Widening as Abstract Domain

Bogdan Mihaila and Alexander Sepp and Axel Simon

Technical University of Munich, Garching b. München, Germany
{firstname.lastname}@in.tum.de

Abstract Verification using static analysis often hinges on precise numeric invariants. Numeric domains of infinite height can infer these invariants, but require widening/narrowing which complicates the fixpoint computation and is often too imprecise. As a consequence, several strategies have been proposed to prevent a precision loss during widening or to narrow in a smarter way. Most of these strategies are difficult to retrofit into an existing analysis as they either require a pre-analysis, an on-the-fly modification of the CFG, or modifications to the fixpoint algorithm. We propose to encode widening and its various refinements from the literature as cofibered abstract domains that wrap standard numeric domains, thereby providing a *modular* way to add numeric analysis to any static analysis, that is, without modifying the fixpoint engine. Since these domains cannot make any assumptions about the structure of the program, our approach is suitable to the analysis of executables, where the (potentially irreducible) CFG is re-constructed on-the-fly. Moreover, our domain-based approach not only mirrors the precision of more intrusive approaches in the literature but also requires fewer iterations to find a fixpoint of loops than many heuristics that merely aim for precision.

Adding numeric domains of infinite height to a static analysis requires that widening and/or narrowing is applied within each loop of the program to ensure termination [7]. Commonly, this is implemented by modifying the fixpoint algorithm to perform upward and downward iterations while a pre-analysis determines necessary widening points. Firstly, downward iterations can be problematic since a widened state can induce a precision loss in other domains that cannot be reverted with the narrowed numeric state [17]. Secondly, determining a minimal set of widening points requires non-trivial algorithms for irreducible control flow graphs (CFGs) [6]. Worse, these algorithms cannot be applied in the context of analyzing machine code, as the CFG is re-constructed on-the-fly while computing the fixpoint [3]. Moreover, narrowing alone is often not enough to obtain precise fixpoints which has been illustrated in many papers that present improved widenings/narrowings [10,11,12,15,17]. All of these approaches require disruptive changes to the fixpoint engine, for instance, tracking several abstract states [10,12], temporarily disabling parts of the CFG [11], performing a pre-analysis with different semantics [13,15], collecting "landmarks" [17] or referring to user-supplied thresholds [5]. This paper shows that widening and its various refinements can be implemented without modifying an existing fixpoint engine, thereby making numeric domains available to analyses that are oblivious to the

```
1 int x = 0;
2 int y = 0;
3 while (x < 100) {
4   x = x + 1;
5   y = y + 1;
6 }
7
```

step	line		intervals		affine	thresholds
			x	y		
1	2		$[0, 0]$		$x = 0$	
2	3		$[0, 0]$	$[0, 0]$	$x = 0, y = 0$	
3	4		$[0, 0]$	$[0, 0]$	$x = 0, y = 0$	$x \leq 99$
4	5		$[1, 1]$	$[0, 0]$	$x = 1, y = 0$	$x \leq 100$
5	6		$[1, 1]$	$[1, 1]$	$x = 1, y = 1$	$x \leq 100$
6	3	$\sqcup$	$[0, 1]$	$[0, 1]$	$x = y$	$x \leq 100$
6'	3'	∇	$[0, 100]$	$[0, 100]$	$x = y$	
7	4		$[0, 99]$	$[0, 99]$	$x = y$	
8	5		$[1, 100]$	$[0, 99]$	$x = y + 1$	
9	6		$[1, 100]$	$[1, 100]$	$x = y$	
10	3	$\sqsubseteq$	$[0, 100]$	$[0, 100]$	$x = y$	
11	7		$[100, 100]$	$[100, 100]$		

Figure 1. Rapid convergence during widening.

challenges of widening [1]. Specifically, we propose to implement the inference of widening points and the various widening heuristics as abstract domains that can be plugged into an analysis in a modular way. This modular approach not only reduces the overall complexity of an analysis, it also facilitates the comparison and combination of various heuristics.

The key idea of our approach is to implement abstract domains as cofibered domains [18], an approach sometimes called "functor domains" [5]. Here, each domain $\mathcal{D}$ has a child $\mathcal{C}$ that it controls. The combined domain is written $\mathcal{D} \triangleright \mathcal{C}$. Only the leaf, namely the interval domain $\mathcal{I}$, has no child. The benefit is that a transfer function of domain $\mathcal{D}$ on a state $s \in \mathcal{D} \triangleright \mathcal{C}$ may execute any number of transfer functions on its child $\mathcal{C}$ before returning a new state s'. We illustrate this idea using a cofibered threshold domain $\mathcal{T}$ and a cofibered affine domain $\mathcal{A}$ to build the domain stack $\mathcal{T} \triangleright (\mathcal{A} \triangleright \mathcal{I})$. A state is written as a tuple $\langle t, \langle a, i \rangle\rangle \in \mathcal{T} \triangleright (\mathcal{A} \triangleright \mathcal{I})$ containing the individual domain states $t \in \mathcal{T}$, $a \in \mathcal{A}$ and $i \in \mathcal{I}$.

Figure 1 presents the analysis of a simple loop over $\mathcal{T} \triangleright (\mathcal{A} \triangleright \mathcal{I})$ where the state of each domain is written in a separate column. The states of the interval and affine domain for steps 1 to 6 are straightforward. The threshold domain tracks all conditions in tests that are redundant, here `x<100`, i.e. $x \leq 99$ in step 3. These so-called predicates are changed by assignments, here yielding $x \leq 100$ after `x=x+1;`. In step 6, the state after one loop iteration is joined with the previous state at line 3, yielding the intervals $[0, 1]$ for both, `x` and `y` together with the affine relation $x = y$ and the threshold $x \leq 100$ since it is still redundant in the joined state. The interim step 6' shows how the state obtained at step 2 is widened with respect to the state at step 6: the threshold domain applies widening on its child, yielding $x, y \in [0, \infty]$ for the interval domain while the affine domain returns the join $x = y$ since its lattice is of finite height. The threshold domain then refines this state by applying the test $x \leq 100$. The affine domain passes this test to its child, the interval domain, but also applies the tests $\sigma(x \leq 100)$ for any substitution $\sigma = [x/y]$ that can be derived from equalities over x. This refines the interval domain to $x, y \in [0, 100]$ as shown as step 6'. Steps 7 to 10 ascertain that this state is indeed a fixpoint of the loop, yielding the post-condition shown as step 11.

The example illustrates two consequences of this cofibered arrangement of domains: firstly, it is a modular way of combining several domains, thus keeping each domain simple; secondly, information can be propagated between domains by applying several operations on a child $\mathcal{C}$ for each operation on the parent $\mathcal{D}$.

One might argue that the modular design itself creates the need for propagation which is unnecessary when using a monolithic domain such as an off-the-shelf polyhedra package [2]. However, combining several simple domains allows for a more flexible trade-off between efficiency and precision by adjusting the interaction between domains [16]. For instance, in all polyhedra packages we tested, the widening operation is reduced to a join when the affine relations are not stable, thereby requiring a third fixpoint iteration for this simple example. In our modular setup, the information in the affine domain is not intermingled with information on variable bounds, thereby allowing the affine domain (which has finite height) to compute a join while the interval domain performs widening. This alternative design yields the same precision while requiring fewer iterations.

The implementation of the various widening strategies builds on the ability to separate various concerns into individual domains. These domains are as follows:

Widening Point Domain: Rather than enhancing a fixpoint engine to identify widening points in loops, we propose a domain that turns a join operation into a widening when it observes that the state is propagated along a back-edge of the CFG. This simple technique for irreducible CFGs [6] and CFGs that are constructed on-the-fly [3] works surprisingly well in practice.

Threshold Domain: We implement widening with thresholds [5,13] but infer the thresholds automatically. We present the basic domain that infers thresholds from tests. Unlike previous work [15] that extracts thresholds from a pre-analysis using the domain of polyhedra [9], only relevant tests are tracked.

Delay Domain: A domain which postpones widening is presented that ensures precise results for loops containing assignments of constants.

Phased Domain: We provide an automatic way to separate the state space of loops into several phases, where phase boundaries are automatically inferred from tests within the loop, similar to guided static analysis [11]. This domain can be seen as an instance of a decision tree domain combinator [8].

Besides these specific domains, our paper makes the following contributions:

- Even though cofibered abstract domains only allow to selectively delay widening or to restrict the result of widening by applying tests, they suffice to implement even the most complex widening heuristic in the literature [11].
- Our domains can be added to existing analyses without modification to the fixpoint engine. Our modular approach allows for combining several heuristics and even to retrofit an existing analysis that has no notion of widening.
- We give experimental evidence of the precision of our widening domains.

The remainder of the paper is organized as follows. The next section introduces notation and defines a domain that determines when to widen. Sections 2 to 4 introduce the threshold, delay, and phased domain to improve precision. Section 5 evaluates our domains before Sect. 6 presents related work and concludes.

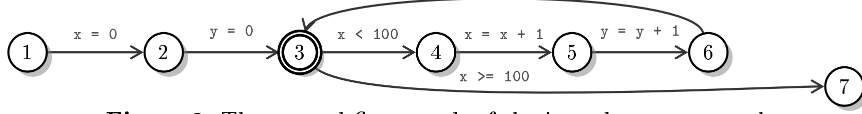

Figure 2. The control flow graph of the introductory example.

1 Preliminaries

This section details the program analysis problem we address. The CFG is represented by a set of vertices labeled $l_1, l_2, \ldots \in \mathit{Lab}$ and directed edges for assignments $l_i \xrightarrow{x=e} l_j$ and tests $l_i \xrightarrow{e\leq 0} l_j$. Given the lattice $\langle \mathcal{D}, \sqsubseteq_{\mathcal{D}}, \sqcup_{\mathcal{D}}, \sqcap_{\mathcal{D}}, \top_{\mathcal{D}}, \bot_{\mathcal{D}} \rangle$ of an abstract domain $\mathcal{D}$, we associate each vertex at l_i with a state $d_i \in \mathcal{D}$ which initially are $d_0 = \top_{\mathcal{D}}$ and $d_i = \bot_{\mathcal{D}}$ for $i \neq 0$. The semantics of an assignment edge $l_i \xrightarrow{x=e} l_j$ in $\mathcal{D}$ is given by $F_i^j = [\![l_i : x = e]\!]^{\mathcal{D}} : \mathcal{D} \to \mathcal{D}$; likewise for test edges. As an example, Fig. 2 shows the CFG of the introductory example of Fig. 1. Here, nodes l_i are labeled with i and an empty edge $l_6 \longrightarrow l_3$ was added so that labels match line numbers. The solution of analysing a program is characterized by a set of equations $s_j \sqsupseteq_{\mathcal{D}} F_i^j(s_i)$, each corresponding to an edge from l_i to l_j. It can be inferred using chaotic iteration [1,6] which picks a location l_i that is not stable and, for all edges $s_i \longrightarrow s_j$ updates s_j to $s_j := s_j \sqcup_{\mathcal{D}} F_i^j(s_i)$.

Inferring numeric information about program variables usually requires the use of abstract domains that have infinite increasing chains such as intervals where $[0,1] \sqsubseteq [0,2] \sqsubseteq [0,3] \ldots$ or convex polyhedra [9]. In these cases, termination of the fixpoint computation is not guaranteed unless at least one widening operator is inserted into each cycle of the graph. The idea of a widening operator is to extrapolate the change in the abstract state between consecutive iterations at a node in the graph. It must obey the following definition [9]:

Definition 1. *Given a domain $\mathcal{D}$, define $\nabla_{\mathcal{D}}^l : \mathcal{D} \times \mathcal{D} \to \mathcal{D}$ such that:*

$$\forall x, y \in \mathcal{D} : x \sqsubseteq_{\mathcal{D}} x \nabla_{\mathcal{D}}^l y$$
$$\forall x, y \in \mathcal{D} : y \sqsubseteq_{\mathcal{D}} x \nabla_{\mathcal{D}}^l y$$

and for all increasing chains $x_0 \sqsubseteq_{\mathcal{D}} x_1 \sqsubseteq_{\mathcal{D}} \ldots$ the increasing chain $y_0 = x_0$, $\ldots y_{i+1} = y_i \nabla_{\mathcal{D}}^l x_{i+1}$ is not strictly increasing.

Consider inserting a widening operator into the equation of the no-op edge from l_6 to l_3, yielding $s_3 := s_3 \nabla_{\mathcal{D}}^l (s_3 \sqcup_{\mathcal{D}} F_6^3(s_6)) = s_3 \nabla_{\mathcal{D}}^l (s_3 \sqcup_{\mathcal{D}} s_6)$. Although termination is now guaranteed, the result of, say, an interval analysis is imprecise: $\{x \in [0,0], y \in [0,0]\} \nabla_{\mathcal{D}}^l \{x \in [0,1], y \in [0,1]\} = \{x \in [0,\infty], y \in [0,\infty]\}$. This stable state can, in principle, be made more precise by replacing the widening with a narrowing operator and re-running the fixpoint computation just for the loop body. However, this requires meddling with the fixpoint engine in order to identify the loop and its in- and outgoing edges and changing the way states are handled: for example our updates $s_j := s_j \sqcup_{\mathcal{D}} F_i^j(s_i)$ are *expansive* ($s_i \sqsubseteq_{\mathcal{D}} s_j \sqcup_{\mathcal{D}} F_i^j(s_i)$) so that the states cannot shrink by evaluating the updates [15]. We

$$
\begin{aligned}
&[\![l : x = e]\!]^{\mathcal{W}} \langle\langle l^w, f^w\rangle, c\rangle && = \langle\langle l, f^w \vee (l < l^w)\rangle, [\![l : x = e]\!]^{\mathcal{C}} c\rangle)\\
&[\![l : e \leq 0]\!]^{\mathcal{W}} \langle\langle l^w, f^w\rangle, c\rangle && = \langle\langle l, f^w \vee (l < l^w)\rangle, [\![l : e \leq 0]\!]^{\mathcal{C}} c\rangle)\\
&\langle w_1, c_1\rangle \sqsubseteq_{\mathcal{W}} \langle w_2, c_2\rangle && = c_1 \sqsubseteq_{\mathcal{C}} c_2\\
&\langle\langle l_1^w, f_1^w\rangle, c_1\rangle \sqcup_{\mathcal{W}} \langle\langle l_2^w, f_2^w\rangle, c_2\rangle && = \begin{cases} \langle\langle l, \mathit{false}\rangle, c_1 \nabla_{\mathcal{C}}^{l}\ c_2\rangle & \text{if } f_1^w \vee f_2^w\\ \langle\langle l, \mathit{false}\rangle, c_1 \sqcup_{\mathcal{C}} c_2\rangle & \text{otherwise}\end{cases}\\
& && \quad \text{where } l = \max(l_1^w, l_2^w)
\end{aligned}
$$

Figure 3. Lattice and transfer functions for the widening point domain.

therefore avoid narrowing altogether to avoid changing the way states are stored. Instead, Sect. 2 and 3 present domains that implement more precise widenings. Before detailing these, we address the task of identifying widening points.

1.1 Inferring Widening Points

For programs made up of well-nested loops, widening is only required at each loop head in the program [5], which renders fixpoint computations relatively straightforward. For programs with irreducible CFGs, it is generally necessary to place more than one widening point in each cycle [6] and, hence, a widening heuristic must not lose precision when widening is applied several times within a loop. This, in turn, implies that a conservative heuristic, that places rather many widening points, suffices. We now present such a heuristic that is appropriate for machine code, implemented as abstract domain $\mathcal{W}$. The domain observes back-edges, that is, information flowing from larger to smaller addresses. Once observed, the next join on $\mathcal{W} \triangleright \mathcal{C}$ translates to a widening on the child $\mathcal{C}$.

For the sake of finding back-edges, we assume that statement labels $l \in \mathit{Lab}$ represent the code address of a statement or test. The widening point domain is given by the lattice $\langle \mathcal{W} \triangleright \mathcal{C}, \sqsubseteq_{\mathcal{W}}, \sqcup_{\mathcal{W}}, \sqcap_{\mathcal{W}}, \top_{\mathcal{W}}, \bot_{\mathcal{W}}\rangle$ where $\mathcal{W} : \mathit{Lab} \times \{\mathit{true}, \mathit{false}\}$ is a tuple of the last program point and a flag indicating if a backward edge has been observed. If set, widening is applied at the next junction node at which point the loop is usually completely traversed. Figure 3 defines the domain operations. The transfer functions for assignment and the lattice functions for subset and join are shown. Each function operates on tuples $\langle w, c\rangle \in \mathcal{W} \triangleright \mathcal{C}$ where $w \equiv \langle l^w, f^w\rangle \in \mathcal{W}$. The transfer functions $[\![\cdot]\!]^{\mathcal{W}}$ on $\mathcal{W}$ apply the corresponding operation $[\![\cdot]\!]^{\mathcal{C}}$ on the child $c \in \mathcal{C}$ while tracking the current label l and whether a backward edge $l^w \to l$ with $l < l^w$ has been observed. The subset test $\sqsubseteq_{\mathcal{W}}$ translates to a subset test on the child, indicating that the $\mathcal{W}$ domain does not actually infer any information about the state of the program and is therefore, per definition, stable. The only effect of the domain is that the $\sqcup_{\mathcal{W}}$ translates to a widening operation on the child if one of the flags is true. Note that this domain may be more precise than a standard algorithm for determining widening points [6] since widening is applied only after a back-edge. For instance, in Fig. 2, widening is only applied when updating node l_3 with a state from l_6 but no widening is applied when propagating the state from l_2 to l_3, as this path is not a back-edge.

The ability to add widening to an analysis without changing the fixpoint can also be carried over to various widening heuristics, as detailed next.

$$
\begin{aligned}
[\![l : x = e]\!]^{\mathcal{T}} \langle t, c\rangle &= \langle [p \mapsto \langle \bar{l}_o, \bar{l}_w\rangle \in t \mid x \notin \mathit{vars}(p)] \cup [\sigma^{-1}(p) \mapsto \langle \bar{l}_o, \bar{l}_w\rangle \mid p \mapsto \langle \bar{l}_o, \bar{l}_w\rangle \in t \\
&\qquad \wedge \sigma = [x/e], \sigma^{-1}(p) \text{ exists}], [\![l : x = e]\!]^{\mathcal{C}} c\rangle \\
[\![l : e \leq 0]\!]^{\mathcal{T}} \langle t, c\rangle &= \langle \mathit{filter}(t'[e \leq 0 \mapsto \langle \{l\}, \emptyset\rangle \mid \exists p \,.\, t(p) = \langle \bar{l}_o, \emptyset\rangle \wedge l \in \bar{l}_o], c), [\![l : e \leq 0]\!]^{\mathcal{C}} c\rangle \\
&\qquad \text{where } t' = [p \mapsto \langle \bar{l}_o, \bar{l}_w\rangle \in t \mid l \notin \bar{l}_o \vee \bar{l}_w \neq \emptyset] \\
\langle t_1, c_1\rangle \sqsubseteq_{\mathcal{T}} \langle t_2, c_2\rangle &= c_1 \sqsubseteq_{\mathcal{C}} c_2 \\
\langle t_1, c_1\rangle \sqcup_{\mathcal{T}} \langle t_2, c_2\rangle &= \langle \mathit{filter}(t, c), c\rangle \text{ where } t = \mathit{join}(t_1, t_2) \text{ and } c = c_1 \sqcup_{\mathcal{C}} c_2 \\
\langle t_1, c_1\rangle \nabla^l_{\mathcal{T}} \langle t_2, c_2\rangle &= \langle \mathit{filter}(t[p_i \mapsto \langle \bar{l}_o, \bar{l}_w \cup \{l\}\rangle \mid \langle \bar{l}_o, \bar{l}_w\rangle \in t \wedge \bar{l}_o \cap \bar{l}_{upd} \neq \emptyset], c), c\rangle \text{ where} \\
&\qquad t = \mathit{join}(\mathit{filter}(t_1, c_2), \mathit{filter}(t_2, c_2)) \wedge c = [\![l : p_1]\!]^{\mathcal{C}} \ldots [\![l : p_n]\!]^{\mathcal{C}} (c_1 \nabla^l_{\mathcal{C}} c_2) \\
&\qquad p_i \in \{e_1 \leq 0, \ldots, e_n \leq 0\} = \{p \in \mathit{dom}(t) \mid t(p) = \langle \bar{l}_o, \bar{l}_w\rangle \wedge l \notin \bar{l}_w\} \text{ and} \\
&\qquad \bar{l}_{upd} = \textstyle\bigcup_{i=1}^{n} \{\bar{l}_o \mid t(p_i) = \langle \bar{l}_o, \bar{l}_w\rangle \wedge \max(e_i, c) = \min_{i=1}^{n}(\max(e_i, c))\}
\end{aligned}
$$

Figure 4. Transfer and lattice functions for the threshold domain.

2 Widening with Thresholds as Abstract Domain

Widening is necessary to ensure termination when a fixpoint is computed over a domain of infinite height. One problem of widening is that the obtained fixpoint is almost always a post-fixpoint, that is, it is larger than the least fixpoint. This section shows how predicates occurring in tests can be used as *thresholds* to restrict the widened state, thereby often giving better results than a narrowing can provide. Let *Pred* be a set of predicates that are used as conditions in tests. We only require that the negation $\neg p$ of $p \in \mathit{Pred}$ exists and that $\neg p \in \mathit{Pred}$ where $\neg(\neg p) \equiv p$. In practice, we gather all tests convertible to linear inequalities and assume integer arithmetic: $\neg(a_1 x_1 + \ldots + a_n x_n \leq c) \equiv a_1 x_1 + \ldots + a_n x_n \geq c + 1$.

The threshold domain is given by the lattice $\langle \mathcal{T} \rhd \mathcal{C}, \sqsubseteq_{\mathcal{T}}, \sqcup_{\mathcal{T}}, \sqcap_{\mathcal{T}}\rangle$ where the universe $\mathcal{T} : \mathit{Pred} \dashrightarrow \wp(\mathit{Lab}) \times \wp(\mathit{Lab})$ is a partial map from redundant tests $p \in \mathit{Pred}$ to two sets of program points. The first set $\bar{l}_o$ contains the program points of the test where p originated. The second set $\bar{l}_w$ denotes the widening points at which p has been used as thresholds. We update $t \in \mathcal{T}$ to $t' = t[p \mapsto l] \in \mathcal{T}$ with $t'(p) = l$ and $t'(q) = t(q)$ for $q \neq p$. In abuse of notation we use $[p \mapsto \ldots]$ to construct a new mapping and $\emptyset$ for the empty map. We enforce the invariant that all tests $p \in \mathit{Pred}$ are redundant in the child domain by applying $\mathit{filter} : \mathcal{T} \times \mathcal{C} \to \mathcal{T}$ which is defined as $\mathit{filter}(t, c) = [p \mapsto t(p) \mid p \in \mathit{dom}(t) \wedge [\![\neg p]\!]^{\mathcal{C}} c = \bot_{\mathcal{C}}]$ where $[\![\neg p]\!]^{\mathcal{C}} c \in \mathcal{C}$ computes a state of the child domain in which the test $\neg p$ has been applied. Note that instead of $[\![p]\!]^{\mathcal{C}} c = c$ we use the cheaper test $[\![\neg p]\!]^{\mathcal{C}} c = \bot_{\mathcal{C}}$.

Figure 4 presents the transfer functions and lattice operations of the threshold domain. An assignment $x = e$ at program point $l \in \mathit{Lab}$ is forwarded to the child. All thresholds that are not affected by the write to x are kept as is (first line) while predicates p that mention x are kept if an inverted substitution σ^{-1} exists where $\sigma = [x/e]$. For instance, consider the assignment `x=x+1;` in Fig. 1 where $t = [x \leq 99 \mapsto \langle \{3\}, \emptyset\rangle] \in \mathcal{T}$ and intervals $\mathcal{C} = \mathcal{I}$ with $x \in [0, 1]$. With $\sigma = [x/x + 1]$, we obtain $\sigma^{-1} = [x/x - 1]$ and $\sigma^{-1}(x \leq 99) = x \leq 100$. Thus, the state after the assignment is $\langle [x \leq 100 \mapsto \langle \{3\}, \emptyset\rangle], x \in [1, 2]\rangle$. Note that the resulting threshold is again 98 units away from the current state space. Indeed, applying a linear substitution ensures that each threshold remains redundant. If $x \notin \mathit{vars}(e)$ or if e is not linear, σ^{-1} does not exist and the threshold is removed.

```
int n = 0;
while (true) {
  if (!read_sec())
    continue;
  if (n<60) {
    n = n+1;
  } else {
    n = 0;
  }
}
```

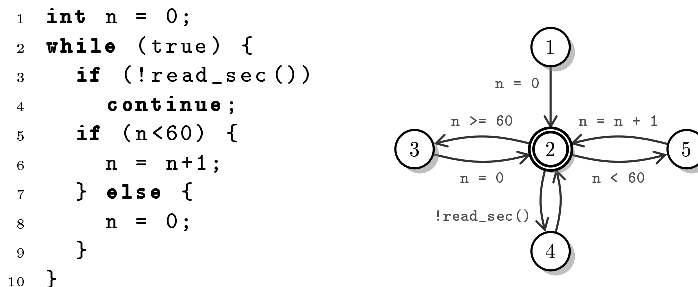

Figure 5. A loop whose fixpoint cannot be obtained by narrowing.

The transfer function for tests replaces tests that originate here ($l \in \bar{l}_o$) and which have not been applied yet ($\bar{l}_w = \emptyset$) with a fresh threshold $e \leq 0$. Here, t' contains the remaining mappings. Tests that happen to actually restrict the incoming state space c are removed by *filter*.

With respect to the lattice operation, the entailment test $\langle t_1, c_1\rangle \sqsubseteq_{\mathcal{T}} \langle t_2, c_2\rangle$ reduces to an entailment test on the child. The join $\langle t_1, c_1\rangle \sqcup_{\mathcal{T}} \langle t_2, c_2\rangle$ uses a function *join* that merges the program points where tests originate and where they are applied point-wise as follows:

$$join(t_1, t_2) = \left[p \mapsto \langle \bar{l}_1^o \cup \bar{l}_2^o, \bar{l}_1^w \cup \bar{l}_2^w \rangle \mid \langle \bar{l}_i^o, \bar{l}_i^w \rangle = \begin{cases} t_i(p) & \text{if } p \in dom(t_i) \\ \langle \emptyset, \emptyset \rangle & \text{otherwise} \end{cases} \right]_{\substack{p \in dom(t_1) \\ \cup dom(t_2)}}$$

Again, applying *filter* removes thresholds that are not redundant in $c_1 \sqcup_{\mathcal{C}} c_2$. Given the collected thresholds, the widening $\langle t_1, c_1\rangle \nabla_{\mathcal{T}}^{l} \langle t_2, c_2\rangle$ is now able to refine the widened child state $c_1 \nabla_{\mathcal{C}}^{l} c_2$ by applying those predicates $e_1 \leq 0$, $\ldots e_n \leq 0$ as tests that have not yet been used at this widening point, that is, for which $l \notin \bar{l}_w$ holds. For each such predicate, we check if $e_i \leq 0$ has actually contributed to restricting $c_1 \nabla_{\mathcal{C}}^{l} c_2$ by checking if the distance $max(e_i, c)$ to the state c is the smallest of all predicates. The set $\bar{l}_{upd}$ is defined to contain the locations of all contributing tests. The idea is that, from two tests $l_1 : x \leq 10, l_2 : x \leq 50$, only l_1 is marked as being applied, thereby allowing the test $x \leq 50$ to serve as a threshold in future widenings. The merged domain state t is then updated so that all tests at locations $l \in \bar{l}_{upd}$ are marked as applied at l. Overall, widening is delayed at most $|\bar{l}_o|$ times at each of the $|\bar{l}_w|$ widening points. Since there are only a finite number of program locations, termination follows.

Widening with thresholds can find least fixpoints where narrowing cannot [12]. Consider the program in Fig. 5 that tracks the seconds within a minute. The loop repeatedly waits for a seconds signal that causes `read_sec` to return `1`. The simplified CFG of the program contains three loops. After propagating $n = 0$ to node 2, the loops through node 3 and 4 are stable. The loop via node 5 yields $n \in [0, 1]$ in node 2 which is widened to $n \in [0, \infty]$. The threshold $n \leq 59$ is transformed by `n=n+1` to $n \leq 60$ and is applied after widening, yielding $n \in [0, 60]$. Narrowing cannot deduce this fixpoint due the cycle via node 4 [12].

$$
\begin{aligned}
\llbracket l : x = e \rrbracket^{\mathcal{D}} \langle d, c \rangle \quad &= \langle d \cup \bar{l}, \llbracket l : x = e \rrbracket^{\mathcal{C}} c \rangle \text{ where } \bar{l} = \begin{cases} \{l\} & \text{if } e \in \mathbb{Z} \\ \emptyset & \text{otherwise} \end{cases} \\
\langle d_1, c_1 \rangle \sqsubseteq_{\mathcal{D}} \langle d_2, c_2 \rangle &= c_1 \sqsubseteq_{\mathcal{C}} c_2 \\
\langle d_1, c_1 \rangle \sqcup_{\mathcal{D}} \langle d_2, c_2 \rangle &= \langle d_1 \cup d_2, c_1 \sqcup_{\mathcal{C}} c_2 \rangle \\
\langle d_1, c_1 \rangle \nabla^l_{\mathcal{D}} \langle d_2, c_2 \rangle &= \langle d_1 \cup d_2, c \rangle \text{ where } c = \begin{cases} c_1 \nabla^l_{\mathcal{C}} c_2 & \text{if } d_2 \backslash d_1 = \emptyset \\ c_1 \sqcup_{\mathcal{C}} c_2 & \text{otherwise} \end{cases}
\end{aligned}
$$

Figure 6. Lattice and transfer functions for the delaying domain.

```
1 int x = 0;
2 int y = 0;
3 while (x < 100) {
4  if (read()) y = 1;
5  x = x + 4;
6 }
7
```

step	line		intervals		congruences	$\mathcal{T}$	·	$\mathcal{D}$
			x	y			·	
1	2		$[0,0]$				·	$\{1\}$
2	3		$[0,0]$	$[0,0]$			·	$\{1,2\}$
3	4		$[0,0]$	$[0,0]$		$x \leq 99$	·	$\{1,2\}$
4	5		$[0,0]$	$[1,1]$		$x \leq 99$	·	$\{1,2,4\}$
5	6		$[4,4]$	$[1,1]$		$x \leq 103$	·	$\{1,2,4\}$
6	3	$\sqcup$	$[0,4]$	$[0,1]$	$x \equiv 4$	$x \leq 103$	·	$\{1,2,4\}$
6'	3'	∇	$[\mathbf{0},\mathbf{100}]$	$[\mathbf{0},\infty]$	$x \equiv 4$		·	$\{1,2,4\}$
7	4		$[0,96]$	$[0,\infty]$	$x \equiv 4$		·	$\{1,2,4\}$
8	5		$[0,96]$	$[0,\infty]$	$x \equiv 4$		·	$\{1,2,4\}$
9	6		$[4,100]$	$[0,\infty]$	$x \equiv 4$		·	$\{1,2,4\}$
10	3	$\sqsubseteq$	$[0,100]$	$[0,\infty]$	$x \equiv 4$		·	$\{1,2,4\}$
11	7		$[100,100]$	$[0,\infty]$			·	$\{1,2,4\}$

Figure 7. Widening after one iteration loses the bound on `y`.

3 Restricting Widening after Constant Assignments

It is widely acknowledged that computing a few iterations of a loop without widening can improve the precision of the computed fixpoint [2]. For instance, the program in Fig. 7 may set the variable `y` to 1 depending on some external event where `read()` may return the value of some sensor in a control software [5,8]. Given the threshold domain $\mathcal{T}$ as-is, the table in Fig. 7 shows how widening the state at step 6' with respect to that at step 2 yields $x \in [0,0] \nabla^l_{\mathcal{I}} [0,4] = [0,\infty]$ and $y \in [0,0] \nabla^l_{\mathcal{I}} [0,1] = [0,\infty]$ where the former interval can be refined by the threshold $x \leq 103$ to $[0,100]$ since $x \equiv 4$, i.e. x is a multiple of four. The loop test `x<100` then yields the precise value for x in step 7. However, the upper bound for y is lost. The common approach to improve the precision is to delay widening [13], that is, to compute another iteration of the loop using the state at step 6. After the second iteration, $y \in [0,1] \nabla^l_{\mathcal{I}} [0,1] = [0,1]$ is as desired.

Rather than fixing the number of times widening should be delayed, we track if widening would alter variables that were set to a constant. To this end, we define a delaying domain given by the lattice $\langle \mathcal{D} \triangleright \mathcal{C}, \sqsubseteq_{\mathcal{D}}, \sqcup_{\mathcal{D}}, \sqcap_{\mathcal{D}} \rangle$ where $\mathcal{D} : \wp(Lab)$ is a set of program points with constant assignments. The transfer functions in Fig. 6 simply collect those program points that assign a constant to a variable. Performing widening on $\mathcal{D}$ will check if this set has increased and, if so, perform a join instead of a widening. For example, in step 6' of Fig. 7, location 4 is new relative to the state at step 2, thereby performing another iteration based on the state at step 6. Note that the delaying domain also delays widening if a new **`if`**-branch becomes enabled that contains a constant assignment. Note further that `i=i+1` is never considered constant so as not to delay widening unnecessarily.

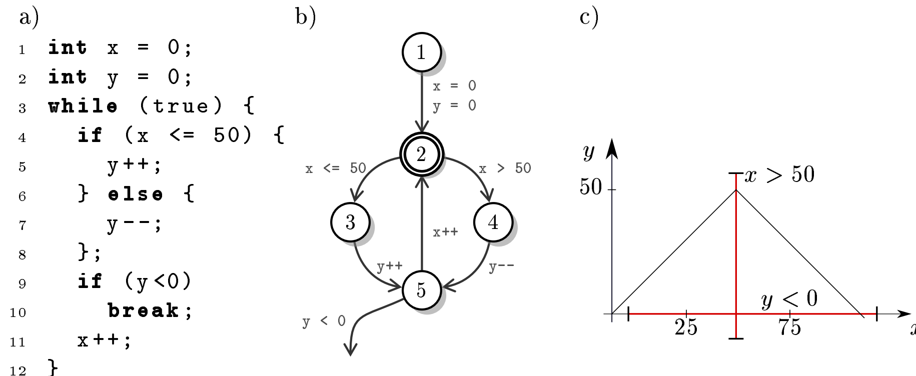

Figure 8. A loop containing phase transitions.

4 Guided Static Analysis as Abstract Domain

Numeric domains are usually convex approximations of the possible set of numeric values. One drawback of convexity is that joining two states can incur a precision loss that cannot later be recovered from. For example, the join of two intervals $[0,5] \sqcup_{\mathcal{I}} [15,20] = [0,20]$ adds the spurious values $6, \ldots 14$ and applying $x \leq 10$ to this state is less precise than applying it to the individual intervals. The idea of guided static analysis [11] is to avoid this kind of precision loss by identifying different phases of a loop and to track a separate state for each phase. The original proposal is formulated in terms of operations that restrict the CFG to increasingly larger sub-graphs and to perform widening/narrowing on these sub-graphs. In this section, we show that the same effect can be obtained by adding a cofibered *phase domain* into the domain hierarchy, thereby avoiding any modification to the fixpoint engine or to the handling of states.

Consider the loop in Fig. 8a) that increments `x`, starting from zero. For the first fifty iterations, `y` is incremented while in the next fifty iterations `y` is decremented. The loop exits in the 102th iteration when `y` becomes negative. The state space is depicted in Fig 8c) where the two hyperplanes annotated with the predicates $p_x \equiv x > 50$ and $p_y \equiv y < 0$ mark the different phase transitions. In particular, observe that the three phases can be characterized by the predicates that hold: for the first phase $\neg p_x \wedge \neg p_y$ holds, for the second phase $p_x \wedge \neg p_y$ and for the third phase $p_x \wedge p_y$. Thus, rather than characterizing the loop phases by enabled sub-graphs of the CFG, we construct an abstract domain that tracks a different child domain for each feasible valuation of the predicates. In a child c that is tracked for the predicates $p_1, \ldots p_n$, we assume that each predicate p_i holds and, lest the domain is imprecise, $[\![\neg p_i]\!]^{\mathcal{C}} c = \bot_{\mathcal{C}}$ for all $i \in [1,n]$. Thus, in the example, the predicates $\neg p_x \wedge \neg p_y$ hold in the state of the first phase c_1 and propagating c_1 over the edge from CFG node 2 to 4 in Fig. 8b) yields an empty state, thereby simulating the fact that this sub-path of the CFG is disabled. Analogous, a state c_2 in which $p_x \wedge \neg p_y$ holds has the path $2 \to 3 \to 5$ disabled since it is guarded by $p_x \equiv x > 50$.

step	line		intervals x	intervals y	affine	phased $c; p_1 : t_1, \ldots p_n : t_n; \bar{p}$	threshold
1	2		$[0,0]$		$x = 0$	$c_1; \emptyset$	
2	3		$[0,0]$	$[0,0]$	$x = 0, y = 0$	$c_1; \emptyset$	
3	5		$[0,0]$	$[0,0]$	$x = 0, y = 0$	$c_1; \{x > 50\}$	$x \leq 50$
4	6		$[0,0]$	$[1,1]$	$x = 0, y = 1$	$c_1; \{x > 50\}$	$x \leq 50$
5	11		$[0,0]$	$[1,1]$	$x = 0, y = 1$	$c_1; \{x > 50, y < 0\}$	$x \leq 50$
6	12		$[1,1]$	$[1,1]$	$x = 1, y = 1$	$c_1; \{x > 50, y < 0\}$	$x \leq 51, y \geq 0$
7	3	$\sqcup$	$[0,1]$	$[0,1]$	$x = y$	$c_1; \{x > 50, y < 0\}$	$x \leq 51, y \geq 0$
8	3'	∇	$[0,50]$	$[0,50]$	$x = y$	$c_1; x > 50 : t_2; \{y < 0\}$	$y \geq 0$
	3'		$[51,51]$	$[51,51]$	$x = 51, y = 51$	$c_2; \{y < 0\}$	
9	5		$[0,50]$	$[0,50]$	$x = y$	$c_1; x > 50 : t_2; \{y < 0\}$	$y \geq 0$
	5					$\bot_{\mathcal{A} \triangleright \mathcal{I}}; \{y < 0\}$	
10	6		$[0,50]$	$[1,51]$	$x = y - 1$	$c_1; x > 50 : t_2; \{y < 0\}$	$y \geq 1$
	6					$\bot_{\mathcal{A} \triangleright \mathcal{I}}; \{y < 0\}$	
11	7					$\bot_{\mathcal{A} \triangleright \mathcal{I}}; x > 50 : t_2; \{y < 0\}$	$y \geq 0$
	7		$[51,51]$	$[51,51]$	$x = 51, y = 51$	$c_2; \{y < 0\}$	
12	8					$\bot_{\mathcal{A} \triangleright \mathcal{I}}; x > 50 : t_2; \{y < 0\}$	$y \geq -1$
	8		$[51,51]$	$[50,50]$	$x = 51, y = 50$	$c_2; \{y < 0\}$	
13	9		$[0,50]$	$[1,51]$	$x = y - 1$	$c_1; x > 50 : t_2; \{y < 0\}$	$y \geq -1, y \geq 1$
	9		$[51,51]$	$[50,50]$	$x = 51, y = 50$	$c_2; \{y < 0\}$	
14	12		$[1,50]$	$[1,50]$	$x = y$	$c_1; x > 50 : t_2; \{y < 0\}$	$y \geq 0$
	12		$[51,52]$	$[50,51]$	$x + y = 102$	$c_2; \{y < 0\}$	
15	3'	$\sqsubseteq$	$[1,50]$	$[1,50]$	$x = y$	$c_1; x > 50 : t_2; \{y < 0\}$	
	3'	∇	$[51,102]$	$[0,51]$	$x + y = 102$	$c_2; \{y < 0\}$	
16	5	$\sqsubseteq$	$[0,50]$	$[0,50]$	$x = y$	$c_1; x > 50 : t_2; \{y < 0\}$	
	5					$\bot_{\mathcal{A} \triangleright \mathcal{I}}; \{y < 0\}$	
17	6	$\sqsubseteq$	$[0,50]$	$[1,51]$	$x = y - 1$	$c_1; x > 50 : t_2; \{y < 0\}$	
	6					$\bot_{\mathcal{A} \triangleright \mathcal{I}}; \{y < 0\}$	
18	7					$\bot_{\mathcal{A} \triangleright \mathcal{I}}; x > 50 : t_2; \{y < 0\}$	
	7		$[51,102]$	$[0,51]$	$x + y = 102$	$c_2; \{y < 0\}$	
19	8					$\bot_{\mathcal{A} \triangleright \mathcal{I}}; x > 50 : t_2; \{y < 0\}$	
	8		$[51,101]$	$[0,50]$	$x + y = 101$	$c_2; y < 0 : t_3; \emptyset$	
	8		$[102,102]$	$[-1,-1]$	$x = 102, y = -1$	$c_3; \emptyset$	
20	9		$[0,50]$	$[1,51]$	$x = y - 1$	$c_1; x > 50 : t_2; \{y < 0\}$	
	9		$[51,100]$	$[0,50]$	$x + y = 101$	$c_2; y < 0 : t_3; \emptyset$	
	9		$[102,102]$	$[-1,-1]$	$x = 102, y = -1$	$c_3; \emptyset$	
21	10		$[102,102]$	$[-1,-1]$	$x = 102, y = -1$	$c_3; \emptyset$	
22	12		$[1,50]$	$[1,50]$	$x = y$	$c_1; x > 50 : t_2; \{y < 0\}$	
	12		$[51,102]$	$[0,51]$	$x + y = 102$	$c_2; y < 0 : \bot_{\mathcal{A} \triangleright \mathcal{I}}; \emptyset$	
23	3	$\sqsubseteq$	$[1,50]$	$[1,50]$	$x = y$	$c_1; x > 50 : t_2; \{y < 0\}$	
	3	$\sqsubseteq$	$[51,102]$	$[0,51]$	$x + y = 102$	$c_2; y < 0 : \bot_{\mathcal{A} \triangleright \mathcal{I}}; \emptyset$	

Figure 9. Computing the fixpoint for the example in Fig. 8.

We implement the ideas of tracking several children depending on which predicates hold in the cofibered *phase domain* that is given by the lattice $\langle \mathcal{P} \triangleright \mathcal{C}, \sqsubseteq_{\mathcal{P}}, \sqcup_{\mathcal{P}}, \sqcap_{\mathcal{P}}, \top_{\mathcal{P}}, \bot_{\mathcal{P}} \rangle$ where $\mathcal{P} : \mathcal{C} \times (\mathit{Pred} \times \mathcal{P})^* \times \wp(\mathit{Pred})$ is a recursive type, representing a multi-way decision tree. A node in this tree $\langle c; p_1 : t_1; \ldots p_n : t_n; \bar{p} \rangle \in \mathcal{P}$ contains a child domain c in which predicates $p_1, \ldots, p_n \in \mathit{Pred}$ do not hold. The node has n sub-trees $t_1, \ldots t_n \in \mathcal{P}$ where p_i holds in t_i. The set $\bar{p} \subseteq \mathit{Pred}$ is a set of predicates that are unsatisfiable and represent phases that have not (yet) been entered. Before we detail the transfer and lattice functions, we consider the fixpoint computation in Fig. 9 using a domain stack $\mathcal{T} \triangleright \mathcal{P} \triangleright \mathcal{A} \triangleright \mathcal{I}$, that is, thresholds wrapping the phase domain, that wraps affine and intervals.

Initially, the phase domain contains a single child domain c_1 and no sub-trees as shown in step 1 of Fig. 9. The idea of the phased domain is to gather all unsatisfiable tests as possible phase predicates, adding them to the set $\bar{p}$. Thus, step 3 adds the predicate $x > 50$ and step 5 adds $y < 0$. Note that, unlike the threshold

$$
\begin{array}{l}
[\![l : x = e]\!]^{\mathcal{T}} \langle c; p_1 : t_1; \ldots p_n : t_n; \bar{p} \rangle = \\
\quad \text{let } c_e = [\![l : x = e]\!]^{\mathcal{C}} c \text{ and } c' = [\![l : \neg p_i]\!]^{\mathcal{C}} \cdots [\![l : \neg p_n]\!]^{\mathcal{C}} c_e \text{ and } c_i = [\![l : p_i]\!]^{\mathcal{C}} c_e \\
\quad \text{and } \langle \tilde{c}_e^i; \mathit{part}^i \rangle = [\![l : x = e]\!]^{\mathcal{T}} t_i \\
\quad \text{and } \tilde{c}_{res}^i = c_i \sqcup_{\mathcal{C}} [\![l : p_i]\!]^{\mathcal{C}} \tilde{c}_i \text{ and } c'' = c' \sqcup_{\mathcal{C}} [\![l : \neg p_1]\!]^{\mathcal{C}} \tilde{c}_e^1 \sqcup_{\mathcal{C}} \cdots \sqcup_{\mathcal{C}} [\![l : \neg p_n]\!]^{\mathcal{C}} \tilde{c}_e^n \\
\quad \text{and } \bar{p}^{red} = \{p \in \bar{p} \mid [\![l : \neg p]\!]^{\mathcal{C}} c'' = \bot_{\mathcal{C}}\} \text{ and } \langle p_{n+1}, \ldots p_{n+k}\} = \bar{p} \setminus \bar{p}^{red} \\
\quad \text{and } c_{res} = [\![l : \neg p_{n+1}]\!]^{\mathcal{C}} \cdots [\![l : \neg p_{n+k}]\!]^{\mathcal{C}} c'' \text{and } c_{new}^{n+j} = [\![l : p_{n+j}]\!]^{\mathcal{C}} c'' \text{ for } j = 1 \ldots k \\
\quad \text{in } \langle c_{res}; p_1 : \langle \tilde{c}_{res}^1; \mathit{part}^1 \rangle; \ldots p_n : \langle \tilde{c}_{res}^n; \mathit{part}^n \rangle; p_{n+1} : \langle c_{new}^{n+1} \rangle; \ldots p_{n+k} : \langle c_{new}^{n+k} \rangle; \bar{p}^{red} \rangle \\
[\![l : e \leq 0]\!]^{\mathcal{T}} \langle c; p_1 : t_1; \ldots p_n : t_n; \bar{p} \rangle = \\
\quad \text{let } \langle \tilde{c}_e^i : \mathit{part}^i \rangle = [\![l : e \leq 0]\!]^{\mathcal{T}} t_i \text{ and } c_e = [\![l : e \leq 0]\!]^{\mathcal{C}} c \\
\quad \text{in if } \bigwedge_{i=1}^n \tilde{c}_e^i = \bot_{\mathcal{T}} \wedge c_e = \bot_{\mathcal{C}} \text{ then } \bot_{\mathcal{T}} \text{ else} \\
\quad \langle c_e; p_1 : \langle \tilde{c}_e^1; \mathit{part}^1 \rangle; \ldots p_n : \langle \tilde{c}_e^n; \mathit{part}^n \rangle; \text{if } [\![l : e > 0]\!]^{\mathcal{C}} c = \bot_{\mathcal{C}} \text{ then } \bar{p} \cup \{e \leq 0\} \text{ else } \bar{p} \rangle \\
\langle c_1; \mathit{part}_1 \rangle \sqsubseteq_{\mathcal{T}} \langle c_2; \mathit{part}_2 \rangle = \\
\quad \text{let } \langle \langle p_1^1 : t_1^1; \ldots p_n^1 : t_n^1; \bar{p}^1 \rangle, \langle p_1^2 : t_1^2; \ldots p_n^2 : t_n^2; \bar{p}^2 \rangle \rangle = \mathit{compatible}(\mathit{part}_1, \mathit{part}_2) \\
\quad \text{in } c_1 \sqsubseteq_{\mathcal{C}} c_2 \wedge \bigwedge_{i=1}^n t_i^1 \sqsubseteq_{\mathcal{T}} t_i^2 \\
\langle c_1; \mathit{part}_1 \rangle \sqcup_{\mathcal{T}} \langle c_2; \mathit{part}_2 \rangle = \\
\quad \text{let } \langle \langle p_1^1 : t_1^1; \ldots p_n^1 : t_n^1; \bar{p}^1 \rangle, \langle p_1^2 : t_1^2; \ldots p_n^2 : t_n^2; \bar{p}^2 \rangle \rangle = \mathit{compatible}(\mathit{part}_1, \mathit{part}_2) \\
\quad \text{in } \langle c_1 \sqcup_{\mathcal{C}} c_2; p_1^1 : t_1^1 \sqcup_{\mathcal{T}} t_1^2; \ldots p_n^1 : t_n^1 \sqcup_{\mathcal{T}} t_n^2; \bar{p}^1 \rangle \\
\langle c_1; \mathit{part}_1 \rangle \nabla_{\mathcal{T}}^l \langle c_2; \mathit{part}_2 \rangle = \\
\quad \text{let } \langle \langle p_1^1 : t_1^1; \ldots p_n^1 : t_n^1; \bar{p}^1 \rangle, \langle p_1^2 : t_1^2; \ldots p_n^2 : t_n^2; \bar{p}^2 \rangle \rangle = \mathit{compatible}(\mathit{part}_1, \mathit{part}_2) \\
\quad \text{and } c_e = [\![l : \neg p_1^1]\!]^{\mathcal{C}} \cdots [\![l : \neg p_n^1]\!]^{\mathcal{C}} c_1 \nabla_{\mathcal{T}}^l c_2 \text{ and } \tilde{c}_e^i; \mathit{part}^i \rangle = \langle [\![l : p_i^1]\!]^{\mathcal{T}} (t_i^1 \nabla_{\mathcal{T}}^l t_i^2); \\
\quad \text{and } \bar{p}^{red} = \{p \in \bar{p} \mid [\![l : \neg p]\!]^{\mathcal{C}} c'' = \bot_{\mathcal{C}}\} \text{ and } \langle p_{n+1}, \ldots p_{n+k}\} = \bar{p} \setminus \bar{p}^{red} \\
\quad \text{and } c_{res} = [\![l : \neg p_{n+1}]\!]^{\mathcal{C}} \cdots [\![l : \neg p_{n+k}]\!]^{\mathcal{C}} c_e \text{and } c_{new}^{n+j} = [\![l : p_{n+j}]\!]^{\mathcal{C}} c_e \text{ for } j = 1 \ldots k \\
\quad \text{in } \langle c_{res}; p_1^1 : \langle \tilde{c}_e^1; \mathit{part}^1 \rangle \ldots p_n^1 : \langle \tilde{c}_e^n; \mathit{part}^n \rangle; p_{n+1} : \langle c_{new}^{n+1} \rangle; \ldots p_{n+k} : \langle c_{new}^{n+k} \rangle; \bar{p}^{red} \rangle
\end{array}
$$

Figure 10. Transfer and lattice functions for the phase domain.

predicates, the phase predicates are not transformed. Once widening is applied in step 8, the subtree $t_2 = c_2; \{y < 0\}$ is added. This new subtree is immediately disabled in step 9 and 10 due to the test `x <= 50`. Analogously, only the subtree t_2 is enabled in steps 11 and 12. Both states are joined in step 13. Incrementing `x` to obtain step 14 poses the challenge that x in c_1 straddles the phase bound $x > 50$. Thus, the state $c_e = [\![\texttt{x++}]\!]^{\mathcal{A} \rhd \mathcal{I}} c_1$ is split into $c_1' = [\![x \leq 50]\!]^{\mathcal{A} \rhd \mathcal{I}} c_e$ and $\tilde{c} = [\![x > 50]\!]^{\mathcal{A} \rhd \mathcal{I}} c_e = \langle x = 51, y = 51 \rangle$. The latter is joined with the updated state of the subtree $[\![\texttt{x++}]\!]^{\mathcal{A} \rhd \mathcal{I}} c_2 = \langle x = 52, y = 50 \rangle$ yielding the downward slope $x + y = 102$ in the second line of step 14. Widening is applied again, thereby consuming the last threshold $y \geq 0$. The same state is propagated in steps 16 and 17 whereas the **else**-branch sees a larger state. Indeed, decrementing `y` in c_2 surpasses the phase threshold $y < 0$, thereby creating a third subtree $t_3 = c_3; \emptyset$ in step 19. Step 20 computes the joined state from which the state at loop exit is split off (step 21). Step 22 increments `x` which again propagates the point $\langle x = 51, y = 51 \rangle$ from c_1 to c_2 as for step 14. A fixpoint is observed in step 23.

The domain operations are formally defined in Fig. 10. We allow for several subtrees per node to cater for sequences of **if**-statements. The assignment $l : x = e$ first computes the effect on the state in the current node c, yielding c_e, and its subtrees t_i, yielding $\tilde{c}_e^i$. The state space that spills over the phase predicates $p_1, \ldots p_n$ is cut off and merged into the respective parent or subtree.

	our analysis						interproc		conc.
example	time	insns.	#wp	steps	iter.	exact	iter.	exact	exact
simple loop Fig. 1	7	14	1	23	2	✓	3+1	✓	✓
nested loops random	7	20	2	42	3	✓	5+2	✓	✓
nested loops random (mod)	7	20	2	43	3	✓	5+2		✓
nested loops medium	4	18	2	39	3	✓	4+2		✓
nested loops hard	4	19	2	40	3	✓	4+2		✓
nested loops hard (mod 1)	5	19	2	48	4	✓	4+2		✓
nested loops hard (mod 2)	10	19	2	96	8		4+3		✓
Halbwachs Fig. 1a [12]	2	9	1	15	2	✓	3+2	✓	✓
Halbwachs Fig. 1b	5	17	2	51	4	✓	4+2		✓
Halbwachs Fig. 1b (mod)	4	17	2	49	4	✓	5+2		✓
Halbwachs Fig. 2a	2	10	1	23	3	✓	3+2	✓	✓
Halbwachs Fig. 2b	3	12	1	28	3	✓	4+1		✓
Halbwachs Fig. 2b (mod)	4	14	1	46	4	✓	3+2	✓	✓
Halbwachs Fig. 4	18	18	2	84	9		4+2		✓
⋆Gopan Fig. 1a [10]	15	14	1	36	4	✓	5+2		
⋆Gopan Fig. 1a (mod)	13	14	1	33	4	✓	5+1		
Chaouch Fig. 2 [15]	2	12	1	19	2	✓	3+2		✓
Chaouch Fig. 3	7	23	1	83	6	✓	4+2		✓
Chaouch Fig. 3 (mod)	4	25	3	66	3	✓	4+1		
Chaouch Fig. 4	2	10	1	22	3	✓	4+1		✓
Chaouch Fig. 5	2	17	2	51	4	✓	4+2		✓
⋆Chaouch Fig. 6	13	14	1	36	4	✓	5+2		

Figure 11. Widening examples

Any previously unsatisfiable phase predicates are checked against the new node state c'' and new subtrees $p_{n+1} : \langle c_{new}^{n+1} \rangle; \ldots p_{n+k} : \langle c_{new}^{n+k} \rangle$; are added. Much simpler is the test $l : x \leq e$ which is applied recursively and is also added as phase predicate to $\bar{p}$ if it is unsatisfiable. The domain operations all rely on a function *compatible* that recursively adds missing phases by adding a subtree $p_i : \bot_{\mathcal{C}}; \ldots \bar{p}$ whenever $p_i : c_i; \ldots ; \bar{p}$ only exists in the respective other domain. The lattice operations $t_1 \sqsubseteq_{\mathcal{P}} t_2$ and $t_1 \sqcup_{\mathcal{P}} t_2$ then reduce to a point-wise lifting of the respective operations on the child domain. Widening is defined similarly to join, however, the phase boundaries are enforced after widening in order to ensure that the various states remain separated by the phase predicates. If widening makes unsatisfiable phase predicates satisfiable, new subtrees are added.

5 Experimental Results

We evaluated the presented domains in our analyzer for machine code [16], using a domain stack $\mathcal{W} \triangleright \mathcal{D} \triangleright \mathcal{T} \triangleright \mathcal{A} \triangleright \mathcal{C} \triangleright \mathcal{I}$ where $\mathcal{C}$ tracks congruences, except for examples marked with $\star$ that use $\mathcal{W} \triangleright \mathcal{D} \triangleright \mathcal{T} \triangleright \mathcal{P} \triangleright \mathcal{A} \triangleright \mathcal{C} \triangleright \mathcal{I}$. The benchmarks in Fig. 11 represent challenging loops that were mostly put forth in the literature [10,12,15]. Our own "nested loops" increase two variables, with various bounds and resets. Examples marked with "(mod)" are modifications of the

same problem. These include changing the loop exit conditions in nested loops or adding loop exit points (**`break`**, **`continue`**), adding further variables or loop counter increments on separate paths through the loop. We also modified examples, where applicable, to contain non-deterministic paths and multiple widening points inside the loops, both features that can be found in irreducible graphs. The measurements are as follows: *insns.* gives the number of instructions in the program; *#wp* is the number of widening back-edges; *steps* the number of instructions the analyzer evaluated to reach the fixpoint; *iter.* is the maximum number of fixpoint iterations at any program point; *exact* denotes if the best interval bounds were found; *time* shows the analysis time in milliseconds. The time shown is the median of 2000 runs on a 2.4 GHz Core i5 machine running Linux.

We compared our results with those of the `Interproc` and `ConcurInterproc` analyzers [15]. For both we used polyhedra with congruences which is the domain that is closest to our domain stack. `Interproc` can count iteration steps but only uses narrowing to refine the post-fixpoint. The table shows that the number of iterations in our analysis is usually smaller than that of `Interproc`, even without the narrowing iterations (which are indicated by $+n$). In all benchmarks, we used no explicit delay. Since most examples are engineered not to work with narrowing, the least fixpoint is rarely obtained. `ConcurInterproc` uses a pre-analysis to infer thresholds but does not perform an iteration count. Assuming that these thresholds are applied to the states after widening, `ConcurInterproc` must require at least as many iterations as the number of upward iterations of `Interproc`. Our precision and that of the threshold widening in `ConcurInterproc` match. Entries where our analysis is less precise than `ConcurInterproc` require a polyhedral invariant that our domains cannot express. For the examples requiring disjunctive invariants `ConcurInterproc` is imprecise in that it infers, for example, $x \in [51, 102]$ for line 10 in Fig. 8. Our benchmarks used for `Interproc` are available on-line at `http://tinyurl.com/cwdg5qr`.

6 Related Work

Many authors address the task of improving widening, be it for specific domains such as polyhedra [2,14], or by altering the way fixpoints are inferred. With respect to the latter, Halbwachs pioneered the idea of using thresholds to refine widening and to delay widening [13]. Thresholds over variables are created from a set of constants, an idea later successfully used in the large [4]. Chaouch et al. [15] recently proposed a pre-analysis to infer thresholds automatically. This pre-analysis uses the polyhedron abstract domain [9] and requires a way to extract individual inequalities from it. Rather than extracting thresholds, widening with landmarks [17] measures the distance of the current state space to the loop condition and extrapolates the state space accordingly. Both approaches require special domain functions, e.g. for widening, and are thus not easily portable between different numeric domains. Our threshold domain is easier to use as it is agnostic to the underlying domain and infers the possible thresholds by itself.

Bagnara et al. generalizes the idea of delaying widening by using a finite number of tokens: a widening may use any non-terminating strategy if there are still tokens to consume [2]. Rather than requiring the user to fix the set of tokens, our delay domain in Sect. 3 uses program points instead of tokens, thereby ensuring termination without depending on user input.

One challenge of using convex numeric domains is the problem of spillage of state into branches of the program or behaviors of the transfer function that cannot be recovered from by narrowing. In this context, Halbwachs et al. propose to re-start the analysis at a different pre-fixpoint from which widening and narrowing infer a new post-fixpoint. The intersection of the previous and the new post-fixpoint is still sound and may be more precise [12]. Rather than removing the spillage, Gopan et al. propose to avoid spillage into currently unreachable branches immediately after widening [10]. They require one state to determine which branches of the loops are enabled and a second state to compute widening and narrowing on the enabled part of the loop. Instead of duplicating the analysis cost by tracking a second abstract state, the authors later propose to directly track which parts of the CFG are enabled [11]. They generalize their idea to track different states for each phase, that is, for each set of enabled branches in a loop. While none of the three approaches require changes to the transfer functions of the domains as was the case for widening with thresholds, each approach requires intrusive changes to the fixpoint engine and the handling of states. Our threshold domain in Sect. 2 has the same functionality as the Guided Static Analysis approach [11] but requires no changes to the way states are handled. Interestingly, the transfer functions of our threshold domain are similar to those of the decision tree domain of Astrée [8]. However, the latter tracks Boolean flags as predicates and requires a user-supplied limit to avoid an exponential explosion. Since our domain creates a tree that mirrors the finite branching inside the loop body, its size is always limited by the program.

6.1 Conclusion

Implementing widening strategies as abstract domains is beneficial due to its modularity and independence of the fixpoint engine. This approach provides equal or better precision combined with fewer iterations required to obtain stability.

References

1. K. Apinis, H. Seidl, and V. Vojdani. Side-Effecting Constraint Systems: A Swiss Army Knife for Program Analysis. In *Asian Symposium on Program Languages and Systems*. Springer, 2012.
2. R. Bagnara, P. M. Hill, E. Ricci, and E. Zaffanella. Precise Widening Operators for Convex Polyhedra. *Science of Computer Programming*, 58(1–2):28–56, 2005.
3. G. Balakrishnan, G. Grurian, T. Reps, and T. Teitelbaum. CodeSurfer/x86 – A Platform for Analyzing x86 Executables. In *Compiler Construction*, volume 3443 of *LNCS*, pages 250–254, Edinburgh, Scotland, April 2005. Springer. Tool-Demonstration Paper.

4. B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Design and Implementation of a Special-Purpose Static Program Analyzer for Safety-Critical Real-Time Embedded Software. In T. Æ. Mogensen, D. A. Schmidt, and I. H. Sudborough, editors, *The Essence of Computation: Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones*, volume 2566 of *LNCS*, pages 85–108. Springer, 2002.
5. 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*, San Diego, California, USA, June 2003. ACM.
6. F. Bourdoncle. Efficient Chaotic Iteration Strategies with Widenings. In D. Bjørner, M. Broy, and I. V. Pottosin, editors, *Formal Methods in Programming and Their Applications*, volume 735 of *LNCS*, pages 128–141, Novosibirsk, Russia, June 1993. Springer.
7. P. Cousot and R. Cousot. Abstract Interpretation and Application to Logic Programs. *Journal of Logic Programming*, 13(2–3):103–179, 1992.
8. P. Cousot, R. Cousot, J. Feret, A. Miné, L. Mauborgne, D. Monniaux, and X. Rival. Combination of Abstractions in the ASTRÉE Static Analyzer. In M. Okada and I. Satoh, editors, *Asian Computing Science Conference*, volume 4435 of *LNCS*, pages 272–300, Tokyo, Japan, December 2006. Springer.
9. P. Cousot and N. Halbwachs. Automatic Discovery of Linear Constraints among Variables of a Program. In *Principles of Programming Languages*, pages 84–97, Tucson, Arizona, USA, January 1978. ACM.
10. D. Gopan and T. Reps. Lookahead Widening. In T. Ball and R. B. Jones, editors, *Computer-Aided Verification*, volume 4144 of *LNCS*, Seattle, Washington, USA, August 2006. Springer.
11. D. Gopan and T. W. Reps. Guided Static Analysis. In H. R. Nielson and G. Filé, editors, *Static Analysis Symposium*, volume 4634 of *LNCS*, pages 349–365, Kogens Lyngby, Denmark, August 2007. Springer.
12. N. Halbwachs and J. Henry. When the Decreasing Sequence Fails. In Antoine Miné and David Schmidt, editors, *Static Analysis Symposium*, volume 7460 of *LNCS*, pages 198–213. Springer, 2012.
13. 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.
14. B. Jeannet and A. Miné. Apron: A Library of Numerical Abstract Domains for Static Analysis. In A. Bouajjani and O. Maler, editors, *Computer Aided Verification*, volume 5643 of *LNCS*, pages 661–667, Grenoble, France, June 2009. Springer.
15. L. Lakhdar-Chaouch, B. Jeannet, and A. Girault. Widening with Thresholds for Programs with Complex Control Graphs. In *Automated Technology for Verification and Analysis*, pages 492–502, Taipei, 2011. Springer.
16. A. Sepp, B. Mihaila, and A. Simon. Precise Static Analysis of Binaries by Extracting Relational Information. In M.Pinzger and D. Poshyvanyk, editors, *Working Conference on Reverse Engineering*, Limerick, Ireland, October 2011. IEEE Computer Society.
17. A. Simon and A. King. Widening Polyhedra with Landmarks. In N. Kobayashi, editor, *Asian Symposium on Programming Languages and Systems*, volume 4279 of *LNCS*, pages 166–182, Sydney, Australia, November 2006. Springer.
18. A. Venet. Abstract Cofibered Domains: Application to the Alias Analysis of Untyped Programs. In *Static Analysis Symposium*, LNCS, pages 366–382, London, UK, 1996. Springer.