Interval Analysis and Machine Arithmetic: Why Signedness Ignorance is Bliss

GRAEME GANGE, The University of Melbourne
JORGE A. NAVAS, NASA Ames Research Center
PETER SCHACHTE, The University of Melbourne
HARALD SØNDERGAARD, The University of Melbourne
PETER J. STUCKEY, The University of Melbourne

The most commonly used integer types have fixed bit-width, making it possible for computations to "wrap around", and many programs depend on this behaviour. Yet much work to date on program analysis and verification of integer computations treats integers as having infinite precision, and most analyses that do respect fixed width lose precision when overflow is possible. We present a novel integer interval abstract domain that correctly handles wrap-around. The analysis is signedness agnostic. By treating integers as strings of bits, only considering signedness for operations that treat them differently, we produce precise, correct results at a modest cost in execution time.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*compilers*; *optimization*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*assertions*; *invariants*; *logics of programs*; *mechanical verification*; F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages—*program analysis*; G.1.0 [**Numerical Analysis**]: General—*Computer arithmetic*

General Terms: Algorithms, Languages, Reliability, Theory, Verification

Additional Key Words and Phrases: Abstract interpretation, interval analysis, LLVM, machine arithmetic, modular arithmetic, overflow, program analysis

1. INTRODUCTION

Most programming languages provide one or more fixed-width integer types. For mainstream languages these are by far the most widely-used integer types. Arithmetic operations on these types do not have the usual integer semantics, but instead obey laws of modular arithmetic. The results of all fixed-width integer operations, including intermediate operations, are truncated to the *bit-width* of the integer type involved. Failing to account for this can easily lead to incorrect results. For example, if signed w-bit integers a and b are known to be non-negative, it does not follow that their sum is, since signed fixed-width addition of positive integers can "wrap around" to produce a negative result.

Any program analysis seeking to accurately reflect the behaviour of fixed-width integer arithmetic must account for the fact that overflow and underflow may lead to incorrect results. In this paper, we shall consider analyses to determine upper and lower bounds for integer variables, so-called *interval analysis*. Most work on interval analysis, for example,

This work is supported by the Australian Research Council, under ARC grant DP110102579.
Authors' addresses: J. Navas, NASA Ames Research Center, Moffett Field CA 94035, USA; jorge.a.navaslaserna@nasa.gov. Remaining authors: Department of Computing and Information Systems, The University of Melbourne, Vic. 3010, Australia; {gkgange,schachte,harald,pstuckey}@unimelb.edu.au.

DOI:http://dx.doi.org/10.1145/0000000.0000000

Su and Wagner [2004], Leroux and Sutre [2007], Gawlitza et al. [2009], and Rodrigues et al. [2013] have ignored this issue of overflow, treating program integers as unbounded, mathematical integers. In a fixed-precision context this can lead to unsound analysis results. Consider for example this program fragment:

```
int i = 1;
while (*) {
    i = i+1;
}
assert(i>0);
```

A verification tool based on mathematical integers will conclude (regardless of the condition controlling the loop) that the assertion must hold: a value starting at one remains positive no matter how many times it is incremented. However, this does not reflect the actual behaviour of fixed-width arithmetic.

This defect is easily corrected by representing the bounds on fixed bit-width integer variables with fixed bit-width values, and correcting the abstract operations to respect those bit-widths. While this avoids incorrect conclusions, however, the precision of such a domain is disappointing. Take the case of a computation $z = x + y$ where x, y, and z are *unsigned* 4-bit variables[1], x is known to lie in the interval $[1100, 1101]$ and y is confined to the interval $[0010, 0011]$. That is, $12 \leq x \leq 13$ and $2 \leq y \leq 3$. Treating these as intervals on $\mathbb{Z}$, we would expect $14 \leq z \leq 16$, however, 16 is not expressible as a 4 bit integer. Thus z could lie in the interval $[1110, 1111]$, or it could overflow to give 0000, so the correct interval for z is $[0000, 1111]$. That is, all precision is lost.

Ironically, if we treated the same bit patterns as *signed* numbers, we would not lose precision. The reader should pause to consider this, bearing in mind that x, y, and z really are unsigned in our example. In the signed interpretation, $x \in [1100, 1101]$ means $-4 \leq x \leq -3$, so we can conclude $z \in [1110, 0000]$. The same bit patterns do not indicate an overflow for signed integers, so here we do not lose precision. As long as we treat $[1110, 0000]$ as a set of bit patterns rather than a set of integers, we can remain indifferent to the signedness of the actual values.

The same effect can arise when we treat signed integers as unsigned. If we know $x \in [0100, 0101]$ and $y \in [0010, 0011]$, where both values are treated as signed, then we conclude $z \in [1000, 0111] = [-8, 7]$, and again we lose all precision. However, if the values are treated as unsigned, we obtain the precise result $z \in [0110, 1000]$.

Thus, perhaps surprisingly, we obtain better precision by ignoring any signedness information about the numbers being manipulated, instead treating them as just bit patterns. And virtue becomes necessity when we wish to analyse low-level code, such as machine code or LLVM, the “Low Level Virtual Machine.” LLVM is rapidly gaining popularity as a target for compilers for a range of programming languages. As a result, the literature on static analysis of LLVM code is growing [Falke et al. 2012; Falke et al. 2013; Sen and Srikant 2007; Teixera and Pereira 2011; Zhang et al. 2010; Zhang et al. 2011]. LLVM IR (Intermediate Representation) carefully specifies the bit-width of all integer values, but does not specify whether they are signed or unsigned. Because for most operations two’s complement arithmetic (treating the inputs as signed numbers) produces the same bit-vectors as unsigned arithmetic, LLVM IR always omits signedness information except for operations that must behave differently for signed and unsigned numbers, such as comparisons. In general it is not possible to determine from LLVM code which values originated as signed variables in the source program and which originated as unsigned. An analysis for LLVM code benefits all compilers that target LLVM code as their back end, so it is fortuitous that signedness information is not needed to infer precise intervals.

[1] We use 4-bit examples and binary notation to make examples more manageable.

The literature on program analysis is vast, and the reader may ask how our approach differs from methods that use similar-looking abstract domains, or methods based on other ideas, such as constraint propagation or bit-blasting. We discuss this in Section 9 when the reader has become familiar with our method and the sense in which it is "signedness agnostic". For now, suffice it to say that our aim has been to develop a static program analysis which maintains the advantages of classical interval analysis, namely speed and scalability, while working correctly and showing better precision in the fixed-width integer context, compared to simpler "overflow aware" approaches. Alternative methods for reasoning about integer bounds tend to sacrifice speed, precision and/or scalability in the face of real-world programs, especially where these involve non-linear arithmetic.

The contributions of this paper can be summarised as follows:

- We adapt the classical integer interval analysis domain to correctly handle fixed-width integer arithmetic without undue loss of precision. The key idea of this domain, which we call "wrapped intervals," is that correctness and precision of analysis can be obtained by letting abstract operations deal with states that are superpositions of signed and unsigned states.
- As an abstract domain, wrapped intervals do not form a lattice. We investigate the ramifications of this and provide remedies for the undesirable consequences. In particular, we show how to generalise a binary upper bound operator to one that finds a minimal upper bound for a set of intervals, without undue precision loss.
- We motivate and provide detailed algorithms for all aspects of the analysis, including so-called widening. Our widening approach is new and is based on the idea of, roughly, doubling the size of an interval in each widening step.
- We establish various results about relations with similar-looking abstract domains, including the fact that the proposed abstract domain is incomparable with (reduced products of) previously proposed value domains.
- We evaluate the resulting analysis on a suite of SPEC CPU 2000 benchmarks and show that it provides higher precision than the classical integer interval analysis for a moderate added cost.

We assume the reader is familiar with basic lattice theory and concepts from the field of abstract interpretation, including Moore families, reduced products of abstract domains, widening and narrowing [Cousot and Cousot 1977; 1979; 1992].

The remainder of this paper is organized as follows. Section 2 reviews the classical integer interval analysis domain. Section 3 introduces wrapped intervals formally, and discusses their use in contexts in which it is not known whether values are signed or unsigned. In Section 4 the abstract domain of wrapped intervals is compared to related reduced-product domains. Section 5 deals with termination and acceleration of the analysis. Section 6 presents the results of experiments and gives an evaluation of cost and benefits. Section 7 employs the domain to reduce the amount of instrumentation code necessary to detect runtime overflows and underflows in C programs, and Section 8 discusses further potential applications. Section 9 discusses previous work to adapt interval analysis to fixed precision integers. Section 10 describes future work and concludes. A preliminary version of this paper appeared as Navas et al. [2012].

2. BASIC INTEGER INTERVAL ANALYSES

The goal of interval analysis is to determine an approximation of the sets of possible values that may be stored in integer-valued variables at various points in a computation. To keep this tractable, interval analysis approximates such a set using only its smallest and largest possible values, taking the specified set to be all integers between those bounds.

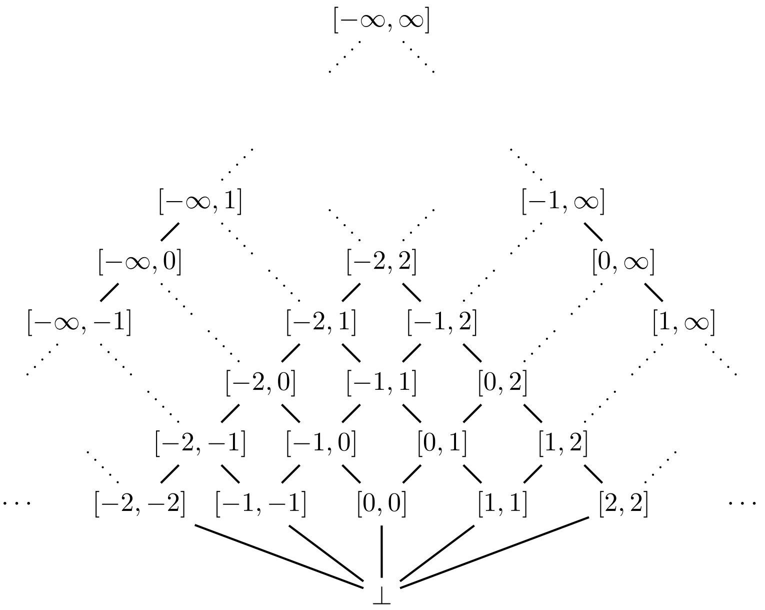


Fig. 1. The classical integer interval domain $\mathcal{I}$

2.1. The classical integer interval domain

Interval analysis is well understood [Nielson et al. 1999; Seidl et al. 2012]. The classical interval lattice $\mathcal{I}$ is shown in Figure 1. Apart from the element $\perp$, which denotes the empty interval, the elements are of the form $[x, y]$, where x ranges over $\mathbb{Z} \cup \{-\infty\}$, y ranges over $\mathbb{Z} \cup \{\infty\}$, and $x \leq y$. (Here of course $\leq$ is the natural extension of $\leq$ on $\mathbb{Z}$, that is, $-\infty \leq x \leq \infty$ for all $x \in \mathbb{Z} \cup \{-\infty, \infty\}$.) The ordering $\sqsubseteq$ of such intervals is obvious, albeit slightly cumbersome to express. Let us define

$$\mathsf{lo}(z) = \begin{cases} \infty & \text{if } z = \perp \\ x & \text{if } z = [x,y] \end{cases} \qquad \mathsf{hi}(z) = \begin{cases} -\infty & \text{if } z = \perp \\ y & \text{if } z = [x,y] \end{cases}$$

Then we can define $z \sqsubseteq z'$ iff $\mathsf{lo}(z') \leq \mathsf{lo}(z) \wedge \mathsf{hi}(z) \leq \mathsf{hi}(z')$. For the join we have:

$$z \sqcup z' = \begin{cases} \perp & \text{if } z = z' = \perp \\ [\mathsf{min}(\mathsf{lo}(z), \mathsf{lo}(z')), \mathsf{max}(\mathsf{hi}(z), \mathsf{hi}(z'))] & \text{otherwise} \end{cases}$$

For the meet, additional care is needed:

$$z \sqcap z' = \begin{cases} \perp & \text{if } z = \perp \text{ or } z' = \perp \\ \perp & \text{if } \mathsf{disjoint}(z, z') \\ [\mathsf{max}(\mathsf{lo}(z), \mathsf{lo}(z')), \mathsf{min}(\mathsf{hi}(z), \mathsf{hi}(z'))] & \text{otherwise} \end{cases}$$

where $\mathsf{disjoint}([x,y],[x',y'])$ holds iff $y < x' \vee y' < x$.

Of central interest in this paper is the handling of arithmetic operations in (wrapped) integer interval analysis. As a reference point, we conclude this section with the well-known definitions of the (abstract versions of) the arithmetic operators + and ×. Abstract addition is defined:

$$z + z' = \begin{cases} \perp & \text{if } z = \perp \text{ or } z' = \perp \\ [\mathsf{lo}(z) + \mathsf{lo}(z'), \mathsf{hi}(z) + \mathsf{hi}(z')] & \text{otherwise} \end{cases}$$

where the + on the right-hand side is addition extended to $\mathbb{Z} \cup \{-\infty, \infty\}$. Abstract multiplication is defined:

$$z \times z' = \begin{cases} \bot & \text{if } z = \bot \\ & \text{or } z' = \bot \\ [min(S), max(S)] \text{ where} & \\ S = \{\mathsf{lo}(z) \times \mathsf{lo}(z'), \mathsf{lo}(z) \times \mathsf{hi}(z'), \mathsf{hi}(z) \times \mathsf{lo}(z'), \mathsf{hi}(z) \times \mathsf{hi}(z')\} & \text{otherwise} \end{cases}$$

For example, to calculate $[-4, 2] \times [3, 5]$, one considers the combinations -4×3, -4×5, 2×3, and 2×5 and identifies the minimum and maximum values. This yields $[-4, 2] \times [3, 5] = [-20, 10]$. Note carefully the central role played by the functions *min* and *max* in this definition.

As is clear from Figure 1, the classical interval domain has infinite ascending chains. Implementations of interval analysis invariably include *widening* [Nielson et al. 1999] to accelerate or ensure termination of the analysis.

2.2. Fixed-precision integer intervals

Adapting the classical interval analysis to the fixed-precision case is not difficult. For an interval analysis over the *unsigned* integers modulo m we define abstract domain $\mathcal{I}^{\mathsf{u}}_m$. The elements of this domain are $\bot$ (for the empty interval) together with the set of *delimited* intervals, $\{[a, b] \mid 0 \leq a \leq b < m\}$. For the *signed* domain $\mathcal{I}^{\mathsf{s}}_m$, the elements are $\bot$ and the delimited intervals $\{[a, b] \mid \lceil \frac{-m}{2} \rceil \leq a \leq b < \frac{m}{2}\}$. For a picture of $\mathcal{I}^{\mathsf{s}}_m$, simply replace, in Figure 1, each lower-bound '$-\infty$' by '$\lceil \frac{-m}{2} \rceil$' and each upper-bound '$\infty$' by '$\lceil \frac{m}{2} \rceil - 1$'. To picture $\mathcal{I}^{\mathsf{u}}_m$, first remove all intervals with negative lower bounds, and then replace '∞' by '$m - 1$'. For the lattice operations, perform the same substitutions—these definitions are unchanged otherwise. For the arithmetic operations we now need to pay attention to the possibility of overflow. For $\mathcal{I}^{\mathsf{u}}_m$, we can conservatively define addition as follows:

$$z + z' = \begin{cases} \bot & \text{if } z = \bot \text{ or } z' = \bot \\ [\mathsf{lo}(z) + \mathsf{lo}(z'), \mathsf{hi}(z) + \mathsf{hi}(z')] & \text{if } \mathsf{hi}(z) + \mathsf{hi}(z') < m \\ [0, m - 1] & \text{otherwise} \end{cases}$$

where the + is normal integer addition. In a similar manner we can define the operations for signed analysis, taking both under- and overflow into account.

Because our interest is in faithfully analysing programs that manipulate (signed or unsigned) native machine integers, we will largely focus on $\mathcal{I}^{\mathsf{s}}_{2^w}$ and $\mathcal{I}^{\mathsf{u}}_{2^w}$, where w is a common integer bit-width.

3. WRAPPED INTEGER INTERVAL ANALYSIS

To accurately capture the behaviour of fixed bit-width arithmetic, we must limit the concrete domain to the values representable by the types used in the program, and correct the implementation of the abstract operations to reflect the actual behaviour of the concrete operations [Simon and King 2007]. As we have seen, a commitment to ordinary ordered intervals $[x, y]$ (either signed or unsigned), when wrap-around is possible, can lead to severe loss of precision.

This suggests that it is better to treat the bounds of an interval as a superposition of signed and unsigned values, allowing ourselves to accommodate both signed and unsigned wrap-around. That is, we treat each bound as merely a bit pattern, considering its signedness only when necessary for the operation involved (such as comparison). We therefore describe the domain as *signedness-agnostic*. We treat each interval as the set of bit patterns beginning with the first bound and obtainable by incrementing this value until the second bound is reached. Not knowing whether these bit patterns are signed or unsigned, we cannot say which is the "lower" and which is the "upper" bound.

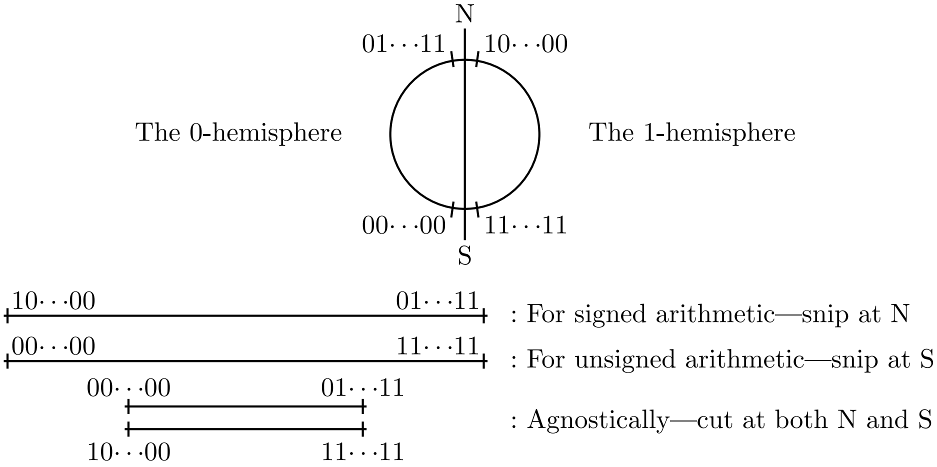


Fig. 2. Three different ways to cut the number circle open

Instead of representing bounds over fixed bit-width integer types as a single range of values on the number line, we handle them as a range of values on a number circle (see Figure 2), or, in the n-dimensional case, as a (closed convex) region of an n-dimensional torus. The unsigned numbers begin with 0 near the "south pole," proceeding clockwise to the maximum value back near the south pole. The signed numbers begin with the smallest number near the "north pole," proceeding clockwise through 0, back to the largest signed number near the north pole.

"Wrapped" intervals are permitted to cross either (or both) poles. Letting an interval start and end anywhere has several advantages:

- It allows for a limited and special type of disjunctive interval information. For example, an interval $x \in [0111, 1001]$ means $7 \leq x \leq 9$ if x is treated as unsigned, and $x = 7 \vee -8 \leq x \leq -7$ if it is treated as signed.
- It results in an abstract domain that is *complemented*: For a wrapped interval t, we can express $x \notin t$ just as readily as $x \in t$. For example, in the analysis of `if (x == 0)` s_1 `else` s_2, we can express `x`'s latitude in each of s_1 and s_2 exactly (namely, $x \in [0000, 0000]$ in case of s_1, and $x \in [0001, 1111]$ in case of s_2).
- Wrapped interval arithmetic better reflects algebraic properties of the underlying arithmetic operations than intervals without wrapping, even if signedness information is available. Consider for example the computation $x + y - z$. If we know x, y, and z are all signed 4-bit integers in the interval $[0011, 0101]$, then we determine $y - z \in [1110, 0010]$, whether using wrapped intervals or not. But wrapped intervals will also capture that $x + y \in [0110, 1010]$, while an unwrapped fixed-width interval analysis would see that this sum could be as large as the largest value 0111 and as small as the smallest 1000, so would derive no useful bounds. Therefore, wrapped intervals derive the correct bounds $[0001, 0111]$ for both $(x+y)-z$ and $x+(y-z)$. The use of ordinary (un-wrapped) intervals, on the other hand, can only derive these bounds for $x + (y - z)$, finding no useful information for $(x + y) - z$, although wrapping is not necessary to represent the final result. This ability to allow intermediate results to wrap around is a powerful advantage of wrapped intervals, even in cases where signedness information is available and final results do not require wrapping.

All up, this small broadening of the ordinary interval domain allows precise analysis of code where signedness information is unavailable. Equally importantly, it can provide increased precision even where all signedness information *is* provided.

As we shall see, the advantages of wrapped intervals do come at a price. The domain of wrapped intervals does not form a lattice, and the consequence is the inconvenience of a great deal of care being necessary in an implementation. In this paper we provide all the necessary details for an efficient implementation and show that, fortunately, the greater craft required in implementation does not translate into algorithms that are substantially slower than those used in classical interval analysis.

3.1. Wrapped intervals, formally

We use $\mathcal{B}_w$ to denote the set of all *bit-vectors* of size w. We will use sequence notation to construct bit-vectors: b^k, where $b \in \{0, 1\}$, represents k copies of bit b in a row, and $s_1 s_2$ represents the concatenation of two bit-vectors s_1 and s_2. For example, $01^4 0^3$ represents 01111000.

We shall apply the usual arithmetic operators, with their usual meanings, to bit-vectors. That is, unadorned arithmetic operators treat bit-vectors identically to their unsigned integer equivalents. Operators subscripted by a number suggest modular arithmetic; more precisely, $a +_n b = (a + b) \bmod 2^n$, and similarly for other operators.

We use $\leq$ for the usual lexicographic ordering of $\mathcal{B}_w$. For example, $0011 \leq 1001$. In the context of wrapped intervals, a relative ordering is more useful than an absolute one. We define

$$b \leq_a c \text{ iff } b -_w a \leq c -_w a$$

Intuitively, this says that starting from point a on the number circle and travelling clockwise, b is encountered no later than c. It also means that if the number circle were rotated to put a at the south pole (the zero point), then b would be lexicographically no larger than c.

Naturally, $\leq_0$ coincides with $\leq$, and reflects the normal behaviour of `<=` on unsigned w-bit integers. Similarly, $\leq_{2^{w-1}}$ reflects the normal behaviour of `<=` on *signed* w-bit integers. When their arguments are restricted to a single hemisphere (see Figure 2), these orderings coincide, but $\leq_0$ and $\leq_{2^{w-1}}$ do not agree across hemispheres.

We view the fixed-width integers we operate on as actually bit-vectors, completely free of signedness information. This accords exactly with how LLVM and assembly languages view integers. However, for convenience, when operations on bit-vectors will be independent of the interpretation, we may sometimes use integers (by default unsigned) to represent bit-vectors. This is just a matter of convenience: by slight extension it allows us to use congruence relations and other modular-arithmetic concepts to express bit-vector relations that are otherwise cumbersome to express. The following definition is a good example.

Definition 3.1. A *wrapped interval*, or *w-interval*, is either an empty interval, denoted $\bot$, a full interval, denoted $\top$, or a delimited interval $\llparenthesis x, y \rrparenthesis$, where x, y are w-width bit-vectors and $x \neq y +_w 1$.[2]

Let $\mathcal{W}_{2^w}$ be the set of w-intervals over width w bit-vectors. The meaning of a w-interval is given by the function $\gamma : \mathcal{W}_{2^w} \to \mathcal{P}(\mathcal{B}_w)$:

$$\begin{array}{lcl} \gamma(\bot) & = & \varnothing \\ \gamma\llparenthesis x, y \rrparenthesis & = & \left\{ \begin{array}{ll} \{x, \ldots, y\} & \text{if } x \leq y \\ \{0^w, \ldots, y\} \cup \{x, \ldots, 1^w\} & \text{otherwise} \end{array} \right. \\ \gamma(\top) & = & \mathcal{B}_w \end{array}$$

For example,

$$\gamma\llparenthesis 1111, 1001 \rrparenthesis = \{1111, 0000, 0001, 0010, 0011, 0100, 0101, 0110, 0111, 1000, 1001\}$$

[2] The condition, which is independent of signed/unsigned interpretation, avoids duplicate names (such as $\llparenthesis 0011, 0010 \rrparenthesis$ and $\llparenthesis 1100, 1011 \rrparenthesis$) for the full interval.

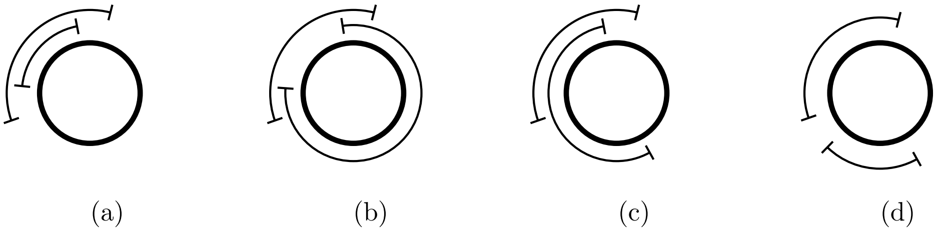


Fig. 3. Four cases of relative position of two w-intervals

represents the signed integers $[-1,7] \cup \{-8,-7\}$ or the unsigned integers $[0,9] \cup \{15\}$. The cardinality of a w-interval is therefore:

$$\begin{array}{lcl} \#(\bot) & = & 0 \\ \#(\!| x, y |\!) & = & (y -_w x +_w 1) \\ \#(\top) & = & 2^w \end{array}$$

In an abuse of notation, we define $e \in u$ iff $e \in \gamma(u)$. Note that $\mathcal{W}_{2^w}$ is complemented. We define the complement of a w-interval:

$$\begin{array}{lcl} \overline{\bot} & = & \top \\ \overline{\top} & = & \bot \\ \overline{(\!| x, y |\!)} & = & (\!| y +_w 1, x -_w 1 |\!) \end{array}$$

3.2. Ordering wrapped intervals

We order $\mathcal{W}_{2^w}$ by inclusion: $t_1 \subseteq t_2$ iff $\gamma(t_1) \subseteq \gamma(t_2)$. It is easy to see that $\subseteq$ is a partial ordering on $\mathcal{W}_{2^w}$ the set is a finite partial order with least element $\bot$ and greatest element $\top$.

We now define membership testing and inclusion for wrapped intervals. For membership testing:

$$e \in u \;\equiv\; u = \top \vee (u = (\!| x, y |\!) \wedge e \leq_x y)$$

Inclusion is defined in terms of membership: either the intervals are identical or else both endpoints of s are in t and at least one endpoint of t is outside s.

$$s \subseteq t \;=\; \begin{cases} \mathit{true} & \text{if } s = \bot \vee t = \top \vee s = t \\ \mathit{false} & \text{if } s = \top \vee t = \bot \\ a \in t \wedge b \in t \wedge (c \notin s \vee d \notin s) & \text{if } s = (\!| a, b |\!), t = (\!| c, d |\!) \end{cases}$$

In guarded definitions like this, the clause that applies is the first (from the top) whose guard is satisfied; that is, an 'if' clause should be read as 'else if'.

Consider the cases of possible overlap between two w-intervals shown in Figure 3. Only case (a) depicts containment, but case (b) shows a situation where each w-interval has its bounds contained in the other. This explains why the third case in the definition of $\subseteq$ requires that $c \notin s$ or $d \notin s$.

While $(\mathcal{W}_{2^w}, \subseteq)$ is partially ordered, it is *not* a lattice. For example, consider the w-intervals $(\!| 0100, 1000 |\!)$ and $(\!| 1100, 0000 |\!)$. Two minimal upper bounds are the incomparable $(\!| 0100, 0000 |\!)$ and $(\!| 1100, 1000 |\!)$, two sets of the same cardinality. So a join operation is not available. By duality, neither is a meet operation.

In fact the domain of wrapped intervals is not a Moore family, that is, it is not closed under conjunction. For example,

$$\gamma(\!| 1000, 0000 |\!) \cap \gamma(\!| 0000, 1000 |\!) = \{0000, 1000\},$$

a *set* which does not correspond to a w-interval. Furthermore, the two w-intervals $\llparenthesis 1000, 0000 \rrparenthesis$ and $\llparenthesis 0000, 1000 \rrparenthesis$ are minimal candidates describing the set $\{0000, 1000\}$ equally well. In other words, there is no unique best abstraction of $\{0000, 1000\}$.

The obvious response to the lack of a join is to seek an "over-join" operation $\widetilde{\sqcup}$ which selects, from the set of possible resulting w-intervals, the one with smallest cardinality. In the case of a tie, any convenient mechanism can be used to select a single result.

3.3. Biased over- and under-approximation of bounds

Since $\mathcal{W}_{2^w}$ is not a lattice, it does not have meet and join operations. However, it is useful to define the best approximations of meet and join we can create. In fact, there are two sensible approximations of each, depending on whether we need an under- or over-approximation: *over-meet* $\widetilde{\sqcap}$ and *over-join* $\widetilde{\sqcup}$ produce over-approximations, and *under-meet* $\underset{\sim}{\sqcap}$ and *under-join* $\underset{\sim}{\sqcup}$ produce under-approximations. These are best understood in terms of the semantic function γ:

$$\begin{aligned}
\gamma(s \mathbin{\widetilde{\sqcup}} t) &\supseteq \gamma(s) \cup \gamma(t), &\quad& \text{minimising } \#(s \mathbin{\widetilde{\sqcup}} t)\\
\gamma(s \mathbin{\underset{\sim}{\sqcap}} t) &\subseteq \gamma(s) \cap \gamma(t), && \text{maximising } \#(s \mathbin{\underset{\sim}{\sqcap}} t)\\
\gamma(s \mathbin{\underset{\sim}{\sqcup}} t) &\subseteq \gamma(s) \cup \gamma(t), && \text{maximising } \#(s \mathbin{\underset{\sim}{\sqcup}} t)\\
\gamma(s \mathbin{\widetilde{\sqcap}} t) &\supseteq \gamma(s) \cap \gamma(t), && \text{minimising } \#(s \mathbin{\widetilde{\sqcap}} t)
\end{aligned}$$

In particular, note that $\widetilde{\sqcup}$ produces a minimal upper bound and $\underset{\sim}{\sqcap}$ produces a maximal lower bound. For the analysis presented in this paper, only $\widetilde{\sqcup}$ and $\widetilde{\sqcap}$ turn out to be useful. However, $\underset{\sim}{\sqcup}$ and $\underset{\sim}{\sqcap}$ would be needed for other analyses, for example a backward analysis to determine the bounds on arguments to a function that would ensure that calls to the function can complete without an index out-of-bounds error. Thus it is worth presenting all four operations.

All four use cardinality to choose among candidate results. To resolve ties we arbitrarily choose the interval with the (lexicographically) smallest left component; hence these are biased algorithms. We use duality to simplify the presentation, shown in Figure 4. In the definition of $\widetilde{\sqcup}$, the first two cases handle $\top$ and $\bot$, as well as Figure 3 (a); the third case handles Figure 3 (b); the fourth and fifth cases handle Figure 3 (c); and the final two cases handle Figure 3 (d). Conversely, in the definition of $\widetilde{\sqcap}$, the first two cases handle $\top$ and $\bot$, as well as Figure 3 (a); the third case handles Figure 3 (d); the fourth and fifth cases handle Figure 3 (c); and the final two cases handle Figure 3 (b).

Unfortunately all these operations have important shortcomings. First, they are not associative; in fact, different ways of associating the operands may yield results with different cardinalities. For example, if $x = \llparenthesis 0010, 0110 \rrparenthesis$, $y = \llparenthesis 1000, 1010 \rrparenthesis$, and $z = \llparenthesis 1110, 0000 \rrparenthesis$, then $(x \widetilde{\sqcup} y) \widetilde{\sqcup} z = \llparenthesis 1110, 1010 \rrparenthesis$ has smaller cardinality than $x \widetilde{\sqcup} (y \widetilde{\sqcup} z) = \llparenthesis 0010, 0000 \rrparenthesis$. Second, none of these operations are monotone. For example, we have $\llparenthesis 1111, 0000 \rrparenthesis \leq \llparenthesis 1110, 0000 \rrparenthesis$ and $\llparenthesis 0110, 1000 \rrparenthesis \mathbin{\widetilde{\sqcup}} \llparenthesis 1111, 0000 \rrparenthesis = \llparenthesis 1111, 1000 \rrparenthesis$. But owing to the left bias, $\llparenthesis 0110, 1000 \rrparenthesis \mathbin{\widetilde{\sqcup}} \llparenthesis 1110, 0000 \rrparenthesis = \llparenthesis 0110, 0000 \rrparenthesis$. As we do not have $\llparenthesis 1111, 1000 \rrparenthesis \leq \llparenthesis 0110, 0000 \rrparenthesis$, $\widetilde{\sqcup}$ is not monotone. We discuss the ramifications of this in Section 5, together with a work-around.

Lack of associativity means we cannot define generalised (variadic) $\widetilde{\bigsqcup}$, $\underset{\sim}{\bigsqcap}$, $\underset{\sim}{\bigsqcup}$, and $\widetilde{\bigsqcap}$ operations by simply folding the corresponding binary operation over a collection of w-intervals, as we are accustomed to do for lattice domains. These operations should be carefully defined to produce the *smallest* w-interval containing all the given w-intervals, the *largest* w-interval contained in each of the given w-intervals, the *largest* w-interval contained in the union of all the given w-intervals, and the *smallest* w-interval containing the intersection of all the given w-intervals, respectively. The necessary specialized algorithms are worthwhile, because it is not uncommon to use repeated joins in program analysis, for example, when analysing a basic block with more than two predecessor blocks. Using repeated binary joins

$$s \mathbin{\widetilde{\sqcup}} t = \begin{cases} t & \text{if } s \subseteq t \\ s & \text{if } t \subseteq s \\ \top & \text{if } s = \llparenthesis a, b \rrparenthesis \land t = \llparenthesis c, d \rrparenthesis \land a \in t \land b \in t \land c \in s \land d \in s \\ \llparenthesis a, d \rrparenthesis & \text{if } s = \llparenthesis a, b \rrparenthesis \land t = \llparenthesis c, d \rrparenthesis \land b \in t \land c \in s \\ \llparenthesis c, b \rrparenthesis & \text{if } s = \llparenthesis a, b \rrparenthesis \land t = \llparenthesis c, d \rrparenthesis \land d \in s \land a \in t \\ \llparenthesis a, d \rrparenthesis & \text{if } s = \llparenthesis a, b \rrparenthesis \land t = \llparenthesis c, d \rrparenthesis \land \\ & \quad (\# \llparenthesis b, c \rrparenthesis < \# \llparenthesis d, a \rrparenthesis \lor (\# \llparenthesis b, c \rrparenthesis = \# \llparenthesis d, a \rrparenthesis \land \ a \leq c)) \\ \llparenthesis c, b \rrparenthesis & \text{otherwise} \end{cases}$$

$$s \mathbin{\widetilde{\sqcap}} t = \begin{cases} s & \text{if } s \subseteq t \\ t & \text{if } t \subseteq s \\ \bot & \text{if } s = \llparenthesis a, b \rrparenthesis \land t = \llparenthesis c, d \rrparenthesis \land a \notin t \land b \notin t \land c \notin s \land d \notin s \\ \llparenthesis c, b \rrparenthesis & \text{if } s = \llparenthesis a, b \rrparenthesis \land t = \llparenthesis c, d \rrparenthesis \land b \in t \land c \in s \\ \llparenthesis a, d \rrparenthesis & \text{if } s = \llparenthesis a, b \rrparenthesis \land t = \llparenthesis c, d \rrparenthesis \land d \in s \land a \in t \\ \llparenthesis a, b \rrparenthesis & \text{if } s = \llparenthesis a, b \rrparenthesis \land t = \llparenthesis c, d \rrparenthesis \land \\ & \quad (\# \llparenthesis a, b \rrparenthesis < \# \llparenthesis c, d \rrparenthesis \lor (\# \llparenthesis a, b \rrparenthesis = \# \llparenthesis c, d \rrparenthesis \land \ a \leq c)) \\ \llparenthesis c, d \rrparenthesis & \text{otherwise} \end{cases}$$

$$s \mathbin{\utilde{\sqcap}} t = \overline{\overline{s} \mathbin{\widetilde{\sqcup}} \overline{t}}$$

$$s \mathbin{\utilde{\sqcup}} t = \overline{\overline{s} \mathbin{\widetilde{\sqcap}} \overline{t}}$$

Fig. 4. Over- and under-approximations of extreme bounds

```
function $\widetilde{\bigsqcup}(S)$
    $f \leftarrow g \leftarrow \bot$
    for $s \in S$ (in order of lex increasing left bound) do
        if $s = \top \lor (s = \llparenthesis x, y \rrparenthesis \land y \leq_0 x)$ then
            $f \leftarrow \mathsf{extend}(f, s)$
    for $s \in S$ (in order of lex increasing left bound) do
        $g \leftarrow \mathsf{bigger}(g, \mathsf{gap}(f, s))$
        $f \leftarrow \underline{\mathsf{extend}(f, s)}$
    return $\mathsf{bigger}(g, \overline{f})$
```

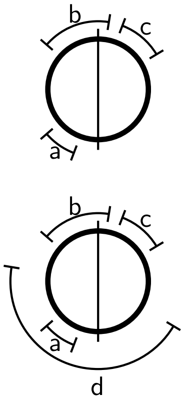


Fig. 5. Finding a minimal upper bound of a set of w-intervals

in such cases will sometimes give weaker results than the generalised approximate least upper bound or greatest lower bound operation [Gange et al. 2013a].

Figure 5 gives an algorithm for computing $\widetilde{\bigsqcup} S$. Intuitively, the algorithm returns the complement of the largest un-covered gap among intervals from S. It identifies this gap by passing through S once, picking intervals lexicographically by their left bounds. However, care must be taken to ensure that any *apparent* gaps, which are in fact covered by w-intervals that cross the south pole and may only be found later in the iteration, are not mistaken for *actual* gaps. We define the gap between two w-intervals as empty if they overlap, or otherwise the clockwise distance from the end of the first to the start of the second:

$$\mathsf{gap}(s, t) \ = \ \begin{cases} \overline{\llparenthesis c, b \rrparenthesis} & \text{if } s = \llparenthesis a, b \rrparenthesis \land t = \llparenthesis c, d \rrparenthesis \land b \notin t \land c \notin s \\ \bot & \text{otherwise} \end{cases}$$

The operation $\mathsf{extend}(s,t)$ produces the w-interval that runs from the start of s to the end of t, ensuring it includes all of s and t:

$$\mathsf{extend}(s,t) = \begin{cases} t & \text{if } s \subseteq t \\ s & \text{if } t \subseteq s \\ \top & \text{if } \overline{s} \subseteq t \\ \llparenthesis a, d \rrparenthesis & \text{otherwise, where } s = \llparenthesis a, b \rrparenthesis, t = \llparenthesis c, d \rrparenthesis \end{cases}$$

The operation $\mathsf{bigger}(s,t)$ is defined:

$$\mathsf{bigger}(s,t) = \begin{cases} t & \text{if } \# t > \# s \\ s & \text{otherwise} \end{cases}$$

The two loops in Figure 5 traverse the set of w-intervals in order of lexicographically increasing left bound; it does not matter where $\top$ and $\bot$ appear in this sequence. The first loop assigns to f the least upper bound of all w-intervals that cross the south pole. The invariant for the second loop is that g is the largest uncovered gap in f; thus the loop can be terminated as soon as $f = \top$. When the loop terminates, all w-intervals have been incorporated in f, so $\overline{f}$ is an uncovered gap, and g is the largest uncovered gap in f. Thus the result is the complement of the bigger of g and $\overline{f}$.

Consider Figure 5 (upper right) as an example. Here no intervals cross the south pole, so at the start of the second loop, $f = g = \bot$, and at the end of the loop, g is the gap between a and b, and f is the interval clockwise from the start of a to the right end of c. Since the complement of f is larger than g, the result in this case is f: the interval from the start of a to the end of c.

For the lower right example of Figure 5, interval d does cross the south pole, so at the start of the second loop, $f = \mathsf{d}$ and $g = \bot$. Now in the second loop, f extends clockwise to encompass b and c, and finally also d, at which point f becomes $\top$. But because the loop starts with $f = \mathsf{d}$, g never holds the gap between a and b; finally it holds the gap between the end of c and the start of d. Now the complement of f is smaller than g so the final result is the complement of g, that is, the interval from the start (right end) of d to the end of c.

The $\widetilde{\bigsqcup}$ operation is useful because it may preserve information that would be lost by repeated use of the over-join. Thus it should always be used when multiple w-intervals must be joined together, such as in the implementation of multiplication proposed in Section 3.4. In fact, a general strategy for improving the precision of analysis is to delay the application of over-joins until all the w-intervals to be joined have become available. Such delays allow multiple uses of $\tilde{\sqcup}$ to be replaced by a single use of $\widetilde{\bigsqcup}$.

While the over-lub of a set of w-intervals S is a smallest w-interval that covers all elements of S, the generalised *under*-lub $\underset{\sim}{\bigsqcup} S$ is a largest w-interval that is entirely covered by elements of S. That is, every value covered by $\underset{\sim}{\bigsqcup} S$ is covered by some element of S. An algorithm for $\underset{\sim}{\bigsqcup} S$ is given in Figure 6.

Like $\widetilde{\bigsqcup} S$ above, this algorithm works by scanning the intervals in S in order of increasing left bound. We keep track of the first contiguous interval f_0, the current interval f and the largest contiguous interval so far p. After we have processed all the intervals, it is possible that the last interval overlaps with the first; if this is the case, we combine the final f with f_0. The largest interval must then be either f or p. We make use of the predicate $\mathsf{overlap}(s,t)$ which checks if there is no gap between the end of s and the beginning of t:

$$\mathsf{overlap}(s,t) \equiv (s = \top) \vee (t = \top) \vee (s = \llparenthesis a, b \rrparenthesis \wedge t = \llparenthesis c, d \rrparenthesis \wedge c \leq_a b)$$

Consider Figure 6 (upper right). We start with $f_0 = a$ and begin scanning from the south pole. As there is no overlap between f_0 and b, the first **while** loop terminates, and we start

```
function $\underset{\sim}{\bigsqcup}([s_1, \ldots, s_n])$
    $[s_1, \ldots, s_n]$ are in order of lex increasing left bound
    $f_0 \leftarrow s_1$
    $i = 2$
    while $i \leq n \wedge \mathsf{overlap}(f_0, s_i)$ do
        $f_0 \leftarrow \mathsf{extend}(f_0, s_i)$
        $i \leftarrow i + 1$
    $p \leftarrow f \leftarrow f_0$
    while $i \leq n$ do
        if $\mathsf{overlap}(f, s_i)$ then
            $f \leftarrow \mathsf{extend}(f, s_i)$
        else
            $p \leftarrow \mathsf{bigger}(p, f)$
            $f \leftarrow s_i$
        $i \leftarrow i + 1$
    if $\mathsf{overlap}(f, f_0)$ then $f \leftarrow \mathsf{extend}(f, f_0)$
    return $\mathsf{bigger}(f, p)$
```

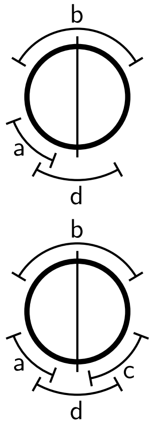


Fig. 6. Finding a maximal under-approximation of a union of w-intervals

the second with $f_0 = f = p = a$. After the first iteration, $f_0 = p = a, f = b$, and after the second and final iteration, $f_0 = a, p = b, f = d$. Finally, as f overlaps with f_0, we expand f to encompass a and d. However, as p is still larger than f, we have $\underset{\sim}{\bigsqcup}\{a, b, d\} = b$.

In the case of Figure 6 (lower right), we have the additional interval c. The algorithm proceeds exactly as before until the second iteration of the second `while` loop, where we encounter c. After this, $f_0 = a, p = b, f = c$. After the third and final iteration, because c and d overlap, $f_0 = a, p = b, f = \mathsf{extend}(c, d)$. We then combine f with f_0 as before; so, we have p covering b, and f covering a, c and d. As f is larger than p, we find $\underset{\sim}{\bigsqcup}\{a, b, c, d\}$ to be the interval from the beginning of c to the end of a.

Finally, the algorithms for $\widetilde{\bigsqcap}$ and $\underset{\sim}{\bigsqcap}$ can easily be defined by duality using the $\widetilde{\bigsqcup}$ and $\underset{\sim}{\bigsqcup}$ operations given above:

$$\widetilde{\bigsqcap} S = \overline{\underset{\sim}{\bigsqcup}\{\overline{s} \mid s \in S\}} \qquad\qquad \underset{\sim}{\bigsqcap} S = \overline{\widetilde{\bigsqcup}\{\overline{s} \mid s \in S\}}$$

The intersection of two w-intervals returns one or two w-intervals, and gives the exact intersection, in the sense that $\bigcup\{\gamma(u) \mid u \in s \cap t\} = \gamma(s) \cap \gamma(t)$.

$$s \cap t = \begin{cases} \varnothing & \text{if } s = \bot \text{ or } t = \bot \\ \{t\} & \text{if } s = t \vee s = \top \\ \{s\} & \text{if } t = \top \\ \{(\!|a, d|\!), (\!|c, b|\!)\} & \text{if } s = (\!|a, b|\!) \wedge t = (\!|c, d|\!) \wedge a \in t \wedge b \in t \wedge c \in s \wedge d \in s \\ \{s\} & \text{if } s = (\!|a, b|\!) \wedge t = (\!|c, d|\!) \wedge a \in t \wedge b \in t \\ \{t\} & \text{if } s = (\!|a, b|\!) \wedge t = (\!|c, d|\!) \wedge c \in s \wedge d \in s \\ \{(\!|a, d|\!)\} & \text{if } s = (\!|a, b|\!) \wedge t = (\!|c, d|\!) \wedge a \in t \wedge d \in s \wedge b \notin t \wedge c \notin s \\ \{(\!|c, b|\!)\} & \text{if } s = (\!|a, b|\!) \wedge t = (\!|c, d|\!) \wedge b \in t \wedge c \in s \wedge a \notin t \wedge d \notin s \\ \varnothing & \text{otherwise} \end{cases}$$

Finally, we define interval difference:

$$s \setminus t = s \,\widetilde{\sqcap}\, \overline{t}$$

3.4. Analysing arithmetic expressions

Addition and subtraction of w-intervals are defined as follows:

$$s + t = \begin{cases} \bot & \text{if } s = \bot \text{ or } t = \bot \\ (\!| a +_w c, b +_w d |\!) & \text{if } s = (\!| a, b |\!),\ t = (\!| c, d |\!), \text{ and } \# s + \# t \leq 2^w \\ \top & \text{otherwise} \end{cases}$$

$$s - t = \begin{cases} \bot & \text{if } s = \bot \text{ or } t = \bot \\ (\!| a -_w d, b -_w c |\!) & \text{if } s = (\!| a, b |\!),\ t = (\!| c, d |\!), \text{ and } \# s + \# t \leq 2^w \\ \top & \text{otherwise} \end{cases}$$

Here, to detect a possible overflow when adding the two cardinalities, standard addition is used. Note that $+_w$ and $-_w$ are signedness-agnostic: treating operands as signed or unsigned makes no difference. Multiplication on w-intervals is more cumbersome, even when we settle for a less-than-optimal solution. The reason is that even though unsigned and signed multiplication are the same operation on bit-vectors, signed and unsigned interval multiplication retain *different* information. The solution requires separating each interval at the north and south poles, so the segments agree on ordering for both signed and unsigned interpretations, and then performing both signed and unsigned multiplication on the fragments.

It is convenient to have names for the smallest w-intervals that straddle the poles. Define the north pole interval $\mathsf{np} = (\!| 01^{w-1}, 10^{w-1} |\!)$ and the south pole interval $\mathsf{sp} = (\!| 1^w, 0^w |\!)$. Define the north and south pole splits of a delimited w-interval as follows:

$$\mathsf{nsplit}(s) = \begin{cases} \varnothing & \text{if } s = \bot \\ \{(\!| a, b |\!)\} & \text{if } s = (\!| a, b |\!) \text{ and } \mathsf{np} \not\subseteq (\!| a, b |\!) \\ \{(\!| a, 01^{w-1} |\!), (\!| 10^{w-1}, b |\!)\} & \text{if } s = (\!| a, b |\!) \text{ and } \mathsf{np} \subseteq (\!| a, b |\!) \\ \{(\!| 0^w, 01^{w-1} |\!), (\!| 10^{w-1}, 1^w |\!)\} & \text{if } s = \top \end{cases}$$

$$\mathsf{ssplit}(s) = \begin{cases} \varnothing & \text{if } s = \bot \\ \{(\!| a, b |\!)\} & \text{if } s = (\!| a, b |\!) \text{ and } \mathsf{sp} \not\subseteq (\!| a, b |\!) \\ \{(\!| a, 1^w |\!), (\!| 0^w, b |\!)\} & \text{if } s = (\!| a, b |\!) \text{ and } \mathsf{sp} \subseteq (\!| a, b |\!) \\ \{(\!| 10^{w-1}, 1^w |\!), (\!| 0^w, 01^{w-1} |\!)\} & \text{if } s = \top \end{cases}$$

Then let the sphere cut be

$$\mathsf{cut}(u) = \bigcup \{\mathsf{ssplit}(v) \mid v \in \mathsf{nsplit}(u)\}$$

For example, $\mathsf{cut}((\!| 1111, 1001 |\!)) = \{(\!| 1111, 1111 |\!), (\!| 0000, 0111 |\!), (\!| 1000, 1001 |\!)\}$.

Unsigned $\times_u$ and signed $\times_s$ multiplication of two delimited w-intervals $(\!| a, b |\!)$ and $(\!| c, d |\!)$ that do not straddle poles are straightforward:

$$(\!| a, b |\!) \times_u (\!| c, d |\!) = \begin{cases} (\!| a \times_w c, b \times_w d |\!) & \text{if } b \times d - a \times c < 2^w \\ \top & \text{otherwise} \end{cases}$$

And, letting msb be the function that extracts the most significant bit from its argument:

$$(\!| a, b |\!) \times_s (\!| c, d |\!) = \begin{cases} (\!| a \times_w c, b \times_w d |\!) & \text{if } \mathsf{msb}(a) = \mathsf{msb}(b) = \mathsf{msb}(c) = \mathsf{msb}(d) \\ & \quad \wedge\ b \times d - a \times c < 2^w \\ (\!| a \times_w d, b \times_w c |\!) & \text{if } \mathsf{msb}(a) = \mathsf{msb}(b) = 1 \wedge \mathsf{msb}(c) = \mathsf{msb}(d) = 0 \\ & \quad \wedge\ b \times c - a \times d < 2^w \\ (\!| b \times_w c, a \times_w d |\!) & \text{if } \mathsf{msb}(a) = \mathsf{msb}(b) = 0 \wedge \mathsf{msb}(c) = \mathsf{msb}(d) = 1 \\ & \quad \wedge\ a \times d - b \times c < 2^w \\ \top & \text{otherwise} \end{cases}$$

Now, signed and unsigned bit-vector multiplication agree for segments that do not straddle a pole. This is an important observation which gives us a handle on precise multiplication

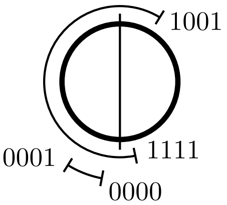


Fig. 7. Example 3.2's two intervals to be multiplied

across arbitrary delimited w-intervals:

$$\llparenthesis a, b \rrparenthesis \times_{us} \llparenthesis c, d \rrparenthesis = (\llparenthesis a, b \rrparenthesis \times_{u} \llparenthesis c, d \rrparenthesis) \cap (\llparenthesis a, b \rrparenthesis \times_{s} \llparenthesis c, d \rrparenthesis)$$

The use of intersection in this definition is the source of the added precision. Each of $\times_u$ and $\times_s$ gives a correct over-approximation of multiplication, and hence the intersection is also a correct over-approximation.

This now allows us to do general signedness-agnostic multiplication by joining the segments obtained from each piecewise hemisphere multiplication:

$$s \times t = \widetilde{\bigsqcup} \{m \mid u \in \mathsf{cut}(s), v \in \mathsf{cut}(t), m \in u \times_{us} v\}$$

Example 3.2. Consider the multiplication $\llparenthesis 1111, 1001 \rrparenthesis \times \llparenthesis 0000, 0001 \rrparenthesis$. The two multiplicand intervals are shown in Figure 7. The cut of the first w-interval is $\{\llparenthesis 1111, 1111 \rrparenthesis, \llparenthesis 0000, 0111 \rrparenthesis, \llparenthesis 1000, 1001 \rrparenthesis\}$, the cut of the second is $\{\llparenthesis 0000, 0001 \rrparenthesis\}$. The three separate segment multiplications give:

(1) $\llparenthesis 1111, 1111 \rrparenthesis \times_u \llparenthesis 0000, 0001 \rrparenthesis = \top$, (or, $[15, 15] \times [0, 1] = \top$)
$\llparenthesis 1111, 1111 \rrparenthesis \times_s \llparenthesis 0000, 0001 \rrparenthesis = \llparenthesis 1111, 0000 \rrparenthesis$ (or, $[-1, -1] \times [0, 1] = [-1, 0]$)
$\therefore \llparenthesis 1111, 1111 \rrparenthesis \times_{us} \llparenthesis 0000, 0001 \rrparenthesis = \{\llparenthesis 1111, 0000 \rrparenthesis\}$
(2) $\llparenthesis 0000, 0111 \rrparenthesis \times_u \llparenthesis 0000, 0001 \rrparenthesis = \llparenthesis 0000, 0111 \rrparenthesis$ (or, $[0, 7] \times [0, 1] = [0, 7]$)
$\llparenthesis 0000, 0111 \rrparenthesis \times_s \llparenthesis 0000, 0001 \rrparenthesis = \llparenthesis 0000, 0111 \rrparenthesis$ (or, $[0, 7] \times [0, 1] = [0, 7]$)
$\therefore \llparenthesis 0000, 0111 \rrparenthesis \times_{us} \llparenthesis 0000, 0001 \rrparenthesis = \{\llparenthesis 0000, 0111 \rrparenthesis\}$
(3) $\llparenthesis 1000, 1001 \rrparenthesis \times_u \llparenthesis 0000, 0001 \rrparenthesis = \llparenthesis 0000, 1001 \rrparenthesis$ (or, $[8, 9] \times [0, 1] = [0, 9]$)
$\llparenthesis 1000, 1001 \rrparenthesis \times_s \llparenthesis 0000, 0001 \rrparenthesis = \llparenthesis 1000, 0000 \rrparenthesis$ (or, $[-8, -7] \times [0, 1] = [-8, 0]$)
$\therefore \llparenthesis 1000, 1001 \rrparenthesis \times_{us} \llparenthesis 0000, 0001 \rrparenthesis$
$= \llparenthesis 0000, 1001 \rrparenthesis \cap \llparenthesis 1000, 0000 \rrparenthesis$
$= \{\llparenthesis 1000, 1001 \rrparenthesis, \llparenthesis 0000, 0000 \rrparenthesis\}$

Applying $\widetilde{\bigsqcup}$, we get the maximally precise result $\llparenthesis 1111, 1001 \rrparenthesis$ ($[15, 9]$ or $[-1, 9]$ depending on signedness). Note the crucial role played by $\times_{us}$ in obtaining this precision. For example, in the first case above, where we have no information about the result of unsigned multiplication ($\llparenthesis 1111, 1111 \rrparenthesis \times_u \llparenthesis 0000, 0001 \rrparenthesis = \top$), we effectively assume that multiplication is signed, obtaining a much tighter result. The role of $\times_{us}$ is to *do signed and unsigned multiplication simultaneously.* ■

Example 3.2 illustrates a rather important point. In Section 1 we showed how it can sometimes be advantageous to perform analysis assuming *signed* integers, while in other cases it is better to assume *unsigned* integers. It is natural to ask: Why not simply perform two analyses, one under each assumption, and combine the results? Example 3.2 shows clearly the weakness of this idea. For the example, both a "signed" analysis and an "unsigned" analysis yields $\top$. In an unsigned analysis, this happens since already $\llparenthesis 1111, 1111 \rrparenthesis \times_u \llparenthesis 0000, 0001 \rrparenthesis = \top$ (case 1). For a signed analysis, note that the three outcomes, $\llparenthesis 1111, 0000 \rrparenthesis$, $\llparenthesis 0000, 0111 \rrparenthesis$, and $\llparenthesis 1000, 0000 \rrparenthesis$, together span all possible values, so again the result is $\top$.

What is different, and important, about our approach is that the signed/unsigned case analysis happens at the "micro-level," throughout the computation, rather than performing the entire computation each way and choosing the best results. This is what we have in mind when we say that the abstract operations deal with superposed signed/unsigned states. The superposition idea is general and works for other operations. We can define *all* abstract operations by "segment case analysis" similar to Example 3.2's. However, this does not always add precision—many abstract operations can be captured using definitions that are equivalent to the case-by-case analysis, but simpler. Sometimes two cases suffice, sometimes one will do. As can be seen in the following, some operations need three cases (and apply cut), others need two (and apply ssplit or nsplit). We have already seen operations that require no segment case analysis at all (addition and subtraction).

Signed and unsigned division are different operations, owing to the need to round towards zero. For example, in unsigned 4-bit integer arithmetic, 1001/0010 yields 0100 (9/2 = 4), while in signed 4-bit integer arithmetic it yields 1101 (-7/2 = -3). We follow LLVM in calling signed and unsigned division sdiv and udiv, respectively.

For unsigned division we define:

$$\mathsf{udiv}(s,t) = \widetilde{\bigsqcup} \{u \,/_u\, (v \setminus \llparenthesis 0, 0 \rrparenthesis) \mid u \in \mathsf{ssplit}(s), v \in \mathsf{ssplit}(t)\}$$

where $/_u$ is defined in terms of usual unsigned integer division (note that, in this context, neither c nor d will be 0):

$$\llparenthesis a, b \rrparenthesis \,/_u\, \llparenthesis c, d \rrparenthesis = \llparenthesis a \,/_u\, d, b \,/_u\, c \rrparenthesis$$

Signed interval division is similarly defined:

$$\mathsf{sdiv}(s,t) = \widetilde{\bigsqcup} \{u \,/_s\, (v \setminus \llparenthesis 0, 0 \rrparenthesis) \mid u \in \mathsf{cut}(s), v \in \mathsf{cut}(t)\}$$

where

$$\llparenthesis a, b \rrparenthesis \,/_s\, \llparenthesis c, d \rrparenthesis = \begin{cases} \llparenthesis a \,/_s\, d, b \,/_s\, c \rrparenthesis & \text{if } \mathsf{msb}(a) = \mathsf{msb}(c) = 0 \\ \llparenthesis b \,/_s\, c, a \,/_s\, d \rrparenthesis & \text{if } \mathsf{msb}(a) = \mathsf{msb}(c) = 1 \\ \llparenthesis b \,/_s\, d, a \,/_s\, c \rrparenthesis & \text{if } \mathsf{msb}(a) = 0 \text{ and } \mathsf{msb}(c) = 1 \\ \llparenthesis a \,/_s\, c, b \,/_s\, d \rrparenthesis & \text{if } \mathsf{msb}(a) = 1 \text{ and } \mathsf{msb}(c) = 0 \end{cases}$$

Example 3.3. Signed-integer interval division $\mathsf{sdiv}(\llparenthesis 0100, 0111 \rrparenthesis, \llparenthesis 1110, 0011 \rrparenthesis)$ (that is, $\mathsf{sdiv}(\llparenthesis 4, 7 \rrparenthesis, \llparenthesis -2, 3 \rrparenthesis)$) yields $\llparenthesis 0001, 1110 \rrparenthesis$ (that is, $\llparenthesis 1, -2 \rrparenthesis$). In this case, the dividend straddles no pole, but the divisor straddles the south pole, and so is split into $\llparenthesis 1110, 1111 \rrparenthesis$ and $\llparenthesis 0001, 0011 \rrparenthesis$, the 0 having been made an endpoint by cut and excised by the difference operation. Now dividing $\llparenthesis 0100, 0111 \rrparenthesis$ by $\llparenthesis 1110, 1111 \rrparenthesis$ (and rounding towards 0) yields $\llparenthesis 1001, 1110 \rrparenthesis$ (that is, $\llparenthesis -7, -2 \rrparenthesis$). And dividing $\llparenthesis 0100, 0111 \rrparenthesis$ by $\llparenthesis 0001, 0011 \rrparenthesis$ yields $\llparenthesis 0001, 0111 \rrparenthesis$ (that is, $\llparenthesis 1, 7 \rrparenthesis$). Application of $\widetilde{\bigsqcup}$ will close the smallest gap between the two: $\llparenthesis 1001, 1110 \rrparenthesis \mathbin{\tilde{\sqcup}} \llparenthesis 0001, 0111 \rrparenthesis = \llparenthesis 0001, 1110 \rrparenthesis$. ∎

LLVM's remainder operations urem and srem are congruent with division's use of rounding towards 0, in the sense that they preserve the invariant $n = (n \,/_s\, k) \times k + \mathit{rem}(n,k)$ for all n and k. In particular, $\mathsf{srem}(n,k)$ has the same sign as n. The Intel X86 instruction set's `IDIV` instruction applied to signed integers, and `DIV` instruction applied to unsigned integers, behave similarly (these instructions yield both the quotient and remainder).

In practice, the remainder operations are almost always used with a fixed value k. In the interval versions, they are more unwieldy than the other arithmetic operations, lacking certain monotonicity properties. More precisely, even when arguments stay within hemispheres, the interval endpoints a, b, c, and d are not sufficient to determine the endpoints of

$\mathsf{urem}(\llparenthesis a, b \rrparenthesis, \llparenthesis c, d \rrparenthesis)$. For example, given the expression $\mathsf{urem}(\llparenthesis 3, 7 \rrparenthesis, \llparenthesis 4, 5 \rrparenthesis)$, and using '%' for the remainder operation on integers, the combinations $3 \,\%\, 4$, $3 \,\%\, 5$, $7 \,\%\, 4$, and $7 \,\%\, 5$ will only reveal the resulting values 2 and 3. However, values from the interval $\llparenthesis 3, 7 \rrparenthesis$ can also produce remainders 0, 1 and 4, when divided by 4 or 5.

Hence we design the abstract remainder operation $\mathsf{urem}(s, t)$ to ignore s, unless the result of the division $s \;/_u\; t$ is a singleton interval. If the result of division is not a singleton then the remainder is considered maximally ambiguous, that is, only bounded from above, by the largest possible modulus. So, defining

$$\mathsf{amb}\llparenthesis a, b \rrparenthesis = \llparenthesis 0, b - 1 \rrparenthesis$$

we have

$$\begin{aligned} \mathsf{urem}(s,t) &= \widetilde{\bigsqcup} \, \{u \;\;\%_u\; (v \setminus \llparenthesis 0, 0 \rrparenthesis) \mid u \in \mathsf{ssplit}(s), v \in \mathsf{ssplit}(t)\} \\ s \;\;\%_u\; t &= \begin{cases} s - (s \;/_u\; t) \times t & \text{if } \#\, (s \;/_u\; t) = 1 \\ \mathsf{amb}(t) & \text{otherwise} \end{cases} \end{aligned}$$

For example, $\mathsf{urem}(\llparenthesis 16, 18 \rrparenthesis, \llparenthesis 12, 14 \rrparenthesis) = \llparenthesis 2, 6 \rrparenthesis$, since $16/14 = 18/12 = 1$.

The case of $\mathsf{srem}(s, t)$ is similar, except for the need to make sure that the resulting sign is that of s.

$$\begin{aligned} \mathsf{srem}(s,t) &= \widetilde{\bigsqcup} \, \{u \;\;\%_s\; (v \setminus \llparenthesis 0, 0 \rrparenthesis) \mid u \in \mathsf{ssplit}(s), v \in \mathsf{ssplit}(t)\} \\ s \;\;\%_s\; t &= \begin{cases} s - (s \;/_s\; t) \times t & \text{if } \#\, (s \;/_s\; t) = 1 \\ \mathsf{sign}(s) \times \mathsf{amb}(\mathsf{sign}(t) \times t) & \text{otherwise} \end{cases} \end{aligned}$$

Here $\mathsf{sign}(s)$ is -1 if $\mathsf{msb}(s) = 1$, and 1 otherwise.

3.5. Analysing bit-manipulating expressions

For the logical operations, it is tempting to simply consider the combinations of interval endpoints, at least when no interval straddles two hemispheres, but that does not work. For example, the endpoints of $\llparenthesis 1010, 1100 \rrparenthesis$ are not sufficient to determine the endpoints of $\llparenthesis 1010, 1100 \rrparenthesis \mathbin{\mathsf{|}} \llparenthesis 0110, 0110 \rrparenthesis$. Namely, $1010 \mathbin{\mathsf{|}} 0110 = 1100 \mathbin{\mathsf{|}} 0110 = 1110$, but $1011 \mathbin{\mathsf{|}} 0110 = 1111$. Instead we use the unsigned versions of algorithms provided by Warren Jr [2003] (pages 58–62), but adapted to w-intervals using a south pole split. We present the method for bitwise-or $\mathsf{|}$; those for bitwise-and and bitwise-xor are similar.

$$s \mathbin{\mathsf{|}} t = \widetilde{\bigsqcup} \, \{u \mathbin{\mathsf{|}_w} v \mid u \in \mathsf{ssplit}(s), v \in \mathsf{ssplit}(t)\}$$

where $\mathsf{|}_w$ is Warren's *unsigned* bitwise or operation for intervals [Warren Jr 2003], an operation with complexity $O(w)$. Note that the signed and unsigned cases have different algorithms, both given by Warren Jr [2003]; Figure 8 shows how to compute the lower and upper bounds in the unsigned case.

Signed and zero extension are defined as follows. We assume words of width w are being extended to width $w + k$, with $k > 0$.

$$\begin{aligned} \mathsf{sext}(s,k) &= \widetilde{\bigsqcup} \, \{\llparenthesis (\mathsf{msb}(a))^k a, (\mathsf{msb}(b))^k b \rrparenthesis \mid \llparenthesis a, b \rrparenthesis \in \mathsf{nsplit}(s)\} \\ \mathsf{zext}(s,k) &= \widetilde{\bigsqcup} \, \{\llparenthesis 0^k a, 0^k b \rrparenthesis \mid \llparenthesis a, b \rrparenthesis \in \mathsf{ssplit}(s)\} \end{aligned}$$

Truncation of a bit-vector a to $k < w$ bits (integer downcasting), written $\mathsf{trunc}(a, k)$, keeps the lower k bits of a bit-vector of length w. Accordingly, we overload $\mathsf{trunc}(s, k)$ to denote

```
function minOr(unsigned a,b,c,d) {
  unsigned e, m = 0x80000000;

  while (m != 0) {
    if (~a & c & m) {
      e = (a | m) & -m;
      if (e <= b) { a = e; break; }
    }
    else if (a & ~c & m) {
      e = (c | m) & -m;
      if (e <= d) { c = e; break; }
    }
    m = m >> 1;
  }
  return a | c;
}
```

```
function maxOr(unsigned a,b,c,d) {
  unsigned e, m = 0x80000000;

  while (m != 0) {
    if (b & d & m) {
      e = (b - m) | (m - 1);
      if (e >= a) { b = e; break; }
      e = (d - m) | (m - 1);
      if (e >= c) { d = e; break; }
    }
    m = m >> 1;
  }
  return b | d;
}
```

Fig. 8. Warren's method (in C) for finding the bounds of $\llparenthesis a, b \rrparenthesis \mid \llparenthesis c, d \rrparenthesis$ in the unsigned case, $w = 32$

a w width w-interval s truncated to a k width w-interval. Truncation is defined as:

$$\mathsf{trunc}(s,k) = \begin{cases} \bot & \text{if } s = \bot \\ \llparenthesis \mathsf{trunc}(a,k), \mathsf{trunc}(b,k) \rrparenthesis & \text{if } s = \llparenthesis a, b \rrparenthesis \wedge a \texttt{>>}_a k = b \texttt{>>}_a k \\ & \quad \wedge \mathsf{trunc}(a,k) \leq \mathsf{trunc}(b,k) \\ \llparenthesis \mathsf{trunc}(a,k), \mathsf{trunc}(b,k) \rrparenthesis & \text{if } s = \llparenthesis a, b \rrparenthesis \wedge (a \texttt{>>}_a k) + 1 \equiv_{2^w} b \texttt{>>}_a k \\ & \quad \wedge \mathsf{trunc}(a,k) \not\leq \mathsf{trunc}(b,k) \\ \top & \text{otherwise} \end{cases}$$

where $\texttt{>>}_a$ is arithmetic right shift. Once truncation is defined, we can easily define left shift:

$$s \texttt{ << } k = \begin{cases} \bot & \text{if } s = \bot \\ \llparenthesis a \texttt{<<} k, b \texttt{<<} k \rrparenthesis & \text{if } \mathsf{trunc}(s, w-k) = \llparenthesis a, b \rrparenthesis \\ \llparenthesis 0^w, 1^{w-k} 0^k \rrparenthesis & \text{otherwise} \end{cases}$$

Logical right shifting ($\texttt{>>}_l$) requires testing if the south pole is covered:

$$s \texttt{ >>}_l\ k = \begin{cases} \bot & \text{if } s = \bot \\ \llparenthesis 0^w, 0^k 1^{w-k} \rrparenthesis & \text{if } \mathsf{sp} \subseteq s \\ \llparenthesis a \texttt{>>}_l k, b \texttt{>>}_l k \rrparenthesis & \text{if } s = \llparenthesis a, b \rrparenthesis \end{cases}$$

and arithmetic right shifting ($\texttt{>>}_a$) requires testing if the north pole is covered:

$$s \texttt{ >>}_a\ k = \begin{cases} \bot & \text{if } s = \bot \\ \llparenthesis 1^k 0^{w-k}, 0^k 1^{w-k} \rrparenthesis & \text{if } \mathsf{np} \subseteq s \\ \llparenthesis a \texttt{>>}_a k, b \texttt{>>}_a k \rrparenthesis & \text{if } s = \llparenthesis a, b \rrparenthesis \end{cases}$$

Shifting with variable shift, for example, $s \texttt{ << } t$, can be defined by calculating the (fixed) shift for each $k \in \llparenthesis 0, w-1 \rrparenthesis$ which is an element of t, and over-joining the resulting w-intervals.

3.6. Dealing with control flow

When dealing with signedness agnostic representations, comparison operations must be explicitly signed or unsigned. Taking the 'then' branch of a conditional with condition $s \leq_0 t$ can be thought of as prefixing the branch with the constraint '`assume` $s \leq_0 t$'. If we assume the program has been normalised such that s and t are variables, then we can tighten the bounds on s and t as they apply to statements only executed if this assumption holds. We compute s' and t' as updated versions of the bounds s and t, respectively, as follows:

$$s' = \begin{cases} \bot & \text{if } t = \bot \\ s & \text{if } 1^w \in t \\ s \mathbin{\widetilde{\sqcap}} \llparenthesis 0^w, b \rrparenthesis & \text{if } t = \llparenthesis a, b \rrparenthesis \end{cases}$$

$$t' = \begin{cases} \bot & \text{if } s = \bot \\ t & \text{if } 0^w \in s \\ t \mathbin{\widetilde{\sqcap}} \llparenthesis a, 1^w \rrparenthesis & \text{if } s = \llparenthesis a, b \rrparenthesis \end{cases}$$

Signed comparison ($\leq_{2^{w-1}}$) is similar, but replaces 1^w by 01^{w-1} and 0^w by 10^{w-1}. If either of s' and t' is $\bot$, we can conclude that the assumption is not satisfiable, so the following statements are unreachable.

It may then be possible to propagate these revised bounds back to the variables from which s and t were computed. For example, if we have the bounds $x = \llparenthesis 0000, 0111 \rrparenthesis$ when executing `s = x+1; t = 3; assume s` $\leq_0$ `t;` then we derive bounds $s = \llparenthesis 0001, 1000 \rrparenthesis, t = \llparenthesis 0011, 0011 \rrparenthesis$ before the assume, and $s' = \llparenthesis 0000, 0011 \rrparenthesis, t' = \llparenthesis 0011, 0011 \rrparenthesis$ after. From this, moreover, we can propagate backwards to derive the tighter bounds $x' = \llparenthesis 0000, 0010 \rrparenthesis$ for x after the assume (using the fact that `+` and `-` are inverse operations).

Finally, at confluence points in the program, such as φ-nodes in LLVM or targets of multiple jumps in assembler, we use over-lub $\widetilde{\bigsqcup}$ to combine bounds from multiple sources.

4. RELATIONSHIP WITH OTHER DOMAINS

In this section, we establish the relationship between wrapped intervals and a range of common value domains. For convenience, we use m to denote the modulus of a given integer domain. We can compare abstract domains with respect to expressiveness. Given abstract domains $\mathcal{A}$ and $\mathcal{B}$ approximating a set of values $\mathbb{V}$, we say $\mathcal{A}$ is at least as expressive as $\mathcal{B}$ (denoted $\mathcal{A} \preceq \mathcal{B}$) if, for every element $y \in \mathcal{B}$ approximating a set $S \subseteq \mathbb{V}$, there is some element $x \in \mathcal{A}$ such that x approximates S, and $\gamma_{\mathcal{A}}(x) \subseteq \gamma_{\mathcal{B}}(y)$. Two domains are incomparable iff $A \not\preceq B$ and $B \not\preceq A$. They are equivalent, denoted $\cong$, iff $A \preceq B$ and $B \preceq A$.

Given a finite (although possibly quite large) set of possible values $\mathbb{V}$, the most expressive possible abstract domain is the power-set domain $\mathcal{P}(\mathbb{V})$.

PROPOSITION 4.1. *For $m \leq 3$, the wrapped-interval domain $\mathcal{W}_m \cong \mathcal{P}(\mathbb{Z}_m)$.*

PROOF. For $m \leq 2$ or intervals of size 1, this is trivial, since $\mathcal{W}_m$ includes $\top$, $\bot$ and all singletons. The following table shows that $\mathcal{W}_3$ can express all elements of $\mathcal{P}(\mathbb{Z}_3)$:

Set	Interval	Set	Interval
$\varnothing$	$\bot$	$\{0, 1\}$	$\llparenthesis 0, 1 \rrparenthesis$
$\{0\}$	$\llparenthesis 0, 0 \rrparenthesis$	$\{0, 2\}$	$\llparenthesis 2, 0 \rrparenthesis$
$\{1\}$	$\llparenthesis 1, 1 \rrparenthesis$	$\{1, 2\}$	$\llparenthesis 1, 2 \rrparenthesis$
$\{2\}$	$\llparenthesis 2, 2 \rrparenthesis$	$\{0, 1, 2\}$	$\top$

Therefore, $\mathcal{W}_m$ is exact for $m \leq 3$. □

For larger m, $\mathcal{W}_m$ cannot express $\{0, 2\}$ exactly, so is less expressive than $\mathcal{P}(\mathbb{Z}_m)$.

One may wonder whether wrapped intervals are equivalent to some finite partitioning of the number circle, or some reduced product of classical intervals. We introduce the notation $\mathcal{I}_m^k$ denote a classical interval domain with the fixed wrapping point k (so the unsigned interval domain is $\mathcal{I}_m^0$, and the signed version is $\mathcal{I}_m^{m/2}$). Let $\mathcal{R}_m^{\{k_1,\ldots,k_p\}}$ denote the reduced product $\mathcal{I}_m^{k_1} \times \cdots \times \mathcal{I}_m^{k_p}$.

PROPOSITION 4.2. *For $m > 3$, $\mathcal{W}_m$ is incomparable with $\mathcal{R}_m^K$ for $1 < |K| < \frac{m}{2}$.*

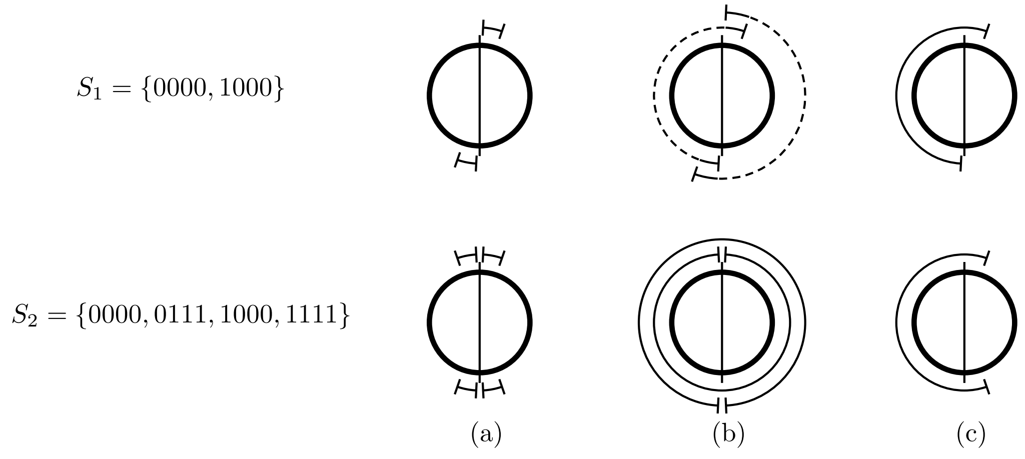


Fig. 9. For two sets S_1 and S_2 of 4-bit values, we show (a) the concrete values, and best approximations under (b) $\mathcal{I}^0_{16} \times \mathcal{I}^8_{16}$ and under (c) $\mathcal{W}_{16}$.

Proof. Let $K = \{k_1, \ldots, k_p\}$ with the k_i in ascending order. As $p < \frac{m}{2}$, there must be some adjacent pair of elements k_i, k_j, where $k_j - k_i \geq 3$.

Consider the set $S = \{k_i + 1, k_j\}$. Under $\mathcal{I}^{k_i}_m$, S is approximated by $[k_i + 1, k_j]$; the approximation under $\mathcal{I}^{k_j}_m$ is $[k_j, k_i + 1]$. The concrete intersection of these intervals is exactly S, so this set can be exactly represented under $\mathcal{R}^K_m$. The possible approximations under $\mathcal{W}_m$ are $\llparenthesis k_i + 1, k_j \rrparenthesis$ and $\llparenthesis k_j, k_i + 1 \rrparenthesis$. The former contains $k_i + 2$, and the latter contains k_i. Therefore $\mathcal{W}_m$ cannot represent S exactly, so $\mathcal{W}_m \not\preceq \mathcal{R}^K_m$.

Now consider the set $S' = \{k_1 - 1, k_1, k_2 - 1, k_2, \ldots, k_p - 1, k_p\}$. This set is covered by the wrapped interval $[k_j - 1, k_i]$, which excludes (at least) $k_i + 1$. For each component domain $\mathcal{I}^k_m$, we have $k \in S', k - 1 \in S'$; so the best approximation under $\mathcal{I}^k_m$ is $\top$. As such, the approximation under $\mathcal{R}^K_m$ is also $\top$. Therefore, $\mathcal{R}^K_m \not\preceq \mathcal{W}_m$.

As $\mathcal{W}_m \not\preceq \mathcal{R}^K_m$, and $\mathcal{R}^K_m \not\preceq \mathcal{W}_m$, the two domains are incomparable. □

A corollary is that the wrapped interval domain $\mathcal{W}_m$ is *incomparable* with the reduced product of signed and unsigned analysis $(\mathcal{I}^0_m \times \mathcal{I}^{m/2}_m)$. Figure 9 gives two concrete examples of this behaviour over 4-bit values ($\mathbb{Z}_{16}$). Note how the reduced product gives a superior representation of $S_1 = \{0000, 1000\}$: Taking the intersection of the two segments for S_1 in column (b) eliminates the dashed portions and gives back S_1 exactly, whereas the wrapped interval approximation (c) contains additional values, such as 0011. On the other hand, the reduced product is inferior in the case of $S_2 = \{0000, 0111, 1000, 1111\}$. Column (b) shows how the reduced product conflates S_2 with $\top$, and column (c) shows the more precise wrapped interval that results. Also note that if the largest gap between elements of K is 2 (which is possible with $|K| \geq \frac{m}{2}$), we have $\mathcal{R}^K_m \cong \mathcal{P}(\mathbb{Z}_m)$.

While wrapped analysis is incomparable with the reduced product of up to $m/2$ classical interval analyses, it would be nice to think that wrapped intervals were uniformly more accurate than a single interval analysis, such as $\mathcal{I}^{m/2}_m$ or $\mathcal{I}^0_m$. Unfortunately this it not necessarily the case. Consider the approximation of the expression $(\{0110\} \cup \{1001\}) \cap \{0110, 0111\}$. In $\mathcal{I}^8_{16}$ the calculation $([0110, 0110] \sqcup [1001, 1001]) \sqcap [0110, 0111]$ yields $[1001, 0110]$ as the result of the $\sqcup$ and finally $[0110, 0110]$. In $\mathcal{W}_{16}$ the calculation $(\llparenthesis 0110, 0110 \rrparenthesis \mathbin{\widetilde{\sqcup}} \llparenthesis 1001, 1001 \rrparenthesis) \mathbin{\widetilde{\sqcap}} \llparenthesis 0110, 0111 \rrparenthesis$ yields $\llparenthesis 0110, 1001 \rrparenthesis$ as the result of the $\widetilde{\sqcup}$ and finally $\llparenthesis 0110, 0111 \rrparenthesis$. Hence the wrapped interval analysis can be less accurate since it can choose an incomparable result of the join, which turns out later to give less precise results. Of course we can easily modify wrapped interval analysis to always prefer a wrapped interval that does not cross the north pole where possible. With that, wrapped interval analysis is uniformly more accurate than signed interval analysis, since if all descriptions cross the north pole then the signed interval analysis must return $\top$.

In the following, we assume that transfer functions over $\mathcal{I}_m^{m/2}$ and $\mathcal{W}_m$ coincide over the unwrapped subset of $\mathcal{W}_m$. That is, for a function f and arguments $x_1, \ldots, x_n \in \mathcal{I}_m^{m/2}$:

$$\begin{array}{l} f_I(x_1, \ldots, x_n) = \top \Leftrightarrow \llparenthesis \frac{m}{2} - 1, \frac{m}{2} \rrparenthesis \in f_{\mathcal{W}}(x_1, \ldots, x_n) \\ f_I(x_1, \ldots, x_n) \neq \top \Rightarrow f_I(x_1, \ldots, x_n) = f_{\mathcal{W}}(x_1, \ldots, x_n) \end{array}$$

Given implementations of f_I and $f_{\mathcal{W}}$ which do not necessarily coincide on $\mathcal{I}_m^{m/2}$, we can still construct strengthened versions f'_I and $f'_{\mathcal{W}}$ that satisfy this requirement (assuming $\widetilde{\sqcap}$ is north-biased) as follows:

$$\begin{array}{rcl} f'_I(X) & = & \begin{cases} f_I(X) \sqcap f_{\mathcal{W}}(X) & \textbf{if } f_{\mathcal{W}}(X) \in \mathcal{I}_m^{m/2} \\ f_I(X) & \textbf{otherwise} \end{cases} \\ f'_{\mathcal{W}}(X) & = & f_I(X) \,\widetilde{\sqcap}\, f_{\mathcal{W}}(X) \end{array}$$

We also require all operations to be monotone with respect to $\top$.

Definition 4.3. A function f over a partially ordered set $(\mathcal{O}, \sqsubseteq)$ is *monotone with respect to* an element x iff $\forall x' \,.\, x' \sqsubseteq x \Rightarrow f(x') \sqsubseteq f(x)$, and $\forall x' \,.\, x \sqsubseteq x' \Rightarrow f(x') \sqsubseteq f(x)$.

Hence f is monotone iff it is monotone with respect to all elements of $\mathcal{O}$. It is not hard to see that all operations on $\mathcal{W}_m$ are monotone with respect to $\top$.

THEOREM 4.4. *$\mathcal{W}_m$ modified to favor intervals that do not straddle the north pole is uniformly more accurate than $\mathcal{I}_m^{m/2}$.*

PROOF. Assume there is some function f for which $f_{\mathcal{W}}$ is not strictly more accurate than f_I. Then, there must be some elements $X \in (\mathcal{I}_m^{m/2})^n$, $X' \in \mathcal{W}_m^n$ such that:

$$\forall\, i \in [1, n] \,.\, x'_i \sqsubseteq x_i,\ f_{\mathcal{W}}(X) \not\sqsubseteq f_{\mathcal{I}}(X')$$

We have $x_i \in \mathcal{I}_m^{m/2}$ and $x'_i \sqsubseteq x_i$ for all i. Hence either $x'_i \in \mathcal{I}_m^{m/2}$, or $x_i = \top$. Now define Y as follows:

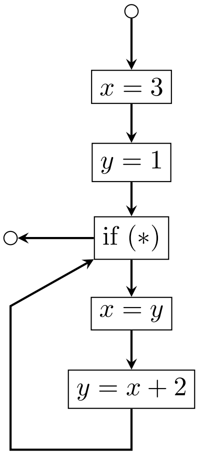
$$y_i = \begin{cases} x'_i & \textbf{if } x'_i \in \mathcal{I}_m^{m/2} \\ \top & \textbf{otherwise} \end{cases}$$

Then we have $X' \sqsubseteq Y \sqsubseteq X$. As f_I and $f_{\mathcal{W}}$ coincide over $\mathcal{I}_m^{m/2}$, we have $f_{\mathcal{W}}(Y) \sqsubseteq f_I(X)$. But since $f_{\mathcal{W}}$ is monotone with respect to $\top$, and Y differs from X' only in elements that are $\top$, we have $f_{\mathcal{W}}(X') \sqsubseteq f_{\mathcal{W}}(Y)$. Therefore, $f_{\mathcal{W}}(X') \sqsubseteq f_{\mathcal{I}}(X)$. □

5. NON-TERMINATION AND WIDENING

Let us now revisit the issue mentioned in Section 3, namely that $\widetilde{\sqcup}$ is neither associative nor monotone. Although the set of w-intervals is finite, the fact that $\widetilde{\sqcup}$ is not monotone raises a major problem: a *least fixed point* may not exist because multiple fixed points could be equally precise, and even worse, when $\widetilde{\sqcup}$ is used in the role of a join operator, the analysis may not terminate.

Figure 10 shows an example where, for simplicity, we assume that x and y are 2-bit integers. In annotating program points we use '1' for the w-interval $\llparenthesis 01, 01 \rrparenthesis$, '3' for $\llparenthesis 11, 11 \rrparenthesis$, '$\overline{0}$' for $\llparenthesis 01, 11 \rrparenthesis$, and '$\overline{2}$' for $\llparenthesis 11, 01 \rrparenthesis$. Note that the result of round 5 is identical to the result



0	**1**	**2**	**3**	**4**	**5**
$(\top,\top)$	$(\top,\top)$	$(\top,\top)$	$(\top,\top)$	$(\top,\top)$	$(\top,\top)$
$(\bot,\bot)$	$(3,\top)$	$(3,\top)$	$(3,\top)$	$(3,\top)$	$(3,\top)$
$(\bot,\bot)$	$(3,1)$	$(3,1)$	$(3,1)$	$(3,1)$	$(3,1)$
$(\bot,\bot)$	$(3,1)$	$(\overline{2},\overline{2})$	$(\overline{2},\overline{0})$	$(\overline{0},\overline{2})$	$(\overline{2},\overline{0})$
$(\bot,\bot)$	$(1,1)$	$(\overline{2},\overline{2})$	$(\overline{0},\overline{0})$	$(\overline{2},\overline{2})$	$(\overline{0},\overline{0})$
$(\bot,\bot)$	$(1,3)$	$(\overline{2},\overline{0})$	$(\overline{0},\overline{2})$	$(\overline{2},\overline{0})$	$(\overline{0},\overline{2})$

Fig. 10. Non-terminating analysis; column i shows (x, y) in round i

of round 3, hence the result will oscillate forever between the annotations given by columns 3 and 4.

While this pathological behaviour can be expected to be rare, a correct and terminating analysis still must take the possibility into account.

In practice, there is an easy solution to the non-termination problem. Since the w-interval domain contains chains of length $O(2^w)$, acceleration is required anyway for practical purposes, even though the domain is finite. Therefore, it seems reasonable to apply a widening operator. The use of widening will ensure termination in our analysis, avoiding the non-monotonicity problem of $\widetilde{\sqcup}$.

In the classical setting, we have a (collecting) semantic domain and an abstract domain, both assumed to be lattices, and a pair (α, γ) of adjoined functions. However, the concept of a Galois connection makes sense also if we define it as a pair of mappings between two posets, or even preordered sets. For now assume that $(A, \sqsubseteq)$ and $(C, \leq)$ are posets. The pair $\alpha : C \to A$ and $\gamma : A \to C$ form a Galois connection iff

$$\alpha(x) \sqsubseteq y \Leftrightarrow x \leq \gamma(y) \tag{1}$$

From this condition follows that (a) α and γ are monotone, (b) $\alpha(\gamma(y)) \sqsubseteq y$ for all $y \in A$, and (c) $x \leq \gamma(\alpha(x))$ for all $x \in C$. In fact, taken together, (a)–(c) are equivalent to (1) [Cousot and Cousot 1977].

If α moreover is surjective it follows that $\alpha \circ \gamma$ is the identity function, and in this case we talk about a Galois surjection. Galois surjections are common in applications to program analysis. However, there are natural examples in program analysis where a non-surjective Galois connection is used.[3]

We usually also assume that we are dealing with (complete) lattices C and A. Having lattices in itself does not guarantee the existence of a Galois connection.

Let C (the concrete domain) be a finite-height meet-semilattice and let $f : C \to C$ be monotone. Let A (the abstract domain) be a partially ordered set with least element $\bot_A$,

[3] An example is given by King and Søndergaard [2010] who abstract a Boolean function to its "congruent closure", as part of a scheme to improve affine congruence analysis [Granger 1991].

and let $g : A \to A$ be a (not necessarily monotone) function approximating f, that is,

$$\forall y \in A \ (f(\gamma(y)) \sqsubseteq \gamma(g(y))) \tag{2}$$

Consider a "g-cycle" $Y = \{y_0, \ldots, y_{m-1}\} \subseteq A$. By this we mean that the set Y satisfies

$$0 \leq i < m \Rightarrow g(y_i) = y_{i+1 \bmod m}$$

Now letting $x_0 = \bigsqcap_{0 \leq i < m} \gamma(y_i)$, we have:

$$\begin{array}{rcll} f(x_0) & \sqsubseteq & f(\gamma(y_i)) & \text{for all } 0 \leq i < m \text{, by monotonicity of } f \\ & \sqsubseteq & \gamma(g(y_i)) & \text{for all } 0 \leq i < m \text{, by (2)} \\ & = & \gamma(y_{i+1 \bmod m}) & \text{for all } 0 \leq i < m \end{array}$$

Hence $f(x_0) \sqsubseteq \bigsqcap_{0 \leq i < m} \gamma(y_i) = x_0$. 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 C has finite height, $\mathit{lfp}(f) \sqsubseteq x_0$. In other words, each element of the g-cycle is a correct result.

Hence we could solve the problem of possible oscillation by checking for cycles at each iteration. This would mean performing Kleene iteration over g as usual, generating the sequence of elements $\bot_A, g(\bot_A), g(g(\bot_A)), \ldots$. Call this sequence $g_0, g_1, g_2, \ldots$. For each $i > 0$, check whether $g_{i-1} \sqsubseteq g_i$. If so, continue as usual; if not, apply loop checking in the evaluation of g_{i+1} and subsequent elements.

In practice, however, we only encounter cycles with constructed, pathological examples. For this reason, it seems acceptable to apply a less precise approach in the form of widening, in particular since this is required anyway, to accelerate convergence of the analysis. Although the set of w-intervals is finite, it contains chains of length $O(2^w)$, and acceleration is regularly needed.

Hence we define an upper bound operator ∇, based on the idea of widening by (roughly) doubling the size of a w-interval. First, $s \nabla \bot = \bot \nabla s = s$, and $s \nabla \top = \top \nabla s = \top$. Additionally,

$$\llparenthesis u, v \rrparenthesis \nabla \llparenthesis x, y \rrparenthesis = \begin{cases} \llparenthesis u, v \rrparenthesis & \text{if } \llparenthesis x, y \rrparenthesis \subseteq \llparenthesis u, v \rrparenthesis \\ \top & \text{if } \# \llparenthesis u, v \rrparenthesis \geq 2^{w-1} \\ \llparenthesis u, y \rrparenthesis \mathbin{\widetilde{\sqcup}} \llparenthesis u, 2v -_w u +_w 1 \rrparenthesis & \text{if } \llparenthesis u, v \rrparenthesis \mathbin{\widetilde{\sqcup}} \llparenthesis x, y \rrparenthesis = \llparenthesis u, y \rrparenthesis \\ \llparenthesis x, v \rrparenthesis \mathbin{\widetilde{\sqcup}} \llparenthesis 2u -_w v -_w 1, v \rrparenthesis & \text{if } \llparenthesis u, v \rrparenthesis \mathbin{\widetilde{\sqcup}} \llparenthesis x, y \rrparenthesis = \llparenthesis x, v \rrparenthesis \\ \llparenthesis x, y \rrparenthesis \mathbin{\widetilde{\sqcup}} \llparenthesis x, x +_w 2v -_w 2u +_w 1 \rrparenthesis & \text{if } u \in \llparenthesis x, y \rrparenthesis \wedge v \in \llparenthesis x, y \rrparenthesis \\ \top & \text{otherwise} \end{cases}$$

Then ∇ is an upper bound operator [Nielson et al. 1999] and we have the property

$$s \nabla t = s \ \vee \ s \nabla t = \top \ \vee \ \# s \nabla t \geq 2 \# s$$

Given $f : \mathcal{W}_{2^w} \to \mathcal{W}_{2^w}$, we define the accelerated sequence $\{f_\nabla^n\}_n$ as follows:

$$f_\nabla^n = \begin{cases} \bot & \text{if } n = 0 \\ f_\nabla^{n-1} & \text{if } n > 0 \wedge f(f_\nabla^{n-1}) \sqsubseteq f_\nabla^{n-1} \\ f_\nabla^{n-1} \nabla f(f_\nabla^{n-1}) & \text{otherwise} \end{cases}$$

Since $\{f_\nabla^n\}_n$ is increasing (whether f is monotone or not) and $\mathcal{W}_{2^w}$ has finite height, the accelerated sequence eventually stabilises. It is undesirable to widen at every iteration, since it gives away precision too eagerly. However, as observed by Gange et al. [2013a], the common practise of widening every $n > 1$ iterations is unsafe for non-lattice domains such as w-intervals, because it is possible that such a sequence will not terminate. Our implementation performs normal Kleene iteration for the first five steps; if that does not find a fixed point, we begin widening at every step. Gange et al. [2013a] discuss several alternative strategies.

Table I. Comparison between unwrapped and wrapped interval analyses with options `-widening 5 -narrowing 2`

Program	T_U	T_W	$\frac{\mathsf{T}_W}{\mathsf{T}_U}$	I	P_U	$\frac{\mathsf{P}_U}{\mathsf{I}}$	P_W	$\frac{\mathsf{P}_W}{\mathsf{I}}$	G_W	$\frac{\mathsf{G}_W}{\mathsf{I}}$
164.gzip	$0.09s$	$0.22s$	2.4	1511	272	18%	309	20%	44	3%
175.vpr	$0.38s$	$1.46s$	3.8	4143	321	8%	378	9%	57	1%
176.gcc	$2.10s$	$4.42s$	2.1	16711	5147	31%	5683	34%	570	3%
186.crafty	$1.19s$	$2.15s$	1.8	17679	3411	19%	3960	22%	562	3%
197.parser	$0.55s$	$1.96s$	3.6	4736	377	8%	445	9%	76	2%
255.vortex	$1.16s$	$2.42s$	2.1	22813	887	4%	974	4%	88	0%
256.bzip2	$0.35s$	$1.01s$	2.9	2529	411	16%	483	19%	86	3%
300.twolf	$0.07s$	$0.20s$	2.9	730	16	2%	20	3%	4	1%

6. EXPERIMENTAL EVALUATION

We implemented wrapped interval analysis for LLVM 3.0 and ran experiments on an Intel Core with a 2.70Gz clock and 8 GB of memory. For comparison we also implemented an unwrapped fixed-width interval analysis using the same fixed point algorithm. Since we analyse LLVM IR, signedness information is in general not available. Therefore, to compare the precision of "unwrapped" and "wrapped" analysis, we ran the unwrapped analysis assuming all integers are signed, similarly to Teixera and Pereira [2011]. We used the Spec CPU 2000 benchmark suite widely used by LLVM testers. The code for the analyses and the fixed point engine is publicly available at `http://code.google.com/p/wrapped-intervals/`.

Tables I, II, and III show our evaluation results. Columns T_U and T_W show analysis times (average of 5 runs) for the unwrapped and wrapped interval analysis, respectively. Column I shows the total number of integer intervals considered by the analyses, Column P_U shows the number of cases where the unwrapped analysis infers a delimited interval, and P_W does the same for wrapped intervals. Finally, column G_W shows the number of cases in which the wrapped analysis gave a more precise result (it is never less precise). In some cases, both analyses produce delimited intervals, but the wrapped interval is more precise. For instance, for 164.gzip (Table I), there are 7 such cases. This explains why, in most cases, $\mathsf{G}_W > \mathsf{P}_W - \mathsf{P}_U$.

Table I shows our results[4] when widening is only triggered if an interval has not stabilized after five fixed point iterations. We implement narrowing simply as two further iterations of abstract interpretation over the whole program once a fixed point is reached. We have tested with greater widening and narrowing values but we did not observe any significant change in terms of precision.

We note that both analyses are fast, and the added cost of wrapped analysis is reasonable. Regarding precision, the numbers of proper intervals (P_U and P_W) are remarkably low compared with the total number of tracked intervals (I). There are three main reasons for this. First, our analysis is intra-procedural only. Second, it does not track global variables or pointers. Third, several instructions that cast non-trackable types (for example, `ptrtoint`, `fptosi`) are not supported. In spite of these limitations, the numbers in column G_W show that wrapped interval analysis does infer better bounds.

In our second experiment (Table II) we tried to mitigate two of the limitations while preserving the widening/narrowing parameter values. The `-instcombine` option uses an intra-procedural LLVM optimization that can remove unnecessary casting instructions by combining two or more instructions into one. The option `-inline 300` mitigates the lack of inter-procedural analysis by performing function inlining if the size of the function is less than 300 instructions but only if LLVM considers it safe to inline them (function pointers cannot be inlined, for example). These two optimizations pay off: the number of

[4] These numbers are the closest to our previous experiment published in Navas et al. [2012]. The main differences with Navas et al. [2012] are due to two factors: different widening/narrowing parameter values and some changes after fixing some bugs.

Table II. Comparison between unwrapped and wrapped interval analyses with options `-widening 5 -narrowing 2 -instcombine -inline 300`

Pgm	T_U	T_W	$\frac{T_W}{T_U}$	I	P_U	$\frac{P_U}{I}$	P_W	$\frac{P_W}{I}$	G_W	$\frac{G_W}{I}$
164	0.25	0.51	2.04	2781	558	20%	649	23%	101	3%
175	0.97	3.58	3.69	7678	790	10%	1014	13%	283	3%
176	10.15	18.96	1.86	92791	26649	28%	32035	34%	5418	5%
186	1.83	3.52	1.92	24118	5949	24%	6842	28%	918	3%
197	0.96	3.03	3.15	6672	938	14%	1255	18%	340	5%
255	2.02	3.74	1.85	37120	2593	6%	2817	7%	225	0%
256	0.29	1.04	3.58	2447	436	17%	535	21%	113	4%
300	1.56	4.56	2.92	17812	654	3%	979	5%	326	1%

Table III. Comparison between unwrapped and wrapped interval analyses with options `-widening 5 -narrowing 2 -instcombine -inline 300 -enable-optimizations`

Pgm	T_U	T_W	$\frac{T_W}{T_U}$	I	P_U	$\frac{P_U}{I}$	P_W	$\frac{P_W}{I}$	G_W	$\frac{G_W}{I}$
164	0.18	0.55	3.05	2580	474	18%	543	21%	73	2%
175	0.90	3.32	3.68	6942	610	8%	804	11%	225	3%
176	10.26	19.19	1.87	94943	26641	28%	31718	33%	5099	5%
186	1.66	3.30	1.98	22787	5321	23%	6068	26%	761	3%
197	0.91	2.98	3.27	6744	742	11%	1050	15%	314	4%
255	2.27	4.87	2.14	36981	2599	7%	2800	7%	202	0%
256	0.27	0.92	3.40	2252	378	16%	454	20%	90	3%
300	1.80	5.06	2.81	16475	452	2%	657	3%	205	1%

Table IV. Number of ties comparing with total number of $\widetilde{\sqcup}$ for the three different options used in Table I, Table II, and Table III

Program	Table I Options		Table II Options		Table III Options	
	Joins	Ties	Joins	Ties	Joins	Ties
164.gzip	20,695	109	22,414	149	29,170	93
175.vpr	79,506	213	199,807	492	176,584	522
176.gcc	878,707	3102	1,377,207	5912	1,415,847	6782
186.crafty	83,668	728	143,466	667	136,924	939
197.parser	95,149	124	136,156	736	140,313	798
255.vortex	250,409	194	312,739	603	929,907	764
256.bzip2	44,613	129	44,783	130	43,898	134
300.twolf	224,425	274	247,680	306	894,890	353

proper intervals increases significantly, both in the unwrapped and in the wrapped case. The analysis time also increases, for each analysis. Note that we only show analysis times of the wrapped and unwrapped analyses, and we omit the analyses times of the LLVM optimizations. The number of variables for which wrapped analysis gave a more precise result is much higher than in the previous experiment, for a reasonable cost in time.

Our third experiment (Table III) repeats the same previous experiment, but with the option `-enable-optimizations`. This option allows other LLVM optimizations such as constant propagation and dead code elimination. Interestingly, these optimizations appear to have little impact, whether we consider precision or time, for either analysis.

Finally, Table IV shows the number of ties that needed to be resolved during the calculation of over-joins, $\widetilde{\sqcup}$. Whenever there is a tie, we choose the interval that avoids covering the north pole. We also ran the same experiment (but only for Table III Options) where the opposite choice is made. For that experiment, all results remained the same, except for 176.gcc where G_W came out as 5392 rather than 5418. There were no cases where unwrapped intervals produced more precise results than wrapped intervals.

Table V. Comparison between unwrapped and wrapped interval analyses in the context of removing redundant instrumentation using the IOC tool

Program	without LLVM optimizations			with LLVM optimizations		
	T	R_U	R_W	T	R_U	R_W
164.gzip	365	210	210	228	55	57
175.vpr	741	136	136	1149	70	70
176.gcc	3027	1442	1471	3030	512	544
186.crafty	3309	1440	1442	1680	265	265
197.parser	751	177	180	727	34	36
255.vortex	746	485	485	288	11	11
256.bzip2	676	271	271	486	85	85
300.twolf	3297	476	479	2873	34	37

7. AN APPLICATION: REMOVAL OF REDUNDANT INSTRUMENTATION

The experiments reported in the previous section show that, on real-world programs, signedness-agnostic wrapped interval analysis finds tighter bounds in many cases, compared to unwrapped, but sound, fixed-width integer interval analysis. However, the results in Tables I–III do not say exactly how much tighter the intervals are, nor does it follow from the results that there are realistic applications (such as program verification) that are able to capitalize on the tighter bounds. To address this, we have conducted a simple experiment with code that has been instrumented by the Integer Overflow Checker IOC [Dietz et al. 2012]. IOC instruments each arithmetic instruction that can yield signed integer overflow, injecting *trap handlers*. As we use LLVM 3.0, we installed its corresponding IOC version and compiled programs with option `-fcatch-undefined-ansic-behavior`.

The results are shown in Table V. Column T shows the total number of trap handlers inserted in the code by IOC. Column R_U shows the number of redundant traps detected by unwrapped intervals and R_W for the case of wrapped intervals. Columns labelled 'without LLVM optimizations' show the case when the analyses are run without any LLVM optimization (only with options `-widening 5 -narrowing 2`). Columns labelled 'with LLVM optimizations' are executed with all LLVM optimizations enabled, that is, with options `-widening 5 -narrowing 2 -instcombine -inline 300 -enable-optimizations`.

Note that the number of redundant traps are often higher in the case of without LLVM optimizations. The reason is that many traps can be removed by constant propagation (used if option `-enable-optimizations` is enabled).

While the improvements are small or absent in most cases, the experiment does indicate that the improved bounds pay off for some applications.

For most cases where an IOC trap block is deemed necessary, the judgement is based on a wrapped interval which is $\top$. In some 60% of these cases, the main reason for arriving at the value $\top$ is the involvement of either (unknown) input or of pointers. This suggests that inter-procedural wrapped analysis and/or support for pointers may pay off for real applications. In any case, since the overhead of using wrapped intervals is relatively small, the wrapped interval analysis appears useful even in its naive form.

Finally, we observe that a wrapped analysis naturally keeps track of "non-zeroness" of variables, while a signed unwrapped analysis cannot, as non-zeroness is a form of disjunctive information. A common case is an interval that starts out as $\top$ but is refined by the wrapped analysis after a conditional of the form **if** $(x \neq 0)$, turning into the wrapped interval $(\!|\, 1, -1 \,|\!)$. This tighter interval does not have any impact in the removal of unnecessary IOC trap blocks but it would be useful for other applications such as CCured [Condit et al. 2003], if the wrapped interval analysis was enhanced to support pointers. CCured adds memory safety guarantees to C programs by enforcing a strong type system at compile time. The parts that cannot be enforced are checked at run time. A wrapped analysis could help CCured remove runtime checks for null pointer dereferences.

Note that the wrapped interval domain is closed under complement. In particular, the complement of a delimited wrapped interval is always a delimited wrapped interval. In contrast, for the classical interval domain, the (best approximation of the) complement of any finite interval is $\top$. This too improves the expressive power of wrapped intervals. For example, given a conditional `if (x>=10 && x<100)`, both wrapped and unwrapped intervals derive useful information for the *then* branch, but only wrapped intervals derive useful information about `x` for the *else* branch.

8. OTHER APPLICATIONS OF WRAPPED INTERVALS

Signedness information is critical in the determination of the potential for under- or overflow. In that context, the improved precision of bounds analysis that we offer is an important contribution.

There is ample evidence [Dietz et al. 2012; Wang et al. 2013] that overflow is very common in real-world C/C++ code. Dietz et al. [2012] suggest, based on scrutiny of many programs, that much use of overflow is intentional and safe (though not portable), but also that the majority is probably accidental. Our interval analysis of course has a broader scope than C/C++, but it is worth mentioning that even in the context of C/C++, overflow problems are not necessarily removed by adherence to coding standards. Wang et al. [2013] remind us of the many aspects of C/C++ that are left undefined by the language specifications. This lack of definition gives an optimizing compiler considerable license, and Wang et al. [2013] show that, in practice, this license is often (mis-)used to undermine safe programming, for example, through removal of mandated overflow checks.

In C/C++, what happens in case of signed over- or underflow is undefined. Many C programmers, however, rely on overflow behaviour that reflects the nature of the underlying machine arithmetic. The following snippet, taken from an early version of C's `atoi`, is typical:

```
char *p;
int f, n;
...
while (*p >= '0' && *p <= '9')
    n = n * 10 + *p++ - '0';
return (f ? -n : n);
```

There are two independent overflow issues. First, in the assignment, if the `+` is evaluated before the `-`, addition may cause overflow. Second, when `f` is non-zero and `n` is the smallest integer, the unary minus causes overflow. This use of overflow is most likely deliberate, and typical of C programmers' reliance on language properties that are plausible, but not guaranteed by the language specification.

Of course the snippet's problematic assignment may be "repaired" by transforming it to

```
n = n * 10 + (*p++ - '0');
```

Recent work [Coker and Hafiz 2013; Logozzo and Martel 2013] considers how to perform such repairs of overflowing expressions automatically. Sometimes a simple rearrangement of operands may suffice, as above. Other possible repair tools include the introduction of type casts. For this application, program analysis (say, interval analysis) is needed.

Also possible is the unintended use of wrap-around, owing, for example, to the subtle semantics of the C language. Simon and King [2007] give this example of a C program intended to tabulate the distribution of characters in a string `s`:

```
char *s;
int dist[256];
...
```

```
while (*s) {
    dist[(unsigned int) *s]++;
    s++;
}
```

and point out the subtle error arising because `*s` is promoted to `int` before the cast to an unsigned integer takes place [Simon and King 2007]. As a result, `dist` can be accessed at indices $[0,\ldots,127] \cup [2^{32}-128,\ldots,2^{32}-1]$, a set which, we should point out, is conveniently captured as a wrapped interval.

9. RELATED WORK

9.1. Intervals using proper integers

Interval analysis is a favourite textbook example of abstract interpretation [Nielson et al. 1999; Seidl et al. 2012]. The classical interval domain $\mathcal{I}$, which uses unbounded integers, was sketched in Section 2.1. Much of the literature on interval analysis uses this domain [Su and Wagner 2004; Leroux and Sutre 2007; Gawlitza et al. 2009]. As discussed in Section 1, such analysis is sound for reasoning about unlimited-precision integers, but unsound in the context of fixed-width machine arithmetic. In particular, the assumption of unbounded integers will lead to problems in the context of low-level languages, including assembly languages, and, as in the case of Rodrigues et al. [2013], LLVM IR.

9.2. Overflow-aware interval analysis

A simple solution to the mismatch between classical interval analysis and the use of finite-precision integers is to amend the analysis to keep track of possible overflow and deem the result of the analysis to be $\top$, that is, void of information, as sketched in Section 2.2. Abstract interpretation based tools such as Astree [Blanchet et al. 2002] and `cccheck` [Fähndrich and Logozzo 2010] use interval analysis (and other kinds of analysis) in an overflow-aware manner. These tools are able to identify expressions that cannot possibly create over- or under-flow. For other expressions, suitable warnings can then be issued.

Regehr and Duongsaa [2006] perform bounds analysis in a wrapping-aware manner, dealing also with bit-wise operations by treating the bounds as bit-vectors. Brauer and King [2010] show how to synthesize transfer functions for such wrapping-aware bounds analysis. Simon and King [2007] show how to make polyhedral analysis wrapping-aware without incurring a high additional cost. These approaches suffer the problem discussed above: when a computed interval spans a wrap-around point, the interval always contains both the smallest and largest possible integer, so all precision is lost.

9.3. Granger's arithmetical congruence analysis

The congruence analysis by Granger [1989] is another example of an "independent attribute" analysis. It is orthogonal to interval analysis, but we mention it here as it plays a role in many proposals for combined analyses. As with classical interval analysis, arithmetical congruence analysis takes $\mathbb{Z}$ as the underlying domain. For the program

```
x = 3;
while (*) {
    x = x+4;
}
```

congruence analysis yields $x \equiv_4 3$, a result which happens to be correct also in the context of 32- or 64-bit integers. However, in general, the analysis is not sound in the context of fixed-precision integers, as is easily seen by replacing '`x+4`' by '`x+5`'.

9.4. Variants of strided intervals

Classical $\mathbb{Z}$-based intervals are sometimes combined with other domains, for added expressiveness. The *modulo intervals* of Nakanishi et al. [1999] are of the form $[i, j]_{n(r)}$, with the reading

$$[i, j]_{n(r)} = \{k \in \mathbb{Z} \mid i \leq k \leq j, k = nm + r, m \in \mathbb{Z}\}$$

Hence they combine arithmetical congruences with classical integer intervals. They were proposed as a tool for analysis to support vectorization. From an abstract interpretation point of view, the set of modulo intervals has shortcomings. Modulo intervals as defined by Nakanishi et al. [1999] can only express finite sets, and so they do not form a complete lattice.

In contrast, Balakrishnan and Reps [2004] utilise an abstract domain which is the reduced product of the classical interval domain and arithmetical congruences. A *reduced interval congruence (RIC)* with stride a is a set $\{ai + d \mid i \in [b, c]\}$, where $[b, c]$ is an element of the classical interval domain $\mathcal{I}$.

Later Reps et al. [2006] introduce the concept of a *strided interval* which is similar to a RIC, but intervals are now of the fixed-precision kind. A w-bit strided interval is of the form $s[a, b]$, with $0 \leq s \leq 2^w - 1$, and with $-2^{w-1} \leq a \leq b < 2^{w-1}$. It denotes the set $[a, b] \cap \{a + is \mid i \in \mathbb{Z}\}$. Hence all values in $s[a, b]$ are (signed) fixed-precision integers, evenly distributed inside the interval $[a, b]$. In other words, the domain of strided intervals is the reduced product domain that combines fixed-width integer intervals with arithmetical congruences. The special case when the stride is 1 gives the standard kind of fixed-precision integer interval.

As with all types of intervals discussed in Sections 9.1–9.4, strided intervals do not allow wrapping. The set of strided intervals is not closed under complement, and is incomparable with the set of wrapped intervals. More precisely, strided intervals cannot express intervals that straddle the north pole, apart from the two-element interval $\llparenthesis 2^{w-1} - 1, -2^{w-1} \rrparenthesis$.

Reps et al. [2006] and Balakrishnan [2007] describe strided-interval abstract versions of many operations. These exclude non-linear arithmetic operations but include bitwise operations, where they draw on Warren Jr [2003], as we do.

9.5. Variants of wrapped intervals

Sen and Srikant [2007] take the approach of Reps et al. [2006] further, promoting the number circle view, as we have done in this paper. This leads to a kind of *strided* wrapped intervals, which Sen and Srikant [2007] call *Circular Linear Progressions* (CLPs) and utilise for the purpose of analysis of binaries. Setting the stride in their CLPs to 1 results in precisely the concept of wrapped intervals used in this paper. Sen and Srikant [2007] provide abstract operations, most of which agree with the operations defined in the present paper, although their analysis is not signedness agnostic in our sense. Multiplication is a case in point; for example, for $w = 4$, a multiplication (signed analysis) such as $[0, 1] \times [7, -8]$ results in $\top$ when CLPs are used, whereas multiplication as defined in this paper produces $\llparenthesis 0, -8 \rrparenthesis$. Sen and Srikant [2007] define many operations by case in a manner that is equivalent to what we have called a north pole cut (as Sen and Srikant [2007] assume signed representation). They do not say how to resolve ties when their "union" operation faces a choice, and they repeatedly refer to the "CLP lattice". However, the CLP domain cannot have lattice structure, as it reduces to the wrapped interval domain when the stride is set to 1 [Gange et al. 2013a]. Hence an analysis with CLPs faces the termination problems discussed in Section 5, unless some remedial action is taken. Sen and Srikant [2007] do not provide an experimental evaluation of CLPs.

Gotlieb et al. [2010] also study wrapped, or "*clockwise*," intervals (without strides). Their aim is to provide constraint solvers for modular arithmetic for the purpose of software verification (other work in this area is described in Section 9.6). They show how to implement abstract addition and subtraction and also how multiplication by a constant can be handled efficiently. Again, a claim that clockwise intervals form a lattice cannot be correct. Gotlieb et al. [2010] assume unsigned representation and general multiplication and bitwise operations are not discussed. The paper presents the unsigned case only and does not address the issues that arise when signedness information is absent. The proposed analysis is not signedness-agnostic in our sense.

In the context of work on the verified C compiler CompCert, Blazy et al. [2013] perform a value analysis of C programs, based on the reduced product of signed and unsigned interval analysis. As we showed in Section 4, wrapped intervals and the reduced product construction are incomparable. The experiments by Blazy et al. [2013] (for $w = 32$) show that, on a collection of some 20 benchmarks, the reduced product finds more "*bounded intervals*" than the wrapped interval analysis that we have presented here. This is wholly unsurprising, as the definition of "bounded" intervals excludes all intervals with cardinality greater than 2^{31}, hence avoiding all cases where wrapped intervals are more precise, as well as such invariants as $x \leq_s 2$ or $x \geq_u 2$. It would be interesting to rerun the experiments of Blazy et al. [2013] without the restriction to bounded intervals.

9.6. Bit-blasting and constraint propagation approaches

It is natural to think of bit-blasting as a method for reasoning about fixed-precision integers, because the bit-level view reflects directly the modulo 2^w nature of the problem. A main attraction of bit-level reasoning is that it can utilise sophisticated DPLL-based SAT solvers, and indeed these are well suited for reasoning about certain bit-twiddling operations. However, methods based on bit-blasting tend to have serious problems with scalability, and bit-blasting does not deal gracefully with non-linear arithmetic operations such as multiplication and division, even in the context of words that are much smaller than 32 bits. These shortcomings are well understood, and we give, in the following, examples of methods have been proposed to make up for the fact that important numerical properties tend to get "lost in translation" when integer relations are blasted into bit relations.

When constraint propagation is applied to reason about programs, it is usually to tackle the problem of program *verification*, rather than program analysis. This makes a considerable difference. In program verification there is a heavy reliance on constraint solvers, and program loops create obstacles that are absent when using abstract interpretation. A constraint solver is a decision procedure, and the constraints of interest are almost always relational, in the sense that they involve several variables. A program analysis is not a decision procedure, and interval analysis, like many other classical analyses, is not relational, but rather is an "independent attribute" analysis [Nielson et al. 1999].

Leconte and Berstel [2006] discuss the potential and dangers of the constraint propagation approach in software verification. A finite-domain *constraint satisfaction problem* (CSP) [Marriott and Stuckey 1998] is a constraint (in conjunctive form) over the variables, together with a mapping D that associates a finite set of values with each variable. The task of a propagator is, given a constraint, to narrow the domains of the variables involved. Consider integer variables x and y and the constraint $2x+2y = 1$. Assume $D(x) = D(y) = [-127, 127]$. A standard propagation step will deduce that x's domain can be narrowed to $[-\frac{253}{2}, \frac{255}{2}]$, that is, to $[-126, 127]$. Using this information, y's domain can now be narrowed to $[-126, 126]$. This allows x to be further narrowed, and so on. The unsatisfiability of the constraint will eventually be discovered, but only after very many propagation steps. (Of course, bit-level reasoning can establish the unsatisfiability of $2x + 2y = 1$ quite easily.) Leconte and Berstel [2006] propose the inclusion of arithmetical congruence constraints in the constraint propagation approach, effectively obtaining a CSP analogue of the RIC domain discussed in Section 9.4, by utilising the analysis of Granger [1989] to develop propagators. This leads to much faster propagation overall.

Bardin et al. [2010] take the ideas of Leconte and Berstel [2006] a step further, by adding a bit-vector solver with propagators for what they call the bitlist domain, $\mathcal{B}L$. An element of this domain is a set of w-width bit-vectors. The set has to be convex in the sense that it can be written as a single bit-vector, with an asterisk denoting an unknown bit. For example, $\langle 0{*}1{*}\rangle$ denotes the set $\{0010, 0011, 0110, 0111\}$.[5] (Only certain sets of cardinality 2^k can be expressed this way. Most integer intervals cannot be expressed like that, and sets that can be expressed as partial bit-vectors are not, in general, intervals. For example, $\langle 0{*}1{*}\rangle = \{2, 3, 6, 7\}$. This is not an issue for Bardin et al. [2010], as the $\mathcal{B}L$ information is meant to complement interval, and congruence, information.) The aim is to combine reasoning about arithmetic operations using the interval/congruence propagation machinery, with reasoning about (certain) bit-twiddling operations using $\mathcal{B}L$. A domain of each type is associated with each variable and maintained.[6] "Channelling" between the different domains is done by propagators specifically defined for the purpose. There is no description of how multiplication and division are handled.

Michel and Van Hentenryck [2012] utilise a domain which is isomorphic to the $\mathcal{B}L$ domain of Bardin et al. [2010]. They give algorithms for the bitwise operations, the comparisons, shifting and addition, providing better propagation for the latter compared with Bardin et al. [2010]. Michel and Van Hentenryck [2012] focus on bit-vectors which are shorter than the underlying machine's bit-width, so can be implemented efficiently using data parallel machine instructions. They do not give experimental results, nor do they discuss non-linear arithmetic operations.

To summarise, all the approaches based on bit-level and/or word-level constraint propagation discussed in this section are incomparable with our analysis. From a constraint reasoning viewpoint, the constraints that we use are simple, "independent attribute" constraints that express membership of an interval. The solvers discussed in this section can reason with more sophisticated properties, including relations that go beyond the "independent attribute" kind.

10. CONCLUSION

Integer arithmetic is a crucial component of most software. However, "machine integers" are a subset of the integers we learned about in school. The dominant use of integers in computers allows only a fixed amount of space for an integer, so not every integer can be represented. Instead we get *fixed-width integer* arithmetic and its idiosyncrasies.

Much of the existing work on interval analysis uses arbitrary precision integers as bounds. Using such an analysis with programs that manipulate fixed-width integers can lead to unsound conclusions. We have presented wrapped intervals, an alternative to the classical interval domain. Our use of wrapped intervals ensures soundness without undue loss of precision, and for a relatively small cost, as we demonstrated in Section 6.

The key is to treat the bounds as bit patterns, letting a wrapped interval denote the set of bit patterns beginning with the left bound and repeatedly incrementing it until the right bound is reached. Wrapped intervals can therefore represent sets that cannot be represented with ordinary intervals, because they "wrap around." For example, a wrapped interval beginning with the largest representable integer and ending with the smallest denotes the set of only those two values.

Viewing integers as bit patterns, the analysis is indifferent to the signedness of the integers, except where relevant to the results being produced. This is ideal for analysis of low level languages such as assembly language and LLVM IR, as these languages treat fixed-width integers as bit strings; only the operations that behave differently in the signed and unsigned case come in two versions. While it is possible to analyse programs correctly under the

[5] Additionally, Bardin et al. [2010] allows the expression of the empty set of bit-vectors.

[6] More precisely, a *set* of *unsigned* integer intervals is maintained per variable.

assumption that all integer values should be interpreted as unsigned (or signed, depending on taste), such an assumption leads to a significant loss of precision.

It is far better for analysis to be signedness-agnostic. We have shown that, if implemented carefully, signedness-agnosticism amounts to more than simply "having a bet each way." Our key observation is that one can achieve higher precision of analysis by making each individual abstract operation signedness-agnostic, whenever its concrete counterpart is signedness-agnostic. This applies to important operations like addition, subtraction and multiplication.

Signedness-agnostic *bounds analysis* naturally leads to wrapped intervals, since signed and unsigned representation correspond to two different ways of ordering bit-vectors. In this paper we have detailed a signedness-agnostic bounds analysis, based on wrapped intervals. The resulting analysis is efficient and precise. It is beneficial even for programs where all signedness information is present.

We have observed that the wrapped interval domain is not a lattice. To compensate, we have presented over- and under-approximations of join and meet. However, these approximations lack some of the properties we expect of joins and meets: they are neither associative nor monotone. The lack of associativity means that repeated joins and meets are not a substitute for variadic least upper bound and greatest lower bound operations, so we have presented both over- and under-approximating variadic least upper bound and greatest lower bound operations. These are generally more precise than repeated approximate binary joins and meets, irrespective of the order in which the binary operations are applied [Gange et al. 2013a].

The lack of monotonicity of meets and joins means that classical fixed point finding methods may fail to terminate. We have presented a widening operator that ensures monotonicity, as well as accelerating convergence. The widening strategy is based on the idea of, roughly, doubling the size of intervals in each widening step. Gange et al. [2013a] discuss, in more general terms, the issues that arise from the use of non-lattice domains (such as the domain of wrapped intervals) in abstract interpretation.

As future work we plan to extend our tools to support interprocedural analysis using wrapped intervals, and also investigate the combination of wrapped intervals with pointer analyses to improve precision.

A worthwhile line of future research is to find ways of generalizing wrapped interval analysis to relational analyses, such as those using octagons [Miné 2006]. To this end, Gange et al. [2013b] study the case of difference logic (constraints $x - y \leq k$) and find that classical approaches such as the Bellman-Ford algorithm cannot readily be adapted to the setting of modular arithmetic. It appears that much of the large body of work on algorithms for relational analysis requires thorough review through the lenses of machine arithmetic.

ACKNOWLEDGMENTS

We would like to thank John Regehr, Jie Liu, Douglas Teixeira and Fernando Pereira for helpful discussions about interval analysis and LLVM.

REFERENCES

Gogul Balakrishnan. 2007. *WYSINWYX: What You See Is Not What You Execute.* Ph.D. Dissertation. University of Wisconsin at Madison, Madison, WI, USA.

Gogul Balakrishnan and Thomas Reps. 2004. Analyzing Memory Accesses in x86 Executables. In *Compiler Construction: Proceedings of the 13th International Conference*, E. Duesterwald (Ed.). Lecture Notes in Computer Science, Vol. 2985. Springer, 5–23.

Sébastien Bardin, Philippe Herrmann, and Florian Perroud. 2010. An Alternative to SAT-Based Approaches for Bit-Vectors. In *Proceedings of the 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'10)*, J. Esparza and R. Majumdar (Eds.). Lecture Notes in Computer Science, Vol. 6015. Springer, 84–98.

Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. 2002. Design and Implementation of a Special-Purpose Static Program Analyzer for Safety-Critical Real-Time Embedded Software. In *The Essence of Computation: Complexity, Analysis, Transformation*, T. Æ. Mogensen, D. A. Schmidt, and I. H. Sudborough (Eds.). Lecture Notes in Computer Science, Vol. 2566. Springer, 85–108.

Sandrine Blazy, Vincent Laporte, Andre Maroneze, and David Pichardie. 2013. Formal Verification of a C Value Analysis Based on Abstract Interpretation. In *Static Analysis*, F. Logozzo and M. Fähndrich (Eds.). Lecture Notes in Computer Science, Vol. 7935. Springer, 324–344.

Jörg Brauer and Andy King. 2010. Automatic Abstraction for Intervals Using Boolean Formulae. In *Static Analysis*, R. Cousot and M. Martel (Eds.). Lecture Notes in Computer Science, Vol. 6337. Springer, 167–183.

Zack Coker and Munawar Hafiz. 2013. Program Transformations to Fix C Integers. In *Proceedings of the 35th International Conference on Software Engineering (ICSE 2013)*. IEEE, 792–801.

Jeremy Condit, Matthew Harren, Scott McPeak, George C. Necula, and Westley Weimer. 2003. CCured in the Real World. In *ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI'03)*. ACM, 232–244.

Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proceedings of the Fourth ACM Symposium on Principles of Programming Languages*. ACM, 238–252.

Patrick Cousot and Radhia Cousot. 1979. Systematic Design of Program Analysis Frameworks. In *Proceedings of the Sixth ACM Symposium on Principles of Programming Languages*. ACM, 269–282.

Patrick Cousot and Radhia Cousot. 1992. Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation. In *International Symposium on Programming Language Implementation and Logic Programming*, M. Bruynooghe and M. Wirsing (Eds.). Lecture Notes in Computer Science, Vol. 631. Springer, 269–295.

Will Dietz, Peng Li, John Regehr, and Vikram Adve. 2012. Understanding Integer Overflow in C/C++. In *Proceedings of the 34th International Conference on Software Engineering*. IEEE, 760–770.

Manuel Fähndrich and Francesco Logozzo. 2010. Static Contract Checking with Abstract Interpretation. In *FoVeOSS*, B. Beckert and C. Marché (Eds.). Lecture Notes in Computer Science, Vol. 6528. Springer, 10–30.

Stephan Falke, Deepak Kapur, and Carsten Sinz. 2012. Termination Analysis of Imperative Programs Using Bitvector Arithmetic. In *Verified Software: Theories, Tools, and Experiments*, R. Joshi, P. Müller, and A. Podelski (Eds.). Lecture Notes in Computer Science, Vol. 7152. Springer, 261–277.

Stephan Falke, Florian Merz, and Carsten Sinz. 2013. LLBNC: Improved Bounded Model Checking of C Programs Using LLVM. In *Tools and Algorithms for the Construction and Analysis of Systems*, N. Piterman and S. Smolka (Eds.). Lecture Notes in Computer Science, Vol. 7795. Springer, 623–626.

Graeme Gange, Jorge A. Navas, Peter Schachte, Harald Søndergaard, and Peter J. Stuckey. 2013a. Abstract Interpretation over Non-Lattice Abstract Domains. In *Static Analysis*, F. Logozzo and M. Fähndrich (Eds.). Lecture Notes in Computer Science, Vol. 7935. Springer, 6–24.

Graeme Gange, Harald Søndergaard, Peter J. Stuckey, and Peter Schachte. 2013b. Solving Difference Constraints over Modular Arithmetic. In *Automated Deduction*, M. Bonacina (Ed.). Lecture Notes in Artificial Intelligence, Vol. 7898. Springer, 215–230.

Thomas Gawlitza, Jérôme Leroux, Jan Reineke, Helmut Seidl, Grégoire Sutre, and Reinhard Wilhelm. 2009. Polynomial Precise Interval Analysis Revisited. In *Efficient Algorithms: Essays Dedicated to Kurt Mehlhorn on the Occasion of His 60th Birthday*, S. Albers, H. Alt, and S. Näher (Eds.). Lecture Notes in Computer Science, Vol. 5760. Springer, 422–437.

Arnaud Gotlieb, Michel Leconte, and Bruno Marre. 2010. Constraint Solving on Modular Integers. In *Proceedings of the Ninth International Workshop on Constraint Modelling and Reformulation (ModRef'10)*.

Philippe Granger. 1989. Static Analysis of Arithmetical Congruences. *International Journal of Computer Mathematics* 30 (1989), 165–190.

Philippe Granger. 1991. Static Analyses of Linear Congruence Equalities among Variables of a Program. In *Theory and Practice of Software Development*. Lecture Notes in Computer Science, Vol. 493. Springer, 167–192.

Andy King and Harald Søndergaard. 2010. Automatic Abstraction for Congruences. In *Verification, Model Checking and Abstract Interpretation*, G. Barthe and M. Hermenegildo (Eds.). Lecture Notes in Computer Science, Vol. 5944. Springer, 197–213.

Michel Leconte and Bruno Berstel. 2006. Extending a CP Solver with Congruences as Domains for Program Verification. In *Proceedings of the First Workshop on Software Testing, Verification and Analysis (CSTVA'06)*, B. Blanc, A. Gotlieb, and C. Michel (Eds.). 22–33.

Jérôme Leroux and Grégoire Sutre. 2007. Accelerated Data-Flow Analysis. In *Static Analysis*, H. Riis Nielson and G. Filé (Eds.). Lecture Notes in Computer Science, Vol. 4634. Springer, 184–199.

Francesco Logozzo and Matthieu Martel. 2013. Automatic Repair of Overflowing Expressions with Abstract Interpretation. In *Semantics, Abstract Interpretation, and Reasoning about Programs*, A. Banerjee, O. Danvy, K.-G. Doh, and J. Hatcliff (Eds.). Electronic Proceedings in Theoretical Computer Science, Vol. 129. 341–357.

Kim Marriott and Peter J. Stuckey. 1998. *Programming with Constraints: An Introduction.* MIT Press.

Laurant D. Michel and Pascal Van Hentenryck. 2012. Constraint Satisfaction over Bit-Vectors. In *Constraint Programming: Proceedings of the 2012 Conference*, M. Milano (Ed.). Lecture Notes in Computer Science, Vol. 7514. Springer, 527–543.

Antoine Miné. 2006. The Octagon Abstract Domain. *Higher-Order and Symbolic Computation* 19, 1 (2006), 31–100.

Tsuneo Nakanishi, Kazuki Joe, Constantine D. Polychronopoulos, and Akira Fukuda. 1999. The Modulo Interval: A Simple and Practical Representation for Program Analysis. In *Parallel Architecture and Compilation Techniques.* IEEE, 91–96. DOI:http://dx.doi.org/10.1109/PACT.1999.807422

Jorge A. Navas, Peter Schachte, Harald Søndergaard, and Peter J. Stuckey. 2012. Signedness-Agnostic Program Analysis: Precise Integer Bounds for Low-Level Code. In *Proceedings of the 10th Asian Symposium on Programming Languages and Systems (APLAS'12)*, R. Jhala and A. Igarashi (Eds.). Lecture Notes in Computer Science, Vol. 7705. Springer, 115–130.

Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. 1999. *Principles of Program Analysis.* Springer.

John Regehr and Usit Duongsaa. 2006. Deriving Abstract Transfer Functions for Analyzing Embedded Software. In *Proceedings of the 2006 SIGPLAN/SIGBED Conference on Language, Compilers, and Tool Support for Embedded Systems (LCTES'06).* ACM Press, 34–43.

Thomas Reps, Gogul Balakrishnan, and Junghee Lim. 2006. Intermediate-Representation Recovery from Low-Level Code. In *Proceedings of the 2006 ACM SIGPLAN Conference on Partial Evaluation and Semantics-Based Program Manipulation.* ACM Press, 100–111.

Raphael E. Rodrigues, Victor H. Sperle Campos, and Fernando M. Quintão Pereira. 2013. A Fast and Low-Overhead Technique to Secure Programs Against Integer Overflows. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO'13).* IEEE, 1–11.

Helmut Seidl, Reinhard Wilhelm, and Sebastian Hack. 2012. *Compiler Design: Analysis and Transformation.* Springer.

Rathijit Sen and Y. N. Srikant. 2007. 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.* IEEE, 39–48.

Axel Simon and Andy King. 2007. Taming the Wrapping of Integer Arithmetic. In *Static Analysis*, H. Riis Nielson and G. Filé (Eds.). Lecture Notes in Computer Science, Vol. 4634. Springer, 121–136.

Zhendong Su and David Wagner. 2004. A Class of Polynomially Solvable Range Constraints for Interval Analysis without Widenings and Narrowings. In *Tools and Algorithms for the Construction and Analysis of Systems*, K. Jensen and A. Podelski (Eds.). Lecture Notes in Computer Science, Vol. 2988. Springer, 280–295.

Douglas d. C. Teixera and Fernando M. Q. Pereira. 2011. The Design and Implementation of a Non-Iterative Range Analysis Algorithms on a Production Compiler. In *Proceedings of the 2011 Brasilian Symposium on Programming Languages.*

Xi Wang, Nickolai Zeldovich, M. Frans Kaashoek, and Armando Solar-Lezama. 2013. Towards Optimization-Safe Systems: Analyzing the Impact of Undefined Behavior. In *Proceedings of 24th ACM Symposium on Operating Systems Principles.* ACM, 260–275.

Henry S. Warren Jr. 2003. *Hacker's Delight.* Addison Wesley.

Chao Zhang, Tielei Wang, Tao Wei, Yu Chen, and Wei Zou. 2010. IntPatch: Automatically Fix Integer-Overflow-to-Buffer-Overflow Vulnerability at Compile-Time. In *Computer Sceurity – ESORICS 2010*, D. Gritzalis, B. Preneel, and M. Theoharidou (Eds.). Lecture Notes in Computer Science, Vol. 6345. Springer, 71–86.

Chao Zhang, Wei Zou, Tielei Wang, Yu Chen, and Tao Wei. 2011. Using Type Analysis in Compiler to Mitigate Integer-Overflow-to-Buffer-Overflow Threat. *Journal of Computer Security* 19, 6 (2011), 1083–1107.

Received July 2013; revised March 2014; accepted July 2014