEANALYSIS algorithm from Fig. 1. These comprise the additions one must make to an existing abstract interpretation in order to get a variance analysis by our method. Often, these are already implicitly present in, or easily added to, an invariance analysis. Throughout this section we presume that we have a concrete semantics together with an over-approximation as defined above.

Seeding is a commonly used technique for symbolically recording computational history in states. In our setting, the SEED operation in Fig. 1 is specific to the abstract domain. Therefore, instead of providing a concrete definition, we specify properties that each instance must satisfy. As a result, this gives significant freedom to the developer of the SEED/WELLFOUNDED pair, as we will see in Section 6 where we define the SEED/WELLFOUNDED pair for a shape analysis domain.

[2] It might be possible to formulate a generalization of our theory without explicit powersets, using projections of certain disjunctive information out of an abstract domain; we opted for the powerset representation for its simplicity.

[3] In fact, the variance analysis could be formulated more briefly using a single over-approximation of the transitive closure $\longrightarrow_P^+$, but we have found that separating INVARIANCEANALYSIS and STEP makes the connection to standard program analysis easier to see.

After seeding has been performed, the VARIANCEANALYSIS algorithm proceeds to use the INVARIANCEANALYSIS to compute variance assertions over pairs of states. In the following development we formalize the encoding and interpretation between relations on pairs of states and predicates on single states.

First, we require a way to identify *ghost state* in the concrete semantics. [In the program logic literature (e.g, see Reynolds [39]), *ghost variables* are specification-only variables that are not changed by a program. We are formulating our definitions at a level where we do not have a description of the state in terms of variables, so we refer to *ghost state*, by analogy with ghost variable.]

DEFINITION 11 (Ghostly Decomposition). *A* ghostly decomposition *of the concrete semantics is a set S_C with*

- *a bijection $\langle \cdot, \cdot \rangle : S_C \times S_C \to D_C$*

such that

- *$\langle g, p \rangle \longrightarrow_P \langle g', p' \rangle$ implies $g = g'$.*
- *$\langle g_1, p \rangle \longrightarrow_P \langle g_1, p' \rangle$ implies $\langle g_2, p \rangle \longrightarrow_P \langle g_2, p' \rangle$*
- *$\mathsf{pc}_P \langle g_1, p \rangle = \mathsf{pc}_P \langle g_2, p \rangle$*

In $S_C \times S_C$ we think of the second component as the real program state and the first as the ghost state. The first two conditions say that ghost state does not change, and that it does not impact transitions on program state.

Given a transition system it is easy to make one with a ghostly decomposition just by copying the set of states. We do not insist, though, that the bijection in the definition be an equality because transition systems are often given in such a way that a decomposition is possible without explicitly using a product. Typically, states are represented as functions $\mathrm{Var} \to \mathrm{Val}$ from variables to values, and if we can partition variables into isomorphic sets of program variables and copies of them, then the basic bijection

$$(A \to V) \times (B \to V) \;\cong\; (A + B \to V)$$

can be used to obtain a ghostly decomposition. In fact, we will use this idea in all of the example analyses defined later.

Given a ghostly decomposition, we obtain a *relational* meaning $\langle\!\langle a \rangle\!\rangle$, which is just $[\![\cdot]\!]$ adjusted using the isomorphism of the decomposition. Formally,

DEFINITION 12 (Relational Semantics). *For any $a \in D^\sharp$, the relation $\langle\!\langle a \rangle\!\rangle \subseteq S_C \times S_C$ is*

$$\langle\!\langle a \rangle\!\rangle \quad \triangleq \quad \{(g, p) \mid \langle g, p \rangle \in [\![a]\!]\}$$

We are using the notation $\langle g, p \rangle$ here for an element of D_C corresponding to applying the bijection of a ghostly decomposition, reserving the notation (g, p) for the tuple in $S_C \times S_C$.

Using this notion we can formally define the requirements for the well-foundedness check in the algorithm of Fig. 1.

DEFINITION 13 (Well-Foundedness Check). *Suppose that A is an over-approximation of a program P with ghostly decomposition. Then a* well-foundedness check *is a map*

$$\text{WELLFOUNDED} : D^\sharp \to \{\mathtt{true}, \mathtt{false}\}$$

such that if $\text{WELLFOUNDED}(a)$ *then $\langle\!\langle a \rangle\!\rangle$ is a well-founded relation.*

Recall that a relation R is well founded iff there does not exist an infinite sequence p such that $\forall i \in \mathbb{N}.\ (p_i, p_{i+1}) \in R$. Then a well-foundedness check must soundly ensure that the relation $\langle\!\langle a \rangle\!\rangle$ on program states is well founded.

For our variance analysis to work properly it is essential that the abstract semantics work in a way that is independent of the ghost state.

DEFINITION 14 (Ghost Independence). *Suppose the concrete semantics has a ghostly decomposition. We say that*

- *$a \in D^\sharp$ is* ghost independent *if*

$$(g, p) \in \langle\!\langle a \rangle\!\rangle \implies \forall g'.\, (g', p) \in \langle\!\langle a \rangle\!\rangle$$

  i.e., *if the predicate $[\![a]\!]$ is independent of the ghost state. Also, $X \subseteq D^\sharp$ is ghost independent if each element of X is.*
- *An over-approximation* preserves ghost independence *if* $\text{INVARIANCEANALYSIS}(P, X)$ *is ghost independent whenever $X \subseteq D^\sharp$ is ghost independent.*

The idea here is just that the abstract semantics will ignore the ghost state and not introduce spurious facts about it.

Curiously, our results do not require that STEP preserves ghost independence, even though it typically will. Preservation of ghost independence is needed, technically, only in the statement $IAs := \text{INVARIANCEANALYSIS}(P, I^\sharp)$ in the VARIANCEANALYSIS algorithm; for seeding to work properly we need that all the elements of Q are ghost independent if all the initial abstract states in $I^\sharp$ are. The formal requirement on the SEED operation, which takes independence into account, is:

DEFINITION 15 (Seeding). *A* seeding function *is a map*

$$\text{SEED} : D^\sharp \to D^\sharp$$

such that if a is ghost independent and $(g, p) \in \langle\!\langle a \rangle\!\rangle$ then $(p, p) \in \langle\!\langle \text{SEED}(a) \rangle\!\rangle$.

$\text{SEED}(a)$ can be thought of as an over-approximation of the diagonal relation on program variables in a. That is, we do not require that SEED exactly copy the state, which would correspond to $\text{SEED}(a) = \{(p, p)\}$ instead of $(p, p) \in \text{SEED}(a)$ in the definition.

4.3 Soundness

To establish the soundness result, we fix:

- a concrete semantics with ghostly decomposition;
- an over-approximation that preserves ghost independence, with a seeding map and sound well-foundedness check;
- a program P and set of initial states $I_P \subseteq D_C$;
- a finite set $I^\sharp \subseteq D^\sharp$ of initial abstract states, each of which is ghost independent, and where $I_P \subseteq [\![I^\sharp]\!]$.

THEOREM 1 (Soundness). *If* $\text{VARIANCEANALYSIS}(P, L, I^\sharp)$ *of Fig. 1 terminates, for L a finite set of program locations; then upon termination, $LTPreds$ will be such that for each $\ell \in L$ and $s \in I_P$, $\mathcal{LT}(P, L, \ell, I_P)$ if $LTPreds[\ell] = \mathtt{true}$.*

As an immediate consequence of Proposition 2 we obtain

COROLLARY 1. *Suppose L is a set of cutpoints. Assume that $LTPreds = \text{VARIANCEANALYSIS}(P, L, I^\sharp)$. In this case P terminates if $\forall \ell \in L.\ LTPreds[\ell] = \mathtt{true}$.*

Now we give the proof of the theorem.

Proof: [Theorem 1] Consider $\ell \in L$ and suppose $LTPreds[\ell] = \mathtt{true}$ on termination of VARIANCEANALYSIS. Let $O = \text{ISOLATE}(P, L, \ell)$ and

$$I_O \;=\; \{t \mid \exists s \in I_P.\, s \longrightarrow_P^* t \;\wedge\; \mathsf{pc}(t) = \ell\}\ .$$

We aim to show that $\mathcal{WF}(O, \ell, I_O)$ holds. The theorem follows at once from this and Proposition 1.

$$\text{Let } Tr_\ell \triangleq \{\ (b, c)\ \mid\ \exists \langle g, b \rangle \in I_O \quad \langle g, b \rangle \longrightarrow_O^+ \langle g, c \rangle \wedge \mathsf{pc}(\langle g, c \rangle) = \ell\}$$

Assume that the algorithm in Fig. 1 has terminated and that $LTPreds[\ell] = \texttt{true}$. First, we have a lemma:

If Tr_ℓ is well founded then $\mathcal{WF}(O, \ell, I_O)$ holds

This lemma is immediate from the transition conditions in Definition 11. So, by the lemma we will be done if we can establish that Tr_ℓ is well founded. For convenience, we define:

$$\textsc{Locs}(\ell_1, \ell_2) \triangleq \{(s,t) \mid \mathsf{pc}(s) = \ell_1 \wedge \mathsf{pc}(t) = \ell_2\}$$

We need to show two things:

1. Wanted: $\{\langle\!\langle r \rangle\!\rangle \cap \textsc{Locs}(\ell,\ell) \mid r \in VAs\}$ is a finite set of well-founded relations.

   VAs is clearly finite, as otherwise the algorithm would not terminate. Therefore we know that there exists a finite disjunctively well-founded decomposition of $(\bigcup_{r \in VAs} \langle\!\langle r \rangle\!\rangle) \cap \textsc{Locs}(\ell,\ell)$ where, for each $r \in VAs$, $\textsc{WellFounded}(r) = \texttt{true}$. This is due to Definition 13, which tells us that, for each $r \in VAs$, $\langle\!\langle r \rangle\!\rangle$ is well-founded. ✓

2. Wanted: $Tr_\ell \subseteq \bigcup_{r \in VAs} \langle\!\langle r \rangle\!\rangle \cap \textsc{Locs}(\ell,\ell)$.

   Assume that there exist $(b,c) \in Tr_\ell$. That is: we have some $\langle g, b\rangle \in I_O$ with $\langle g, b\rangle \longrightarrow_O^+ \langle g, c\rangle$ and $\mathsf{pc}(\langle g, c\rangle) = \ell$. By over-approximation, $\langle g, b\rangle \in \bigcup [\![IAs]\!]$. Thus, there exists a $q \in IAs$ such that $\langle g, b\rangle \in [\![q]\!]$. Since the start states in $I^\sharp$ are ghost independent, and InvarianceAnalysis preserves ghost independence, we obtain that q is ghost independent. We obtain from Definition 15 that $(b,b) \in \langle\!\langle \textsc{Seed}(q) \rangle\!\rangle$. By ghostly decomposition, $\langle b, b\rangle \longrightarrow_O^+ \langle b, c\rangle$, and so by over-approximation for Step followed by InvarianceAnalysis, there exists $r \in VAs$ where $\langle b, c\rangle \in [\![r]\!]$. By the definition of $\langle\!\langle \cdot \rangle\!\rangle$ this means that $(b,c) \in \langle\!\langle r \rangle\!\rangle$. Thus, because $\mathsf{pc}(q) = \mathsf{pc}(r) = \ell$, $Tr_\ell \subseteq \bigcup_{r \in VAs} \langle\!\langle r \rangle\!\rangle \cap \textsc{Locs}(\ell,\ell)$. ✓

We can now prove that Tr_ℓ is well founded as follows. The two facts just shown establish that $Tr_\ell \subseteq T_1 \cup \cdots \cup T_n$ for a finite collection of well-founded relations given by VAs (note that the union need not itself be well founded). Further, by the definition of Tr_ℓ it follows that $Tr_\ell = Tr_\ell^+$. So we know $Tr_\ell^+ \subseteq T_1 \cup \cdots \cup T_n$ for a finite collection of well-founded relations. By the result of [37] it follows that Tr_ℓ is well founded. □

We close this section with two remarks on the level of generality of our formal treatment.

Remark: On Relational Abstract Domains. A "relational abstract domain" is one in which relationships between variables (like $x < y$ or $x = y$) can be expressed. Polyhedra and Octagon are classic examples of relational domains, while the Sign and Interval domains are considered non-relational. The distinguishing feature of Sign and Interval is that they are *cartesian*, in the sense that the abstract domain tracks the cartesian product of properties of program variables ([15], p.10), independently. It has been suggested that our variance analyses might necessarily rely on having a relational (or non-cartesian) abstract domain, because in the examples above we use equalities to record initial values of variables.

But, consider the Sign domain. For each program variable x the Sign domain can record whether x's value is positive, negative, or zero. If the value cannot be put into one of these categories, it is $\top$. We can define a seeding function, where $\textsc{Seed}(F)$ assigns to each seed variable x_s the same abstract value as x. For example, if F is $\texttt{positive}(x) \wedge \texttt{negative}(y)$ then $\textsc{Seed}(F)$ is

$$\texttt{positive}(x) \wedge \texttt{negative}(y) \wedge \texttt{positive}(x_s) \wedge \texttt{negative}(y_s)$$

This seeding function satisfies the requirements of Definition 15.

The Sign domain is an almost useless termination analysis; it can prove

```
while(x<0 && k>0) x = x * x + k;
```

but not a typical loop that increments or decrements a counter. We have mentioned it only to illustrate the technical point that our definition of seeding does not rely on being able to specify the equality between normal and ghost state. In this sense, our formal treatment is perhaps more general than might at first be expected.

A more significant illustration of this point will be given in Section 6.

Remark: On Relational Analyses. A "relational analysis" is one where an abstract element over-approximates a relation between states (the transition relation) rather than a set of individual states. This notion is often used in interprocedural analysis, for example in the Symbolic Relational Separate Analysis of [21]. (This sense of "relational" is not to be confused with that in "relational abstract domain"; the same word is used for distinct purposes in the program analysis literature.)

It has been suggested [17] that our use of ghost state above is a way to construct a relational analysis from a standard one (where states are over-approximated). Indeed, it would be interesting to rework our theory using a formulation of relational analyses on a level of generality comparable to standard abstract interpretation where, say, the meaning map had type $[\![\cdot]\!] : D^\sharp \to 2^{D_C \times D_C}$ rather than $[\![\cdot]\!] : D^\sharp \to 2^{D_C}$. In this sense, our formal treatment here is probably not as general as possible; we plan to investigate this generalization in the future. Among other things, such a formulation should allow us to use cleverer representations of (over-approximations of) relations than enabled by our use of ghost state; see [21], Section 9, for pointers to several such representations.

5. Variance analyses for numerical domains

The pieces come together in this section. By instantiating VarianceAnalysis with several numerical abstract domains, we obtain induced variance analyses and compare them to existing termination proof tools on benchmarks. As the results in Fig. 4 show, the induced variance analyses yield state-of-the-art termination provers. The two domains used, Octagon [34] and Polyhedra [23], where chosen because they represent two disparate points in the cost/precision spectrum of abstract arithmetic domains.

Instances of* Seed *and* WellFounded *for numerical domains. Before we begin, we must be clear about the domain of states. We presume that we are given a fixed finite set Var of variables with $\mathsf{pc} \in \text{Var}$.[4] Concrete states are defined to be mappings from variables to values $D_C \triangleq \text{Var} \to \mathcal{V}$ that (for simplicity of the presentation) are limited to a set of numerical values $\mathcal{V}$ (where $\mathcal{V}$ could be the integers, the real numbers, etc.). The abstract states $D^\sharp$ are defined to be conjunctions of linear inequalities over $\mathcal{V}$. We assume that each abstract state includes a unique equality $\mathsf{pc} = c$ for a fixed program location constant c. This gives us the way to define the projection $\mathsf{pc}^\sharp$ required by an over-approximation (Definition 10).

Next, in order to define seeding we presume that the variables in Var are of two kinds, *program variables* and *ghost variables*. The former may be referenced in a program, while the latter can be referenced in abstract states but not in programs themselves. The programs are just `goto` programs with assignment and conditional branching based on expressions with Boolean type (represented in flow-graph form as in Definition 2).

[4] For simplicity we are ignoring the issue of variables in F that are not in-scope at certain program locations. This can be handled, but at the expense of considerable complexity in the formulation.

We assume a disjoint partitioning $\mathrm{GVar} \cup \mathrm{PVar}$ of the set of variables Var where $\mathtt{pc} \in \mathrm{PVar}$. We presume a bijective mapping $\rho : \mathrm{PVar} \to \mathrm{GVar}$ that associates ghost variables to program variables. This furnishes the isomorphisms

$$(\mathrm{GVar} \to \mathcal{V}) \times (\mathrm{PVar} \to \mathcal{V}) \cong (\mathrm{PVar} \to \mathcal{V}) \times (\mathrm{PVar} \to \mathcal{V}) \cong \mathrm{Var} \to \mathcal{V}$$

from which we obtain the bijection $\langle \cdot, \cdot \rangle : S_C \times S_C \to D_C$ required by Definition 11 (where $S_C = \mathrm{PVar} \to \mathcal{V}$ and $D_C = \mathrm{Var} \to \mathcal{V}$).

At this point we have everything that is needed to define the SEED and WELLFOUNDED functions:

$$\begin{array}{ll} \textsc{Seed}(F) & \triangleq \quad F \wedge \bigwedge_{v \in \mathrm{PVar}} \{v = \rho(v)\} \\ \textsc{WellFounded}(F) & \triangleq \quad \textsc{WfCheck}(\rho(\mathrm{PVar}), \mathrm{PVar}, F) \end{array}$$

SEED uses ρ to add equalities between ghost and program variables. The well-foundedness check calls either RANKFINDER [36] or POLYRANK [6, 7], which take an input formula and then report whether or not the formula denotes a well-founded relation.

We will not give the explicit definitions of the semantics of concrete programs, or of the corresponding definitions of INVARIANCEANALYSIS and STEP on the particular abstract domains, referring instead to [23, 34]. The concrete dynamic semantics $\longrightarrow_P$ satisfies the required conditions of Ghostly Decomposition (Definition 11) because ghost variables do not appear in programs. Because these variables are never modified by the program the functions STEP and INVARIANCEANALYSIS will preserve ghost independence of abstract states.

Furthermore, the seeding function in this section satisfies Definition 15. Also, WELLFOUNDED satisfies Definition 13 as a result of soundness of the well-foundedness checker (RANKFINDER [36] or POLYRANK [6, 7]). Thus we have given the definitions needed to obtain a specific variance analysis as an instance of the framework in the previous section.

Example. Let $\mathrm{PVar} = \{x, y, pc\}$, $\mathrm{GVar} = \{x_s, y_s, pc_s\}$, and $\rho = \{(x, x_s), (pc, pc_s), (y, y_s)\}$ is a bijective mapping. Let s be the Octagon state $x < y \wedge pc = 10$. Thus,

$$\textsc{Seed}(s) = x < y \wedge pc = 10 \wedge x = x_s \wedge pc = pc_s \wedge y = y_s$$

If we execute the command sequence

$$x := x + 1; \mathtt{assume}(x < y); \mathtt{goto}\ 10$$

from s, the abstract transfer function should produce a state q:

$$q \triangleq x < y \wedge pc = 10 \wedge x \geq x_s + 1 \wedge pc_s = pc \wedge y_s = y$$

$\langle\!\langle q \rangle\!\rangle$ is a well-founded relation because x is increasing while being less than y, and y is unchanging; x cannot increase forever and yet remain less than an unchanging y. Indeed, when RANKFINDER is passed this formula with the ghost and program variables as the "from" and "to" variables, it reports that it can find a ranking function—confirming that $\langle\!\langle q \rangle\!\rangle$ is a well-founded relation.

Experiments. In order to evaluate the utility of our approach for arithmetic domains we have instantiated it using analyses based on the Octagon and Polyhedra domains and then compared these analyses to several known termination tools. The tools used in the experiments are as follows:

O) OCTATERM is the variance analysis induced by OCTANAL [34] composed with a post-analysis phase (see below). OCTANAL is included in the Octagon domain library distribution. During these experiments OCTATERM was configured to return "Terminating" in the case that each of the variance assertions inferred entailed their corresponding local termination predicate. The WFCHECK operation was based on RANKFINDER.

P) POLYTERM is the variance analysis similarly induced from an invariance analysis POLY based on the New Polka Polyhedra library [30].[5]

PR) A script suggested in [5] that calls the tools described in the POLYRANK distribution [6, 7] with increasingly expensive command-line options.

T) TERMINATOR [14].

These tools, except for TERMINATOR, were all run on a 2GHz AMD64 processor using Linux 2.6.16. TERMINATOR was executed on a 3GHz Pentium 4 using Windows XP SP2. Using different machines is unfortunate but somewhat unavoidable due to constraints on software library dependencies, etc. Note, however, that TERMINATOR running on the faster machine was still slower overall, so the qualitative results are meaningful. In any case, the running times are somewhat incomparable since on failed proofs TERMINATOR produces a counterexample path, OCTATERM and POLYTERM give a suspect pair of states, while POLYRANK gives no information. Also, note that the script used to call POLYRANK will never terminate for a divergent input program; the tool may quickly fail for a given set of command-line options, but the script will simply try increasingly expensive options forever.

Fig. 4 contains the results from the experiments performed with these provers. For example, Fig. 4(a) shows the outcome of the provers on example programs included in the OCTANAL distribution. Example 3 is an abstracted version of heapsort, and Example 4 of bubblesort. In this case OCTATERM is the clear winner of the tools. POLYRANK performs poorly on these cases due to the fact that any fully-general translation scheme from programs with full-fledged control-flow graphs to POLYRANK's input format will at times confuse the domain-specific rank-function search heuristics used in POLYRANK.

Fig. 4(b) contains the results from experiments with the 4 tools on examples from the POLYRANK distribution.[6] The examples can be characterized as small but famously difficult (*e.g.* McCarthy's 91 function). We can see that, in these cases, neither TERMINATOR nor the induced provers can beat POLYRANK's hand-crafted heuristics. POLYRANK is designed to support very hard but also carefully expressed examples. In this case each of these examples from the POLYRANK distribution are written such that POLYRANK's heuristics find a termination argument.

Fig. 4(c) contains the results of experiments on fragments of Windows device drivers. These examples are small because we currently must hand-translate them before applying all of the tools but TERMINATOR. In this case OCTATERM again beats the competition. However, we should keep in mind that the examples from this suite that were passed to TERMINATOR contained pointer aliasing, whereas aliasing was removed by hand in the translations used with POLYRANK, OCTATERM and POLYTERM.

From these experiments we can see that the technique of inducing variance analyses with VARIANCEANALYSIS is promising. For programs of medium difficulty (*i.e.* Fig. 4(a) and Fig. 4(c)), OCTATERM is many orders of magnitude faster than the existing program termination tools for imperative programs.

Remark: On Octagon versus Polyhedra for variance analysis. Example 1 from Fig. 4(b) demonstrates that, by moving to a more precise abstract domain (*i.e.* moving from Octagon to Polyhedra), we get a more powerful induced variance analysis. For another

[5] POLY uses the same code base as OCTANAL but calls an OCaml module for interfacing with New Polka, provided with the OCTANAL distribution.

[6] Note also that there is no benchmark number 5 in the original distribution. We have used the same numbering scheme as in the distribution so as to avoid confusion.

	1		2		3		4		5		6	
O	0.11	✓	0.08	✓	6.03	✓	1.02	✓	0.16	✓	0.76	✓
P	1.40	✓	1.30	✓	10.90	✓	2.12	✓	1.80	✓	1.89	✓
PR	0.02	✓	0.01	✓	T/O	-	T/O	-	T/O	-	T/O	-
T	6.31	✓	4.93	✓	T/O	-	T/O	-	33.24	✓	3.98	✓

(a) Results from experiments with termination tools on arithmetic examples from the Octagon Library distribution.

	1		2		3		4		6		7		8		9		10		11		12	
O	0.30	†	0.05	†	0.11	†	0.50	†	0.10	†	0.17	†	0.16	†	0.12	†	0.35	†	0.86	†	0.12	†
P	1.42	✓	0.82	✓	1.06	†	2.29	†	2.61	†	1.28	†	0.24	†	1.36	✓	1.69	†	1.56	†	1.05	†
PR	0.21	✓	0.13	✓	0.44	✓	1.62	✓	3.88	✓	0.11	✓	2.02	✓	1.33	✓	13.34	✓	174.55	✓	0.15	✓
T	435.23	✓	61.15	✓	T/O	-	T/O	-	75.33	✓	T/O	-	T/O	-	T/O	-	T/O	-	T/O	-	10.31	†

(b) Results from experiments with termination tools on arithmetic examples from the POLYRANK distribution.

	1		2		3		4		5		6		7		8		9		10	
O	1.42	✓	1.67	⊘	0.47	⊘	0.18	✓	0.06	✓	0.53	✓	0.50	✓	0.32	✓	0.14	⊘	0.17	✓
P	4.66	✓	6.35	⊘	1.48	⊘	1.10	✓	1.30	✓	1.60	✓	2.65	✓	1.89	✓	2.42	⊘	1.27	✓
PR	T/O	-	T/O	-	T/O	-	T/O	-	0.10	✓	T/O	-	T/O	-	T/O	-	T/O	-	0.31	✓
T	10.22	✓	31.51	⊘	20.65	⊘	4.05	✓	12.63	✓	67.11	✓	298.45	✓	444.78	✓	T/O	-	55.28	✓

(c) Results from experiments with termination tools on small arithmetic examples taken from Windows device drivers. Note that the examples are small as they must currently be hand-translated for the three tools that do not accept C syntax.

Figure 4. Experiments with 4 termination provers/analyses. **O** is used to represent OCTATERM, an Octagon-based variance analysis. **P** is POLYTERM, a Polyhedra-based variance analysis. The **PR** rows represent the results of POLYRANK [5]. **T** represents TERMINATOR [14]. Times are measured in seconds. The timeout threshold was set to 500s. ✓="a proof was found". †="false counterexample returned". T/O = "timeout". ⊘="termination bug found". Note that pointers and aliasing from the device driver examples were removed by a careful hand translation when passed to the tools **O**, **P**, and **PR**.

example of how the Polyhedra-based variance analysis is more precise, consider the following program fragment:

```
while(x+y>z) {
    if(nondet()) {
        x=x-1;
    } else {
        if(nondet()) {
            y=y-1;
        } else {
            z=z+1;
        }
    }
}
```

POLYTERM can prove that this program is terminating when execution starts in a state where both x and y are larger than z, but OCTATERM reports a false bug because the Octagon domain only tracks two-variable inequalities.

Remark: On Disjunction. As mentioned above, if the underlying abstract domain of an induced termination analyzer does not support some level of disjunction, then the termination analysis results are likely to be quite imprecise. Because disjunctive completion is expensive (exponential) and there is no canonical solution, abstract orders and widening operations must be tailored for the application. For our present empirical evaluation we use an extraction method after the fixed-point analysis has been performed in order to find disjunctive invariance/variance assertions. The precise degree of dependence that termination proofs have on disjunctive completion, or an approximation thereof, is an important direction for future work that we hope the existence of the VARIANCEANALYSIS algorithm will catalyze.

6. Variance analyses from shape analyses

SONAR [3] is an invariance analysis tool which tracks the sizes of summarized or abstracted heap structures. SONAR was first used in the MUTANT termination prover, which implements an algorithm from which that in Fig. 1 is generalized. MUTANT has been used to prove the termination of Windows OS device driver dispatch routines whose termination depends on arguments about the changing shape of the heap during the dispatch routine's execution. Due to isolation, SONAR's induced variance analysis (*i.e.,* the analysis resulting from SONAR and Fig. 1) is more powerful than the original MUTANT. As an example consider the following loop where we assume, before entering into the loop, that `x` is a pointer to a circular list:

```
1       z = x;
2       do {
3           z = z->next;
4           y = z;
5           while (y != x) {
6               y = y->next;
7           }
8       } while (z != x)
```

SONARTERM can prove this example terminating, while MUTANT cannot.

SONARTERM is an interesting case of an induced variance analysis, as it demonstrates how SEED does not need to be the most precise approximation of the diagonal relation, and it is also an example of how WELLFOUNDED can do additional abstraction on an already abstract state before attempting to prove it well-founded.

Elements of SONAR's abstract domain $D^\sharp$ are of the form $\Pi \wedge \Sigma$, where Σ is a spatial formula represented as a $*$-conjoined set of

possibly inductive predicates expressed in separation logic [40], and Π is a conjunction of arithmetic inequalities over variables DVar that describe the number of inductive unwindings (depth) of the inductive predicates in Σ. SONAR is path-sensitive in a way that can be expressed as a control flow based trace partitioning [33] where the analyzer dynamically computes a partition by merging partitions when the reachable states can still be precisely represented. Alternately, this trace partitioning can be seen as a dynamically computed control-flow graph elaboration *a la* [41]. As for the numerical domains, in order to define the projection $\mathsf{pc}^\sharp$ required by an over-approximation (Definition 10), we assume that the Π part of each abstract state includes a unique equality $\mathsf{pc} = c$ for a fixed program location constant c.

Before presenting the details of the SONAR instantiation we begin with a small example. Consider an abstract state s such that

$$s \triangleq \mathsf{ls}^k(x,y) * \mathsf{ls}^{k'}(y,x) \wedge k{>}0 \wedge k'{>}0$$

This is an invariant of the loop at line 5 of the example above and, informally speaking, s states that x is a pointer to a linked list segment such that following the trail of pointers in the `next` fields for k steps (for some k) will lead to a node at address y. Note that, if we follow the `next` fields from y (for k' steps), we will get back to the original node at x. Additionally, due to the $*$, we know that there is no aliasing between the first and second lists: they occupy disjoint memory. In this case $\text{SEED}(s)$ equals:

$$\mathsf{ls}^k(x,y) * \mathsf{ls}^{k'}(y,x) \wedge k{>}0 \wedge k'{>}0 \wedge k{=}k_s \wedge k'{=}k'_s$$

Note that we are only copying arithmetic variables, not pointers. If we symbolically execute this new state through the instruction sequence `y = y->next; assume(y!=x);` then this could lead to the symbolic state s' (amongst others):

$$s' \triangleq \mathsf{ls}^k(x,y) * \mathsf{ls}^{k'}(y,x) \wedge k_s{>}0 \wedge k'{>}0 \wedge k{=}k_s{+}1 \wedge k'{=}k'_s{-}1$$

$\text{WELLFOUNDED}(s')$ will project a relation between states (k_s, k'_s) and (k, k') such that

$$k_s{>}0 \wedge k'{>}0 \wedge k{=}k_s{+}1 \wedge k'{=}k'_s{-}1$$

This relation can be proved well-founded by both RANKFINDER and POLYRANK.

***Instance of* SEED *and* WELLFOUNDED.** For SONARTERM, we assume a partitioning $\text{GVar} \cup \text{PVar}$ of the set of variables Var, and assume a set of depth variables $\text{DVar} \subset \text{PVar}$. We assume that $\mathsf{pc} \in \text{PVar} \setminus \text{DVar}$, and that the program neither reads from nor writes to the ghost variables GVar. The set of concrete program states D_C is then defined:

$$\begin{array}{rlrl} \text{GStack} & \triangleq \text{GVar} \to \text{Val} & \text{PStack} & \triangleq \text{PVar} \to \text{Val} \\ \text{Stack} & \triangleq \text{Var} \to \text{Val} & \text{Heap} & \triangleq \text{Loc} \rightharpoonup_{\text{fin}} \text{Val} \\ \text{GState} & \triangleq \text{GStack} \times \text{Heap} & \text{PState} & \triangleq \text{PStack} \times \text{Heap} \end{array}$$
$$D_C \triangleq \text{Stack} \times \text{Heap}$$

We assume a bijective mapping $\rho : \text{PVar} \to \text{GVar}$, thus giving us an isomorphism

$$\text{GStack} \times \text{PStack} \;\cong\; \text{PStack} \times \text{PStack} \;\cong\; \text{Stack}$$

which then yields isomorphisms

$$\text{GState} \times \text{PState} \;\cong\; \text{PState} \times \text{PState} \;\cong\; D_C$$

to obtain $\langle \cdot, \cdot \rangle : S_P \times S_P \to D_C$ (where $S_P = \text{PState}$), the bijection required by a Ghostly Decomposition (Definition 11).

The following four equations define an instantiation for the operations required to induce a variance analysis from SONAR:

$$\begin{array}{rcl} \text{SEED}(\Pi \wedge \Sigma) & \triangleq & (\Pi \wedge \Sigma \wedge \bigwedge_{v \in \text{fDV}(\Pi \wedge \Sigma)} \{v = \rho(v)\}) \\ \text{SEED}(\top) & \triangleq & \top \end{array}$$
$$\begin{array}{rcl} \text{WELLFOUNDED}(\Pi \wedge \Sigma) & \triangleq & \text{WFCHECK}(\rho(\text{DVar}), \text{DVar}, \Pi) \\ \text{WELLFOUNDED}(\top) & \triangleq & \mathsf{false} \end{array}$$

where $\text{fDV}(\Pi \wedge \Sigma)$ denotes the set of depth variables appearing in $\Pi \wedge \Sigma$, and WFCHECK could be tools such as POLYRANK or RANKFINDER. The domain element $\top$ is used in SONAR to represent the case where memory-safety could not be established by the abstract interpretation. Notice that these definitions ignore the spatial part Σ, and treat only the depth variables. In particular, WELLFOUNDED is constant in the spatial part, and SEED plants no information about the spatial part. In this way, SEED is not the best approximation of the diagonal relation, and so is an example that exercises the looseness of Definition 15.

The bijection for ghostly decomposition and WELLFOUNDED/ SEED operations just defined are the necessary additions to the SONAR invariance analysis to obtain the SONARTERM variance analysis. We refer to [3] for the remaining details of the SONAR analysis.

7. Related work

A number of termination proof methods and tools have been reported in the literature. Examples include the size-change principle for purely functional programs (*e.g.* [31]), the dependency pairs approach for term-rewrite systems (*e.g.* [26]), rank-function synthesis for imperative programs with linear arithmetic assignments (*e.g.* [6, 9, 10, 42, 38]) and even non-linear imperative programs [16]. Many of these tools and techniques use specific and fixed abstractions. The size-change principle, for example, builds an analysis around the program's call-graph. Rank-function synthesis techniques, for example, support limited control-flow graph structures (*e.g.* single unnested while loops, perhaps without conditionals), meaning that support for general purpose programs requires an abstraction before rank synthesis can be performed. The work presented here is not tied to a fixed abstraction.

This paper generalizes the previous work on MUTANT in [3]. Thus, given that it is a generalization, some overlap in contributions is to be expected. SONAR's induced variance analysis, SONARTERM, is unique to this paper and is also more accurate than MUTANT. There, we described a specific termination checking tool, concentrating on a particular domain (shape analysis) and focusing much of our attention to the underlying invariance analysis used (SONAR). Here, we introduce the notion of variance analysis, describe a general method of implementing the analysis with invariance analyses, show the general algorithm at work in several contexts, and give a proof of the soundness of parameterized variance analyses. These contributions are all unique to the new work.

The work in this paper builds on the fundamental result of [37] on disjunctively well-founded relations, which shows that a relation Rel is well-founded if and only if its transitive closure Rel^+ is a subset of a finite union $T_1 \cup \cdots \cup T_n$ of well-founded relations (called a transition invariant). The result in [37] led to a method of constructing termination arguments using counterexample-guided refinement [12]. The idea is that when an inclusion check $R^+ \subseteq T_1 \cup \cdots \cup T_i$ fails (so that we do not have an over-approximation), a counterexample can be used to produce a new well-founded relation T_{i+1}. Then if the inclusion $R^+ \subseteq T_1 \cup \cdots \cup T_i \cup T_{i+1}$ holds, well-foundedness of R has been established; if not the abstraction can be refined again using a counterexample.

TERMINATOR is a symbolic model checker for termination and liveness properties that is based on this idea, as described in [11,

13, 14]. The difficulty in TERMINATOR is that checking the validity of the termination argument (*i.e.* checking the invariance property that proves the validity of the termination argument) is extremely expensive. This was shown in Section 5, which presents the first known experimental evaluation of tools like TERMINATOR and POLYRANK.

In contrast, here we directly compute an over-approximation, $T_1 \cup \cdots \cup T_n$, where the inclusion $R^+ \subseteq T_1 \cup \cdots \cup T_n$ holds by construction. Unlike in TERMINATOR, the T_i are not guaranteed to be well-founded: we have to check that. As we have seen, this has some advantages as regards speed and the ability to tune precision.

The work here takes the opposite perspective from TERMINATOR. We show how an over-approximating $T_1 \cup \cdots \cup T_n$ can quickly be computed using an off-the-shelf abstract interpretation. We do not need to check the inclusion, as TERMINATOR must, because it holds by construction. But we must still check that each T_i is well-founded. So, although justified by the same result on disjunctively well-founded relations, we use of this result in a fundamentally different way than does TERMINATOR: we get over-approximation by construction, but have to check for well-foundedness, while TERMINATOR's candidate termination arguments $T_1 \cup \cdots \cup T_i$ are disjunctively well-founded by construction, but it may not be an over-approximation. Our technique is not dependent on counterexample-driven refinement or on predicate abstraction.

The work in [37] uses the result on disjunctively well-founded relations to justify several inference rules for *transition invariants*; a transition invariant is an over-approximation of the transition relation of a program, when restricted to reachable states. It would be straightforward to modify the VARIANCEANALYSIS algorithm to compute transition invariants. As it stands, given a program with transition relation R, our variance assertions can be seen as transition invariants for the transitive closure R^+, restricted to where the start and end states of the transition relation are at the same program location. They are what we need to reason about whether a location is visited infinitely often, and to formulate the more refined local termination predicates which give added modularity to our analysis.

The notions of seeding and ghost variables are basic and have been used many times [3, 4, 16, 32, 39].

8. Conclusion

We have introduced the notion of a variance assertion together with a class of tools called variance analyses. Furthermore, we have developed a generic method of inducing variance analyses from invariance analyses, and shown how several analyses developed through the method lead to termination provers that are competitive with known termination provers.

The proposed approach has several unique advantages over known existing approaches. For example: The induced variance analyses reuse the machinery of the underlying invariance analysis to quickly and automatically find a (disjunctive) candidate termination argument. The argument itself can often be expressed using a less precise, and thus more efficient, domain—*e.g.* Octagon versus Polyhedra as in Section 2.

Furthermore, the induced variance analyses are truly *analyses*. That is, like invariance analyses infer invariance assertions, the induced variance analyses infer variance assertions. As invariance assertions can be used to prove safety properties (amongst other uses), variance assertions are useful when proving termination and probably other liveness properties.

For example, if termination cannot be proved directly by proving well-foundedness of each disjunct in the VARIANCEANALYSIS algorithm, the well-founded subset of the variance assertion could be passed to a tool like TERMINATOR which could directly use it during further attempts to prove termination or other liveness properties. This would undoubtedly lead to faster termination provers, with at least as much precision as is now possible. A more interesting question is: *Can the combination of approaches prove new programs terminating that cannot be proved terminating with the approaches separately?* This arises from the fact that TERMINATOR sometimes suffers in cases where the loop variables are initialized to constant values (*e.g.* `for(i=0;i<N;i++);`). In this case TERMINATOR might (as it is based on predicate abstraction [27]) produce an infinite number of predicates (`i==0`, `i==1`, etc). TERMINATOR has heuristics that attempt to mitigate this issue, but these techniques often fail. OCTATERM and POLYTERM, for example, do not suffer from this problem. Thus, we might be able to use the information from failed runs of OCTATERM to help TERMINATOR overcome its limitations in contexts like this.

Finally, liveness properties (including termination) for finite-state systems amount to invariance properties. All known model checkers that support liveness properties for finite-state systems make use of this fact. Biere *et al.* [4] go so far as to simply convert the liveness checking problem for finite-state systems into safety. Thus, liveness and termination only become distinct from safety in the context of infinite-state systems. It is known, though, that liveness properties for infinite state systems can be reduced to fair termination (see [11, 43]). It therefore seems likely that the ideas in this paper could be used for liveness analyses other than termination.

The variance analyses we have given as instances of this paper's proposed algorithm are built from invariance analyses that are concerned with established ingredients of termination provers: linear arithmetic and size of data structures. However, there exist many invariance analyses built on a broad spectrum of abstractions. It will be exciting to see what kinds of variance analyzers will come of combining these abstract domains with VARIANCEANALYSIS.

Acknowledgments. We are grateful to Domagoj Babic, Nick Benton, Aaron Bradley, Andreas Blass, Patrick Cousot, Georges Gonthier, Alexey Gotsman, Arie Gurfinkel, Antoine Miné, Andreas Podelski, Helmut Veith and Hongseok Yang for discussions and comments that helped to improve the paper. Chawdhary was supported by a Microsoft PhD studentship. Distefano and O'Hearn acknowledge the support of the EPSRC.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Priciples, Techniques, and Tools*. 1986.

[2] A. W. Appel. *Modern Compiler Implementation in ML*. 1998.

[3] J. Berdine, B. Cook, D. Distefano, and P. O'Hearn. Automatic termination proofs for programs with shape-shifting heaps. In *CAV'06: International Conference on Computer Aided Verification*, 2006.

[4] A. Biere, C. Artho, and V. Schuppan. Liveness checking as safety checking. In *FMICS'02: Formal Methods for Industrial Critical Systems*, 2002.

[5] A. Bradley. Personal communication. Aaron Bradley's suggested script that iteratively applies the tools described in [7] and [6] with increasingly expensive options, June 2006.

[6] A. Bradley, Z. Manna, and H. Sipma. Termination of polynomial programs. In *VMCAI'05: Verification, Model Checking, and Abstract Interpretation*, 2005.

[7] A. R. Bradley, Z. Manna, and H. B. Sipma. The polyranking principle. In *ICALP'05: International Colloquium on Automata, Languages and Programming*, 2005.

[8] C. Calcagno, D. Distefano, P. O'Hearn, and H. Yang. Beyond reachability: Shape abstraction in the presence of pointer arithmetic. In *SAS'06: Static Analysis Symposium*, 2006.

[9] M. Colón and H. Sipma. Synthesis of linear ranking functions. In *TACAS'01: Tools and Algorithms for the Construction and Analysis of Systems*, 2001.

[10] M. Colón and H. Sipma. Practical methods for proving program termination. In *CAV'02: International Conference on Computer Aided Verification*, 2002.

[11] B. Cook, A. Gotsman, A. Podelski, A. Rybalchenko, and M. Vardi. Proving that programs eventually do something good. In *POPL'06: Principles of Programming Languages*, 2006.

[12] B. Cook, A. Podelski, and A. Rybalchenko. Abstraction refinement for termination. In *SAS'05: Static Analysis Symposium*, 2005.

[13] B. Cook, A. Podelski, and A. Rybalchenko. Termination proofs for systems code. In *PLDI'06: Programming Language Design and Implementation*, 2006.

[14] B. Cook, A. Podelski, and A. Rybalchenko. Terminator: Beyond safety. In *CAV'06: International Conference on Computer Aided Verification*, 2006.

[15] P. Cousot. The calculational design of a generic abstract interpreter. In M. Broy and R. Steinbrüggen, editors, *Calculational System Design*. 1999.

[16] P. Cousot. Proving program invariance and termination by parametric abstraction, Lagrangian relaxation and semidefinite programming. In *VMCAI'05: Verification, Model Checking, and Abstract Interpretation*, 2005.

[17] P. Cousot. Personal communication, 2006.

[18] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL'77: Principles of Programming Languages*, 1977.

[19] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *POPL'79: Principles of Programming Languages*, 1979.

[20] P. Cousot and R. Cousot. Abstract interpretation frameworks. J. Log. Comput. 2(4), pp511-547, 1992.

[21] P. Cousot and R. Cousot. Modular static program analysis. In *CC'02: Conference of Compiler Construction*, 2002.

[22] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The ASTRÉE analyzer. In *ESOP'05: European Symposium on Programming*, 2005.

[23] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL'78: Principles of Programming Languages*, 1978.

[24] D. Distefano, P. W. O'Hearn, and H. Yang. A local shape analysis based on separation logic. In *TACAS'06: Tools and Algorithms for the Construction and Analysis of Systems*, 2006.

[25] R. W. Floyd. Assigning meanings to programs. In *Mathematical Aspects of Computer Science*, 1967.

[26] J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke. Automated termination proofs with AProVE. In *RTA'04: Rewriting Techniques and Applications*, 2004.

[27] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *CAV'97: International Conference on Computer Aided Verification*, 1997.

[28] B. S. Gulavanii and S. K. Rajamani. Counterexample driven refinement for abstract interpretation. In *TACAS'06: Tools and Algorithms for the Construction and Analysis of Systems*, 2006.

[29] H. Jain, F. Ivancic, A. Gupta, I. Shlyakhter, and C. Wang. Using statically computed invariants inside the predicate abstraction and refinement loop. In *CAV'06: International Conference on Computer Aided Verification*, 2006.

[30] B. Jeannet. NewPolka polyhedra library. http://pop-art.inrialpes.fr/people/bjeannet/newpolka/index.html.

[31] C. S. Lee, N. D. Jones, and A. M. Ben-Amram. The size-change principle for program termination. In *POPL'01: Principles of Programming Languages*, 2001.

[32] Z. Manna and A. Pnueli. Axiomatic approach to total correctness of programs. *Acta Informatica*, 1974.

[33] L. Mauborgne and X. Rival. Trace partitioning in abstract interpretation based static analyzers. In *ESOP'05: European Symposium on Programming*, 2005.

[34] A. Miné. The Octagon abstract domain. *Higher-Order and Symbolic Computation*. (to appear).

[35] A. Pnueli. The temporal logic of programs. In *18th IEEE Symposium on Foundations of Computer Science*, 1977.

[36] A. Podelski and A. Rybalchenko. A complete method for the synthesis of linear ranking functions. In *VMCAI'04: Verification, Model Checking, and Abstract Interpretation*, 2004.

[37] A. Podelski and A. Rybalchenko. Transition invariants. In *LICS'04: Logic in Computer Science*, 2004.

[38] A. Podelski and A. Rybalchenko. Transition predicate abstraction and fair termination. In *POPL'05: Principles of Programming Languages*, 2005.

[39] J. C. Reynolds. *The Craft of Programming*. London, 1981.

[40] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS'02: Symposium on Logic in Computer Science*, 2002.

[41] S. Sankaranarayanan, F. Ivancic, I. Shlyakhter, and A. Gupta. Static analysis in disjunctive numerical domains. In *SAS'06: Static Analysis Symposium*, 2006.

[42] A. Tiwari. Termination of linear programs. In *CAV'04: International Conference on Computer Aided Verification*, 2004.

[43] M. Y. Vardi. Verification of concurrent programs: The automata-theoretic framework. *Ann. Pure Appl. Logic*, 51(1-2):79–98, 1991.