Abstract Interpretation over Non-Lattice Abstract Domains

Graeme Gange, Jorge A. Navas, Peter Schachte,
Harald Søndergaard, and Peter J. Stuckey

Department of Computing and Information Systems,
The University of Melbourne, Victoria 3010, Australia
{gkgange,jorge.navas,schachte,harald,pstuckey}@unimelb.edu.au

Abstract. The classical theoretical framework for static analysis of programs is abstract interpretation. Much of the power and elegance of that framework rests on the assumption that an abstract domain is a lattice. Nonetheless, and for good reason, the literature on program analysis provides many examples of non-lattice domains, including non-convex numeric domains. The lack of domain structure, however, has negative consequences, both for the precision of program analysis and for the termination of standard Kleene iteration. In this paper we explore these consequences and present general remedies.

1 Introduction

The goal of static analysis is to automatically infer useful information about the possible runtime states of a given program. Because different information is pertinent in different contexts, each analysis specifies the abstraction of the computation state to use: the *abstract domain* of the analysis.

Where the abstract domain has certain desirable properties, the abstract interpretation framework of Cousot and Cousot [1, 2] provides an elegant generic analysis algorithm. Under certain reasonable assumptions, the method is guaranteed to terminate with a sound abstraction of all possible program states. In particular, the abstract interpretation framework requires that the abstract domain be a lattice, and that the functions that specify how program operations affect the abstract program state be monotone.

In this paper, we focus on a class of abstract domains that do not form lattices; in particular they may not provide least upper bound and greatest lower bound operations. Such abstract domains are commonly proposed in the literature because they strike a balance, providing more detail than is offered by simpler lattice domains, while being computationally more tractable than more complex lattice domains. The reader will find several examples in Section 4.

The common response to the lack of meets and joins is to arbitrarily choose suitable but *ad hoc* lower and upper bound operators to use instead (we call them “quasi-meets” and “quasi-joins”). However, as we show, this begets other problems, such as lack of associativity of upper and lower bound operators, and a lack of monotonicity, with repercussions for termination. We pinpoint some inevitable consequences of straying from the classical abstract interpretation framework and exemplify these, together with example-specific solutions.

The reader is expected to be familiar with basic concepts in order theory and abstract interpretation. In Section 2 we refresh some of these concepts, to fix our terminology and to define a notion of quasi-lattice. In Sections 3.2 and 3.3 we prove that non-lattice domains fail to preserve many principles of reasoning that we have come to depend upon in the implementation of abstract interpretation, and we discuss the ensuing problems. In Section 4 we briefly present some of the non-lattice domains found in the literature on program analysis, and show how the problems manifest themselves. In Section 5 we catalogue various remedies. Section 6 concludes.

Contributions: In this paper we

– study the impact of using quasi-lattice abstract domains, in particular the effect on precision and termination;
– identify quasi-lattice domains in the literature on program analysis and use these to exemplify the issues that the general study lays bare; and
– outline modifications to classical abstract interpretation that are sufficient to guarantee soundness and termination, while maintaining reasonable precision of analysis.

2 Lattices and quasi-lattices

An important aim of this paper is to facilitate a discussion of "non-lattice-ness" and its consequences. The program analyses discussed in the paper are not new; they only serve to exemplify the phenomena we want to discuss. In this section we first recapitulate well-known concepts from order theory, then introduce a kind of "almost-but-not-lattice" with properties that are found in several recently proposed abstract domains.

Definition 1 (Ultimately cyclic and stationary sequences). Let $\mathbb{N} = \{0, 1, 2 \ldots\}$ and X be a set. An ω-sequence (or just *sequence*) of X-elements $[x_0, x_1, \ldots] = [x_i \mid i \in \mathbb{N}]$ is a total mapping s from $\mathbb{N}$ to X with $s(i) = x_i$. The sequence is *ultimately cyclic* iff $\exists k, m \in \mathbb{N}\ \forall n \in \mathbb{N}\ : n \geqslant k \rightarrow x_n = x_{m+n}$. In this case we refer to $[x_k, \ldots, x_{k+m-1}]$ as the sequence's *ultimate cycle.* The sequence is *ultimately stationary* iff it has an ultimate cycle of size 1. In this case we refer to the cycle's single element as the sequence's *final element.* □

Definition 2 (Bounded poset). Consider a binary relation $\sqsubseteq$, defined on a set D. The relation is a *partial order* iff it is

1. reflexive: $\forall d \in D : d \sqsubseteq d$
2. transitive: $\forall d_1, d_2, d_3 \in D : d_1 \sqsubseteq d_2 \wedge d_2 \sqsubseteq d_3 \rightarrow d_1 \sqsubseteq d_3$
3. antisymmetric: $\forall d_1, d_2 \in D : d_1 \sqsubseteq d_2 \wedge d_2 \sqsubseteq d_1 \rightarrow d_1 = d_2$

A set equipped with a partial order is a *poset*. If the poset $\langle D, \sqsubseteq \rangle$ has a *least* element $\bot$ and a greatest element $\top$ (that is, elements $\bot, \top \in D$ such that for all $d \in D$, $\bot \sqsubseteq d \sqsubseteq \top$) then D is *bounded*. Two elements $x, y \in D$ are *comparable* iff $x \sqsubseteq y$ or $y \sqsubseteq x$; otherwise they are *incomparable*. □

Definition 3 (Chain). A sequence $[x_i \in X \mid i \in \mathbb{N}]$ is a *chain* iff $\forall j, k \in \mathbb{N}$: $x_j \sqsubseteq x_k \vee x_k \sqsubseteq x_j$, that is, all elements are comparable. □

Definition 4 (Monotonicity). Let $\langle D, \sqsubseteq \rangle$ be a poset. A function $f : D \to D$ is *monotone* iff $\forall x, y \in D : x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y)$. □

Note that the composition of monotone functions is monotone.

Definition 5 (Upper and lower bounds). Let $\langle D, \sqsubseteq \rangle$ be a bounded poset. For any $X \subseteq D$ we say that $y \in D$ is an *upper bound* (*lower bound*) of X, written $X \sqsubseteq y$ ($y \sqsubseteq X$) iff $\forall x \in X : x \sqsubseteq y$ ($\forall x \in X : y \sqsubseteq x$). An *upper-bound operator* $U : \mathscr{P}(D) \to D$ is a function which assigns to each set X some upper bound $U(X)$. A *lower-bound operator* is defined analogously. Given a set $X \subseteq D$, a *least upper bound* of X is an element $z \in D$ which satisfies two conditions:

(a) z is an upper bound of X, and
(b) for each upper bound y of X, $z \sqsubseteq y$.

Dually, a *greatest lower bound* z of X satisfies

(a) z is a lower bound of X, and
(b) for each lower bound y of X, $y \sqsubseteq z$.

A *minimal* upper bound of X is an element z satisfying

(a) z is an upper bound of X, and
(b) for each upper bound y of X, $y \sqsubseteq z \Rightarrow y = z$.

A *maximal* lower bound is defined dually. We let *lower*(X) denote the set of lower bounds of X. □

We follow Nielson *et al.* [12] in putting no further requirements on an upper bound operator U. It is well-known that, in general, a least upper bound of X may not exist, but when it does, it is unique. We write the least upper bound as $\bigsqcup X$. Similarly, when it exists, the greatest lower bound of X is denoted $\bigsqcap X$. As usual, we define $x \sqcup y = \bigsqcup\{x, y\}$ and $x \sqcap y = \bigsqcap\{x, y\}$, and we refer to these operations as "join" and "meet", respectively.

Infinite chains do not, in general, have least upper bounds.[1]

Definition 6 (Chain-complete poset). The poset $\langle D, \sqsubseteq \rangle$ is *chain-complete* iff every chain $C \subseteq D$ has a least upper bound $\bigsqcup C$. □

[1] An infinite chain in D may not even have a *minimal* bound in D [5], witness the chain $\{x \in \mathbb{Q} \mid x^2 < 2\}$.

Definition 7 (Continuity). A function $f : D \to D$ is *continuous* iff $\bigsqcup\{f(x) \mid x \in C\} = f(\bigsqcup C)$ for all non-empty chains $C \subseteq D$. □

Definition 8 (Complete lattice). A complete *lattice* $L = \langle D, \sqsubseteq \rangle$ is a poset $\langle D, \sqsubseteq \rangle$ such that each $X \subseteq D$ has a least upper bound $\bigsqcup X$ and a greatest lower bound $\bigsqcap X$. □

Note that a complete lattice is bounded, by definition.

Definition 9 (Lattice). A *lattice* $L = \langle D, \sqsubseteq \rangle$ is a poset $\langle D, \sqsubseteq \rangle$ such that, for all $x, y \in D$, $\bigsqcup\{x, y\}$ and $\bigsqcap\{x, y\}$ exist. A lattice which is a bounded poset is a *bounded* lattice. □

Given a lattice $\langle D, \sqsubseteq \rangle$, the meet $\sqcup$ and the join $\sqcap$ have many desirable algebraic properties. In particular, they are monotone, idempotent, commutative and associative. It follows that, since a least upper bound exists for each 2-element set, a least upper bound exists for each finite set $X \subseteq D$.

Definition 10 (Quasi-lattice). A *quasi-lattice* $\langle Q, \sqsubseteq, \widetilde{\bigsqcup}, \widetilde{\bigsqcap} \rangle$ is a bounded poset $\langle Q, \sqsubseteq \rangle$ satisfying the following conditions:

1. $\widetilde{\bigsqcup}$ is an upper-bound, and $\widetilde{\bigsqcap}$ a lower-bound, operator on Q.
2. For all $x, y \in Q$, $\widetilde{\bigsqcup}\{x, y\}$ is a minimal upper bound of $\{x, y\}$.
3. For all $x, y \in Q$, $\widetilde{\bigsqcap}\{x, y\}$ is a maximal lower bound of $\{x, y\}$.
4. $\langle Q, \sqsubseteq \rangle$ has a "butterfly", that is, for some $a, b \in Q$, the set $\{a, b\}$ has a minimal, but no least, upper bound in Q. □

We refer to the $\widetilde{\bigsqcup}$ and $\widetilde{\bigsqcap}$ operations as "quasi-join" and "quasi-meet", respectively. Again, we define $x \mathbin{\tilde{\sqcup}} y = \widetilde{\bigsqcup}\{x, y\}$ and $x \mathbin{\tilde{\sqcap}} y = \widetilde{\bigsqcap}\{x, y\}$. Note that $\tilde{\sqcup}$ and $\tilde{\sqcap}$ are idempotent and commutative, by definition. Requirements 2 and 3 guarantee that, given two comparable elements, $\tilde{\sqcup}$ returns the larger, and $\tilde{\sqcap}$ the smaller. It also ensures that various absorption laws hold, such as $x \mathbin{\tilde{\sqcup}} x = x$, $(x \mathbin{\tilde{\sqcap}} y) \mathbin{\tilde{\sqcup}} y = y$, and $x \mathbin{\tilde{\sqcap}} (x \mathbin{\tilde{\sqcup}} y) = x$. Note that $\tilde{\sqcup}$ over-approximates each of its elements, that is, $\tilde{\sqcup}$ is conservative in the sense of abstract interpretation. It is therefore a sound replacement for the missing join. In contrast, $\tilde{\sqcap}$ is not a conservative replacement for meet. Hence it will not play a significant role in the rest of the paper. Abstract domains usually capture conjunction (or intersection of sets of runtime states) without precision loss; but not so disjunction.

Consider the bounded poset whose Hasse diagram is shown in Figure 1. It is not a lattice since it has a "butterfly"; namely, there is no least upper bound of $\{a, b\}$ (in fact, for some a, b, c, d, this structure is embedded in any bounded poset which is not a lattice). It is, however, a quasi-lattice, for several different choices of upper-bound operator. Nevertheless, each choice has shortcomings, as we now show.

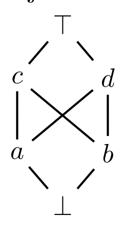


Fig. 1. A "butterfly".

Theorem 1. *In a quasi-lattice $\langle Q, \sqsubseteq, \widetilde{\bigsqcup}, \widetilde{\bigsqcap} \rangle$, $\tilde{\sqcup}$ and $\tilde{\sqcap}$ are neither monotone nor associative.*

Proof. We show this for $\tilde{\sqcup}$ only; the case of $\tilde{\sqcap}$ is similar. Let $\{x, y\} \subseteq Q$ be a set for which no least upper bound exists, and let m and m' be distinct minimal upper bounds for the set. Let $u = x \tilde{\sqcup} y$. We have either (1) $m \sqsubseteq u$ or $m' \sqsubseteq u$ (or both), or we have (2) m, m', u are pairwise incomparable. In case (1) we can assume, without loss of generality, that $m \sqsubseteq u$. Note that in this case, $u \not\sqsubseteq m'$. Hence, whether we are in case (1) or (2), $u \not\sqsubseteq m'$.

Now, to see that $\tilde{\sqcup}$ is not monotone, note that while $y \sqsubseteq m'$ (Rule 2), we also have

$$(x \tilde{\sqcup} y) = u \not\sqsubseteq m' = (x \tilde{\sqcup} m').$$

To see that $\tilde{\sqcup}$ is not associative, note that

$$(x \tilde{\sqcup} (y \tilde{\sqcup} m')) = m' \neq (u \tilde{\sqcup} m') = ((x \tilde{\sqcup} y) \tilde{\sqcup} m'). \square$$

Definition 11 (Fixed point). A *fixed point* of a function $f : D \to D$ is an element $x \in D$ such that $f(x) = x$. The set $fp(f)$ of fixed points of f is $\{x \in D \mid f(x) = x\}$. $\square$

Theorem 2 (From the Knaster-Tarski fixed point theorem). *Let L be a complete lattice and let $f : L \to L$ be monotone. Then $fp(f)$ is a non-empty complete lattice, and $\bigsqcap fp(f)$ is the least fixed point of f.*

We denote the least fixed point of f by $lfp(f)$.

Kleene iteration is a procedure which, given $x \in D$ and function $f : D \to D$, computes the sequence $iter_f(x) = [f^j(x) \mid j \in \mathbb{N}]$ but stops as soon as a final element (a fixed point of f) is reached; if $iter_f(x)$ is not ultimately stationary, the process does not terminate. If f is monotone and there is no infinite ascending chain in D then $iter_f(\bot)$, where $\bot$ is the least element of D, is an ultimately stationary chain whose final element is $lfp(f)$.

Abstract interpretation is concerned with the approximation of program semantics, expressed as fixed points of "semantic" functions. There are essentially two fixed point approximation approaches, one based on Galois connections, and one based on "widening" [3].

Definition 12 (Galois connection). Let $\langle D, \sqsubseteq_D \rangle$ and $\langle X, \sqsubseteq_X \rangle$ be posets, and let $\alpha : D \to X$ and $\gamma : X \to D$ be monotone functions. Then $\langle D, \alpha, \gamma, X \rangle$ is a *Galois connection* between D and X iff:

$$d \sqsubseteq_D \gamma(x) \Leftrightarrow \alpha(d) \sqsubseteq_X x \quad \text{for all } d \in D \text{ and } x \in X \square$$

The intuition is that $\gamma(x)$ expresses an abstract property x in concrete terms (that is, γ provides the "meaning" of x) whereas $\alpha(d)$ expresses the concrete property d as best as it can, given X's more limited expressiveness. We can naturally express the fact that x approximates (or is an abstraction of) d by $d \sqsubseteq_D \gamma(x)$, or alternatively by $\alpha(d) \sqsubseteq_X x$, and when these two characterisations coincide, we have a Galois connection.

We can also express the fact that a function $g : X \to X$ approximates $f : D \to D$ point by point: For all $x \in X$, we have

$$f(\gamma(x)) \sqsubseteq \gamma(g(x))$$

An important result [2] states that for a chain-complete poset D (as well as for a complete lattice D) and Galois connection $\langle D, \alpha, \gamma, X \rangle$, if the monotone $g : X \to X$ approximates the monotone $f : D \to D$ then $\mathit{lfp}(f) \sqsubseteq_D \gamma(\mathit{lfp}(g))$, that is, g's least fixed point is a sound approximation of f's.

Widening-based approaches to fixed point approximation are based on the following concept.

Definition 13 (Widening). Let $\langle D, \sqsubseteq \rangle$ be a bounded poset, a widening operator $\nabla : D \times D \to D$ satisfies the following two conditions:

1. $\forall\, x, y \in D : x \sqsubseteq (x \nabla y) \wedge y \sqsubseteq (x \nabla y)$.
2. For any increasing chain $x_0 \sqsubseteq x_1 \sqsubseteq x_2 \sqsubseteq \ldots$ the alternative chain defined as $y_0 = x_0$ and $y_{k+1} = (y_k \nabla x_{k+1})$ stabilizes after a finite number of steps.

In general, a widening operator is not commutative and it is not necessarily monotone. In practice it is common to combine the Galois connection approach with the widening approach, by resorting to the use of a widening operator only after a while, to enforce or accelerate convergence. We discuss this point further, in the context of non-lattice domains, in Section 5.2.

3 The use of quasi-joins

There are important consequences of the absence of lattice properties. Here we discuss three important ramifications.

3.1 Impact on predictability of analysis

A “join node” in a control flow graph may be at the confluence of edges that come from many different nodes. In lattice-based analysis this is where a least upper bound operation is used to combine the incoming pieces of information. Commonly, this is computed through repeated use of a binary join operation $\sqcup$. In the lattice context, the order in which these binary join operations are applied is irrelevant—any order will produce the least upper bound.

In non-lattice-based analysis, the absence of a $\sqcup$ forces us to define a proxy, $\tilde{\sqcup}$, which produces an upper bound, and ideally, a minimal one. However, as shown by Theorem 1, a minimal upper bound operation is not associative. In other words, different orders of application of $\tilde{\sqcup}$ may lead to different results, and in fact, some may be less precise than others.

The order in which the elements are combined depends on quirks of the analysis implementation, details that rightly should have no bearing on the result. One consequence is unpredictable analyses: insignificant changes to a program (or to the analyzer itself) may have significant consequences for analysis results.

3.2 Impact on precision

A brute-force approach to attacking the order-dependency problem is to exhaustively consider all possible orders of $\tilde{\sqcup}$ applications, choosing the best one. In this way we may hope to synthesize $\widetilde{\bigsqcup}$ from $\tilde{\sqcup}$.

Alas, this is not possible. Perhaps surprisingly, it turns out that in a quasi-lattice, one may not calculate a minimal upper bound of a finite set X by performing a sequence of binary quasi-joins, in spite of the fact that each quasi-join produces a minimal upper bound. The next theorem expresses this precisely.

Theorem 3. *There exists some finite bounded poset $\langle D, \sqsubseteq \rangle$ for which a minimal-upper-bound operator $U : \mathcal{P}(D) \to D$ cannot be obtained through repeated application of a (any) binary minimal upper bound $\tilde{\sqcup} : D^2 \to D$.*

Proof. The proof is by construction. Consider a bounded poset representing containment of elements within triples. The Hasse diagram for the case that has four atoms is shown in Figure 2. Note that, for each 3-element set $\{x, y, z\}$, there is a least upper bound xyz. Hence, for this least upper bound to be produced via repeated use of a quasi-join $\tilde{\sqcup}$, it would have to be the case that, for any triple xyz, one of $(x \,\tilde{\sqcup}\, y)$, $(x \,\tilde{\sqcup}\, z)$, and $(y \,\tilde{\sqcup}\, z)$ is xyz. Consider, however, the triple-containment poset over six elements a–f. In this case, there are $\binom{6}{3} = 20$ distinct triples. However, there are only $\binom{6}{2} = 15$ pairs of elements. So there must be some triple $x'y'z'$ such that, for all pairs of elements a, b, $a \,\tilde{\sqcup}\, b \neq x'y'z'$. Then computing $x' \,\tilde{\sqcup}\, y' \,\tilde{\sqcup}\, z'$ must yield $\top$, rather than the minimal upper bound $x'y'z'$. $\square$

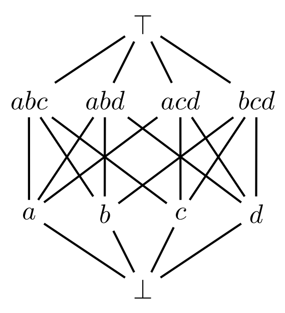


Fig. 2. A quasi-lattice with 10 elements

It follows that it is impossible to automatically synthesize a generalized minimal upper bound operator from primitive quasi-joins.

The lesson from this section and the previous one is that, in the non-lattice based case, a $\tilde{\sqcup}$ is not a suitable substitute for a minimal upper bound operator. Such an operator needs to be carefully crafted for the non-lattice domain at hand.

3.3 Impact on termination

With quasi-lattices, including the concrete examples we turn to in Section 4, we have structures which are *almost* lattices, but lack a monotone upper-bound operation $\tilde{\sqcup}$. We now show that this lack of monotonicity has ramifications for the usual approach of solving recursive dataflow equations via Kleene iteration, even when the quasi-lattice is finite.

Theorem 4. *For any quasi-lattice Q with binary upper-bound operation $\tilde{\sqcup}$, there are elements $x \in Q$ and monotone functions $f : Q \to Q$ for which Kleene iteration of $g = \lambda y \,.\, x \,\tilde{\sqcup}\, f(y)$ fails to terminate.*

Proof. From Theorem 1 we know that $\tilde{\sqcup}$ is not monotone. Hence there are $x, y, y' \in Q$ such that

$$y \sqsubseteq y', \;\; x \,\tilde{\sqcup}\, y \not\sqsubseteq x \,\tilde{\sqcup}\, y'$$

Either $(x \,\tilde{\sqcup}\, y') \sqsubset (x \,\tilde{\sqcup}\, y)$, or else $x \,\tilde{\sqcup}\, y$ and $x \,\tilde{\sqcup}\, y'$ are incomparable. Define the function $f : Q \to Q$ as follows:

$$f(v) = \begin{cases} y & \textbf{if } v \sqsubseteq x \,\tilde{\sqcup}\, y' \\ y' & \textbf{otherwise} \end{cases}$$

f is monotone, since:

$$f(v) \not\sqsubseteq f(v') \Rightarrow f(v) = y' \wedge f(v') = y \Rightarrow v \not\sqsubseteq x \,\tilde{\sqcup}\, y' \wedge v' \sqsubseteq x \,\tilde{\sqcup}\, y' \Rightarrow v \not\sqsubseteq v'$$

Note that $f(x \,\tilde{\sqcup}\, y') = y$, and $f(x \,\tilde{\sqcup}\, y) = y'$. Now consider Kleene iteration of g. Assuming $f(x \,\tilde{\sqcup}\, \bot) = y$, we observe the sequence of values:

$$[\bot, x \,\tilde{\sqcup}\, y, x \,\tilde{\sqcup}\, y', x \,\tilde{\sqcup}\, y, x \,\tilde{\sqcup}\, y', \ldots]$$

This sequence will alternate between the two values indefinitely. The same oscillation occurs if we instead assume $f(x \,\tilde{\sqcup}\, \bot) = y'$. □

At first the construction in the proof of Theorem 4 may seem artificial. However, Kleene iteration of functions of the form $\lambda y \,.\, x \sqcup f(y)$ captures exactly how one usually solves dataflow equations for simple loops, of the form shown in Figure 3. The proof of the theorem therefore gives us a recipe for constructing programs whose non-lattice-based analysis will fail to terminate, unless some remedial action is taken. Section 4 illustrates this for three concrete examples of non-lattice domains.

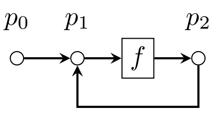


Fig. 3. A loop involving repeated use of monotone function f

4 Examples of non-lattice abstract domains

In this section, we review some recent abstract domains from the literature which do not form a lattice. In each case we sketch the abstract domain and the resulting analysis. (We necessarily skip many details—the reader is referred to the cited papers for detail.) In each case we show how the phenomena identified in Sections 3.2 and 3.3 play out for the domain.

4.1 Wrapped intervals (w-intervals)

Navas *et al.* [11] describe an abstract domain for reasoning about arithmetic operations over fixed-width machine integers. Where unbounded integers can be seen to sit on an infinite number line, fixed-width integers exist on a fixed-size number circle. One approach to handling machine arithmetic is to select a fixed wrapping point on the number circle, and represent values as intervals in the range $[v_{min}, v_{max}]$. For example, Regehr and Duongsaa [13] perform bounds analysis in a sound, wrapping-aware manner (dealing also with bit-wise operations) but as their analysis uses conventional intervals, precision is lost when sets of values cross the selected wrapping point.

Example 1 (Traditional intervals lose precision over machine arithmetic). Consider the interval $x = [0, 2]$ over 4-bit unsigned integers. The feasible values for $x - 1$ are $\{15, 0, 1\}$; however, as both 0 and 15 are contained in this set, the resulting interval is $\top$. $\square$

A natural alternative is to let intervals "wrap" [8, 11, 15]. The *wrapped intervals* (or w-intervals) of Navas *et al.* [11] still approximate a set of values as a single interval. However, there is no fixed wrapping point; a wrapped interval can begin or end at any point on the number circle.

More formally, a *w-interval* is either an empty interval, denoted $\perp$, a full interval, denoted $\top$, or a delimited interval $\llparenthesis x, y \rrparenthesis$, where x, y are w-width bit-vectors. Let $\mathcal{B}$ be the set of all *bit-vectors* of size w, and let b^k denote k copies of bit $b \in \{0, 1\}$ in a row. Then, the concretization function is defined as:

$$\begin{aligned}
\gamma(\perp) &= \varnothing \\
\gamma\llparenthesis x, y \rrparenthesis &= \begin{cases} \{x, \ldots, y\} & \text{if } x \leqslant y \\ \{0^w, \ldots, y\} \cup \{x, \ldots, 1^w\} & \text{otherwise} \end{cases} \\
\gamma(\top) &= \mathcal{B}
\end{aligned}$$

In the case of Example 1, the corresponding wrapped interval is $\llparenthesis 15, 1 \rrparenthesis$, which succinctly represents the set of feasible values $\{15, 0, 1\}$. Unfortunately, while there is a partial ordering $\sqsubseteq$ over the set of wrapped intervals, there is no longer a unique upper bound for any pair of intervals; accordingly, the domain clearly is not a lattice. In fact, using the upper bound $\tilde{\sqcup}$ given in [11], this domain is a quasi-lattice. Therefore, by Theorem 1, $\tilde{\sqcup}$ is neither associative nor monotone.

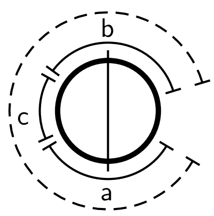
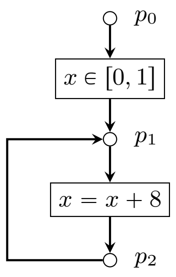


Fig. 4. The dashed interval is the minimal upper bound of $\{a, b, c\}$

Example 2 (Quasi-join over the wrapped intervals is not associative). In the context of 4-bit unsigned arithmetic, consider the three w-intervals $a = \llparenthesis 13, 2 \rrparenthesis$, $b = \llparenthesis 6, 10 \rrparenthesis$, and $c = \llparenthesis 3, 5 \rrparenthesis$. These are shown in Figure 4. Consider that we apply the binary quasi-join $\tilde{\sqcup}$ as follows: $(a \mathbin{\tilde{\sqcup}} b) \mathbin{\tilde{\sqcup}} c = \llparenthesis 6, 2 \rrparenthesis \mathbin{\tilde{\sqcup}} \llparenthesis 3, 5 \rrparenthesis = \llparenthesis 6, 5 \rrparenthesis = \top$.



0	**1**	**2**	**3**
$\llparenthesis 0,1 \rrparenthesis$	$\llparenthesis 0,9 \rrparenthesis$	$\llparenthesis 8,1 \rrparenthesis$	$\llparenthesis 0,9 \rrparenthesis$
$\llparenthesis 8,9 \rrparenthesis$	$\llparenthesis 8,1 \rrparenthesis$	$\llparenthesis 0,9 \rrparenthesis$	$\llparenthesis 8,1 \rrparenthesis$

Fig. 5. Non-terminating analysis of wrapped intervals over 4-bit integers; column i shows the interval for x in round i

However, the minimal upper bound can be obtained as $a \mathbin{\tilde{\sqcup}} (b \mathbin{\tilde{\sqcup}} c) = \llparenthesis 13,2 \rrparenthesis \mathbin{\tilde{\sqcup}} \llparenthesis 3,10 \rrparenthesis = \llparenthesis 13,10 \rrparenthesis$. □

Example 3 (Any minimal quasi-join over wrapped intervals is non-monotone). Consider again the domain of intervals over 4-bit integers, with $x = \llparenthesis 0,1 \rrparenthesis$, $y = \llparenthesis 8,9 \rrparenthesis$, $x' = \llparenthesis 0,9 \rrparenthesis$ and $y' = \llparenthesis 8,1 \rrparenthesis$. Clearly, $x \sqsubseteq x'$ and $y \sqsubseteq y'$. Assume we have a quasi-join $\tilde{\sqcup}$ which selects a minimal upper bound. The two candidates for $x \mathbin{\tilde{\sqcup}} y$ are $\llparenthesis 0,9 \rrparenthesis$ and $\llparenthesis 8,1 \rrparenthesis$. Assume we let $x \mathbin{\tilde{\sqcup}} y = \llparenthesis 0,9 \rrparenthesis$. Then we have $x \mathbin{\tilde{\sqcup}} y = \llparenthesis 0,9 \rrparenthesis \not\sqsubseteq \llparenthesis 8,1 \rrparenthesis = y' = x \mathbin{\tilde{\sqcup}} y'$. Similarly, if $x \mathbin{\tilde{\sqcup}} y = \llparenthesis 8,1 \rrparenthesis$, we have $x \mathbin{\tilde{\sqcup}} y = \llparenthesis 8,1 \rrparenthesis \not\sqsubseteq \llparenthesis 0,9 \rrparenthesis = x' = x' \mathbin{\tilde{\sqcup}} y$. □

Given that $\tilde{\sqcup}$ is non-monotone, we can construct an instance where the analysis in the form of Kleene iteration does not terminate.

Example 4 (Analysis with w-intervals does not terminate when $\tilde{\sqcup}$ is used as the join operator). Figure 5 shows an example for bit-width $w = 4$. Recall from Example 3 that there are two equally good representations of the interval $\llparenthesis 0,1 \rrparenthesis \mathbin{\tilde{\sqcup}} \llparenthesis 8,9 \rrparenthesis$; namely $\llparenthesis 0,9 \rrparenthesis$ and $\llparenthesis 8,1 \rrparenthesis$. Assume we pick $\llparenthesis 0,9 \rrparenthesis$; the other case is symmetric. In round 1, we compute $p_2 = \llparenthesis 0,9 \rrparenthesis + 8 = \llparenthesis 8,1 \rrparenthesis$. At the beginning of round 2, we compute $p_1 = \llparenthesis 0,1 \rrparenthesis \mathbin{\tilde{\sqcup}} \llparenthesis 8,1 \rrparenthesis = \llparenthesis 8,1 \rrparenthesis$ (since $\llparenthesis 0,1 \rrparenthesis \sqsubseteq \llparenthesis 8,1 \rrparenthesis$), and p_2 then becomes $\llparenthesis 0,9 \rrparenthesis$. Since $\llparenthesis 0,1 \rrparenthesis$ is also contained in $\llparenthesis 0,9 \rrparenthesis$, p_1 becomes $\llparenthesis 0,9 \rrparenthesis$ in round 3, and we return to the state observed in round 1. The analysis will forever oscillate between the states shown in columns 1 and 2. □

4.2 Donut domains

Most numerical domains are restricted to convex relations between variables; however, it is often useful to allow limited forms of non-convex reasoning. *Donut domains* [6] are constructed as the set difference of two convex domains $\langle A_1, \leqslant_1 \rangle$ and $\langle A_2, \leqslant_2 \rangle$ (relative to a given concrete powerset domain). We first consider an idealized form of donut domains. An abstract value $(x_1, x_2) \in A_1 \backslash A_2$ is interpreted according to the concretization function:

$$\gamma(x_1, x_2) = \gamma_1(x_1) \backslash \gamma_2(x_2)$$

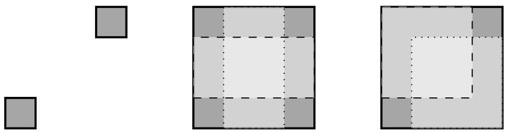


Fig. 6. The pair of intervals $(x, y \in [-2, -1])$ and $(x, y \in [1, 2])$ has four minimal upper bounds (with respect to the inclusion ordering). In each case, the convex hull remains the same, but the *hole* component can exclude either of the rectangular regions between the two squares, or one of the two corner rectangles.

x_1 is an over-approximation of the set of reachable states, and x_2 is an under-approximation of the set of unreachable states. Assuming a suitable normalization operation, this induces a partial order over the abstract values:

$$(x_1, x_2) \sqsubseteq (y_1, y_2) \text{ iff } \gamma(x_1, x_2) \sqsubseteq \gamma(y_1, y_2)$$

This partial order clearly does not form a lattice, as there may be many minimal upper bounds of a given pair of elements.

Example 5 (The donut domain of intervals does not have a least upper bound). Consider computing the least upper bound of $a = (x, y \in [1, 2])$ and $b = (x, y \in [-2, -1])$. Four minimal upper bounds are illustrated in Figure 6. All the minimal upper bounds are of the form $(x, y \in [-2, 2]) \wedge \neg p(x, y)$ for some "hole" constraint p. For example, $p(x, y)$ may be $(x \in [-2, 2]) \wedge (y \in [-1, 1])$, or we could have $(x \in [-2, 1]) \wedge (y \in [-1, 2])$. Even though all four choices are minimal with respect to $\sqsubseteq$, the concretization of the rectangular bounds shown in the centre is larger than that of the square bounds shown to the right. □

Given that this ordering lacks a least upper bound, any precise quasi-join will necessarily suffer the same precision and non-termination problems present in other non-lattice domains.

Example 6 (Any minimal quasi-join for the donut domain over intervals is non-associative). Consider again the intervals discussed in Example 5. Assume the quasi-join chooses the upper-left square $p = (x \in [-2, 1]) \wedge (y \in [-1, 2])$ as the hole. Then $(a \mathbin{\tilde{\sqcup}} b) \mathbin{\tilde{\sqcup}} p$ has no hole, where the minimal upper bounds have the non-empty holes $(x \in [1, 2]) \wedge (y \in [-2, 1])$ and $(x \in [-1, 2]) \wedge (y \in [-2, -1])$. For any other minimal choice made by $\tilde{\sqcup}$, we can select p similarly such that $(a \mathbin{\tilde{\sqcup}} b) \mathbin{\tilde{\sqcup}} p$ is strictly larger than $a \mathbin{\tilde{\sqcup}} (b \mathbin{\tilde{\sqcup}} p)$. □

We now turn our attention to the formulation of donut domains presented in [6]. Given the difficulty, in general, of computing a minimal convex under-approximation of the complement of a pair of donuts, it is unsurprising that a simplified join is presented instead. The authors first define a slightly different ordering over abstract values. They define

$$(x_1, x_2) \leqslant_{1\backslash 2} (y_1, y_2) = x_1 \leqslant_1 y_1 \wedge \left(\overline{\gamma_1(x_1)} \cup \gamma_2(x_2) \supseteq \overline{\gamma_1(y_1)} \cup \gamma_2(y_2)\right)$$

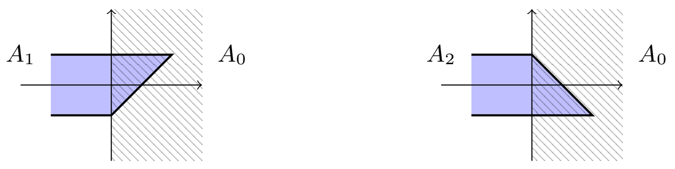


Fig. 7. The two donut objects $\langle A_1, A_0\rangle$ and $\langle A_2, A_0\rangle$ have identical concretizations, but are incomparable under the ordering $\leqslant_{1\backslash 2}$.

The bracketed component of this definition is precisely the partial order used above; $\leqslant_{1\backslash 2}$ is then a subset of $\sqsubseteq$. We suspect a misprint has crept in here, since otherwise, for some donut domains, there are values with identical concretization which are incomparable under $\leqslant_{1\backslash 2}$. An example of this is given in Figure 7. The donut objects $\langle A_1, A_0\rangle$ and $\langle A_2, A_0\rangle$ are built from octagons; A_0 can be expressed as $x \geqslant 0$, A_1 as $-4 \leqslant y \leqslant 4 \wedge x \leqslant y + 4$, and A_2 as $-4 \leqslant y \leqslant 4 \wedge x \leqslant -y + 4$.

The join is defined as $(x_1, x_2) \mathrel{\tilde{\sqcup}_{1\backslash 2}} (y_1, y_2) = (x_1 \sqcup_1 y_1, (x_1, x_2)\breve{\sqcap}(y_1, y_2))$, where

$$(x_1, x_2)\breve{\sqcap}(y_1, y_2)) = \tilde{\alpha}((\gamma_2(x_2) \cap \gamma_2(y_2)) \cup (\gamma_2(x_2) \cap \overline{\gamma_1(y_1)}) \cup (\gamma_2(y_2) \cap \overline{\gamma_1(x_1)}))$$

The first component of the quasi-join is simply the join over A_1. $\breve{\sqcap}$ computes the complement by taking the intersection of each hole with the complement of the other value. Note that this only reasons about existing holes, and does not synthesize additional holes from gaps between the convex hulls; in the case of Example 6, $\tilde{\sqcup}_{1\backslash 2}$ simply takes the convex hull of the pair. Once the complement is computed, it is mapped back to A_2 by $\tilde{\alpha}$; this differs from α_2 in that $\tilde{\alpha}$ under-approximates the set of concrete states, rather than over-approximates. The definition of $\tilde{\alpha}$ is left unspecified; it is assumed to select one of the possibly many convex under-approximations of the complement.

As the previous paragraph illustrates, the donut domain, with the operations provided in [6], is not a quasi-lattice. The domain could, however, be turned into a quasi-lattice, by providing precise (minimal) upper bound operations, as these do exist. In any case, we can construct a non-terminating instance for donut domains in a similar manner as for wrapped intervals.

Example 7 (Analysis with the donut domain over intervals does not terminate). Figure 8 shows a non-terminating example for donut domains. We start with the constraint $x, y \in [-2, 5] \wedge \neg(x, y \in (2, 4))$. At the beginning of round 1, given this definition of $\cup_{1\backslash 2}$, there are two minimal choices of hole. Assume we pick the hole in the top-right. This gives us the value $(x, y \in [-5, 5], x, y \in (2, 4))$. Applying f, we get $p_2 = (x, y \in [-5, 5], x, y \in (-4, -2))$. Notice that this contains p_0. In round 2, we then get $p_1 = (x, y \in [-5, 5], x, y \in (-4, -2))$. Then $p_2 = (x, y \in [-5, 5], x, y \in (2, 4))$. This again contains p_0, so round 3 returns to the state observed in round 1. As for the case of wrapped intervals, the analysis oscillates forever between the states observed in rounds 1 and 2. □

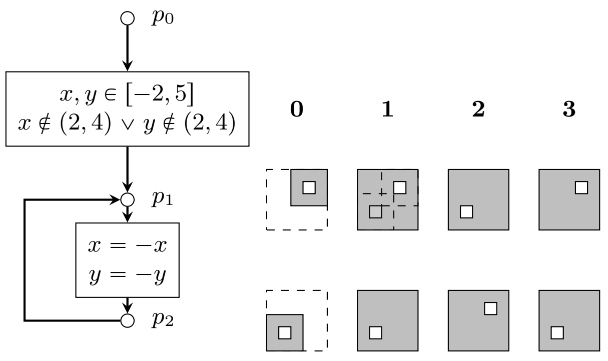


Fig. 8. Non-terminating analysis of the donut domain over intervals; column i shows the possible values for (x, y) in round i. The top entry in round 1 illustrates the two minimal upper bounds.

4.3 Segmentations for array content analysis

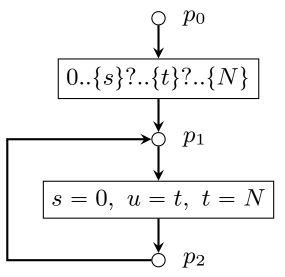
Cousot, Cousot and Logozzo [4] propose a novel approach to the analysis of array-processing code. The idea (a further development of work by Gopan, Reps and Sagiv [7], and by Halbwachs and Péron [9]) is to summarise the content of an array by using "segmentations". These are compact descriptions that combine information about the order of indices with summary information about the content of delineated array segments. More precisely, a segmentation is of the form

$$\{e^1_1 \ldots e^1_{m_1}\}\ P_1\ \{e^2_1 \ldots e^2_{m_2}\}\ P_2\ \cdots P_{n-1}\ \{e^n_1 \ldots e^n_{m_n}\}$$

where each e is an index expression and P_j is a description that applies to every array element between index e^{j-1} and index e^j. The lumping together (between a pair of curly brackets) of several index expressions indicates that these expressions are aliases, that is, $e^j_1 = e^j_2 = \cdots = e^j_{m_j}$. In our examples, index expressions will be constants or program variables, but they could be more complex. Unless otherwise indicated, the interval from e^j to e^{j+1} is assumed to be definitely non-empty. To indicate that it may be empty, the curly brackets surrounding e^{j+1} are followed by a '?'. As examples of what segmentations tell us about index relations, a segmentation of form $\{0\}..\{s\}?..\{t\}?..\{N\}$ tells us that $0 \leqslant s \leqslant t < N$, whereas one of form $\{0\ s\}..\{u\}?..\{t\ N\}$ tells us that $0 = s \leqslant u < t = N$.

Segmentation unification is the process of, given two segmentations with "compatible" extremal segment bounds (in general for the same array), modifying both segmentations so that they coincide. By "compatible" we mean that the first and last segment bounds have a non-empty intersection.

Segmentation unification is a key operation since it is the core of the join. Sec 11.4 of [4] states the problem of segmentation unification admits a partially ordered set of solutions, but in general, not forming a lattice.



0	**1**	**2**
$\{0\}..\{s\}?..\{t\}?..\{N\}$	$\{0\}..\{s\}?..\{N\}$	$\{0\}..\{s\}?..\{t\}?..\{N\}$
$\{0\ s\}..\{u\}?..\{t\ N\}$	$\{0\ s\}..\{t\ N\}$	$\{0\ s\}..\{u\}?..\{t\ N\}$

Fig. 9. Non-terminating analysis with array segmentations; column i shows the segmentation in round i

For instance, unifying $\{0\}..\{a\}..\{b\}..\{c\}$ with $\{0\}..\{b\}..\{a\}..\{c\}$ results in two incomparable minimal solutions: $\{0\}..\{a\}..\{c\}$ and $\{0\}..\{b\}..\{c\}$.

The authors describe a greedy pseudo-algorithm that scans left-to-right, keeping a point-wise consistent subset of the ordering. They also describe a *2 look-ahead* approach (in contrast with the greedy 1 look-ahead algorithm) which takes the next segment into account when unifying. As we should expect, both of these quasi-joins are non-monotone.

Example 8 (1 look-ahead segment unification is non-monotone). Let segment $A = \{0\}..\{s\}..\{t\}..\{N\}$, $B = \{0\}..\{u\}..\{s\}..\{N\}$, and $B' = \{0\}..\{s\}..\{N\}$. Clearly, $B \sqsubseteq B'$. However, we have $A \,\tilde{\sqcup}\, B = \{0\}..\{N\}$, and $A \,\tilde{\sqcup}\, B' = \{0\}..\{s\}..\{N\}$. $\square$

The 2 look-ahead algorithm is also clearly non-monotone, such as in the case where $A = \{0\}..\{a\}..\{b\}..\{N\}$ and $B = \{0\}..\{b\}..\{c\}..\{N\}$. (In this case, the 1 look-ahead algorithm would see that $\{a\}$ and $\{b\}$ do not match, discarding both, then again for $\{b\}$ and $\{c\}$, returning the segmentation $\{0\}..\{N\}$).

In both cases, we can construct a non-terminating instance following the structure of Theorem 4. We present only an example for the 1 look-ahead algorithm; the 2 look-ahead case can be constructed in a very similar fashion.

Example 9 (Analysis with the array segmentation domain does not terminate using the 1 look-ahead unification algorithm). Figure 9 shows an example over variables $\{s, t, u\}$. Initially, we have $0 \leqslant s \leqslant t < N$. Reaching p_2, we have the segmentation $\{0\ s\} \ldots \{u\}? \ldots \{t\ N\}$. Unifying at p_1, the $\{t\}$ and $\{u\}$ partitions are discarded, yielding $\{0\} \ldots \{s\}? \ldots \{N\}$. Since t is not in the current segmentation, u is omitted in the next round, and we reconstruct the original segmentation. It is interesting to note that this oscillates between two *comparable* values. $\square$

This domain does not satisfy the quasi-lattice conditions using either of the described $\tilde{\sqcup}$ definitions. For example, using the 1 look-ahead algorithm, we have $x = \{0\}..\{s\}..\{t\}..\{N\} \sqsubseteq y = \{0\}..\{t\}..\{N\}$, but $x \,\tilde{\sqcup}\, y = \{0\}..\{N\} \neq y$. However, a related quasi-lattice could be constructed by using a more precise $\tilde{\sqcup}$ which selects amongst the set of minimal upper bounds.

5 Abstract interpretation over bounded posets

We have seen that, even for a domain satisfying the relatively strict quasi-lattice requirements, being a non-lattice has negative consequences for predictability, precision and termination of Kleene iteration. We now consider abstract interpretation over abstract domains that are only required to be bounded posets.

5.1 Non-associative quasi-joins

The lack of associativity of a quasi-join cannot of itself compromise the total correctness of the abstract interpretation algorithm. However, we have seen that it can cause a loss of precision. It also means that different fixed point algorithms may lead to different results, further complicating the design of an abstract interpretation framework.

Usually analysis frameworks will define a generalized quasi-join operation in terms of a binary quasi-join:

$$\widetilde{\bigsqcup}\{x_1, \ldots, x_n\} = (\cdots (x_1 \mathbin{\tilde{\sqcup}} x_2) \mathbin{\tilde{\sqcup}} x_3 \cdots) \mathbin{\tilde{\sqcup}} x_n$$

and similarly for $\widetilde{\bigsqcap}$ (if this operation is needed). Because quasi-joins are not associative, clearly the ordering of the x_i is important. Theorem 3 shows us that in some cases no ordering will produce a minimal upper bound, so it is preferable to specify a generalized quasi-join operation directly. For example, in the case of the wrapped interval domain, Navas *et al.* [11] present a generalized quasi-join operation defined in terms of a binary quasi-join that produces the minimal upper bound with a complexity of $O(n \log(n))$, where n is the number of w-intervals. The generalized quasi-join can compute the minimal solution by first ordering the w-intervals lexicographically. Then, it repeatedly applies the binary quasi-join to the ordered sequence while keeping track of the largest gap which is not covered by the application of the binary quasi-joins. We refer to [11] for details about the algorithm.

5.2 Non-monotone quasi-joins

As discussed in Section 3.3, in abstract interpretation, we often wish to find the least fixed point of a function defined in terms of abstract operations (transfer functions) and joins. When using quasi-lattices, and hence quasi-joins, we may find ourselves seeking the least fixed point of a non-monotone function. In that setting, Theorem 2 does not apply, so we do not know whether a least fixed point exists, or, if it does, how to compute it. As we have seen, standard Kleene iteration may not terminate. We now show, however, that a generalized Kleene iteration algorithm will produce a sound result, under reasonable assumptions.

Theorem 5. *Let C be a complete lattice and $f : C \to C$ be continuous. Let A be a bounded poset with least element $\bot$, and let $\gamma : A \to C$ be given. If the (not necessarily monotone) $g : A \to A$ approximates f, that is,*

$$\forall y \in A : f(\gamma(y)) \sqsubseteq \gamma(g(y)) \tag{1}$$

and the sequence $g* = [\bot, g(\bot), g^2(\bot), \ldots]$ is ultimately cyclic, then for every y in the ultimate cycle of $g*$, $\mathit{lfp}(f) \sqsubseteq \gamma(y)$.

Proof. Let $Y = [y_0, \ldots, y_{m-1}]$ be the ultimate cycle of $g*$ and let $x_0 = \bigsqcap_{0 \leqslant i < m} \gamma(y_i)$, We then have:

$$\begin{aligned} f(x_0) &\sqsubseteq f(\gamma(y_i)) && \text{for all } 0 \leqslant i < m\text{, by monotonicity of } f \\ &\sqsubseteq \gamma(g(y_i)) && \text{for all } 0 \leqslant i < m\text{, by (1)} \\ &= \gamma(y_{i+1 \bmod m}) && \text{for all } 0 \leqslant i < m \end{aligned}$$

Hence $f(x_0) \sqsubseteq \bigsqcap_{0 \leqslant i < m} \gamma(y_i) = x_0$. Clearly $\bot_C \sqsubseteq x_0$, so by monotonicity of f, and the transitivity of $\sqsubseteq$, $f^k(\bot_C) \sqsubseteq x_0$ for all $k \in \mathbb{N}$. As f is continuous, $\mathit{lfp}(f) \sqsubseteq x_0$, so for each $y \in Y$, $\mathit{lfp}(f) \sqsubseteq \gamma(y)$. □

This has two important ramifications for use in abstract interpretation:

1. Kleene iteration will not cycle (repeat) before finding a sound approximation of the true set of concrete states; and
2. In a finite abstract domain, it will reach this result in finite time.

Thus Kleene iteration can safely be used for abstract interpretation over bounded poset abstract domains, as long as we generalize the loop detection algorithm to detect cycles of cardinality greater than one. We can use the fact that any ultimately cyclic sequence must include a subsequence x_i, x_{i+1} such that $x_i \not\sqsubseteq x_{i+1}$ to reduce the overhead of the loop check. Also, since every element of the ultimate cycle is a sound approximation, we are free to return a cycle element e for which $\gamma(e)$ has minimal cardinality. We assume we are supplied with a function **better**(x_1, x_2) that returns the x_i for which $\gamma(x_i)$ has the smaller cardinality.

Algorithm 6 (Generalised Kleene Iteration)

```
procedure ULT_CYCLE(g)
    result ← $\bot$
    repeat
        prev ← result
        result ← g(result)
    until prev $\not\sqsubseteq$ result
    while prev $\neq$ result do
        result ← g(g(result))
        prev ← g(prev)
    end while
    next ← g(result)
    while prev $\neq$ next do
        result ← better(result, next)
        next ← g(next)
    end while
    return result
end procedure
```

The ***repeat*** loop searches for the beginning of an ultimate cycle while repeatedly applying g. Note that the ***until*** condition is a strict inequality and hence, if there is a fixed point (that is, $prev = result$) the loop will also terminate. The first ***while*** loop iterates until it completes the ultimate cycle.[2] By Theorem 5, any solution obtained from any element in this cycle is a sound approximation of the least fixed point. The second ***while*** loop then chooses the most precise member of the cycle using the **better** function.

This algorithm performs very similarly to Kleene iteration in cases where the Kleene sequence is an ascending chain. In other cases it is costlier. Where performance is preferred to precision, the final ***while*** loop can safely be omitted.

A more efficient, but even less precise, algorithm can be had by forcibly ensuring that the Kleene sequence is increasing by defining

$$g'(x) = x \mathbin{\tilde{\sqcup}} g(x).$$

Then the standard Kleene iteration algorithm can be used on g'. Note that where the Kleene sequence for g is an ascending chain, all of these approaches yield the same result at approximately the same cost.

Example 10 (Forced climbing on wrapped intervals). Consider the program given in Example 1. After round 1, we have $p_1 = (\!|0, 9|\!)$, $p_2 = (\!|8, 1|\!)$. Where previously we compute the updated value of p_1 as $(\!|0, 1|\!) \mathbin{\tilde{\sqcup}} (\!|8, 1|\!)$, we now compute $p_1 = (\!|0, 9|\!) \mathbin{\tilde{\sqcup}} (\!|0, 1|\!) \mathbin{\tilde{\sqcup}} (\!|8, 1|\!) = \top$. The updated value of p_2 also becomes $\top$, and we have reached a fixed point. □

Where the abstract domain is infinite, or just intractably large, another mechanism must be used to hasten termination, at the cost of some loss of precision. As has been previously observed [14], widening may be used to ensure termination:

> *The widening technique can be useful even outside abstract interpretation. For example, in a normal dataflow analysis, we can use it to make sure that the series of abstract values computed for a given program point by the analysis iterations is an ascending chain, even if the transfer functions are not monotone.*

However, care must be taken. In many cases, widening is used periodically during Kleene iteration, giving up precision only when convergence appears too slow. If this is done for Kleene iteration over a non-monotone function, it is possible that the progress that is ensured by the occasional widening step is lost in successive non-widening steps, leading to an infinite loop. That is, if the

[2] This part exploits Floyd's "tortoise and hare" principle [10], and requires only two values at a time to be remembered. However, it requires more applications of g than are strictly needed. If computation of g is expensive, it may be preferable to use a hash table to store values returned by g, and to simplify the loop body so that g is called just once per iteration.

underlying function is not monotone, applying widening occasionally will not make it monotone. For example, if $g(x) = y$, $g(y) = z$, $g(z) = x$, and $x \nabla y = z$, and the widening operation is applied on odd steps, then the Kleene sequence $[x, x \nabla g(x) = z, g(z) = x, \ldots]$ is not ultimately stationary. Thus to ensure termination, perhaps after a finite number of non-widening steps, widening must be performed at each step. Alternatively some other measure must be taken between widening steps (such as taking the quasi-join with the previous result) to ensure the function is monotone.

6 Conclusion

In the pursuit of increased precision, it is tempting to step outside the lattice-based framework of abstract interpretation. In the absence of a join operation, the obvious response is to seek a "quasi-join" which provides minimal upper bounds. We have shown, however, that such a quasi-join cannot always be generalized to a "minimal-upper-bound operation". This means that the precision of analysis results depends on arbitrary and insignificant design decisions that should be immaterial to the analysis. Equally, even a small semantics-preserving change to the surface structure of a subject program may have great impact on precision. Finally, the quasi-join's inevitable lack of monotonicity easily leads to non-termination of Kleene iteration.

We have exemplified these phenomena with three recently proposed non-lattice abstract domains. Usually when such domains are proposed, their proponents provide remedial tricks that overcome the problems we discuss, including non-termination of analysis. In particular, widening may be used to ensure termination. We have argued that, even so, care must be exercised if widening is interleaved with non-widening steps. Finally we have provided strategies for adapting standard Kleene iteration to the context of non-monotone functions defined on bounded posets, including forced climbing, widening, and the use of a generalised, loop-checking variant of Kleene iteration.

Acknowledgments

We wish to thank the anonymous reviewers for many insightful suggestions. This work was supported through ARC grant DP110102579.

References

1. Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the Fourth Annual Symposium on Principles of Programming Languages*, pages 238–252. ACM, 1977.
2. Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Proceedings of the Sixth Annual Symposium on Principles of Programming Languages*, pages 269–282. ACM, 1979.

3. Patrick Cousot and Radhia Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In M. Bruynooghe and M. Wirsing, editors, *International Symposium on Programming Language Implementation and Logic Programming*, volume 631 of *LNCS*, pages 269–295. Springer, 1992.
4. Patrick Cousot, Radhia Cousot, and Francesco Logozzo. A parametric segmentation functor for fully automatic and scalable array content analysis. In *Proceedings of the 38th Annual Symposium on Principles of Programming Languages*, pages 105–118. ACM, 2011.
5. B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order.* Cambridge University Press, 1990.
6. Khalil Ghorbal, Franjo Ivančić, Gogul Balakrishnan, Naoto Maeda, and Aarti Gupta. Donut domains: Efficient non-convex domains for abstract interpretation. In V. Kuncak and A. Rybalchenko, editors, *Verification, Model Checking and Abstract Interpretation*, volume 7148 of *LNCS*, pages 235–250, 2012.
7. Denis Gopan, Thomas Reps, and Mooly Sagiv. A framework for numeric analysis of array operations. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 338–350. ACM, 2005.
8. Arnaud Gotlieb, Michel Leconte, and Bruno Marre. Constraint solving on modular integers. In *Proceedings of the Ninth International Workshop on Constraint Modelling and Reformulation*, September 2010.
9. Nicolas Halbwachs and Mathias Péron. Discovering properties about arrays in simple programs. *SIGPLAN Notices*, 43:339–348, 2008.
10. Donald E. Knuth. *The Art of Computer Programming*, volume 2. Addison-Wesley, second edition, 1981.
11. Jorge A. Navas, Peter Schachte, Harald Søndergaard, and Peter J. Stuckey. Signedness-agnostic program analysis: Precise integer bounds for low-level code. In R. Jhala and A. Igarashi, editors, *APLAS 2012: Proceedings of the 10th Asian Symposium on Programming Languages and Systems*, volume 7705 of *Lecture Notes in Computer Science*, pages 115–130. Springer, 2012.
12. Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis.* Springer, 1999.
13. John Regehr and Umit Duongsaa. Deriving abstract transfer functions for analyzing embedded software. In *LCTES '06: Proceedings of the 2006 ACM SIGPLAN/SIGBED Conference on Language, Compilers, and Tool Support for Embedded Systems*, pages 34–43. ACM, 2006.
14. Alexandru Sălcianu. Notes on abstract interpretation, 2001. Unpublished Manuscript, `www.mit.edu/~salcianu`.
15. R. Sen and Y. N. Srikant. Executable analysis using abstract interpretation with circular linear progressions. In *Proceedings of the Fifth IEEE/ACM International Conference on Formal Methods and Models for Codesign*, pages 39–48. IEEE, 2007.