Refining Abstract Interpretations

Bhargav S. Gulavani[a], Supratik Chakraborty[a], Aditya V. Nori[b], Sriram K. Rajamani[b]

[a] *IIT Bombay*
[b] *Microsoft Research Bangalore*

Abstract

Abstract interpretation techniques prove properties of programs by computing abstract fixpoints. All such analyses suffer from the possibility of false errors. We present a dag-based abstraction refinement technique to automatically refine such abstract interpretations to reduce false errors. This technique refines precision loss due to widen operator by a new interpolated widen operator and refines precision loss due to join operator by disjunctions. We prove the soundness and progress properties of this abstraction refinement procedure.

Key words: Program correctness, Abstract interpretation, Interpolation, CEGAR

1. Introduction

Abstract interpretation [5] is a general technique to compute sound fixpoints for programs. Such fixpoint computations have to lose precision in order to guarantee termination. However, precision losses can lead to false errors. Recently, there has been an increased interest in automatically refining abstract interpretations. A first attempt in this direction was made in [8] where widen was refined by convex hull in the polyhedra domain. In [7] we improved upon these techniques by designing (a) an interpolated widen operator that refines widen using interpolants, (b) a new algorithm to implicitly handle disjunctions that occur during refinement, and (c) a new algorithm to refine abstract interpretations that use the join operator to merge abstract states at program locations where conditional branches merge. We have built a tool Dagger that implements these ideas. In [7] we compared Dagger with recent related works and presented its empirical evaluation. These results showed that Dagger outperforms a number of available tools such as Slam [2], Blast [10], Armc [11], and our earlier tool [8] on a range of benchmarks.

The focus of our earlier work [7] was on the practical implementation of dag based abstraction refinement techniques in Dagger, and on the evaluation of Dagger's performance and accuracy vis-a-vis other tools. In [7], we had therefore explained only the basic principles of abstraction and refinement required for implementing Dagger. In the present paper, we provide an in-depth formal analysis of our dag-based abstraction and refinement algorithms, focussing on soundness and progress properties. This provides theoretical guarantees about our technique that are extremely useful (especially progress guarantees) in a practical setting.

Preliminaries. Let V be a finite set of variables, each over potentially infinite domains. A state s is a valuation of all variables in V. Let Σ be the (possibly infinite) set of all possible states. A program P_V over a set of variables V is a six-tuple $(L, E, R, l_0, l_x, \mathsf{Image}^\flat)$, where (i) L is a finite set of control locations in the program, representing possible valuations of the program counter, (ii) $E \subseteq L \times L$ is a set of control flow edges, (iii) $R \subseteq L$ is a set of error locations, (iv) l_0 is the initial program location, which is not in R, and cannot be the target of any control flow edge, (v) l_x is the exit location of the program, and cannot be the source of any control flow edge, and (vi) $\mathsf{Image}^\flat$ is a function $\wp(\Sigma) \times E \to \wp(\Sigma)$. $\mathsf{Image}^\flat(\{\sigma\}, l, l')$ returns the states obtained by starting at state σ and executing the statements along the control flow edge $(l, l') \in E$. The preimage operation $\mathsf{Preimage}^\flat$ is defined as the inverse of the image operation. We overload the $\mathsf{Image}^\flat$ and $\mathsf{Preimage}^\flat$ operations to operate over a set of states and sequence of edges in the obvious way.

We assume that the control flow graphs of our programs are connected reducible graphs (i.e. arising from structured programs without goto's), and that every location has either one or two incoming control flow edges. We also assume that locations in R have exactly one incoming edge. An edge $e \in E$ is a *backedge* if it closes a cycle during a depth first traversal of the graph $\langle L, E\rangle$, starting at location l_0. A location l is called a *merge* location if it has two incoming edges, neither of which is a backedge. A location l is called a *loophead* if it has two incoming edges, and exactly one is a backedge

A control location l is said to be reachable if there exists a path $(l_0, l_1, \ldots, l_n, l)$ in the control flow graph, such that $\mathsf{Image}^\flat(\Sigma, (l_0, l_1, \ldots, l_n, l))$ is not $\emptyset$. Our goal is to check if any error location $l_e \in R$ is reachable. A true counterexample cx is a sequence of control flow locations $(l_0, l_1, \ldots, l_n, l_e)$ such that $l_e \in R$ and $\mathsf{Image}^\flat(\Sigma, cx) \neq \emptyset$. The counterexample $cx : (l_0, l_1, \ldots, l_n, l_e)$ is called spurious if $\mathsf{Image}^\flat(\Sigma, cx) = \emptyset$, and its length is $n + 1$.

Following [5], we use abstract interpretation of a program P_V over an abstract domain $\langle \Sigma^\sharp, \sqsubseteq, \top, \bot, \sqcup, \sqcap\rangle$ that is assumed to be a complete lattice. The abstract domain is related to the concrete domain $\langle \wp(\Sigma), \subseteq, \Sigma, \emptyset, \cup, \cap\rangle$ by a Galois connection (α, γ). In particular, we consider abstract domains where elements of $\Sigma^\sharp$ are formulas over V in a fragment of first order logic closed under Craig interpolation [6]. Every formula represents a set of concrete states. The abstract image operation $\mathsf{Image}(s, l, l')$ takes an abstract element s and a control flow edge $(l, l') \in E$ and returns the abstract element s' obtained by abstractly executing the statements along the control flow edge (l, l'). The abstract $\mathsf{Preimage}$ operation is analogously defined. We assume that the Image and $\mathsf{Preimage}$ operations are exact, i.e., for any $s \in \Sigma^\sharp$ and $(l, l') \in E$, $\mathsf{Image}^\flat(\gamma(s), l, l') = \gamma(\mathsf{Image}(s, l, l'))$ and $\mathsf{Preimage}^\flat(\gamma(s), l, l') = \gamma(\mathsf{Preimage}(s, l, l'))$.

Widen. The widen [5] operator $\nabla : \Sigma^\sharp \times \Sigma^\sharp \to \Sigma^\sharp$ is a binary operator such that for all $s_1, s_2 \in \Sigma^\sharp$, we have (i) $s_1, s_2 \sqsubseteq s_1 \nabla s_2$, and (ii) for any increasing sequence $s_0 \sqsubseteq s_1 \sqsubseteq \ldots$, the sequence defined by $s'_0 = s_0$, and $s'_k = s'_{k-1} \nabla s_k$, for all $k \geq 1$, eventually stabilizes, i.e., $\exists i \geq 0$ such that $s'_j = s'_i$ for all $j \geq i$.

The *bounded widen* [3] operator $\nabla_T : \Sigma^\sharp \times \Sigma^\sharp \to \Sigma^\sharp$ is a widen operator parameterized by a set T of abstract elements such that for any $s_t \in T$, if $s_1 \sqsubseteq s_t$ and $s_2 \sqsubseteq s_t$, then $s_1 \nabla_T s_2 \sqsubseteq s_t$.

Interpolant. For any two elements $s, s_e \in \Sigma^\sharp$ such that $s \sqcap s_e = \bot$, $s_I \in \Sigma^\sharp$ is said to be an interpolant [6, 9] of s and s_e if (i) $s \sqsubseteq s_I$, (ii) $s_I \sqcap s_e = \bot$, and (ii) the formula representing s_I has only those free variables that are common to the formulas representing s and s_e.

Interpolated Widen. Let s be the abstract state computed at a loophead at some point during abstract fixpoint computation, and let s' be the abstract state at the same loophead after one more execution of the loop body. Let s_e denote the abstract error state at the loophead, obtained by backward propagation (using the Preimage opertion) from an error location. The widen operator $s \nabla s'$ is used to guarantee termination when computing an abstract fixpoint. However, the imprecision introduced by widen may lead to false errors. This can happen if $(s \sqcup s') \sqcap s_e = \bot$, but $(s \nabla s') \sqcap s_e \neq \bot$. In this case, we propose to pick an interpolant s_I of $s \sqcup s'$ and s_e, and use a bounded widen operator $s \nabla_{\{s_I\}} s'$ to compute the abstract fixpoint. Such a bounded widen operator that uses interpolants as bounds is called *interpolated widen*. Note that since $s \sqcup s' \sqsubseteq s_I$, $s \nabla_{\{s_I\}} s'$ is guaranteed to be smaller than s_I in $\sqsubseteq$ ordering, and hence $(s \nabla_{\{s_I\}} s') \sqcap s_e = \bot$. Further, since $s \nabla_T s'$ is a valid widening operator for any finite set T of bounds, the abstract fixpoint computation is guaranteed to terminate.

A primary insight of interpolated widen is captured by the following lemma, which says that if the parameter T of a bounded widen contains an interpolant, then a false error that occurs due to the imprecision of widen can be avoided. Its proof follows from the definitions of bounded widen and interpolants.

Lemma 1. *Let $s, s', s_e \in \Sigma^\sharp$ such that $(s \sqcup s') \sqcap s_e = \bot$. Let $s_I \in \Sigma^\sharp$ be an interpolant of $(s \sqcup s')$ and s_e, and let $T \subseteq \Sigma^\sharp$ be any set such that $s_I \in T$. Then $(s \sqcup s') \sqsubseteq (s \nabla_T s') \sqsubseteq s_I$ and $(s \nabla_T s') \sqcap s_e = \bot$.*

2. Dag-based abstraction refinement

Our abstraction algorithm AbstractDAG (shown in Fig. 1) performs abstract fixpoint computation on the control flow graph of a given program. It stores the intermediate states obtained during the computation in the form of a dag. It joins states computed along different paths to the same merge location by $\sqcup$ operator and uses ∇_T operator at loopheads to ensure termination of the abstract fixpoint computation. If an error location is found reachable during the computation, it calls the refinement algorithm RefineDAG (shown in Fig. 1). RefineDAG either confirms a true counterexample or recovers the precision lost by $\sqcup$ and ∇_T operators in order to eliminate spurious counterexamples.

```
/*Global: Program P_V, interpolant set T,
abstract computation dag G*/
1. AbstractDAG
n_0 ← ⟨l_0, ⊤⟩; T ← ∅
l′ ← l_0
loop
    if l′ is a merge location
        Create a new node n′ = ⟨l′, ⊥⟩; s′ ← ⊥
        for all n = ⟨l, s⟩ such that (l, l′) ∈ E
            if ¬Covered(l′, Image(s, l, l′)) then
                s′ ← s′ ⊔ Image(s, l, l′)
                Add an edge from n to n′
        end for
        if s′ = ⊥ then Delete n′
        else n′ ← ⟨l′, s′⟩
    else
        for all n = ⟨l, s⟩ such that (l, l′) ∈ E
            img ← Image(s, l, l′)
            if ¬Covered(l′, img) then
                if l′ is loophead then
                    s′′ ← Sel(S̃_{l′,n}, img)
                    img ← s′′ ∇_T (s′′ ⊔ img)
                Add n′ = ⟨l′, img⟩ as child of n
        end for
    if ∃n_e = ⟨l_e, s_e⟩, l_e ∈ R then
        l′ ← RefineDAG(n_e)
    else if l′ is prog. exit location then
        "program correct"; exit
    else
        l′ ← NextInWTOOrder(l′)
end loop
```

```
2. RefineDAG(n_e)
BackPropagateError(n_e)
If err(n_0) ≠ {⊥} then "prog. incorrect"; exit
S ← {n = ⟨l, s⟩ | s ⩀ err(n) ≠ {⊥}}
for all n′ = ⟨l′, s′⟩ ∈ S in rev topological order
    if l′ is a merge location then
        if ∃ immediate predecessor p = ⟨l, s⟩ of
            n′, and s ⩀ err(p) = {⊥} then
            DeleteDescendents(n′)
            Split(n′)
    else
        Let ⟨l, s⟩ = Parent(n′), ψ = err(n′)
        if Image(s, l, l′) ⊓ ψ = ⊥ then
            /* n′ is a refinement node;
            and l′ is a loophead */
            s′ ← ApplyRefinement(n′, l′, ⟨l, s⟩, ψ)
            DeleteDescendents(n′)
end for
Let n = ⟨l, s⟩ = node with least WTO
    index among refined nodes
return NextInWTOOrder(l)
```

```
3 ApplyRefinement(n′, l′, ⟨l, s⟩, ψ)
img ← Image(s, l, l′)
s′′ ← Sel(S̃_{l′,n′}, img)
if (s′′ ⊔ img) ⩀ ψ = {⊥} then
    for all ψ̂ ∈ ψ
        T ← T ∪ Interpolate(s′′ ⊔ img, ψ̂)
    return s′′ ∇_T (s′′ ⊔ img)
else
    return img
```

Figure 1: Abstract and Refine procedures for dag based exploration

During the abstract fixpoint computation, intermediate states are stored in the form of a dag $G = (N, A, n_0)$ where N is the set of nodes, A is the set of edges and n_0 is the root node. Each node in N represents an abstract state that is stored as a pair $\langle l, s\rangle \in L \times \Sigma^\sharp$. The dag thus constructed gives a model to perform counterexample driven refinement whenever an error location is reached during the abstract computation.

Let function $\mathsf{Depth} : N \to \mathbb{N}$ give for each node $n \in N$ the number of edges along the path from n_0 to n. Let $\mathsf{Covered}(l, s)$ be a predicate that returns True iff either $s = \bot$ or there exists a node $n = \langle l, s'\rangle$ in the dag such that $s \sqsubseteq s'$.

In a dag based scheme, when we wish to join abstract states at a merge location, we must have already computed abstract states at all locations leading to the merge location. We therefore use a weak topological ordering (WTO) [4] for traversing control flow locations, and compute the order of nodes to be processed using a recursive iteration strategy. WTO, denoted $\preceq$, is a hierarchical ordering of program locations that groups sets of locations from L, except l_0 and l_x, into components. l_0 and l_x are the first and last elements, respectively, in the WTO order. A component consists of a sequence of locations followed by a (possibly empty) sequence of sub-components and locations. The first location of any component is a loophead. For an edge $u \to v$ of the control flow graph that is not a backedge, we have, $u \preceq v$. In the recursive iteration strategy, we first stabilize (i.e., find a fixpoint of) nested components before stabilizing outer components. Function NextInWTOOrder used by AbstractDAG takes a location l, and returns the location following l in the WTO order, in the current component. If there does not exist a location following l in the current component, it returns the first location of the current component (loophead) if the current component has not yet stabilized; otherwise it returns the first location following the current component in the WTO order.

The computation of $\mathsf{AbstractDAG}$ starts with an abstract dag having a single node $n_0 = \langle l_0, \top\rangle$. Consider a node $n = \langle l, s\rangle$ and control flow edge (l, l'), such that l' is not a merge location. If $\mathsf{Covered}(l', \mathsf{Image}(s, l, l'))$ returns False, a new node $n' = \langle l', s'\rangle$ is added as a child of n in the dag; otherwise a new node is not added. If (l, l') is not a backedge then s' is obtained as $\mathsf{Image}(s, l, l')$. Otherwise s' is computed using interpolated widen as follows.

Let $S_{l,n}$ denote the set of maximal abstract states among the abstract states at the predecessors of node n with location l. Let $\widetilde{S}_{l,n}$ be the sequence of states in $S_{l,n}$ ordered by the depth of their corresponding nodes in the dag. As we will explain later, because of refinements there can be more than one node with maximal abstract state among the predecessors of node n with location l. Therefore, when we reach loopheads, we need to widen a set $S_{l,n}$ of abstract states with the newly computed abstract state $img = \mathsf{Image}(s, l, l')$. We compute the widened result s' as $\mathsf{Sel}(\widetilde{S}_{l',n}, img) \nabla_T (\mathsf{Sel}(\widetilde{S}_{l',n}, img) \sqcup img)$. The function $\mathsf{Sel}(\widetilde{S}_{l',n}, img)$ returns the abstract state s_i from the sequence $\widetilde{S}_{l',n}$ with the largest index i satisfying the property that there does not exist any j such that $(s_j \sqcup img) \sqsubset (s_i \sqcup img)$. This computation defines a new powerset widen operator $\nabla^p_T : \wp(\Sigma^\sharp) \times \Sigma^\sharp \to \wp(\Sigma^\sharp)$ that takes a set $S \subseteq \Sigma^\sharp$ and an element $s \in \Sigma^\sharp$, and returns the set of maximal elements in $S \cup \{\mathsf{Sel}(\widetilde{S}, s)\, \nabla_T\, (\mathsf{Sel}(\widetilde{S}, s) \sqcup s)\}$, where $\widetilde{S}$ is the unique sequence obtained by ordering the elements of S according to (any) predefined order. Lemma 2, presented below, formalizes the convergence property of ∇^p_T. This property ensures that the abstract fixpoint computation is guaranteed to terminate if no refinements are done in between.

For a merge location l', $\mathsf{AbstractDAG}$ joins the abstract states along all paths reaching l' using the $\sqcup$ operator (unless the state computed along a path is already covered), thereby giving rise to a dag. The abstract states for locations that are predecessors of l' in the control flow graph are guaranteed to have been already computed because of the WTO order of processing locations.

Lemma 2. *Let $S_0 \subseteq \Sigma^\sharp$ be a non-empty finite set. Consider the sequence $S_1 = S_0\, \nabla^p_T\, s_0$, $S_2 = S_1\, \nabla^p_T\, s_1, \ldots,$ where $s_i \in \Sigma^\sharp$ for all $i \geq 0$. There exists $u \geq 0$ such that $S_v = S_u$ for all $v \geq u$.*

Proof. Since ∇_T is a widen operator on the base abstract domain $\Sigma^\sharp$, the reflexive, transitive closure of the relation $\{(s_1, s) \in \Sigma^\sharp \times \Sigma^\sharp \mid \exists s_2 \in \Sigma^\sharp, s_1 \sqsubset s_2 \wedge s = s_1 \nabla_T s_2\}$ defines a partial ordering $\vdash_{\nabla_T}$ on $\Sigma^\sharp$ that satisfies the ascending chain condition [1]. It is now straightforward to show by induction on i that $|S_i| \leq |S_0|$ for all $i \geq 0$. This is because, by definition of ∇^p_T, $|S_{i+1} \setminus S_i| \leq 1$, and also, there exists $s' = \mathsf{Sel}(\widetilde{S}_i, s_i) \in S_i$ that is smaller in $\sqsubseteq$ order than $s' \nabla_T (s' \sqcup s_i)$. Hence $s' \notin S_{i+1}$ and $|S_{i+1}| \leq |S_i|$. Similarly it follows from the definition of ∇^p_T that for every $a \in S_{i+1}$, where $i \geq 0$, there exists $b \in S_i$ such that $b \vdash_{\nabla_T} a$. Since $\vdash_{\nabla_T}$ is transitive, it follows that for every $a \in S_i$, there exists $c \in S_0$ such that $c \vdash_{\nabla_T} a$. Thus, every element of every S_i ($i \geq 0$) must be in a non-decreasing chain starting from some element in S_0. Combining the facts that (i) there are a finite number of elements in S_0, (ii) $\vdash_{\nabla_T}$ satisfies the ascending chain condition, and (iii) $|S_i| \leq |S_0|$, it follows immediately that there exists $u \geq 0$ such that $S_v = S_u$ for all $v \geq u$. $\square$

Let us illustrate the abstraction and refinement procedure through an example. Consider a program $\mathtt{l_0} :\ \mathtt{x := 0};\ \mathtt{l_1} : \mathtt{while}(*)\ \{\mathtt{x++}\};\ \mathtt{l_2} : \mathtt{if(x == 0)}$ $\mathtt{l_3} : \mathtt{error}$. The abstract computation starts from the root $n_0 = \langle l_0, \top \rangle$. It first computes a fixpoint for the while loop. In the process it adds the node $n_1 = \langle l_1, x = 0 \rangle$ as a child of n_0 and the node $n_2 = \langle l_1, x \geq 0 \rangle$ as a child of n_1. The state at node n_2 is obtained by applying ∇_T. Furthermore, the node $n_3 = \langle l_2, x \geq 0 \rangle$ is added as a child of n_2 and $n_4 = \langle l_3, x = 0 \rangle$ is added as a child of n_3. This adds a node with error location l_3 to the dag.

If a node $n_e = \langle l_e, s_e \rangle$ with $l_e \in R$ (the set of error locations) gets added to the dag, then an error location is found reachable, and $\mathsf{RefineDAG}(n_e)$ is invoked. The use of a dag G to represent the abstract computation implies that when an error location is reached at node n_e, there are potentially multiple paths from the root n_0 to n_e in G. Let the subgraph of G containing all paths from n_0 to n_e be called the counterexample-dag G_e. We now analyze *all* paths in G_e to determine if n_e is indeed reachable. Every path in G_e corresponds to a unique potential counterexample and conversely, every counterexample in G_e has a unique path associated with it. In our running example, the counterexample dag is simply the path $(n_0, n_1, n_2, n_3, n_4)$.

Procedure $\mathsf{RefineDAG}$ either finds a true counterexample in G_e or if all counterexamples are spurious, it replaces a set of imprecise operations by more precise ones along every path in G_e. Refinement proceeds by first computing a *set of abstract error preimages*, $err(n)$, at each node n in G_e. The function $\mathsf{BackPropagateError}$ computes $err(n)$ by initializing $err(n_e)$ to $\{\top\}$. For a node n, it computes $err(n)$ as the set union of the preimage of every element in $err(n')$, for every successor n' of n in G_e. We also define the operator $\widehat{\cap} : \Sigma^\sharp \times \wp(\Sigma^\sharp) \to \wp(\Sigma^\sharp)$ as $a \,\widehat{\cap}\, S = \{a \sqcap s \mid s \in S\}$. We say that node $n' = \langle l', s' \rangle$ is a *refinement node with respect to predecessor* $n = \langle l, s \rangle$ if $s' \widehat{\cap}\, err(n') \neq \{\perp\}$ and $s \,\widehat{\cap}\, err(n) = \{\perp\}$. The goal of refinement at such a node n' is to improve the precision of computation of s' from s, so that the new abstract state at n' does not intersect $err(n')$. This is always possible because Image and $\mathsf{Preimage}$ operations are assumed to be exact, and hence $\mathsf{Image}(s, l, l') \,\widehat{\cap}\, err(n') = \{\perp\}$.

Refinement is done at node $n' = \langle l', s' \rangle$ in one of the following ways: (i) If l' is a merge location and n' has predecessors $n_1 = \langle l_1, s_1 \rangle, \ldots, n_k = \langle l_k, s_k \rangle$, the function $\mathsf{Split}(n')$ deletes n' and all its incoming edges. Then it creates k new nodes $m_1, \ldots, m_k$, where $m_i = \langle l', \mathsf{Image}(s_i, l_i, l') \rangle$. (ii) If l' is a loophead, we refine node n' with procedure $\mathsf{ApplyRefinement}$, as shown in Fig. 1. The refinement proceeds as follows. Let $\widetilde{S}_{l',n'}$ be the sequence of abstract states at predecessors of n' as defined earlier, $n = \langle l, s \rangle$ be the immediate predecessor of n' in the dag, and img be $\mathsf{Image}(s, l, l')$. If $s'' = \mathsf{Sel}(\widetilde{S}_{l',n'}, img)$ and $(s'' \sqcup img) \widehat{\cap} \psi =$

$\perp$, then we can compute an interpolant between $(s'' \sqcup img)$ and every element of ψ and refine the interpolated widen ∇_T. The result of interpolated widen is then guaranteed not to intersect ψ. However, if $(s'' \sqcup img) \sqcap \psi \neq \perp$, it could be that $s'' \sqcap \psi = \perp$ or $s'' \sqcap \psi \neq \perp$. In either case, the abstract state at the current node can be refined to img, since $img \sqcap \psi = \perp$ (as checked in line 11 of $\mathsf{RefineDAG}$). If s'' indeed intersects ψ, the counterexample passing through $\langle l', s'' \rangle$ will be detected and analyzed when analyzing a different path in the dag. After this refinement, all descendants of n' are deleted by the function $\mathsf{DeleteDescendents}(n')$.

Consider a counterexample $cx : (l_0, \ldots, l_m, l_e)$ in dag G_e, where l_e is the error location. Suppose $n_i = \langle l_i, s_i \rangle$ is the refinement node on the path corresponding to cx. Then the refined abstract state s'_i computed at n_i is such that $\mathsf{Image}(s'_i, (l_i, \ldots, l_m, l_e)) = \perp$. However, in the subsequent abstract computation, due to the application of over-approximate operations like ∇_T and $\sqcup$, the abstract state obtained at l_e starting from s'_i and proceeding along the path $l_{i+1}, \ldots, l_m, l_e$ may be non-$\perp$. Hence the counterexample cx may reappear in a subsequent abstract computation. While this is indeed undesirable, our refinement ensures local progress guarantees, as formalized in Lemma 3 below.

Consider a merge node n' that is a refinement node with respect to predecessor n but not with respect to predecessor n''. Suppose no ancestor of n is a refinement node while n'' has an ancestor p that is a refinement node. In this case, if we apply refinement at p before n', node n' will be deleted and no counterexample corresponding to a path through n and n' would have any of its nodes refined. To circumvent this, refinement nodes are refined in reverse topological order. For a spurious counterexample $cx : (l_0, \ldots, l_m, l_e)$ in G_e, which defines a unique sequence of nodes $n_0, \ldots, n_m, n_e$ in G_e, let us define the *culprit node* as the node $n_i = \langle l_i, s_i \rangle$ with the greatest depth such that $\mathsf{Image}(s_i, (l_i, \ldots, l_m, l_e)) = \perp$. The refinement of nodes in reverse topological order ensures the following *local progress guarantee.*

Lemma 3. *If $err(n_0) = \perp$ then* $\mathsf{RefineDAG}(n_e)$ *increases the depth of culprit nodes along each counterexample in G_e.*

Proof sketch. Since $err(n_e) = \{\top\}$ and $n_0 = \langle l_0, \top \rangle$, it follows that if $err(n_0) = \perp$, then there exists a refinement node along every path from n_0 to n_e. By definition, if n_i is a culprit node in counterexample cx, then every node $n_j = \langle l_j, s_j \rangle$ that preceeds n_i in cx must have $\mathsf{Image}(s_j, (l_j, \ldots, l_e)) = \perp$. Therefore processing refinement nodes in reverse topological order ensures that the depth of culprit node increases along each counterexample. $\Box$

If the refinement algorithm chooses nodes other than refinement nodes in reverse topological order for splitting or converting ∇_T to $\sqcup$ operation, then the depth of culprit node does not necessarily increase. In this case even local progress cannot be guaranteed. In our running example, $\mathsf{RefineDAG}$ refines the abstract state at node n_2 by replacing the widened result with the precise abstract image. The abstract state at the refined node n'_2 is $x = 1$. The subsequent abstract computation computes the node $n'_3 = \langle l_1, x \geq 1 \rangle$ as a successor of n'_2. The resulting dag has two nodes n_1 and n'_3 with location l_1

and having maximal abstract states. Hence the fixpoint for the while loop is $x = 0 \vee x \geq 1$. The abstract computation further adds the nodes $n_2'' = \langle l_2, x = 0 \rangle$ as a child of n_1 and $n_3'' = \langle l_3, x = 0 \rangle$ as a child of n_2''. This abstract dag now has a previously unexplored path corresponding to the counterexample (l_0, l_1, l_2, l_3). The subsequent call to RefineDAG identifies the true counterexample.

Theorem 1 formalizes the correctness of RefineDAG when it outputs "program incorrect" at line 2 (Fig. 1). Its proof follows from the assumption that the abstract operations Image and Preimage are exact and the fact that $n_0 = \langle l_0, \top \rangle$.

Theorem 1. *$err(n_0) \neq \{\bot\}$ iff there exists a path $(n_0, \ldots, n_e)$ in G_e such that $(l_0, \ldots l_e)$ represents a feasible execution of the program, where $n_i = \langle l_i, s_i \rangle$.*

Theorem 2. *If* AbstractDAG *outputs "program correct" (line 25) then none of the error locations is reachable.*

PROOF SKETCH. l_x is the last location in WTO order. Hence once l_x is reached in AbstractDAG, every reachable program location must have been processed at least once. The function NextInWTOOrder ensures that fixpoint is computed for every location. After RefineDAG terminates, abstract computation starts from the least location (in WTO order) from among the locations of nodes affected by refinement. NextInWTOOrder ensures that fixpoint is recomputed for every location affected by RefineDAG. The failure of check in line 22 of AbstractDAG (Fig. 1) ensures that none of the error locations are reachable in the fixpoint thus computed. □

Progress Guarantees: It would be desirable to prove that our dag-based abstraction refinement scheme has the following *progress* property: *Once a counterexample is eliminated it remains eliminated forever and every counterexample is eliminated in a bounded number of refinements.* There are two reasons why the procedure in Fig. 1 may not ensure this. First, it does not keep track of refinements (splitting of joined nodes) performed earlier. Since successive abstract computations can introduce previously unexplored program paths (as illustrated by our running example), previously split nodes may be joined in subsequent abstract computation. This is possible if an intermediate refinement step refines a predecessor of the split nodes, consequently deleting all of the split nodes. This can indeed cause a previously eliminated counterexample to reappear. Second, the interpolated widen operation is in general non monotonic, i.e., $s_1' \sqsubseteq s_1$ and $s_2' \sqsubseteq s_2$ does not necessarily imply $(s_1' \nabla s_2') \sqsubseteq (s_1 \nabla s_2)$. This may result in larger abstract states (in $\sqsubseteq$ order) in successive abstract computations. In addition, the above operations can potentially undo the increase in depth of culprit nodes effected by earlier invocations of RefineDAG. To remedy this and ensure global progress, we must keep track of all earlier refinements and use monotonic operations. We propose doing this by using a *Hint* dag H, which is a generalization of list based *hints* used in [8]. Like in dag G, every node in H has a location and abstract state associated with it. As G is explored and refined, we maintain the invariant that G is a subgraph of H. Thus, there is a correspondence map *corr* from nodes in G to nodes in H. For a node $n = \langle l, s \rangle$ in G such that $corr(n) = h = \langle l', s' \rangle$ in H, we have $l = l'$ and $s = s'$.

Initially, H has just one node $h_0 = \langle l_0, \top \rangle$. The node n_0 in G corresponds to h_0. Procedure AbstractDAG refers to H when adding new nodes and edges to G. If the abstract computation for a new node n is already represented in H (possibly because of an earlier exploration or refinement), AbstractDAG mimics that computation and maps the new node created in G to the corresponding node h in H. The abstract state computed at n in G is then intersected ($\sqcap$ operation) with the abstract state at h. The result is stored as the abstract state at n and also at h. If, however, the abstract computation for a new node n is not already represented in H, AbstractDAG proceeds as depicted in Fig. 1 and creates corresponding nodes and edges in H to record this abstract computation. When refinement is done at a node n in G, procedure RefineDAG records this refinement in H. This includes updating the structure of H to record conversion of joins to splits. In addition, every node h in H corresponding to a refined node n in G stores the refined abstract state of n. While nodes in G may be deleted by DeleteDescendents during refinement, the corresponding nodes in H are never deleted. Thus the dag H monotonically grows.

Theorem 3. *The depth of the culprit node along any spurious counterexample cx is strictly increasing in successive counterexample-dags containing cx.*

PROOF SKETCH. Lemma 3 ensures that the depth of culprit node along all counterexamples in the counterexample-dag increases due to refinement. Every time the depth of the culprit node along a counterexample increases due to refinement, it is recorded in the $Hint$ dag H. The depth of a culprit node along a counterexample in G is no less than its depth in H, because no nodes are deleted from the $Hint$ dag H and because abstract states in G are obtained by intersecting with the abstract states stored at the corresponding nodes in H. $\square$

Since all operations other than interpolated widen are monotone, a $Hint$ dag that stores abstract states only at loophead nodes also gives the same guarantee as in Theorem 3. It follows from Theorem 3 that *a spurious counterexample $cx : (l_0, \ldots, l_m, l_e)$ having k locations whose corresponding abstract states are computed by ∇_T or $\sqcup$ operations is eliminated in at most k invocations of* RefineDAG *on a counterexample-dag containing cx.*

3. Conclusion

We presented a dag-based refinement algorithm to automatically tune the precision of fixpoint computation dynamically and to reduce the number of false errors produced by abstract interpretation. We showed that our abstraction refinement is sound and it guarantees global progress in a formal sense.

References

[1] R. Bagnara, P. Hill, and E. Zaffanella. Widening operators for powerset domains. Technical Report 344, University of Parma, Italy, 2004.

[2] T. Ball and S. K. Rajamani. The SLAM project: debugging system software via static analysis. In *POPL 02*, pages 1–3, 2002.

[3] N. Bjørner, A. Browne, and Z. Manna. Automatic generation of invariants and assertions. In *Principles and Practice of Constraint Programming (CP)*, pages 589–623, 1995.

[4] F. Bourdoncle. Efficient chaotic iteration strategies with widenings, 1993.

[5] 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*, pages 238–252, 1977.

[6] W Craig. Three uses of the herbrand-gentzen theorem in relating model theory and proof theory. volume 22, pages 269–285, 1957.

[7] B. S. Gulavani, S. Chakraborty, A. V. Nori, and S. K. Rajamani. Automatically refining abstract interpretations. In *TACAS*, pages 443–458, 2008.

[8] B. S. Gulavani and S. K. Rajamani. Counterexample driven refinement for abstract interpretation. In *TACAS 06*, pages 474–488, 2006.

[9] T. Henzinger, R. Jhala, R. Majumdar, and K. McMillan. Abstractions from proofs. In *POPL*, 2004.

[10] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL*, pages 58–70, 2002.

[11] A. Rybalchenko and A. Podelski. Armc: The logical choice for software model checking with abstraction refinement. In *PADL*, 2007.