Improving the results of program analysis by abstract interpretation beyond the decreasing sequence[⋆]

Rémy Boutonnet, Nicolas Halbwachs

Univ. Grenoble Alpes, CNRS, Grenoble INP, VERIMAG, 38000 Grenoble, France

Abstract. The classical method for program analysis by abstract interpretation consists in computing first an increasing sequence using an extrapolation operation, called widening, to correctly approximate the limit of the sequence. Then, this approximation is improved by computing a decreasing sequence without widening, the terms of which are all correct, more and more precise approximations. It is generally admitted that, when the decreasing sequence reaches a fixpoint, it cannot be improved further. As a consequence, most efforts for improving the precision of an analysis have been devoted to improving the limit of the increasing sequence. In a previous paper, we proposed a method to improve a fixpoint after its computation. This method consists in computing from the obtained solution a new starting value from which increasing and decreasing sequences are computed again. The new starting value is obtained by projecting the solution onto well-chosen components. The present paper extends and improves the previous paper: the method is discussed in view of some example programs for which it fails. A new method is proposed to choose the restarting value: the restarting value is no longer a simple projection, but is built by gathering and combining information backward the widening nodes in the basic solution. Experiments show that the new method properly solves all our examples, and improves significantly the results obtained on a classical benchmark.

1 Introduction

Program analysis by abstract interpretation [CC77] consists in computing an upper approximation of the reachable states of the program, as an upper approximation of the least fixpoint of an abstract semantic function in a suitable abstract lattice of properties. When the abstract lattice is of infinite depth, the standard approach [CC76,CC77] consists in computing an *increasing sequence* whose convergence is forced using a *widening operator*; then, from the obtained limit of the increasing sequence, one can improve the solution by computing a *decreasing sequence*, by iterating the function without widening. The decreasing sequence may either stop at a fixpoint of the semantic function, or be infinite, but since all its terms are correct solutions, one can stop the computation after a fixed number of terms, or limit its length using a *narrowing operator*.

The precision of the result depends both on the ability of the widening operator to "guess" a precise limit of the increasing sequence, and on the information gathered during the decreasing sequence. Intuitively, the increasing sequence extrapolates the behaviour of the program from the first steps of its execution, while the decreasing sequence gathers information about the end of the execution of the program, its loops, or more generally, the way the strongly connected components of its control structure are exited.

While significant effort has been devoted to improving the precision of the limit of the increasing sequence (see Section 3 for a quick survey), little attention has been paid to the decreasing sequence. It is generally admitted that, when the decreasing sequence reaches a fixpoint, it cannot be improved. However, it appears that such a fixpoint can be far from the least fixpoint, so improving it may have a significant influence.

More specifically, we shall see in Section 2 some examples of numerical programs where the decreasing sequence provides poor improvements, for the following reason: the body of a loop (or a strongly connected

[⋆] This work has been partially supported by the European Research Council under the European Union's Seventh Framework Programme(FP/2007-2013) / ERC Grant Agreement nr. 306595 "STATOR".

component of the control structure) is made of two sets of paths, one of them ensuring the boundedness of some quantity (e.g., a variable), while the other does not modify that quantity. As a consequence, if the considered quantity is widened to infinity, the fact that it is not modified by some paths prevents the decreasing sequence from finding a better bound.

Contribution and structure of the paper. In this paper, we propose a method starting from the result of the decreasing sequence, and trying to improve it as follows: From the obtained solution, a new value is computed which will be used as the starting point of new increasing and decreasing sequences, providing a new solution which can be intersected with the previous one. The essential problem concerns the choice of the new starting value, that we call a "seed". The present paper recalls and extends a conference paper [HH12], from which the first sections are essentially borrowed: after presenting our examples (Section 2), and reviewing the main related work (Section 3), we introduce in Section 4 the necessary definitions and notation. Section 5 presents the general restarting method. In Section 6, we recall and discuss the method proposed in [HH12] to select the restarting "seed". In view of this discussion, we propose an improvement of the seed construction in Section 7, which is shown to properly deal with all our examples. Both methods have been implemented on top of the static analyzer Pagai [HMM12], so we can perform deeper experiments (Section 9) and compare the methods with each other and with the basic analysis.

2 Motivating examples

We first illustrate the problem with some very simple examples.

Example 1 is a classical example of what can be obtained by interval analysis [CC76]. Fig. 1.a shows the control-flow graph[1] (CFG) of a very simple loop incrementing a variable i from 0 to 100, together with a table detailing its analysis[2]. The increasing sequence consists of 2 iterations; at iteration 2, the widening is applied, and the sequence converges. Iteration 3 shows the descending sequence, which reaches a fixpoint in 1 step. The results are the best possible: $i \in [100, 100]$ at the end of the loop.

Now, the CFG shown in Fig.1.b is obtained from the preceding one by nesting a second loop on another variable j within the first one. Computed values are pairs of intervals, one for i and one for j. The increasing sequence converges after 4 steps. Again, the descending sequence reaches a fixpoint in 1 step, but now, the result for i is imprecise: $i \in [100, \infty]$ at the end. The reason is that the nested loop neither modifies nor tests the variable i; so, as soon as its interval has been widened to $[0, \infty]$, it will remain unchanged in the inside loop. Notice that the same phenomenon happens if we select both loop heads as widening nodes (see Note 1 in Section 4 on the selection of widening nodes).

Example 2: Our second example illustrates a situation which occurs commonly in reactive programs, cyclically sampling their environment. A first program (Fig. 2.a) is just an infinite loop with a counter modulo 60. A precise fixpoint is reached after two steps of increasing sequence and one descending step.

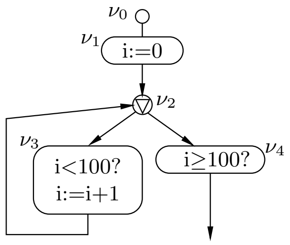
Now, assume that we do not want to count all loop iterations, but only when some external event (e.g., a "second") is detected[3] . This is done by the program of Fig. 2.b. As before, the increasing sequence converges after 2 steps, but now, the limit is a fixpoint, so there is no decreasing sequence, and the upper bound of the counter is missed.

This second example can be simply explained as follows: let $(L, \sqsubseteq, \sqcup, \sqcap, \top, \bot)$ be the abstract lattice, and F be an abstract semantic function from L to L. Let $G = Id \sqcup F$ (i.e., $\lambda X.X \sqcup F(X)$). Then, it is easy to see that G has the same least fixpoint as F, but G is *extensive* (i.e., $\forall X \in L, X \sqsubseteq G(X)$). As a consequence, any postfixpoint of F is a fixpoint of G. So, while the limit of the increasing sequence with

[1] We use a form of CFG that should be self-explanatory. It is made precise in §4.2.

[2] The table shows, at each iteration of the analysis and for each node of the graph, the value (i.e., the interval for i) attached to the node; since this value is computed only when the value of a preceding node changes, cells are left empty when their value is not computed.

[3] These two programs could have been generated by compiling the Esterel programs [BS91] "every 60 tick do emit minute" and "every 60 second do emit minute".



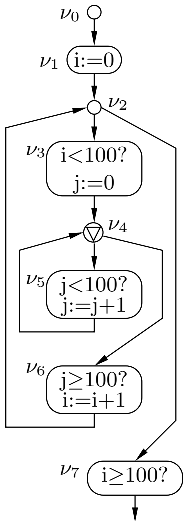
Control Flow Graph

iterations

nodes	1	2	3 ↓
ν_0	$[-\infty, \infty]$		
ν_1	$[0, 0]$		
ν_2	$[0, 0]$	$[0, \infty]$	$[0, 100]$
ν_3	$[1, 1]$	$[1, 100]$	$[1, 100]$
ν_4	$\perp$	$[100, \infty]$	$[100, 100]$
		convergence	fixpoint

Analysis

(a) A classical example where the decreasing sequence reaches the least fixpoint

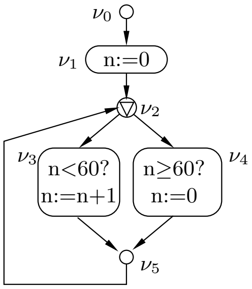


iterations

nodes	1	2	3	4	5 ↓
ν_0	$[-\infty, \infty]$ $[-\infty, \infty]$				
ν_1	$[0, 0]$ $[-\infty, \infty]$				
ν_2	$[0, 0]$ $[-\infty, \infty]$		$[0, 1]$ $[-\infty, \infty]$	$[0, \infty]$ $[-\infty, \infty]$	
ν_3	$[0, 0]$ $[0, 0]$		$[0, 1]$ $[0, 0]$	$[0, 99]$ $[0, 0]$	
ν_4	$[0, 0]$ $[0, 0]$	$[0, 0]$ $[0, \infty]$	$[0, \infty]$ $[0, \infty]$	$[0, \infty]$ $[0, \infty]$	$[0, \infty]$ $[0, 100]$
ν_5	$[0, 0]$ $[1, 1]$	$[0, 0]$ $[1, 100]$	$[0, \infty]$ $[1, 100]$		$[0, \infty]$ $[1, 100]$
ν_6	$\perp$	$[1, 1]$ $[100, \infty]$	$[1, \infty]$ $[100, \infty]$		$[1, \infty]$ $[100, 100]$
ν_7	$\perp$	$\perp$		$[100, \infty]$ $[100, \infty]$	$[100, \infty]$ $[100, 100]$
				convergence	fixpoint

(b) A nested loop prevents the decreasing sequence from getting precise results

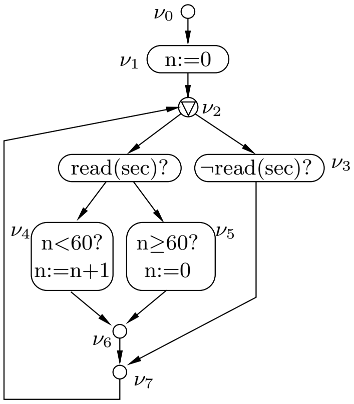
Fig. 1. Example 1 – Nested loops



	1	2	3 ↓
ν_0	$[-\infty,\infty]$		
ν_1	$[0,0]$		
ν_2	$[0,0]$	$[0,\infty]$	$[0,60]$
ν_3	$[1,1]$	$[1,60]$	$[1,60]$
ν_4	$\bot$	$[0,0]$	$[0,0]$
ν_5	$[1,1]$	$[0,60]$	$[0,60]$

convergence fixpoint

(a) An infinite loop with a counter modulo 60

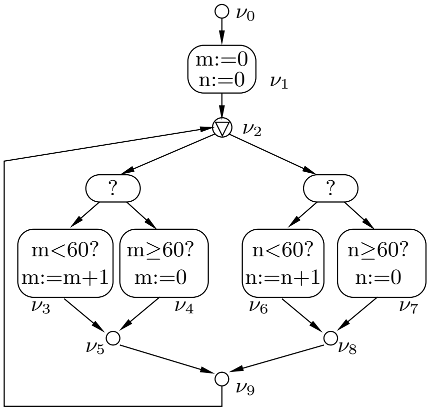


	1	2
ν_0	$[-\infty,\infty]$	
ν_1	$[0,0]$	
ν_2	$[0,0]$	$[0,\infty]$
ν_3	$[0,0]$	$[0,\infty]$
ν_4	$[1,1]$	$[1,60]$
ν_5	$\bot$	$[0,0]$
ν_6	$[1,1]$	$[0,60]$
ν_7	$[0,1]$	$[0,\infty]$

fixpoint

(b) A loop counting the occurrences of "seconds"

Fig. 2. Example 2 – Intermittent counting



	1	2
ν_0	$[-\infty,\infty]$ $[-\infty,\infty]$	
ν_1	$[0,0]$ $[0,0]$	
ν_2	$[0,0]$ $[0,0]$	$[0,\infty]$ $[0,\infty]$
ν_3	$[1,1]$ $[0,0]$	$[1,60]$ $[0,\infty]$
ν_4	$\bot$	$[0,0]$ $[0,\infty]$
ν_5	$[1,1]$ $[0,0]$	$[0,60]$ $[0,\infty]$
ν_6	$[0,0]$ $[1,1]$	$[0,\infty]$ $[1,60]$
ν_7	$\bot$	$[0,\infty]$ $[0,0]$
ν_8	$[0,0]$ $[1,1]$	$[0,\infty]$ $[0,60]$
ν_9	$[0,1]$ $[0,1]$	$[0,\infty]$ $[0,\infty]$

fixpoint

Fig. 3. Example 3 – Double intermittent counting

F may be a strict postfixpoint of F — which can be improved by a decreasing sequence —, this limit will be a fixpoint of G, meaning that there is no decreasing sequence with G. This is exactly what happens with our example, where the dummy branch in the loop of Fig. 2.b adds an identity term to the semantic function at the widening node.

Example 3: Fig. 3 shows a variation of the previous example, where two bounded counters are executed conditionally. In this case, each counter computation plays the role of the dummy branch in Example 2, since it does not affect the other counter. As a consequence, the decreasing sequence cannot improve the result of the increasing sequence and misses the upper bound of both counters.

3 Related works

Beside research proposing new abstract domains, many existing works aim at fighting the imprecision of analysis, considered to be essentially due to the widening operation. Apart from proposals of systematic design of widening and narrowing operators [CZ11], one can distinguish at least three main tracks: (1) designing improved widening operators, generally dedicated to some specific domains; (2) avoiding or minimising the use of widening, by focusing either on some classes of programs or on some classes of abstract domains; (3) applying widening and narrowing using specific strategies.

Improving widening operators. Many proposals concern the improvement of widening operators[4], especially for the polyhedra domain [CH78,BHRZ03]. Limited widening — widening *up to* or *with thresholds* [Hal93,HPR97,BCC$^+$03] — consists in choosing — generally from the conditions appearing in the program, or some propagation of them [LCJG11] — some tentative constraints used as limits to the widening. Widening *with landmarks* [SK06] follows the same idea, but the selection of limits is made dynamically. Widening *with care set* [WYGI07] and *interpolated widening* [GCNR08] make use of a proof objective. Some of these proposals can properly deal with some of our examples, mainly because they can reach a precise solution at the end of the increasing sequence. Our approach is compatible with any widening operation.

Avoiding widening. Other authors try to avoid the use of widening. *Acceleration techniques* [BGP97,WB98,CJ98,BFLP03,GH06,GS14] are dedicated to some classes of programs or loops, the effect of which can be exactly computed. Other approaches can be applied only with some kinds of domains — namely "weakly relational domains" [Min04] or "templates" [SSM04,SSM05] — in which *policy iteration* [SW04,CGG$^+$05,GS07,KMW16] allows least fixpoints to be precisely computed. These methods generally solve our problem, but they are restricted either to some class of programs or to some abstract domains. Notice that, for simplicity, all our examples are analyzable with the interval abstract domain, so they are properly dealt with by policy iteration. However, a simple variable change would produce versions of these programs that would need strongly relational domains like polyhedra. Such an example will be considered in §8.4.

Widening strategies. An obvious way of improving the precision of the widening is to *delay* its application [Hal93,BCC$^+$03], i.e., applying it only after a fixed number of exact steps or intermittently, or applying it after some *loop unrolling* [Gou01,PGM03]. Some strategies adapt the application of the widening according to the discovery of *new feasible paths* [Hal93,HPR97,BCC$^+$03] in the program. In particular, [GR06a,GR07] propose a very clever strategy, called *guided static analysis*, where a succession of increasing-decreasing sequences are computed for more and more feasible paths of the program. [ASV13] and [ASS$^+$16] generalize this approach, applying widening and narrowing in an interleaved manner, in the context of infinite equation systems and non monotone abstract functions. [AS13] also proposes specific strategies for alternating increasing and decreasing phases. [GR06b] refines the widening strategy using counter-examples with respect to a proof objective. *Stratified analysis* [ML12] is a succession of analyses

[4] Defining the quality of a widening operator is difficult: since a widening is not monotone, by essence, a locally more precise widening is not guaranteed to lead to a more precise limit at the end of the sequence. This is why most proposed improvements are justified experimentally.

concerning more and more variables, according to their dependencies. While some of these methods can work on some of our examples, none of them specifically addresses the problem of improving the result of the decreasing sequence. Our approach is different, but compatible with existing methods, as it can be used in combination with them.

Our approach can be situated in the framework of [Cou15], which proposes, in particular, to improve an imprecise solution by computing again an increasing iteration *interpolating* the existing solution using *dual-narrowing* ([Cou15], §9). Our proposal differs in that we still apply an extrapolation sequence with widening, but we put more emphasis on the selection of the starting point of the new increasing sequence.

4 Definitions and notation

4.1 Abstract lattice

As said before, we assume that the analysis makes use of an abstract complete lattice $(L, \sqsubseteq, \sqcup, \sqcap, \top, \bot)$. We assume this lattice to be of infinite depth. The lattice operations are supposed to be available, together with a widening operator ∇, and the interpretation of each program statement s as a function (predicate transformer) $f_s : L \mapsto L$.

4.2 Control-flow graph

A control-flow graph (CFG) is a graph $(\mathcal{N}, \mathcal{E})$, where

- the finite set $\mathcal{N} = \{\nu_1, ..., \nu_k\}$ is made of 3 types of nodes: start nodes, junction nodes, and statement nodes. With each statement node ν_i is associated a function $f_i : L \mapsto L$.
- $\mathcal{E} \subseteq \mathcal{N} \times \mathcal{N}$ is the set of edges. Start nodes have no incoming edge, statement nodes have one incoming edge, junction nodes have several incoming edges.

Remark: for simplicity, each node has a single output, possibly leading to several successor nodes. This means that the classical “test nodes” are transformed into pairs of conditions appearing in statement nodes, whose associated functions intersect their arguments with the condition of the test (“then” part) or its negation (“else” part).

4.3 Semantic equations

The analysis will associate with each node ν_i of the CFG an abstract value $X_i \in L$, these abstract values being defined by a system of recursive equations:

$$\forall i = 1..k,\ X_i = \begin{cases} \top & \text{if } \nu_i \text{ is a start node} \\ f_i(X_j) & \text{if } \nu_i \text{ is a statement node and } (\nu_j, \nu_i) \in \mathcal{E} \\ \displaystyle\bigsqcup_{(\nu_j,\nu_i)\in\mathcal{E}} X_j & \text{if } \nu_i \text{ is a junction node} \end{cases}$$

We will often write this system of equations as a vectorial fixpoint equation: $\boldsymbol{X} = F(\boldsymbol{X})$ in the lattice L^k.

4.4 Increasing sequence

Since the lattice L is of infinite depth, the Kleene sequence $\boldsymbol{X}_0 = \bot^k,\ \ \boldsymbol{X}_{\ell+1} = F(\boldsymbol{X}_\ell)$ may be infinite. The classical approach consists in computing the increasing sequence $\boldsymbol{Y}_0 = \bot^k, \boldsymbol{Y}_{\ell+1} = \boldsymbol{Y}_\ell \nabla F(\boldsymbol{Y}_\ell)$; from the properties of the widening operator, this sequence is guaranteed to converge after a finite number of steps towards a limit $\boldsymbol{Y}^\nabla$, which is a postfixpoint of F, i.e., $F(\boldsymbol{Y}^\nabla) \sqsubseteq \boldsymbol{Y}^\nabla$. The increasing sequence is computed in a chaotic way (cf. Figures 1, 2 and 3), by propagating changes along the paths of the CFG, and since the widening operation loses information, it is only applied on a selected set $\mathcal{W}$ of *widening nodes* intersecting each loop of the CFG.

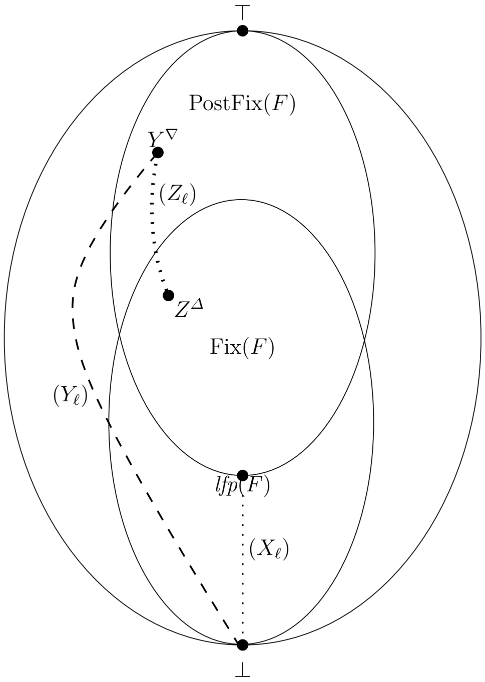


Fig. 4. Increasing and decreasing sequences

Note 1 (on the selection of widening nodes). The set $\mathcal{W}$ of widening nodes must be as small as possible, to minimise the number of applications of the widening operator. Since finding a minimal cutting set $\mathcal{W}$ is an NP-complete problem (apart for specific kinds of graphs [Sha79]), the heuristic classically applied is the method of *strongly connected subcomponents* proposed by Bourdoncle [Bou93]: the method recursively uses Tarjan's algorithm [Tar72] to find the strongly connected components (SCC) of a directed graph, together with an entry node to each SCC. Entry nodes are the targets of back edges, so *they are all junction nodes.* Bourdoncle's method consists in adding all SCC entry nodes to $\mathcal{W}$, then removing them from the graph and recursively apply Tarjan's algorithm to the rest of each SCC. The result is a hierarchy of *strongly connected subcomponents* (SCSC), each of which being cut by a junction node in $\mathcal{W}$. An obvious improvement of this method (which we did not find published anywhere) consists in considering again the hierarchy of SCSC bottom-up, checking whether each SCSC is disconnected by the cut-points of its children. For instance, on the CFG of Fig. 1(b), a first application of Tarjan's algorithm finds one non-trivial SCC, $c_1 = \{\nu_2, \nu_3, \nu_4, \nu_5, \nu_6\}$, with entry node ν_2. Removing node ν_2 and applying again the algorithm provides the SCSC $c_2 = \{\nu_4, \nu_5\}$, with entry node ν_4, whose removal disconnects the graph. So, Bourdoncle's method provides $\mathcal{W} = \{\nu_2, \nu_4\}$. Now, since the cut-point ν_4 of the leaf SCSC c_2 also disconnects the father SCSC c_1, it is enough to choose $\mathcal{W} = \{\nu_4\}$, as done in Fig. 1(b).

4.5 Decreasing sequence

The limit Y^∇ of the increasing sequence is a post-fixpoint of F. If it is a strict post-fixpoint — i.e., if $F(Y^\nabla) \sqsubset Y^\nabla$ —, it can be improved by computing a decreasing sequence $Z_0 = Y^\nabla$, $Z_{\ell+1} = F(Z_\ell)$, without widening. This sequence can be infinite, or reach a fixpoint of F. In practice, it generally reaches a fixpoint after very few steps. Now, since all of its terms are post-fixpoints of F, hence correct approximations of the least fixpoint, one can stop it after a fixed number of steps, or force its convergence using a *narrowing operator*[5]. In the following, we will note Z^Δ the last term of the descending sequence, and we will generally assume that Z^Δ is a fixpoint; however, the results still hold if Z^Δ is a strict post-fixpoint.

[5] Notice that widening and narrowing are *not* dual. Following [Cou15], widening is an extrapolation operator (its result is outside the range of its operands) while narrowing is an interpolation (its result is between its operands).

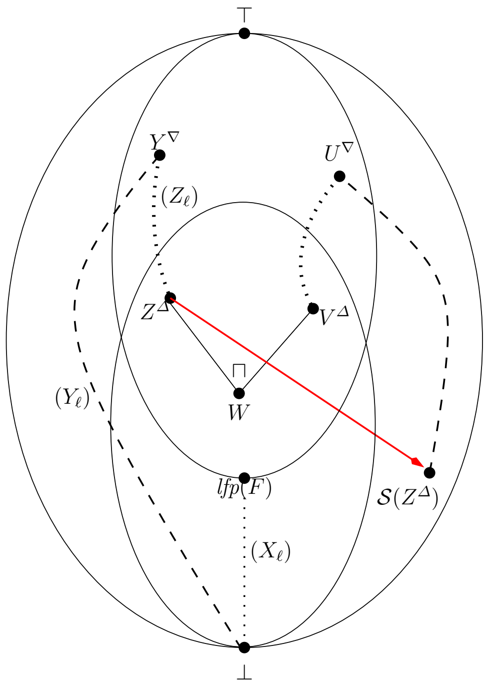


Fig. 5. Restarting from a new seed

Fig. 4 classically illustrates the sequences in the abstract lattice: $\mathrm{PostFix}(F)$ (resp., $\mathrm{Fix}(F)$) represents the set of postfixpoints (resp. fixpoints) of F. (X_ℓ) is the (generally infinite) Kleene's sequence, (Y_ℓ) is the (finite) increasing sequence (with widening), leading to the post-fixpoint Y^∇, and (Z_ℓ) is the decreasing sequence of post-fixpoints providing a solution Z^Δ.

5 Restarting from a well-chosen "seed"

5.1 An intuition of the method

Let us look again at Example 1.b (see Fig. 1(b)). At the widening node ν_4, during the decreasing sequence, we have $Z_4 = Z_3 \sqcup Z_5$. At the end of the decreasing sequence, we find $Z_4^\Delta = (i \in [0,\infty], j \in [0,100])$, while $Z_3^\Delta = (i \in [0,99], j \in [0,0])$ and $Z_5^\Delta = (i \in [0,\infty], j \in [1,100])$. Obviously, $i \in [0,99]$, found in Z_3^Δ, is a correct invariant, which is lost in Z_4^Δ because of the least upper bound with $i \in [0,\infty]$ imprecisely found in Z_5^Δ. We will say that Z_5^Δ *pollutes* Z_3^Δ at ν_4. Our idea is to start again a propagation of Z_3^Δ after resetting Z_5^Δ to $\bot$.

5.2 Generalised sequences

More generally, from the result Z^Δ of the decreasing sequence, a new value $\mathcal{S}(Z^\Delta)$ will be computed, that we call the *seed* from Z^Δ. Then, a new increasing sequence will be computed from $\mathcal{S}(Z^\Delta)$, leading to a postfixpoint U^∇, from which a new decreasing sequence will provide a new solution V^Δ. The greatest lower bound $W = Z^\Delta \sqcap V^\Delta$ is also a solution, which is better than both Z^Δ and V^Δ. This process is illustrated by Fig. 5.

The choice of the restarting seed will be the subject of the next two sections. Here, we make precise the computation of the new increasing and decreasing sequences. Restarting an iteration from an arbitrary point requires some changes in the definition of the iteration sequences. We must ensure that a widened sequence starting from an arbitrary point (not necessarily a pre-fixpoint) is increasing; moreover, widening operators are generally designed under the assumption that their first operand is smaller than the second one. We introduce the following notations: Let F be a monotone function[6] from L to L. Let $X \in L$. Then,

- we note $F^{\nabla}(X)$ the limit of the sequence $Y_0 = X,\ Y_{\ell+1} = Y_\ell \nabla (X \sqcup F(Y_\ell))$;
- we note $F^{\nabla\Delta}(X)$ the last term Z^{Δ} of a descending sequence (Z_ℓ) starting at $Z_0 = F^{\nabla}(X)$

Remarks:

- The second operand $X \sqcup F(Y_\ell)$ of the widening is always greater than the first one, and the increasing sequence (Y_ℓ) is indeed increasing. Obviously, $F^{\nabla}(X)$ is the classical approximation of the least fixpoint of the function $\lambda Y.(X \sqcup F(Y))$, i.e., of the least fixpoint of F greater than X.
- Notice also that, with $X = \bot$, these definitions of sequences and limits match the classical ones recalled in §4.
- For any X, $F^{\nabla\Delta}(X)$ is a correct approximation of the least fixpoint of F, i.e., $\forall X \in L,\ F^{\nabla\Delta}(X) \sqsupseteq$ *lfp*(F).
- Neither F^{∇} nor $F^{\nabla\Delta}$ is monotone. As a consequence, there can be some X such that $F^{\nabla\Delta}(\bot) \not\sqsubseteq F^{\nabla\Delta}(X)$, i.e., such that the limit obtained from X is more precise than, or incomparable with, the one computed by the classical iteration. Now, $F^{\nabla\Delta}(\bot) \sqcap F^{\nabla\Delta}(X)$ is a postfixpoint of F, i.e., a correct solution better than both the available ones.

Example: Coming back to Example 1.b, Fig. 6 shows the computation of the new sequences starting from the seed assigning the value $(i \in [0, 99], j \in [0, 0])$ to the widening node, $\bot$ elsewhere. Notice the difference between the initial and restarted sequences (Fig. 7).

5.3 Improvement

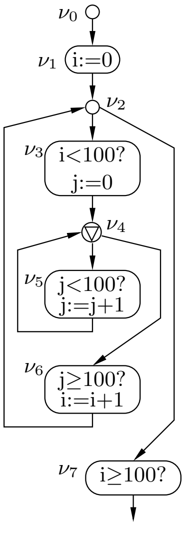
Since the final solution will be the greatest lower bound $F^{\nabla\Delta}(\bot) \sqcap F^{\nabla\Delta}(X)$, an obvious improvement is to perform the greatest lower bound operation at each step of the new increasing sequence, i.e., computing $Y_0 = X, Y_{\ell+1} = (Y_\ell \nabla (X \sqcup F(Y_\ell))) \sqcap Z^{\Delta}$.

With this improvement, in the previous example, the increasing sequence at widening node becomes $Y_0 = ([0, 99][0, 0]), Y_1 = ([0, 99][0, \infty]) \sqcap ([0, \infty], [0, 100]) = ([0, 99][0, 100])$, thus converges in 1 step to the fixpoint, without any decreasing sequence.

Remark on limited widening and termination: The improvement proposed above is a special case of "limited widening" [HPR97,BCC+03,LCJG11] (see Section 3): one uses an operator ∇_I defined by $X \nabla_I Y = (X \nabla Y) \sqcap I$, where I is a fixed abstract value (a known invariant, as in our case, or a tentative invariant with $X \sqsubseteq I$ and $Y \sqsubseteq I$). Now, the fact that ∇ satisfies the chain condition[7] does not imply that ∇_I does so, and, in principle, the increasing sequence built using ∇_I can be infinite. However, in practice, the very same arguments used to prove the chain condition for ∇ can be used to prove it for ∇_I: for instance, in the case of intervals, the chain condition for the classical widening [CC76] results from the fact that either $X \nabla Y = X$ (convergence) or the set of finite bounds of intervals in $X \nabla Y$ is strictly included in the set of finite bounds in X; the same argument can be used to prove the chain condition for ∇_I, for any I. In the case of polyhedra, for the classical widening [CH78] and variants [BHRZ03], either $X \nabla Y = X$ (convergence), or the dimension of $X \nabla Y$ is strictly greater than the dimension of X, or the set of constraints of $X \nabla Y$ is strictly included in the one of X; the same property holds for ∇_I.

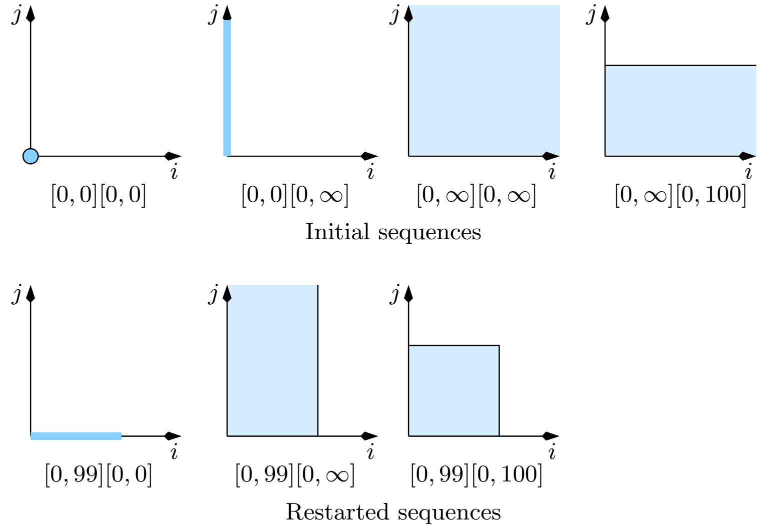
[6] For simplicity, we consider only the case of monotone abstract functions. The extension to non-monotone functions could be considered, since such functions appear in some contexts — like analysis by induction on the syntax, or when using some partially reduced products of domains [BCC+03], or in context-sensitive interprocedural analysis [ASV12,ASV13,ASS+16].

[7] The *chain condition*, imposed to a widening operator ∇, states that for any increasing sequence $X_0 \sqsubseteq X_1 \sqsubseteq X_2 \sqsubseteq \ldots$ of abstract values, the sequence $Y_0 = X_0$, $Y_{n+1} = Y_n \nabla X_{n+1}$ is not strictly increasing (i.e., converges after a finite number of terms).



	0	1	2	3	4 ↓
ν_0	$\bot$	$\top$			
ν_1	$\bot$	$[0,0]$ $[-\infty,\infty]$			
ν_2	$\bot$	$[0,0]$ $[-\infty,\infty]$		$[0,100]$ $[-\infty,\infty]$	
ν_3	$\bot$	$[0,0]$ $[0,0]$		$[0,99]$ $[0,0]$	
ν_4	$[0,99]$ $[0,0]$	$[0,99]$ $[0,0]$	$[0,99]$ $[0,\infty]$		$[0,99]$ $[0,100]$
ν_5	$\bot$	$[0,99]$ $[1,1]$	$[0,99]$ $[1,100]$		
ν_6	$\bot$	$\bot$	$[1,100]$ $[100,\infty]$		$[1,100]$ $[100,100]$
ν_7	$\bot$	$\bot$			$[100,100]$ $[-\infty,\infty]$
				convergence	fixpoint

Fig. 6. Example 1 — Restarting increasing-decreasing sequences



$[0,0][0,0]$ $[0,0][0,\infty]$ $[0,\infty][0,\infty]$ $[0,\infty][0,100]$

Initial sequences

$[0,99][0,0]$ $[0,99][0,\infty]$ $[0,99][0,100]$

Restarted sequences

Fig. 7. Example 1 — Successive values at widening node

6 Choice of a seed: "*select&project*" [HH12]

The choice of a seed consists in selecting, from the solution Z^{Δ} reached after the standard increasing-decreasing sequences, the values associated with widening nodes in the restarting seed. It is a heuristic process: each chosen value must neither be too large — and in particular, it should not be affected by an over-widening performed during the initial sequence — nor too small — otherwise it could be the source of the same over-widening that occurred in the initial sequence.

The problem can be limited to a strongly connected component of the control flow graph, since the analysis of a complex graph considers each SCC in turn. We first recall and discuss the choice proposed in [HH12], a method that we call "*select&project*".

Let us consider an SCC with only one widening node, ν_i (see Fig. 8). That node is a junction node (from the way widening nodes are selected, see Note 1 §4.4). The abstract value at ν_i depends on those at the preceding nodes in the SCC, say $\nu_{j_1}, \ldots, \nu_{j_m}$, and on abstract values propagated from outside (start nodes and preceding SCCs), which we note Y_i^0 since it is the first value not equal to $\bot$ at node ν_i during the increasing sequence. The semantic equations considered during the descending sequence can be subsumed as:

$$Z_i = Y_i^0 \sqcup Z_{j_1} \sqcup \ldots \sqcup Z_{j_m} \quad , \quad Z_{j_\ell} = F_{i,j_\ell}(Z_i), \ell = 1..m$$

where F_{i,j_ℓ} denotes the propagation function along the paths from ν_i to ν_{j_ℓ}.

At the end of the sequence, we have $Z_i^{\Delta} \sqsupseteq Y_i^0 \sqcup Z_{j_1}^{\Delta} \sqcup \ldots \sqcup Z_{j_m}^{\Delta}$ (an equality if $\boldsymbol{Z}^{\Delta}$ is a fixpoint).

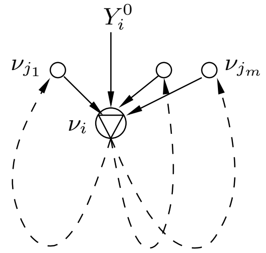


Fig. 8. An SCC with one widening node

[HH12] proposes to select one of the components $Z_{j_\ell}^{\Delta}$ and to assign $Y_i^0 \sqcup Z_{j_\ell}^{\Delta}$ as a restarting value. As a matter of fact, Y_i^0 clearly consists of states which would not be shown to be unreachable and will belong to any solution. Now, a component $Z_{j_\ell}^{\Delta}$ is likely to improve the solution only if it is strictly smaller than Z_i^{Δ}. It may happen for two reasons:

- either some "initial states" in Y_i^0 have been left on the paths from ν_i to $\nu_{j\ell}$; this case is not interesting since, as said before, Y_i^0 will be included in any solution.
- or, during its propagation along these paths, Z_i^{Δ} has been "truncated" by some condition and/or "projected" by some assignment; this is the interesting case which can add some information.

As a consequence, $Z_{j_\ell}^{\Delta}$ will be selected only if $Y_i^0 \sqcup Z_{j_\ell}^{\Delta} \sqsubset Z_i^{\Delta}$. Moreover, it is useless to propagate again Y_i^0, which already provided the existing result Z^{Δ}. So, $Z_{j_\ell}^{\Delta}$ will be selected only if $Z_{j_\ell}^{\Delta} \not\sqsubseteq Y_i^0$.

Example: This selection process selects the convenient restarting value for ν_5 in our example 1.b (Fig. 1(b) and §5.1). We have $Z_5^{\Delta} = ([0, \infty], [0, 99])$, and $Y_5^0 = ([0, 0][0, 0])$. The two incoming values are $Z_3^{\Delta} = ([0, 99][0, 0])$ and $Z_6^{\Delta} = ([0, \infty], [1, 100])$. Now,

- $Y_5^0 \sqcup Z_6^{\Delta} = ([0, \infty], [0, 100]) = Z_5^{\Delta}$, so Z_6^{Δ} is not selected (only initial values with $j = 0$ have been left).
- $Y_5^0 \sqcup Z_3^{\Delta} = ([0, 99][0, 0]) \sqsubset Z_5^{\Delta}$ and $Z_3^{\Delta} \not\sqsubseteq Y_5^0$, this is why $Y_5^0 \sqcup Z_3^{\Delta}$ was selected for restarting in Fig. 6.

Discussion: This method is easily extended to multiple widening nodes. Notice also that it is completely independent of the abstract lattice used. However, it suffers some weaknesses that are highlighted by our examples.

- The "pollution" of precise values may occur elsewhere than at widening nodes. In our Example 2 (Fig. 2.b) it occurs at junction node ν_7: the initial sequences terminate with $Z_2^\Delta = Z_7^\Delta = Z_3^\Delta = (n \in [0,\infty])$, while the precise value to be selected is $Z_6^\Delta = (n \in [0,60])$. So the convenient values to be kept in the seed may have to be searched upward the widening nodes.
- Nothing is said about how to proceed when several values $Z_{j_\ell}^\Delta$ could be selected. Moreover, our Example 3 (Fig. 3) shows that the precise value may be a *combination* of incoming values at a junction node. In that example, the initial sequences terminate with $Z_2^\Delta = Z_9^\Delta = (m \in [0,\infty], n \in [0,\infty])$, while $Z_5^\Delta = (m \in [0,60], n \in [0,\infty])$ and $Z_8^\Delta = (m \in [0,\infty], n \in [0,60])$ are more precise but incomparable with each other. Here, obviously, the precise value to restart with would be the combination $Z_5^\Delta \sqcap Z_8^\Delta$.

7 Choice of a seed, new proposal: "*improve&project*"

7.1 Restarting values at widening nodes

As before, we propose to select a seed all components of which are $\bot$, except those corresponding to widening nodes. This choice makes the computation order of the restarting sequence easier to define: since it is important to apply the widening as late as possible, the new increasing sequence is computed by propagating values from the widening nodes in the SCC until nothing can change, then performing the widening at some widening nodes, and initiate another propagation.

Following the previous discussion, the restarting value at a widening node is a "combination" of "contributions" recursively provided by predecessor nodes:

$$\mathcal{S}(Z)[\nu_i] = \begin{cases} \mathit{combine}\left(\{\mathit{contrib}(Z,\nu_j) \;\mid\; (\nu_j,\nu_i) \in \mathcal{E}\}\right) & \text{if } \nu_i \in \mathcal{W} \\ \bot \quad \text{otherwise} \end{cases} \tag{1}$$

Now, we have to define the functions *contrib* and *combine*.

7.2 Extracting "contributions" to restarting values

We already noticed that restarting values should represent sets of states that are likely to be effectively reachable, i.e., not polluted by overwidening. On the other hand, they should be as large as possible, not to provoke further overwidening in the restarted sequences. Our definition of the *contrib* function is based on the following remarks:

- Some components of Z^Δ may have been overwidened. We must detect them as much as possible, and prevent them to pollute more precise values. The worst consequence of overwidening is to produce unbounded values: intervals or templates with infinite bounds, polyhedra with rays, regular expressions with "$*$", ... In our proposal, bounded values will be trusted to represent reachable states: if Z_i^Δ is bounded, the contribution of the node ν_i will be the value Z_i^Δ unchanged.
- Information is lost at junction nodes: the contribution of such a node will be a specific "combination" (defined in the next subsection) of incoming contributions. This combination is intended to prevent the pollution of precise values by overwidened ones.
- Contributions are searched backward from a widening node. To prevent infinite recursion, this recursive search must stop at widening nodes.
- Finally, for statement nodes, the incoming contribution is propagated normally according to the semantics of the node.

According to these remarks, we define the contribution of a node ν_i as follows:

$$\mathit{contrib}(Z,\nu_i) = \begin{cases} Z_i, \quad \text{if } Z_i \text{ is bounded, or if } \nu_i \in \mathcal{W} \\ \mathit{combine}\left(\{\mathit{contrib}(Z,\nu_j) \mid (\nu_j,\nu_i) \in \mathcal{E}\}\right), \text{if } \nu_i \text{ is a junction node} \\ f_i(\mathit{contrib}(Z,\nu_j)), \quad \text{if } \nu_i \text{ is a statement node and } (\nu_j,\nu_i) \in \mathcal{E} \end{cases} \tag{2}$$

Boundedness and rays: The previous definition uses the notion of boundedness of abstract values. Here, we make this notion precise in the case of classical numerical domains. Intervals [CC76], octagons [Min01], templates [SSM05] are special cases of convex polyhedra [CH78]. A polyhedron P is unbounded if it contains rays, i.e., if there exist vectors $\boldsymbol{v}$ such that $\forall x \in P, \forall \mu \geq 0, x + \mu\boldsymbol{v} \in P$. If the set of rays of P is not empty, it is generated by a finite set $\mathcal{R}(P)$ of extremal rays, i.e., each ray of P is a positive combination of extremal rays. As a consequence, a convex polyhedron is bounded if and only if $\mathcal{R}(P) = \emptyset$. $\mathcal{R}(P)$ is generally available from polyhedra libraries based on the double-representation [BRZH02,JM09]. Concerning intervals, octagons or templates, the representation of extremal rays is even simpler, since they correspond simply to infinite bounds of intervals or constraints. We will use the following equivalence relation between abstract values:

$$P \overset{R}{\sim} Q \overset{\text{def}}{\iff} \mathcal{R}(P) = \mathcal{R}(Q)$$

7.3 Combining incoming contributions at junction nodes

The combination function is to be used to compute the contribution of junction nodes (equation 2), as well as to obtain the restart value at widening nodes (equation 1). Its role is to extract a maximal value resulting from a set of incoming contributions, while avoiding the "pollution" by overwidened values.

Our definition is based on the following remarks:

- the presence of rays in an abstract value is an indication that it may result from an overwidening. So, the combination should avoid the propagation of rays.
- as a consequence, when combining values with different sets of rays, we choose to return their greatest lower bound.
- on the other hand, in presence of values with the same set of rays, there is no reason to return a more bounded combination, so we return their least upper bound.
- as noted before, the first value $Y^0[\nu]$ different from $\bot$ associated with a node ν will always be included in the contribution of ν. It will be added to the incoming contributions before combining them.

As a consequence, to combine the set S of incoming contributions at a junction node ν, we propose to return

$$\mathit{combine}\,(S) = \bigsqcap_{S_p \in S/\overset{R}{\sim}} \left(Y^0[\nu] \sqcup \bigsqcup S_p\right)$$

i.e, to partition the set S of incoming contributions into values with the same sets of rays ($S_p \in S/\overset{R}{\sim}$), to compute the least upper bound of each S_p, together with $Y^0[\nu]$, and to return the greatest lower bound of the results.

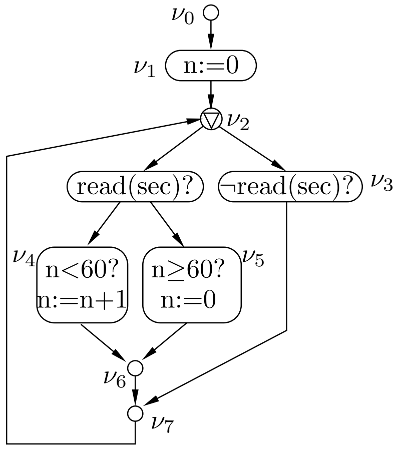
8 Examples

8.1 Example 2 (Fig. 2.b)

From the widening node ν_2, contributions are requested from ν_7, then, separately

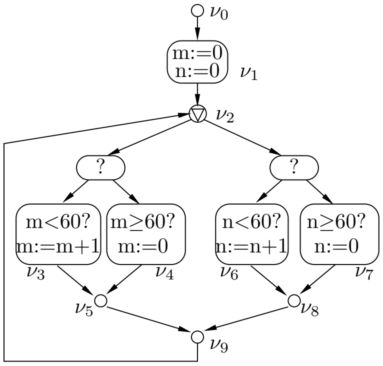
- from ν_6, the value of which is bounded ($n \in [0, 60]$)
- from $\nu 3$, then from ν_2, the widening node which returns its value ($n \in [0, \infty]$), propagated forward at ν_3.

Now, at junction node ν_7 — with $Y^0[\nu_7] = (n \in [0, 1])$ —, the incoming contributions from ν_6 and ν_3 are combined into their greatest lower bound ($n \in [0, 60]$), which is returned back to ν_2 as the restarting value. It is an invariant, and the best one.



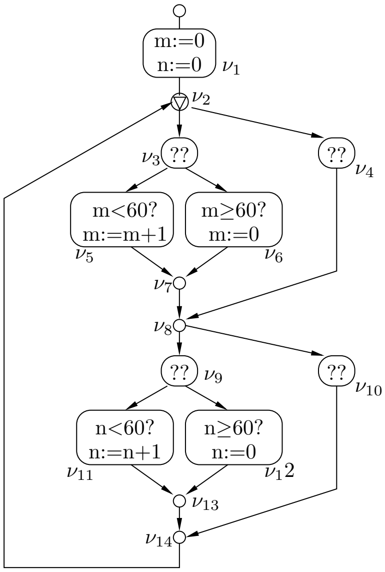
8.2 Example 3 (Fig. 3)

From ν_2, contribution is requested from ν_9, then separately from ν_5 and ν_8 (which associated values are both unbounded). The request for contributions is propagated to $\nu_3, \nu_4, \nu_6, \nu_7$ and finally ν_2 which will all return their associated values without any change. At junction nodes ν_5 and ν_8, the incoming contributions have the same rays, so their least upper bound is returned, i.e., $(m \in [0,60], n \in [0,\infty])$ and $(m \in [0,\infty], n \in [0,60])$, respectively. These values are transmitted back to ν_9, and since their sets of rays differ, their greatest lower bound $(m \in [0,60], n \in [0,60])$ is transmitted back to ν_2 as the restarting value. It is again the best invariant.



8.3 An additional example

Let us consider now two event counters composed sequentially. The initial sequences provide the solution Z^{Δ}, with $Z_2^{\Delta} = Z_{14}^{\Delta} = Z_{10}^{\Delta} = Z_8^{\Delta} = Z_4^{\Delta} = (m \in [0,\infty], n \in [0,\infty]$, and $Z_7^{\Delta} = (m \in [0,60], n \in [0,\infty]$, $Z_{13}^{\Delta} = (m \in [0,\infty], n \in [0,60]$. None of these values are bounded, so contributions are requested backward along all the paths from ν_2 to ν_2. Contributions from ν_7 $(m \in [0,60], n \in [0,\infty])$ and ν_4 $(m \in [0,\infty], n \in [0,\infty])$ are intersected at ν_8, providing $(m \in [0,60], n \in [0,\infty])$, so ν_{13} returns $(m \in [0,60], n \in [0,60])$, while ν_{10} returns $(m \in [0,60], n \in [0,\infty])$, and a new intersection at ν_{14} returns the precise value $(m \in [0,60], n \in [0,60])$ for restarting at ν_2.



8.4 A polyhedral version of Example 1

We already mentionned that, for simplicity, all our examples use the lattice of intervals, and that they could be precisely analysed using policy iteration [SW04,CGG+05,GS07,KMW16]. Now we present an example (already used in [HH12]) transformed from our Example 1 by performing an affine variable change. This example cannot be properly analysed with intervals nor octagons, and the template to be used is not obvious, so policy iteration cannot be used. Using the lattice of polyhedra, our method discovers a constraint that does not appear in the program, and which could not be found by any existing limited widening [Hal93,HPR97,BCC+03,LCJG11,SK06,WYGI07].

Fig 9 shows the CFG of the example, together with a graphical representation of its behaviour (the succession of states at ν_5). The initial increasing-decreasing sequences (see Fig 10, (a) and (b)) terminate with $Z_5^{\Delta} = (0 \le j \le 2i, j \le 4)$. Since $Z_3^{\Delta} = (0 \le i \le 3, j = 0)$ is bounded, it is the contribution of ν_3; ν_7 contributes with $Z_7^{\Delta} \sqcup Y_7^0 = (2 \le j \le 2i, j \le 4) \sqcup (i = j = 0) = (0 \le j \le 2i, j \le 4)$, and the restarting value is the greatest lower bound of these contributions, i.e., $(0 \le i \le 3, j = 0)$ (Fig 10 (c)). The new increasing-decreasing sequences provide the precise invariant $(0 \le j \le 2i \le j + 6, j \le 4)$ at ν_5 (Fig 10 (e)). This result is obtained directly (i.e., from (c) to (e)) if the terms of the increasing sequences are intersected with Z^{Δ}, applying the improvement of §5.3. Notice that the constraint $2i \le j + 6$ does not appear anywhere in the program, and could not be used to limit the widening.

9 Experimental results

Both methods have been implemented in the prototype tool Pagai [HMM12]. Pagai computes numerical invariants in programs expressed in the LLVM internal representation [LA04]. In this representation, a

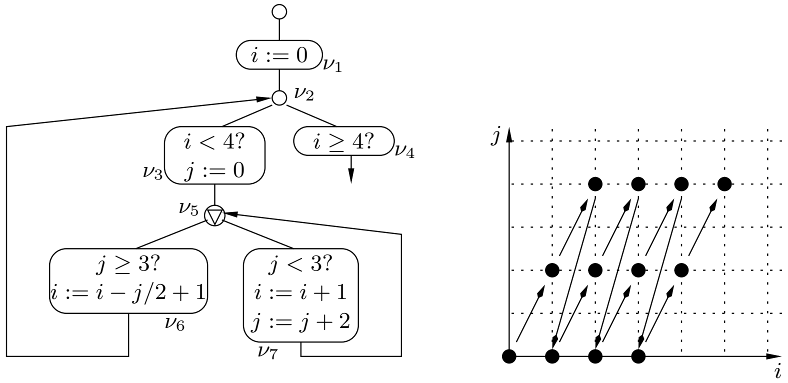
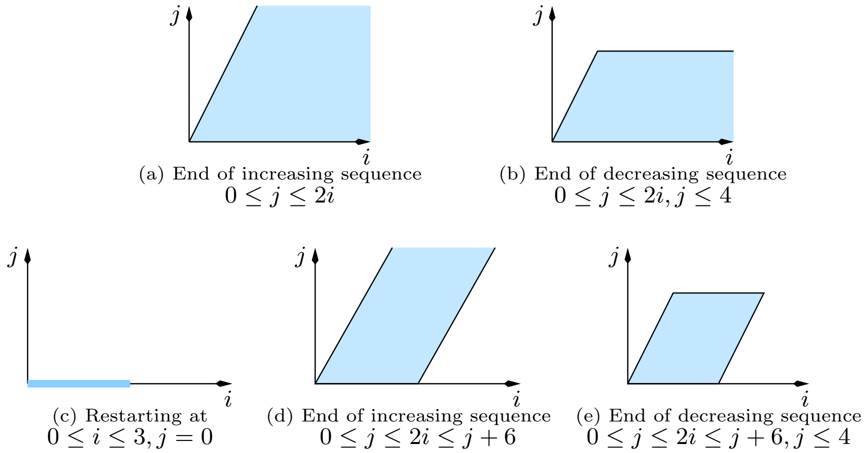


Fig. 9. A polyhedral example



(a) End of increasing sequence
$0 \leq j \leq 2i$

(b) End of decreasing sequence
$0 \leq j \leq 2i, j \leq 4$

(c) Restarting at
$0 \leq i \leq 3, j = 0$

(d) End of increasing sequence
$0 \leq j \leq 2i \leq j + 6$

(e) End of decreasing sequence
$0 \leq j \leq 2i \leq j + 6, j \leq 4$

Fig. 10. Analysis of the polyhedral example, values at ν_5

function is a graph of basic blocks. The analyser takes as input such an LLVM file (that can be obtained from a C, C++, Fortran program by llvm-gcc or clang), and outputs for each basic block a numerical inductive invariant over the variables that are live at the head of this block. We can choose among several abstract domains : convex polyhedra, octagons, intervals, etc. through the Apron library [JM09].

All our experiments are made with the polyhedra abstract domain. Widening is never delayed, and decreasing sequences are limited to 5 terms. All function calls are inlined using the "always inline" LLVM transformation pass. The analysis times are those obtained on an Intel Xeon E5-2630 v3 2.40 Ghz machine with 32 GB of RAM and 20 MB of cache running Linux.

Table 1 compares these methods with the basic analysis, in term of number of constraints found in the loop invariants. As expected, the *improve&project* method finds the precise invariants for all examples. Analysis times are hardly significant for such tiny programs.

Now, obviously, our example programs are very specific. So, it is important to experiment our approach on an existing benchmark. We conducted such a experiment on the benchmark[8] from the Mälardalen WCET research group, which contains various, small or medium size programs, such as sorts, matrix transformations, fft, etc. The goal is, again, to compare the basic method with *select&project* and *improve&project* methods. A precise, quantitative comparison of results would be difficult — involving, for

[8] www.mrtc.mdh.se/projects/wcet/benchmarks.html

	basic		*select&project*		*improve&project*	
program	#const.	time (s)	#const.	time (s)	#const.	time (s)
example 1.a	2	0.012	2	0.008	2	0.008
example 1.b	3	0.016	4	0.016	4	0.016
example 2.a	2	0.012	2	0.012	2	0.016
example 2.b	1	0.008	1	0.008	2	0.016
example 3 §8.2	2	0.016	2	0.016	4	0.028
seq. modulo §8.3	2	0.016	2	0.024	4	0.028
ex1 poly §8.4	3	0.024	4	0.024	4	0.028

Table 1. Experiments on the examples

instance, the computation of volumes of polyhedra. Here, we just count the number of loop invariants (i.e., results at widening nodes) which are strictly more precise with one method than with the basic one.

Table 2 gives, for each program of the benchmark, the time in seconds of the analysis with the basic method, the number of widening points, then the percentage of widening points where each new method improves the results of the basic method, and the overhead of each new method (i.e., the factor by which the basic analysis time is multiplied). These results are graphically represented in Fig. 9.

The results are significantly improved, by one method or the other, in 18 over 32 programs. This shows that the phenomenon we considered is by no way specific to our examples, but happens quite often in usual programs. As expected, the *improve&project* method generally gives better results than *select&project*, with some exceptions (janne_complex.c and select.c) due to the fact that *select&project* may improve bounded values that are ignored by *improve&project*. Here again, many very short analysis times are hardly relevant. However, overhead values must be commented: for most programs, the cost of new methods is around twice the cost of the basic analysis, which is not surprising since they involve new increasing-decreasing sequences computations. Larger overheads correspond to longer sequences in the improvement phase. The largest overhead ($\sim\times 7$, for adpcm.c with *improve&project*) corresponds to a very significant improvement of the results ($\sim 70\%$ of the 27 widening nodes). There are also a few less good cases, like fibcall.c, where the cost of *improve&project* is high ($\times 4$) for no benefit.

10 Conclusion

We have presented two methods for improving results of program analysis by abstract interpretation in infinite abstract lattices. Both methods consist in restarting increasing and decreasing sequences from well-chosen values. They differ in the way the restarting values are constructed:

- The so-called *select&project* method was already proposed in [HH12], and consists in selecting some components of the basic solution, according to some criteria. This method is fully independent of the abstract lattice.
- The new method, which we call *improve&project*, tries to improve the restarting values by gathering and combining information backwards the widening nodes in the basic solution. This methods makes use of a notion of ray (infinite direction), which is easy to define in classical numerical abstract lattices (intervals, octagons, polyhedra, ...).

While the design of these methods was strongly guided by a small set of specific example programs, experiments show that they work well for a large proportion of programs taken from a classical benchmark. The results of the *improve&project* method are generally more precise, with some exceptions due to the fact that it can only improve unbounded solutions, in contrast with the previous *select&project* method. An obvious improvement would be to apply both methods and return the intersection of their results,

			% improved widening pts values		Overhead w.r.t. basic	
Program	basic analysis time (s)	# widening pts	*select& project*	*improve& project*	*select& project*	*improve& project*
adpcm.c	2.65	27	14.81	70.37	2.57	6.93
bs.c	0.03	1	0.00	0.00	1.33	1.00
cnt.c	0.04	4	50.00	50.00	1.75	2.25
compress.c	0.42	11	0.00	54.55	1.88	1.50
cover.c	1.75	3	0.00	0.00	1.55	1.91
crc.c	0.07	6	33.33	33.33	1.71	1.57
duff.c	0.04	1	0.00	0.00	1.00	1.25
edn.c	0.08	12	16.67	16.67	1.75	1.25
expint.c	0.02	3	0.00	0.00	1.50	1.50
fac.c	0.01	1	0.00	0.00	2.00	3.00
fdct.c	0.04	2	0.00	0.00	1.25	1.25
fft1.c	0.27	30	23.33	20.00	1.74	1.85
fibcall.c	0.01	1	0.00	0.00	2.00	4.00
fir.c	0.04	2	50.00	50.00	1.50	1.00
insertsort.c	0.02	2	50.00	50.00	1.50	1.00
janne_complex.c	0.02	2	50.00	0.00	2.50	3.00
jfdctint.c	0.04	3	0.00	0.00	1.00	1.25
lcdnum.c	0.08	1	0.00	0.00	1.00	1.63
lms.c	0.13	12	16.67	66.67	2.46	2.15
ludcmp.c	0.07	11	45.45	45.45	1.29	1.57
matmult.c	0.03	7	42.86	57.14	2.67	2.67
minver.c	0.09	17	29.41	47.06	2.44	2.11
ndes.c	0.11	12	8.33	50.00	1.64	1.45
ns.c	0.05	4	25.00	75.00	1.20	1.60
nsichneu.c	0.81	1	0.00	0.00	1.19	1.49
prime.c	0.02	2	0.00	0.00	2.00	2.50
qsort-exam.c	0.10	6	0.00	0.00	1.30	1.10
qurt.c	0.12	3	0.00	0.00	1.83	1.50
select.c	0.13	4	75.00	0.00	1.77	1.54
sqrt.c	0.04	1	0.00	0.00	1.75	1.00
ud.c	0.06	11	45.45	45.45	1.67	1.83
whet.c	0.06	11	9.09	0.00	1.67	1.33

Table 2. Methods comparison on Mälardalen benchmark

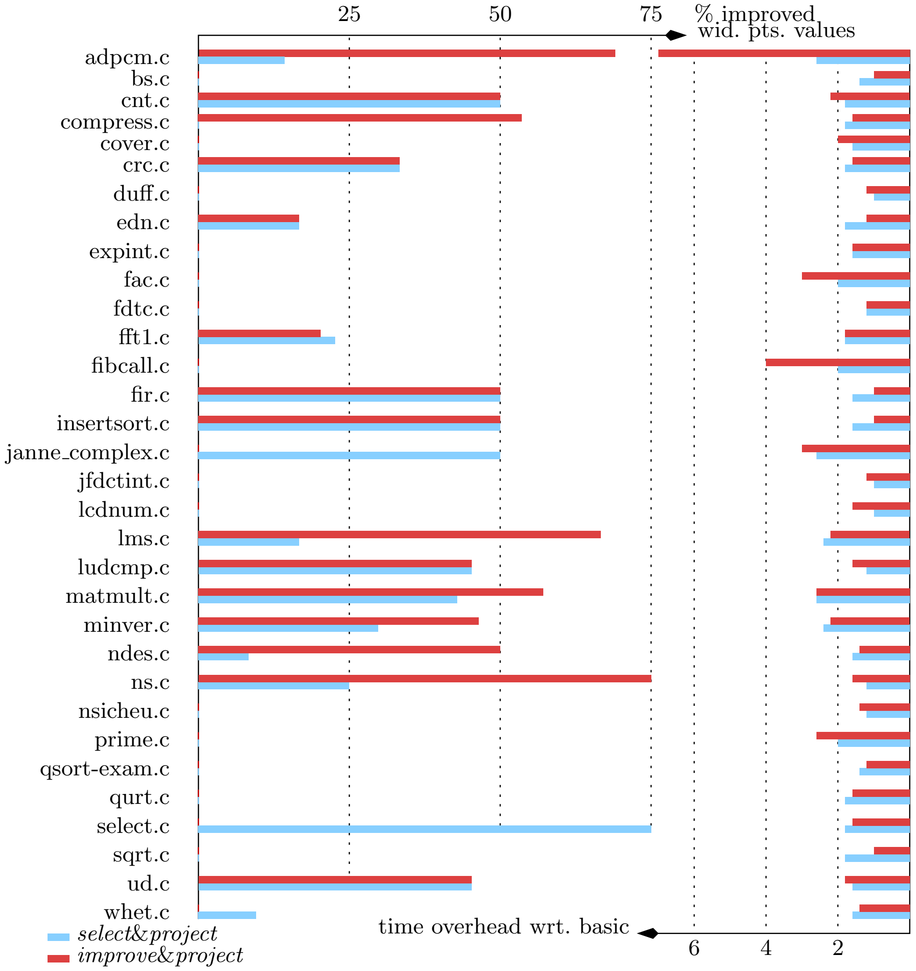


Fig. 11. Graphical view of results on Mälardalen benchmark

but the time overhead is clearly an obstacle. Another, cheaper way of combining both methods would be to combine the restarting values they construct, before computing new increasing-decreasing sequences once. These methods are fully compatible with all other techniques for improving the precision of abstract interpretation in infinite abstract lattices.

Acknowledgements: Julien Henry implemented the first version of the *select&project* method in his analyzer Pagai. We are also indebted to the reviewers for their helpful comments.

References

AS13. G. Amato and F. Scozzari. Localizing widening and narrowing. In *Static Analysis International Symposium, SAS 2013*, pages 25–42, Seattle, WA, June 2013.

ASS$^+$16. G. Amato, F. Scozzari, H. Seidl, K. Apinis, and V. Vojdani. Efficiently intertwining widening and narrowing. *Science of Computer Programming*, 120:1–24, 2016.

ASV12. K. Apinis, H. Seidl, and V. Vojdani. Side-effecting constraint systems: A swiss army knife for program analysis. In *Asian Symposium on Programming Languages and Systems, APLAS 2012*, pages 157–172, Kyoto, Japan, December 2012.

ASV13. K. Apinis, H. Seidl, and V. Vojdani. How to combine widening and narrowing for non-monotonic systems of equations. In *Programming Language Design and Implementation, PLDI 2013*, pages 377–386, Seattle, WA, June 2013.

BCC+03. B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *Programming Language Design and Implementation, PLDI 2003*, pages 196–207, San Diego, California, June 2003.

BFLP03. S. Bardin, A. Finkel, J. Leroux, and L. Petrucci. Fast: Fast acceleration of symbolic transition systems. In *Computer Aided Verification, CAV 2003*, pages 118–121, Boulder, Colorado, July 2003.

BGP97. T. Bultan, R. Gerber, and W. Pugh. Symbolic model checking of infinite state systems using Presburger arithmetic. In *Computer Aided Verification, CAV 1997*, pages 400–411, Haifa, Israel, June 1997.

BHRZ03. R. Bagnara, P. M. Hill, E. Ricci, and E. Zaffanella. Precise widening operators for convex polyhedra. In *Static Analysis Symposium, SAS 2003*, pages 337–354, San Diego, California, USA, 2003.

Bou93. F. Bourdoncle. Efficient chaotic iterations strategies with widening. In *International Conference on Formal Methods in Programming and their applications*, pages 128–141. LNCS 735, Springer Verlag, 1993.

BRZH02. R. Bagnara, E. Ricci, E. Zaffanella, and P. M. Hill. Possibly not closed convex polyhedra and the Parma Polyhedra Library. In *Static Analysis Symposium, SAS 2002*, pages 213–229, Madrid, Spain, September 2002.

BS91. F. Boussinot and R. de Simone. The Esterel language. *Proceedings of the IEEE*, 79(9):1293–1304, September 1991.

CC76. P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *2nd International Symposium on Programming*. Dunod, Paris, 1976.

CC77. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Principles of Programming Languages, POPL 1977*, pages 238–252, Los Angeles, California, January 1977.

CGG+05. A. Costan, S. Gaubert, E. Goubault, M. Martel, and S. Putot. A policy iteration algorithm for computing fixed points in static analysis of programs. In *Computer-Aided Verification, CAV 2005*, pages 462–475, Edinburgh, Scotland, July 2005.

CH78. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Principles of Programming Languages, POPL 1978*, pages 84–96, Tucson, Arizona, January 1978.

CJ98. H. Comon and Y. Jurski. Multiple counters automata, safety analysis and Presburger arithmetic. In *Computer Aided Verification, CAV 1998*, pages 268–279, Vancouver, Canada, June 1998.

Cou15. P. Cousot. Abstracting induction by extrapolation and interpolation. In *Verification, Model Checking, and Abstract Interpretation, VMCAI 2015*, pages 19–42, Mumbai, India, January 2015.

CZ11. A. Cortesi and M. Zanioli. Widening and narrowing operators for abstract interpretation. *Computer Languages, Systems & Structures*, 37(1):24–42, 2011.

GCNR08. B. S. Gulavani, S. Chakraborty, A. V. Nori, and S. K. Rajamani. Automatically refining abstract interpretations. In *Tools and Algorithms for Construction and Analysis of Systems,TACAS 2008*, pages 443–458, Budapest, Hungary, March 2008.

GH06. L. Gonnord and N. Halbwachs. Combining widening and acceleration in linear relation analysis. In *Static Analysis Symposium, SAS 2006*, pages 144–160, Seoul, Korea, August 2006.

Gou01. E. Goubault. Static analyses of the precision of floating-point operations. In *Static Analysis Symposium, SAS 2001*, pages 234–259, Paris, France, July 2001.

GR06a. D. Gopan and T. Reps. Lookahead widening. In *Computer Aided Verification, CAV 2006*, pages 452–466, Seattle, WA, August 2006.

GR06b. B. S. Gulavani and S. K. Rajamani. Counterexample driven refinement for abstract interpretation. In *Tools and Algorithms for Construction and Analysis of Systems, TACAS 2006*, pages 474–488, Vienna, Austria, March 2006.

GR07. D. Gopan and T. W. Reps. Guided static analysis. In *Static Analysis Symposium, SAS 2007*, pages 349–365, Kongens Lyngby, Denmark, August 2007.

GS07. T. Gawlitza and H. Seidl. Precise fixpoint computation through strategy iteration. In *European Symposium on Programming, ESOP 2007*, pages 300–315, Braga, Portugal, March 2007.

GS14. L. Gonnord and P. Schrammel. Abstract acceleration in linear relation analysis. *Science of Computer Programming*, 93:125–153, 2014.

Hal93. N. Halbwachs. Delay analysis in synchronous programs. In *Computer-Aided Verification, CAV 1993*, pages 333–346, Elounda, Greece, July 1993.

HH12. N. Halbwachs and J. Henry. When the decreasing sequence fails. In *Static Analysis Symposium, SAS 2012*, pages 198–213, Deauville, France, September 2012.

HMM12. J. Henry, D. Monniaux, and M. Moy. PAGAI: a path sensitive static analyzer. *Electr. Notes Theor. Comput. Sci.*, 289:15–25, 2012.

HPR97. N. Halbwachs, Y.E. Proy, and P. Roumanoff. Verification of real-time systems using linear relation analysis. *Formal Methods in System Design*, 11(2):157–185, August 1997.

JM09. B. Jeannet and A. Miné. Apron: a library of numerical abstract domains for static analysis. In *Computer Aided Verification, CAV 2009*, pages 661–667, Grenoble, France, July 2009.

KMW16. E. Karpenkov, D. Monniaux, and P. Wendler. Program analysis with local policy iteration. In *Verification, Model Checking, and Abstract Interpretation, VMCAI 2016*, pages 127–146, St Petersburg, Florida, January 2016.

LA04. C. Lattner and V. Adve. LLVM: a compilation framework fopr lifelong program analysis & transformation. In *Code Generation and Optimization, CGO 2004*, pages 75–86, Washington, DC, August 2004.

LCJG11. L. Lakhdar-Chaouch, B. Jeannet, and A. Girault. Widening with thresholds for programs with complex control graphs. In *Automated Technology for Verification and Analysis, ATVA 2011*, pages 492–502, Taipei, Taiwan, October 2011.

Min01. A. Miné. The Octagon abstract domain. In *Eighth Working Conference on Reverse Engineering, WCRE 2001*, pages 310–319, Stuttgart, Germany, October 2001.

Min04. A. Miné. *Weakly relational numerical abstract domains*. PhD thesis, Ecole Polytechnique, Paris, 2004.

ML12. D. Monniaux and J. Le Guen. Stratified static analysis based on variable dependencies. *Electr. Notes Theor. Comput. Sci.*, 288:61–74, 2012.

PGM03. S. Putot, E. Goubault, and M. Martel. Static analysis-based validation of floating-point computations. In *Numerical Software with Result Verification, International Dagstuhl Seminar*, pages 306–313. LNCS 2991, Springer Verlag, 2003.

Sha79. A. Shamir. A linear time algorithm for finding minimum cutsets in reducible graphs. *SIAM Journal on Computing*, 8(4):645–655, November 1979.

SK06. A. Simon and A. King. Widening Polyhedra with Landmarks. In *Asian Symposium on Programming Languages and Systems, APLAS 2006*, pages 166–182, Sydney, Australia, November 2006.

SSM04. S. Sankaranarayanan, H. Sipma, and Z. Manna. Constraint-based linear relations analysis. In *Static Analysis Symposium, SAS 2004*, pages 53–68, Verona, Italy, August 2004.

SSM05. S. Sankaranarayanan, H. Sipma, and Z. Manna. Scalable analysis of linear systems using mathematical programming. In *Verification, Model-checking, and Abstract Intepretation, VMCAI 2005*, pages 25–41, Paris, France, January 2005.

SW04. Z. Su and D. Wagner. A class of polynomially solvable range constraints for interval analysis without widenings and narrowings. In *Tools and Algorithms for Construction and Analysis of Systems, TACAS 2004*, pages 280–295, Barcelona, Spain, March 2004.

Tar72. R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1:146–160, 1972.

WB98. P. Wolper and B. Boigelot. Verifying systems with infinite but regular state spaces. In *Computer-Aided Verification, CAV'98*, pages 88–97, Vancouver, Canada, June 1998.

WYGI07. C. Wang, Z. Yang, A. Gupta, and F. Ivančić. Using counterexamples for improving the precision of reachability computation with polyhedra. In *Computer-Aided Verification, CAV 2007*, pages 352–365, Berlin, Germany, July 2007.