Localizing widening and narrowing

Gianluca Amato and Francesca Scozzari

Dipartimento di Economia
Università "G. d'Annunzio" di Chieti-Pescara
{gamato,fscozzari}@unich.it

Abstract. We show two strategies which may be easily applied to standard abstract interpretation-based static analyzers. They consist in 1) restricting the scope of widening, and 2) intertwining the computation of ascending and descending chains. Using these optimizations it is possible to improve the precision of the analysis, without any change to the abstract domains.

1 Introduction

In abstract interpretation-based static analysis, the program to analyze is typically translated into a set of equations describing the abstract program behavior, such as:

$$\begin{cases} x_1 = \Phi_1(x_1, \ldots, x_n) \\ \quad\vdots \\ x_n = \Phi_n(x_1, \ldots, x_n) \end{cases} \tag{1}$$

Each index $i \in \{1, \ldots, n\}$ represents a control point of the program and each Φ_i is a monotone operator. The variables in the equations range over an *abstract domain* A, which is a poset whose elements encode the properties we want to track. The analysis aims at computing the least solution of this system of equations.

In theory, it is possible to find the (exact) least solution of the system with a Kleene iteration, starting from the least element in A^n. However, in practice, many abstract domains have infinite ascending chains, therefore this procedure may not terminate. In other cases, domains may have very long finite ascending chains that would make this procedure impractical. The standard solution to these problems is to use *widening*, which ensures the termination of the analysis in exchange of a certain loss in precision.

Widening has been extensively studied, and we can find in the literature many different widenings for the most common abstract domains. Furthermore, many domain-independent techniques have been developed to improve widening precision, such as delayed widening, widening with threshold [9] and lookahead widening [17]. There are alternatives to the use of widening, such as acceleration operators [16] and strategy/policy iteration [11, 15]. However, acceleration only works for programs with restricted linear assignments, while strategy/policy iteration is restricted to template domains [26]. Therefore, widening is the only general applicable mechanism.

Using widening in the Kleene iteration, we still get an ascending chain which stabilizes on a post-fixpoint of Φ. It is common practice to improve the precision of the analysis continuing the iteration, taking this post-fixpoint as a starting point for a descending chain. Every element of the latter is an over approximation of the least fixpoint, therefore it is possible to stop the sequence at any moment without losing correctness. Sometimes a narrowing operator may be used, with the same purpose.

While widening and ascending chains have been extensively studied, little attention has been devoted to descending chains. One of the few papers, if not the only one, which tackles the subject is [20]. Nonetheless, descending chains (with or without narrowing) are often needed to get a decent precision.

In this paper we propose two strategies for improving the way standard abstract interpretation-based static analyzers are engineered. The first improvement regards widening and ascending chains. The second improvements regards descending chains, in particular the way ascending and descending chains may be intertwined.

1.1 Improving widening

Widening is defined by Cousot and Cousot [12] as a binary operation $\nabla : A \times A \to A$ over an abstract domain A, with the property that, given a sequence $x_0, \ldots, x_i, \ldots$ of abstract elements, the sequence $y_0 = x_0$, $y_{i+1} = y_i \nabla x_i$ is eventually constant.

It is possible to select a set of widening points $W \subseteq \{1, \ldots, n\}$ among all the control points of the program and replace, for each $i \in W$, the i-th equation in the system (1) with

$$x_i = x_i \nabla \Phi_i(x_1, \ldots, x_n) \ .$$

When W is admissible, i.e., every loop in the *dependency graph* of the system of equations contains at least one element in W, then any chaotic iteration sequence terminates. The choice of the set W of widening points may influence both termination and precision, thus should be chosen wisely.

Bourdoncle's algorithm [10] returns an admissible set of widening points. When the equations are generated by a control-flow graph, this set contains all the loop junction nodes. For structured programs, these widening points are exactly the loop heads of the program. This means that, if $i \in W$, the corresponding equation is

$$x_i = x_{in} \vee x_{back} \ ,$$

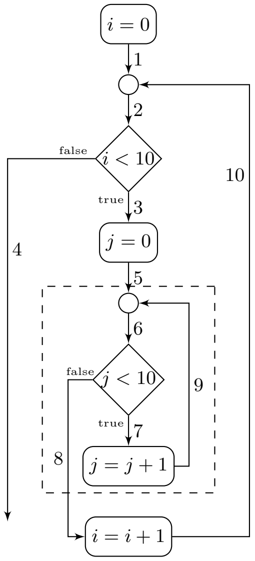
where the control points i, *in* and *back* are, respectively, the head of the loop, the input to the loop and the tail of the loop. The standard application of widening yields the equation

$$x_i = x_i \nabla (x_{in} \vee x_{back}) \ .$$

We believe that this is a source of imprecision, and show that, under certain conditions, it is possible (and generally better) to replace this equation with

$$x_i = x_{in} \vee (x_i \nabla x_{back}) \ . \qquad (2)$$

```
i = 0
while (i<10) {
  j = 0
  while (j<10)
    j = j+1
  i = i+1
}
// Invariant: i = 10
```



$$
\begin{aligned}
x_1 &= [0,0] \times [-\infty,\infty] \\
x_2 &= x_1 \vee x_{10} \\
x_3 &= x_2 \wedge ([-\infty,9] \times [-\infty,\infty]) \\
x_4 &= x_2 \wedge ([10,\infty] \times [-\infty,\infty]) \\
x_5 &= \mathit{first}(x_3) \times [0,0] \\
x_6 &= x_5 \vee x_9 \\
x_7 &= x_6 \wedge ([-\infty,\infty] \times [-\infty,9]) \\
x_8 &= x_6 \wedge ([-\infty,\infty] \times [10,\infty]) \\
x_9 &= x_7 + ([0,0] \times [1,1]) \\
x_{10} &= x_8 + ([1,1] \times [0,0])
\end{aligned}
$$

Fig. 1. The example program `nested`.

The last equation suggests that, when a junction node is entered from outside the loop, widening is replaced by least upper bound, and when a junction node is entered from inside the loop, widening is performed only on values generated inside the loop. We call *localized widening* the use of widening according to Eq. 2. Localized widening is mostly useful in the case of nested loops, where x_{in} does not change while analyzing the inner loop.

Consider the program in Figure 1 and the corresponding system of equations. Bourdouncle's algorithm outputs the set of widening points $W = \{2, 6\}$. Consider the trace of the analysis given in Figure 2, which is limited to the ascending chain and uses a recursive iteration strategy with the standard widening on the interval domain.

In the result, both x_2 and x_6 have infinite upper bounds for i. The problem is that, the second time we enter the inner loop, the new value of x_6 is computed as

$$
\begin{aligned}
x_6 \triangledown (x_5 \vee x_9) &= \{i = 0, j \geq 0\} \triangledown (\{i \in [0,9], j = 0\} \vee \{i = 0, j \in [1,10]\}) \\
&= \{i = 0, j \geq 0\} \triangledown \{i \in [0,9], j \in [0,10]\} \\
&= \{i \geq 0, j \geq 0\} \enspace .
\end{aligned}
$$

If we compute the descending sequence starting from here, we get $x_2 = \{i \geq 0\}$ and $x_6 = \{i \geq 0, j \in [0,10]\}$. Note that, while for j we got optimal bounds, the analysis cannot determine that $i = 10$ at the end of the loops. The problem is that the descending iteration cannot improve the upper bound for i for the variable x_5, since i is not used in the inner loop. This is a well known problem, and [20] gives a detailed presentation of the issue.

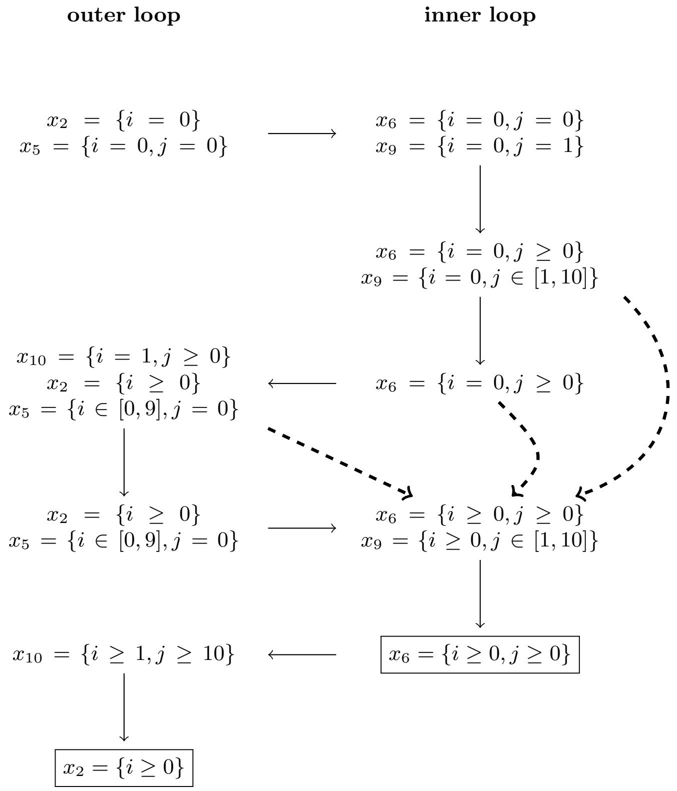

outer loop
inner loop
$x_2 = \{i = 0\}$
$x_5 = \{i = 0, j = 0\}$
$x_6 = \{i = 0, j = 0\}$
$x_9 = \{i = 0, j = 1\}$
$x_6 = \{i = 0, j \geq 0\}$
$x_9 = \{i = 0, j \in [1, 10]\}$
$x_{10} = \{i = 1, j \geq 0\}$
$x_2 = \{i \geq 0\}$
$x_5 = \{i \in [0, 9], j = 0\}$
$x_6 = \{i = 0, j \geq 0\}$
$x_2 = \{i \geq 0\}$
$x_5 = \{i \in [0, 9], j = 0\}$
$x_6 = \{i \geq 0, j \geq 0\}$
$x_9 = \{i \geq 0, j \in [1, 10]\}$
$x_{10} = \{i \geq 1, j \geq 10\}$
$\boxed{x_6 = \{i \geq 0, j \geq 0\}}$
$\boxed{x_2 = \{i \geq 0\}}$

Fig. 2. Trace of the analysis (only the ascending chain) for the program in Figure 1 using standard widening. Boxed values are final values. Solid arrows depict the order of execution, while dashed arrows show which input values are used to compute a new value.

If we use localized widening as described in Eq. 2, the second time we enter the inner loop the new value of x_6 is computed as:

$$\begin{aligned} x_5 \vee (x_6 \mathbin{\nabla} x_9) &= \{i \in [0,9], j = 0\} \vee (\{i = 0, j \geq 0\} \mathbin{\nabla} \{i = 0, j \geq 1\}) \\ &= \{i \in [0,9], j = 0\} \vee \{i = 0, j \geq 0\} \\ &= \{i \in [0,9], j \geq 0\} \ . \end{aligned}$$

At the end of the ascending chain we get $x_2 = \{i \geq 0\}$, $x_6 = \{i \in [0,9], j \geq 0\}$ and $x_{10} = \{i \in [1,10], j \geq 10\}$. This is enough to obtain, after the descending chain, the required result $x_2 = \{i \in [0,10]\}$ and $x_6 = \{i \in [0,9], j \in [0,10]\}$.

1.2 Improving descending sequences

The way ascending and descending sequences interact has never been fully clarified. The standard technique is to first compute an over approximation of the least solution of the equations with an ascending chain, and later refine the solution with a descending sequence. However, in the presence of nested loops, other choices are possible. In particular, when using a recursive strategy for the ascending sequence [10], it seems natural to intertwine ascending and descending sequences.

Consider again the program in Figure 1. Using either a recursive or an iterative strategy with the standard widening, the ascending chain determines the following invariants for the loops:

$$x_2 = \{i \geq 0\} \qquad x_6 = \{i \geq 0, j \geq 0\}$$

As shown in the previous section, precision may be partially recovered using a descending iteration, obtaining:

$$x_2 = \{i \geq 0\} \qquad x_6 = \{i \geq 0, j \in [0, 10]\}$$

However, it is possible to view the nodes inside the dashed rectangle in Figure 1 as if they were a single node, with input edge 5 and output edge 8. The abstract transformer for the new node is obtained performing a standard analysis of the inner loop, comprised of both ascending and descending chain. In this case, we have the following results:

$$\begin{array}{lll}
x_2^0 = \bot & x_2^1 = \{i = 0\} & x_2^2 = \{i \in [0, \infty]\} \\
x_5^0 = \bot & x_5^1 = \{i = 0, j = 0\} & x_5^2 = \{i \in [0, 9], j = 0\} \\
x_8^0 = \bot & x_8^1 = \{i = 0, j = 10\} & x_8^2 = \{i \in [0, 9], j = 10\}
\end{array}$$

where the values for x_8 are computed by considering the dashed rectangle as a whole. Every time it is considered, the analysis of the inner loop starts from the beginning, independently from the results of previous iterations. The last row is the fixpoint of the ascending chain (of the outer loop). Then, the descending chain (of the outer loop) begins:

$$\begin{array}{ll}
x_2^{\downarrow 0} = \{i \in [0, \infty]\} & x_2^{\downarrow 1} = \{i \in [0, 10]\} \\
x_5^{\downarrow 0} = \{i \in [0, 9], j = 0\} & x_5^{\downarrow 1} = \{i \in [0, 9], j = 0\} \\
x_8^{\downarrow 0} = \{i \in [0, 9], j = 10\} & x_8^{\downarrow 1} = \{i \in [0, 9], j = 10\}
\end{array}$$

In this case, we are able to prove that in the head of the outer loop $i \in [0, 10]$, since in the ascending chain we do not lose the information that $i \in [0, 9]$ holds in the inner loop.

If we look at the entire procedure without considering the abstraction given by the dashed rectangle, it happens that ascending and descending sequences are intertwined. While an ascending sequence is going on in the outer loop, either an ascending or descending sequence is going on in the inner loop. We call *localized narrowing* this strategy of intertwining ascending and descending chains. Here we use the term narrowing broadly, to mean not only the standard narrowing operator [13] but any procedure producing a descending chain.

1.3 Plan of the paper

Localized widening and narrowing may improve precision, but it is not completely clear whether they may be applied without compromising correctness and termination.

In Section 2, we show that localized widening is correct and terminates for any iteration strategy. In Section 3, we show that localized narrowing is correct and guarantees termination. Section 4 shows that localized widening and narrowing may improve precision not only w.r.t. standard abstract interpretation, but even when compared to other optimizations.

2 Localized widening

We now formalize the treatment of widening presented in Section 1.1. We show the conditions that allow to replace the standard widening with the localized one and prove the correctness of the resulting analysis.

In the following, we denote with $\varPhi$ a system of equations as in (1), where each variable x_i ranges over a poset A, and each $\varPhi_i : A^n \to A$ is a monotone function. With an abuse of notation, we denote with $\varPhi = (\varPhi_1, \dots, \varPhi_n)$ the function $\varPhi : A^n \to A^n$ obtained as the product of the $\varPhi_i$'s.

2.1 Preliminaries

We use the standard definition of widening, as appeared for the first time in [12].

Definition 1 (Widening [12]). *A* widening *for the poset A is a binary operator $\triangledown : A \times A \to A$ such that:*

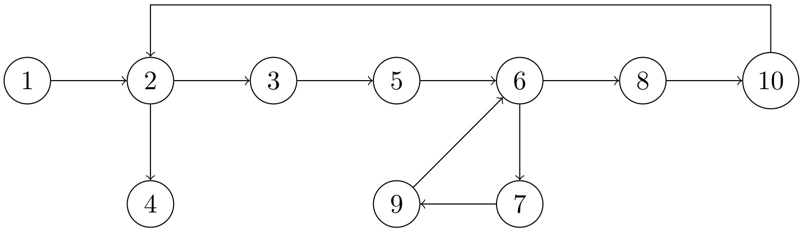
1. *$x \leq x \triangledown y$,*
2. *$y \leq x \triangledown y$,*
3. *for every sequence $(x_i)_{i \in \omega}$, the sequence $y_0 = x_0$, $y_{i+1} = y_i \triangledown x_i$ is eventually constant.*

In [14] a different definition of widening is introduced, where the convergence of the sequence (y_i) is ensured only if the sequence (x_i) is ascending. Note that, every widening $\tilde{\triangledown}$ satisfying [14] may be transformed in a widening $\triangledown$ satisfying [12] by defining

$$x \triangledown y = x \tilde{\triangledown} (x \vee y) \ . \tag{3}$$

Definition 2 (Dependency graph). *The* dependency graph *of the system of equations $\varPhi$ is a directed graph with nodes $\{1, \dots, n\}$ and an edge $i \to j$ iff x_i occurs in $\varPhi_j$.*

Example 1. The dependency graph for the system in Figure 1 is:



The nodes in the dependency graph correspond to the edges in the control-flow graph.

We recall from [10] the definitions of hierarchical ordering and weak topological ordering.

Definition 3 (Hierarchical ordering [10]). *A* hierarchical ordering *of a set S is a well-parenthesized permutation of this set without two consecutive "(".*

In other words, a hierarchical ordering is a string over the alphabet S augmented with left and right parenthesis. The elements between two matching parentheses are called a *component* and the first element of a component is called the *head*. The innermost component containing an element l is denoted by $\mathsf{comp}(l)$, and its head is denoted by $\mathsf{head}(l)$, when they exist. The set of heads of the components containing the element l is denoted by $\omega(l)$.

Example 2. For the dependency graph in Example 1, two hierarchical orderings are 1 2 3 4 5 6 7 8 9 10 and 1 (2 3 5 (6 7 9) 8 10) 4. In the second ordering, the heads are 2 and 6 and we have $\mathsf{head}(7) = 6$ and $\mathsf{head}(3) = 2$.

A hierarchical ordering induces a total ordering, that we denote by $\preceq$, corresponding to the permutation of the elements.

Definition 4 (Weak topological ordering [10]). *A* weak topological ordering *of a directed graph (w.t.o. for short) is a hierarchical ordering of its nodes such that for every edge $u \to v$, either $u \prec v$ or $v \preceq u$ and $v \in \omega(u)$.* [1]

Example 3. For the graph given in Example 1, a possible weak topological ordering is 1 (2 3 5 (6 7 9) 8 10) 4.

Every weak topological ordering of the dependency graph of Φ determines a set of admissible widening points (the set of all the heads) and two iteration strategies for solving the equations in Φ: an *iterative* and a *recursive* strategy.

In the recursive strategy, we apply the equations in the order given by the w.t.o., but every time we enter a new component, we loop within that component until all its values are stabilized. The iterative strategy is similar, but with the ordering obtained by removing all parentheses except the ones for the outermost component.

[1] In [10], the first condition was $u \prec v \land v \notin \omega(u)$. However, the second conjunct is implied by the first one.

2.2 Localizing widening

In the following, assume given a system Φ, its dependency graph and an associated weak topological ordering. An admissible set of widening points is implicitly defined as the set of all the heads in the weak topological ordering.

Definition 5 (Loop join node). *A* loop join node *is a node $l \in [1, n]$ in the dependency graph of Φ such that l is the head of a component and $\Phi_l(x_1, \ldots, x_n) = x_{v_1} \vee \cdots \vee x_{v_m}$ for some $\{v_1, \ldots, v_m\} \subseteq [1, n]$.*

Given a loop join node l, let $\{v^i_1, \ldots, v^i_r\}$ and $\{v^b_1, \ldots, v^b_s\}$ be the partition of $\{v_1, \ldots, v_m\}$ such that $v^i_j \notin \mathsf{comp}(l)$ and $v^b_j \in \mathsf{comp}(l)$. Elements of the two sets are called *input nodes* and *back nodes* respectively. We define $x^{in}_i = x_{v^i_1} \vee \cdots \vee x_{v^i_r}$ and $x^{back}_i = x_{v^b_1} \vee \cdots \vee x_{v^b_s}$.

Example 4. Nodes 2 and 6 in the system in Figure 1 are loop join nodes.

Intuitively, the above definition allows us to distinguish between join nodes generated by while loops and join nodes generated by if statements. In the first case, we separate the edges coming from inside the loop, denoted by x^{in}_i, and the edges coming from outside the loop, denoted by x^{back}_i. Note that the conditions on input and back nodes, i.e., $v^i_j \notin \mathsf{comp}(l)$ and $v^b_j \in \mathsf{comp}(l)$, are equivalent to $v^i_j \prec l$ and $l \preceq v^b_j$.

Definition 6. *We denote by $\Phi^\vee$ a new system of equations derived from Φ and such that, for each head node i, the i-th equation is replaced as follows:*

- *if i is a loop join node, by $x_i = x^{in}_i \vee (x_i \mathbin{\nabla} x^{back}_i)$;*
- *if i is not a loop join node, by $x_i = x_i \mathbin{\nabla} \Phi_i(x_1, \ldots, x_n)$.*

The idea is that any input coming from outside of a component does not need to be guarded by the widening. In fact, either the input does not belong to any loop (and therefore it has a constant value after the first iteration) or it belongs to a loop, and therefore it is already guarded by another outer widening which ensures that it will not increase forever. This reasoning works, however, only assuming that all the head nodes are widening nodes.

Example 5. If Φ is the system in Figure 1 whose heads are 2 and 6, we have that $\Phi^\vee$ is the same as Φ but for the following equations: $x_2 = x_1 \vee (x_2 \mathbin{\nabla} x_{10})$, $x_6 = x_5 \vee (x_6 \mathbin{\nabla} x_9)$.

We can now prove that localized widening guarantees termination, using any fair iteration sequence. First of all, we clarify what we mean with fair iteration sequence.

An *iteration sequence* starting from $D \in A^n$ is a possibly infinite sequence (X^j) with elements $X^j \in A^n$ such that:

- $X^0 = D$.
- X^j for $j > 0$ is obtained from X^{j-1} by applying one of the equations in $\Phi^\vee$.

In the following we denote with $\delta(j)$ the equation chosen to compute X^j.

Definition 7 (Enabled equation). *Given an iteration sequence (X^j), we say that equation i is* enabled *in step k when*

- *either i has never been chosen before, i.e., $\{l \in [1, k-1] \mid \delta(l) = i\} = \emptyset$;*
- *or let $m = \max\{l \in [1, k-1] \mid \delta(l) = i\}$ the last choice of i: there is $l \in [m, k-1]$ with $\delta(l) = u$, $u \to i$ is an edge in the dependency graph, $X^l > X^{l-1}$.*

An equation is enabled when its execution may produce a new value. An equation is not enabled when its execution cannot produce a new value, that is $X_i^j = \Phi_i^\vee(X^j)$. A fair iteration sequence is an iteration sequence where some enabled equations are eventually executed.

Definition 8 (Fair iteration sequence). *A* fair iteration sequence *starting from $D \in A^n$ is an iteration sequence (X^j) starting from D such that, for any step j, there exists $j' \geq j$ such that the equation $\delta(j')$ is enabled.*

The sequence terminates when it is not possible to choose any equation. It is immediate to see that both the iterative and recursive strategies compute fair iteration sequences. Moreover, any work-list based iteration sequence is fair.

Theorem 1. *Given a system of equations Φ and $D \in A^n$ a pre-fixpoint of Φ, any fair iteration sequence starting from D over $\Phi^\vee$ terminates on a post-fixpoint of Φ greater than D.*

There is one peculiarity when we use localized widening we should be aware of. While not specified in the definition, in the standard application of widening in the form $x_i = x_i \nabla \Phi_i(x_1, \ldots, x_n)$, it is always the case that $\Phi_i(x_1, \ldots, x_n) \geq x_i$. This does not hold anymore with localized widening. Some libraries of abstract domains, such as PPL [8] or APRON [23], implement widening under the assumption that the second argument is bigger then the first one. In this case, the same trick of Eq. 3 may be used: it is enough to replace $x \triangledown y$ with $x \triangledown (x \vee y)$.

3 Localized Narrowing

Looking from a different perspective, what localized widening does is to decouple the analysis of the inner components from the analysis of the outer components. Each component is analyzed almost as if it were a black box. We say "almost" because every time the black box is entered, we remember the last value of the variables and continue the analysis from that point: we are still computing a fixpoint using a chaotic iteration strategy.

However, we can push further the idea of the black box, as we have shown in Section 1.2. This allows to intertwine ascending and descending sequences in order to reach better precision and generally pursuing different strategies which do not follow in the umbrella of chaotic iteration. In this section, we are going to formalize and generalize the example given in Section 1.2.

3.1 More on w.t.o. and dependency graphs

First of all, we make an assumption to simplify notation: we consider only systems of equations with a join regular w.t.o., according to the following definition.

Definition 9 (Join regular w.t.o.). *A w.t.o. for the dependency graph of the system of equations Φ is* join regular *iff all the heads of the components are loop join nodes.*

The reason why we use join regular w.t.o. is that, as shown for localized widening, it is possible to separate the information coming from outer components from the information coming from inner components, giving better chance of optimizations. If in the head node i we have the equation $x_i = \Phi_i(x_a, x_b)$ and Φ_i is not a join, it is not clear whether it is possible to separate the contribution of x_a and x_b.

We could easily extend the algorithm to work on non join regular graphs, along the lines of Definition 6, which chooses the right widening to be applied. However, we think it is not a particularly heavy restriction, since systems of equations used in static analysis generally come out from flow graphs or labelled transition systems, and may be rewritten in such a way that the only equations with more than one variable in right-hand side are of the form $x_i = x_{v_1} \vee \cdots \vee x_{v_n}$, therefore allowing a join regular w.t.o.

The recursive algorithm we are going to present works on the components of the w.t.o. In order to iterate over components and nodes, it uses the concepts of segments and top-level elements.

Definition 10 (Segment). *A* segment *is a set $S \subseteq [1, n]$ such that there exists a well-parenthesized substring of the w.t.o. which contains exactly the elements in S.*

Example 6. Consider the w.t.o. 1 (2 3 5 (6 7 9) 8 10) 4 from Example 3. Some of the possible segments are $\{6, 7\}$ and $\{3, 5, 6, 7, 9, 8\}$, while $\{5, 6, 7\}$ and $\{3, 6, 7, 9\}$ are not segments, because the substring 5 (6 7 is not well-parenthesized and $\{3, 6, 7, 9\}$ does not come from a substring.

Intuitively, a segment corresponds to a piece of a program which is syntactically correct, where loops are not broken apart. It is immediate to see that every component C is a segment. Moreover, if C is a component with head h, then $C \setminus \{h\}$ is a segment. Finally, the entire $[1, n]$ is a segment.

Definition 11 (Top-level elements). *A* top-level element *of a segment S is an element $t \in S$ such that $\omega(t) \cap S \subseteq \{t\}$.*

Example 7. Consider the w.t.o. 1 (2 3 5 (6 7 9) 8 10) 4 from Example 3. The top-level elements of the segment $\{3, 5, 6, 7, 9, 8\}$ are 3, 5, 6 and 8.

Algorithm 1 Analysis based on localized narrowing

The algorithm requires a system of equations Φ with a join regular w.t.o. and a global map $x : [1, n] \to A$ to keep track of the current value of the variables.

```
Require: $S$ is a segment in the w.t.o. of $\Phi$
procedure Analyze(S)
    for all $j \leftarrow tl(S)$ do                                  ▷ extracted in w.t.o.
        if $j$ is head of a component then
            AnalyzeComponent(comp($j$))
        else
            $x_j \leftarrow \Phi_j(x_1, \ldots, x_n)$
        end if
    end for
end procedure

Require: $C$ is a component in the w.t.o. of $\Phi$
procedure AnalyzeComponent($C$)
    $i \leftarrow$ head of the component $C$
    $input \leftarrow \bigvee\{x_l \mid l \to i, l \notin C\}$                 ▷ Input from outer components
    ⟨ initialize $candidateInv \geq input$ ⟩
    repeat                                                         ▷ Start of ascending phase
        $x_i \leftarrow candidateInv$
        Analyze($C \setminus \{i\}$)
        $candidateInv \leftarrow x_i \triangledown \bigvee\{x_l \mid l \to i, l \in C\}$   ▷ Widening with back edges
    until $candidateInv \leq x_i$                                      ▷ End of ascending phase
    while ⟨ eventually false condition ⟩ do                           ▷ Start of descending phase
        $x_i \leftarrow \Phi_i(x_1, \ldots, x_n)$
        $y \leftarrow x$
        Analyze($C \setminus \{i\}$)
        $x \leftarrow x \wedge y$
    end while                                                      ▷ End of descending phase
end procedure
```

3.2 The algorithm

Algorithm 1 is the formalization and generalization of the procedure illustrated in Section 1.2. It depends on a system of equations Φ with a join regular w.t.o. and on a global map $x : [1, n] \to A$ which contains the initial value and keeps track of the current value of variables. There are two procedures mutually recursive. The procedure AnalyzeComponent has a parameter which is a component of the w.t.o., and calls Analyze to analyze the equations which are part of the component, with the exception of the head. The head is analyzed directly within AnalyzeComponent, using widening to ensure convergence. The procedure Analyze takes as input a segment of the w.t.o., and iterates over the top-level elements, either executing equations directly, or calling AnalyzeComponent for nested components. The entry point of the algorithm is the procedure Analyze. To analyze the entire system of equations, we call Analyze($[1, n]$) with x initialized to $\bot$.

The procedure AnalyzeComponent depends on a policy, which initializes *candidateInv*: the value for *candidateInv* may be chosen freely, subject to the condition $candidateInv \geq input$, where *input* is the join of all edges coming into the join node from the outer components. It starts with an ascending phase, where all the nodes on the component are dealt with, either directly, or with a recursive call for nodes which are part of a nested component. Then it follows a descending phase where the $\wedge$ operator is used to refine the result. The lines 21 and 23 are used to enforce that x is descending. Termination of the descending phase is ensured by the condition in line 19 which should be eventually false. A typical check is obtained by performing a given number of descending steps before giving up. A narrowing operating could be used instead to enforce termination.

3.3 Initialization policies

Let us consider some of the possible initialization policies. The simplest one is the Restart policy, given by

```
$candidateInv \leftarrow input$                    ▷ Restart policy      (4)
```

With this policy, every time Analyze is called on a component, all the results of the previous analyses are discharged and the analysis starts from the beginning. This is exactly the behavior we have shown in Section 1.2.

When the outer component is in the ascending phase, this is mostly a waste, since each time AnalyzeComponent is called with an input value which is bigger than the previous one. Hence, even the resulting invariant should be bigger than the one previously computed. We use "should" since non-monotonicity of widening makes this statement potentially false. Nonetheless, it is probably better for efficiency reasons not to start the analysis from the beginning. To this purpose, we can use the Continue policy, which joins the new input with the previous invariant.

```
$candidateInv \leftarrow x_i \vee input$           ▷ Continue policy     (5)
```

Were not for the intertwining of ascending and descending sequences, this would correspond to the use of localized widening. The Continue policy has a different drawback. When the outer component is in the descending phase, successive inputs generally form a descending chain. Starting from the last invariant may preclude a more precise result to be found. The Hybrid policy tries to balance efficiency and precision.

```
if $input = oldinput_i$ then                       ▷ Hybrid policy
    return
else if $input < oldinput_i$ then
    $candidateInv \leftarrow input$
else                                                                     (6)
    $candidateInv \leftarrow x_i \vee input$
end if
$oldinput_i \leftarrow input$
```

```
i = 0
while (TRUE) {
  i = i+1
  j = 0
  while (j<10) {
    // Inv: 0 ≤ i ≤ 10
    j = j+1
  }
  if (i>9) i = 0
}
```

```
i = 0
while (i<4)  {
  j = 0
  while (j<4)  {
    // Inv:  i ≤ j + 3
    i = i+1
    j = j+1
  }
  i = i-j+1
}
```

```
i = 0
while (TRUE) {
   // Inv:  i ≥ 0
   j = 0
   while (j<10) {
      j = j+1
   }
   i = i+11-j
}
```

Fig. 3. From left to right: programs `hybrid`, `hh` from [20] and `nested2`.

This policy needs a global map *oldinput* : $H \to A$, where H is the set of loop heads, to keep track of old input values.

The Hybrid policy behaves either as the Restart or Continue policy, according to the relation between the new input and the old one. The program `hybrid` in Figure 3 is an example where the Hybrid strategy is more precise then the Continue strategy. At the end of the ascending phase of the outer loop, the inner invariant is $1 \leq i, 0 \leq j < 10$. At the second iteration of the outer descending phase, the inner loop is called with input $1 \leq i \leq 10, j = 0$. However, this is joined with the previous invariant, and since i is not used in the inner loop, the improvement in precision is lost. With the Hybrid strategy, when the inner loop is called with input $1 \leq i \leq 10, j = 0$, since it is smaller then the previous input $1 \leq i, j = 0$, the analysis starts from the beginning, and the invariant of the inner loop is updated.

Since the combination of ascending and descending sequences is not something commonly considered, Algorithm 1 requires a correctness proof.

Theorem 2. *Algorithm 1 terminates and the global map x resulting from the call to* Analyze($[1, n]$) *is a post-fixpoint of the set of equations Φ.*

Note that, differently from the case of standard iteration strategies, we are not sure that the post-fixpoint resulting from Analyze($[1, n]$) is greater than the original value of x. If we want to find a solution bigger than a given $D \in A^n$, we may modify the algorithm accordingly, or insert the lower bound in the equations.

4 Related works

In the abstract interpretation literature, many efforts have been devoted to improve the precision of the analysis by modifying the standard procedure of an ascending chain with widening followed by a descending chain.

Some frameworks propose a complete departure from the model of iterative sequences, such as the acceleration operators [16] and the strategy/policy iteration [11, 15]. These are not compatible with localized widening and narrowing.

Other proposals refine the model of iterative sequences, and can be applied together with our optimizations. We recall the main ones, comparing them with our results.

Gopan and Reps' guided static analysis [18] is a technique were standard program analysis is applied to a sequence of program restrictions, which are essentially obtained by removing some edges from the control-flow graph of the program. Each restriction is analyzed starting from the result of the previous restrictions, until the original program is analyzed. Due to non-monotonicity of widening, this procedure may improve precision, especially in the case where loops contain different phases. In guided static analysis, the analyzer is treated as a black box, and therefore it may be immediately replaced with an analyzer implementing localized widening and narrowing. Henry at al. [21] enhance guide static analysis by combining it with path-focusing [24], in order to avoid merging infeasible paths. This optimization helps in finding precise disjunctive invariants, avoiding the use of disjunctive completion. The basic idea is to exploit an SMT-solver to find feasible paths, which are gradually discovered and analysed.

Guided static analysis and the strategies we propose try to fix complementary defects of standard iterative sequences. Guided static analysis focuses on improving analysis of loops whose behavior evolves along time, while localization improves results of nested loops.

Similar arguments hold for Monniaux and Le Guen's stratified static analysis by variable dependency [25]. The idea is similar to guided static analysis in that restrictions of the program are considered, but in this case the restriction is not on the edges of the control-flow graph, but on the variables. Successive approximations of the program are considered, where later approximations consider more variables than former ones. The result of an approximation is used within the successive approximations to restrict the results. If in a program node it turns out that $i \geq 0$, then the same should hold for all the successive approximations. This approach requires to modify the standard abstract interpretation procedure to use results from the previous restrictions, but the modifications may also be applied immediately to our localized strategies.

Halbwachs and Henry [20] propose a procedure to improve the result of static analysis which consists in successive static analysis phases. After each phase, the result of some special nodes are chosen, and another analysis is restarted from that point. As for guided static analysis, the standard analysis procedure is considered as a black box, and therefore can be combined with localization.

While localized narrowing seems to be more precise, we believe that combining localized widening with the optimizations in [20] is not worth, and we would obtain essentially the same results of localized widening, since both proposals essentially improve the result of nested loops. For an experimental comparison, see Section 4. We think, however, that localized widening is simpler to implement and may be easily integrated with other techniques.

Finally, localization may directly exploit any improvement to the design and implementation of widening operators (such as delayed widening, widening with threshold [9], lookahead widening [17], etc. . .), since we use standard definitions for widening, although applied in a different way.

4.1 Examples and Experiments

We have performed three different experiments to validate our techniques, and we plan to make more in the future. In these experiments we have used three tools: Interproc, Pagai and our prototype Jandom[2].

Jandom implements both localized widening and narrowing, and is the successor of our previous analyzer Random [7, 3]. Random implements many numerical abstract domains, included the recent parallelotopes [6] and template parallelotopes [4, 2, 5, 1].

Interproc [22] performs inter-procedural analysis of a simple imperative language. It support standard abstract interpretation analysis, policy iteration for intervals and guided static analysis. Pagai [21] is a path-sensitive static analyzer. It implements several different techniques, such as lookahead widening, guided static analysis, path focusing, and the optimizations to narrowing in [20].

As a first experiment, we tried to understand whether localized widening or narrowing was in use in other analyzers (apart from Random and our prototype). Therefore, we tried the program `nested` in Figure 1, and the programs in Figure 3 in Interproc and Pagai, using standard abstract interpretation over the domain of closed polyhedra. In Interproc we used delayed widening with 4 delays and a two step descending sequence. None of the two analyzers, with this standard settings, were able to prove the optimal invariant. After this experiment, and given the current literature, we are confident that localized narrowing and widening have never been implemented before.

Later, we used Pagai on the same programs, but selecting different known optimization techniques, and comparing the results with that of localized widening and narrowing. The aim was not to provide a full evaluation, but to give an idea of the kind of programs that may benefit from localization or other techniques. Table 1 reports the result of the comparison, using the domains of strict convex polyhedra. The results show that localized narrowing with the hybrid policy proves the required invariant for all the programs, but this is hardly surprising since programs were chosen ad-hoc. Lookahead widening and guided static analysis do not work very well with this examples, but this was expected since the aim of these optimizations is different than ours. The optimized narrowing in [20] behaves better, since it was developed to improve precision on nested loops too.

As a rough evaluation of the overhead of our strategies, we count the number of times that the widening and narrowing operators are executed. It happens that, for all the programs in Table 1, using the localized widening only, we execute 8 widenings and 4 narrowings, using the localized narrowing with the continue policy we execute 8 widenings and 8 narrowings, and using the localized narrowing with the hybrid policy, we execute 11 widenings and 8 narrowings.

[2] `https://github.com/amato-gianluca/Jandom`

program	loc_widening	continue	hybrid	guided	lookahead	narrowing
`nested`	yes	yes	yes	no	no	yes
`nested2`	no	yes	yes	no	yes	no
`hybrid`	no	no	yes	no	no	yes
`hh`	yes	no	yes	no	no	yes

Table 1. Results of the comparison (loc_widening=localized widening, continue=localized narrowing with continue policy, hybrid=localized narrowing with hybrid policy, guided=guided static analysis, lookahead=lookahead widening, narrowing=optimized narrowing in [20]).

Finally, we implemented localized widening in Pagai. As a testament of the simplicity of the idea, the core of the implementation, which is everything but user-interface, only required to modify one line of code. This was possible since Pagai puts a widening on each head node, as required in Theorem 1. Using Pagai we executed the benchmarks of the Mälardalen WCET research group [19], which contains programs such as sorts, matrix transformations, fft, simple loops, etc. . . We compare the result of standard abstract interpretation with and without localized widening.

We analyzed a total of 114 functions. In 29 of these we improved the results of the analysis. In particular, there are a total of 379 head nodes, and for 164 of them we improved the result. In no case we got worse results than standard widening. This improvement was obtained by reducing at the same time the number of iterations. With the standard widening the analysis took a total of 19522 ascending steps and 23415 descending steps, while with localized widening we had 19363 ascending steps and 22528 descending steps. Nonetheless, the analysis with localized widening took 5.25 seconds, against 4.49 seconds of the classic analysis. We argue that it took more time despite a reduction in the number of steps since the join operator is more costly than widening.

We also compared the results of localized widening and the refined narrowing in [20]. For the 379 heads nodes, we got more precise results in 91 cases, worse results in 2 cases, while we had incomparable results in other 2 cases.

Overall, the results of localized widening are excellent, because it can improve precision even considerably without incurring in performance penalties.

More evaluations should be performed on localized narrowing. Potentially it can improve precision much more than localized widening alone. However, the performance penalty it may incur is bigger. Also, its implementation is more difficult, and that is why we have not performed a similar comparison in Pagai as for localized widening.

5 Conclusions

We have shown two strategies for improving precision of abstract interpretation. Localized widening is simple, effective and has negligible computational cost.

Therefore, can be easily implemented in already existent abstract analyzers. Localized narrowing is more complex, potentially slower but generally more precise than localized widening. More experiments should be conducted to check the power and applicability of the latter.

References

1. Gianluca Amato, James Lipton, and Robert McGrail. On the algebraic structure of declarative programming languages. *Theoretical Computer Science*, 410(46):4626–4671, 2009.
2. Gianluca Amato, Maurizio Parton, and Francesca Scozzari. Deriving numerical abstract domains via principal component analysis. In Radhia Cousot and Matthieu Martel, editors, *17th International Symposium, SAS 2010, Perpignan, France, September 14-16, 2010, Proceedings*, volume 6337 of *Lecture Notes in Computer Science*, pages 134–150. Springer, Berlin Heidelberg, 2010.
3. Gianluca Amato, Maurizio Parton, and Francesca Scozzari. A tool which mines partial execution traces to improve static analysis. In H Barringer and *et al.*, editors, *First International Conference, RV 2010, St. Julians, Malta, November 1-4, 2010. Proceedings*, volume 6418 of *Lecture Notes in Computer Science*, pages 475–479. Springer, Berlin Heidelberg, 2010.
4. Gianluca Amato, Maurizio Parton, and Francesca Scozzari. Discovering invariants via simple component analysis. *Journal of Symbolic Computation*, 47(12), 2012.
5. Gianluca Amato and Francesca Scozzari. Observational completeness on abstract interpretation. *Fundamenta Informaticae*, 106(2–4):149–173, 2011.
6. Gianluca Amato and Francesca Scozzari. The abstract domain of parallelotopes. In Jan Midtgaard and Matthew Might, editors, *The Fourth International Workshop on Numerical and Symbolic Abstract Domains (NSAD 2012)*, volume 287 of *Electronic Notes in Theoretical Computer Science*, pages 17–28. Elsevier, November 2012.
7. Gianluca Amato and Francesca Scozzari. Random: R-based Analyzer for Numerical Domains. In Nikolaj Bjrner and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning 18th International Conference, LPAR-18, Mérida, Venezuela, March 11-15, 2012. Proceedings*, volume 7180 of *Lecture Notes in Computer Science*, pages 375–382. Springer, 2012.
8. Roberto Bagnara, Patricia M. Hill, and Enea Zaffanella. The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Science of Computer Programming*, 72(1–2):3–21, 2008.
9. Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jèrôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. A static analyzer for large safety-critical software. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI'03)*, pages 196–207, San Diego, California, USA, June 7–14 2003. ACM Press.
10. François Bourdoncle. Efficient chaotic iteration strategies with widenings. In Dines Bjørner, Manfred Broy, and Igor V. Pottosin, editors, *Formal Methods in Programming and Their Applications, International Conference Academgorodok, Novosibirsk, Russia June 28 July 2, 1993 Proceedings*, volume 735 of *Lecture Notes in Computer Science*, pages 128–141. Springer, Berlin Heidelberg, 1993.

11. A. Costan, Stephane Gaubert, Eric Goubault, Matthieu Martel, and Sylvie Putot. A policy iteration algorithm for computing fixed points in static analysis of programs. In Kousha Etessami and Sriram K. Rajamani, editors, *Computer Aided Verification, 17th International Conference, CAV 2005, Edinburgh, Scotland, UK, July 6-10, 2005. Proceedings*, volume 3576 of *Lecture Notes in Computer Science*, pages 462–475, Berlin Heidelberg, 2005. Springer.
12. Patrick Cousot and Radhia Cousot. Static determination of dynamic properties of programs. In *Proceedings of the Second International Symposium on Programming*, pages 106–130, Paris, France, 1976. Dunod.
13. Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM Press, New York, NY, USA, January 1977.
14. Patrick Cousot and Radhia Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In Maurice Bruynooghe and Martin Wirsing, editors, *Programming Language Implementation and Logic Programming, 4th International Symposium, PLILP '92 Leuven, Belgium, August 26–28, 1992, Proceedings*, volume 631 of *Lecture Notes in Computer Science*, pages 269–295. Springer, Berlin Heidelberg, 1992. Invited paper.
15. Thomas Martin Gawlitza and Helmut Seidl. Solving systems of rational equations through strategy iteration. *ACM Transactions on Programming Languages and Systems*, 33(3):1–48, April 2011.
16. Laure Gonnord and Nicolas Halbwachs. Combining widening and acceleration in linear relation analysis. In Kwangkeun Yi, editor, *Static Analysis, 13th International Symposium, SAS 2006, Seoul, Korea, August 29-31, 2006. Proceedings*, volume 4134 of *Lecture Notes in Computer Science*, pages 144–160, Berlin Heidelberg, 2006. Springer.
17. Denis Gopan and Thomas Reps. Lookahead widening. In Thomas Ball and Robert B. Jones, editors, *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006. Proceedings*, volume 4144 of *Lecture Notes in Computer Science*, pages 452–466, Berlin Heidelberg, 2006. Springer.
18. Denis Gopan and Thomas Reps. Guided static analysis. In Hanne Riis Nielson and Gilberto Filé, editors, *Static Analysis, 14th International Symposium, SAS 2007, Kongens Lyngby, Denmark, August 22-24, 2007.*, volume 4634 of *Lecture Notes in Computer Science*, pages 349–365. Springer, Berlin Heidelberg, 2007.
19. Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. The Mälardalen WCET benchmarks – past, present and future. In Björn Lisper, editor, *Proc. 10th International Workshop on Worst-Case Execution Time Analysis (WCET2010)*, pages 137–147, Brussels, Belgium, July 2010. OCG.
20. Nicolas Halbwachs and Julien Henry. When the decreasing sequence fails. In Antoine Miné and David Schmidt, editors, *Static Analysis, 19th International Symposium, SAS 2012, Deauville, France, September 11-13, 2012. Proceedings*, volume 7460 of *Lecture Notes in Computer Science*, pages 198–213, Berlin Heidelberg, 2012. Springer.
21. Julien Henry, David Monniaux, and Matthieu Moy. PAGAI: A path sensitive static analyser. *Electronic Notes in Theoretical Computer Science*, 289:15–25, 2012.
22. Bertrand Jeannet. *Interproc Analyzer for Recursive Programs with Numerical Variables*. INRIA, 2004. Software and documentation are available at the fol-

lowing URL: `http://pop-art.inrialpes.fr/interproc/interprocweb.cgi`. Last accessed: 2013-04-03.

23. Bertrand Jeannet and Antoine Miné. APRON: A library of numerical abstract domains for static analysis. In Ahmed Bouajjani and Oded Maler, editors, *Computer Aided Verification, 21st International Conference, CAV 2009, Grenoble, France, June 26 – July 2, 2009. Proceedings*, volume 5643 of *Lecture Notes in Computer Science*, pages 661–667. Springer, Berlin Heidelberg, 2009.
24. David Monniaux and Laure Gonnord. Using bounded model checking to focus fixpoint iterations. In Eran Yahav, editor, *Static Analysi, 18th International Symposium, SAS 2011, Venice, Italy, September 14-16, 2011. Proceedings*, volume 6887 of *Lecture Notes in Computer Science*, pages 369–385. Springer, Berlin Heidelberg, 2011.
25. David Monniaux and Julien Le Guen. Stratified static analysis based on variable dependencies. In Damien Massé and Laurent Mauborgne, editors, *Proceedings of the Third International Workshop on Numerical and Symbolic Abstract Domains, NSAD 2011*, volume 288 of *Electronic Notes in Theoretical Computer Science*, pages 61–74. Elsevier, December 2012.
26. Sriram Sankaranarayanan, Henny B. Sipma, and Zohar Manna. Scalable analysis of linear systems using mathematical programming. In Radhia Cousot, editor, *Verification, Model Checking, and Abstract Interpretation, 6th International Conference, VMCAI 2005, Paris, France, January 17-19, 2005. Proceedings*, volume 3385 of *Lecture Notes in Computer Science*, pages 25–41. Springer, Berlin Heidelberg, January 2005.