Computation of the reduced product between the interval and known bit domains

Alexander Arthur
The University of Queensland
alex@alegs.xyz

1 Preface

This document was prepared for an audience mostly unfamiliar with abstract interpretation. If you know what a reduced product is, you could happily skip ahead to Section 7.

2 Introduction

As long as they completed the first or second grade or elementary school, the reader should be familiar with the interval domain in the context of static analysis. For those who failed kindergarten, a static analysis of a program over the interval domain produces an interval of the form $[a, b]$ for each variable in the program, where the concrete value of each variable is guaranteed to be within its interval for every possible execution of the program being analysed.

3 Motivation

Unfortunately, static analyses are not omnipotent[citation needed] and thus an interval analysis sometimes produces results that are less precise than would otherwise be possible.

```
1 X ← 1
2 while X < 10 do
3 |  X ← X + 2
4 done
5 if X ≥ 12
6 |  X ← 0
7 endif
```

Algorithm 1: Blatantly stolen from Antoine Miné [1]

As an example, at Line 4 an interval analysis could use a combination of the loop body and guard to conclude that $X \in [11, 12]$. However, at Line 5 the analysis must consider the body of the if statement (since[1] $[12, +\infty] \cap [11, 12] \neq \emptyset$), and so by Line 7 the interval expands back to $X \in [0, 12]$.

The intelligent reader could observe that Algorithm 1 is free from non-determinism, and that therefore $X = 11$ at the end of every execution of the program. We will aim to use the results of another analysis —a known bits analysis—to refine the results of the interval analysis. Another reason to do this is that BASIL's[2] current interval analysis falls on its face given any binary operation other than addition and subtraction, so supplementing it with information from the known bits analysis will be beneficial as well.

4 Known Bits Analysis

Computers represent numbers using ones and zeroes[citation needed], and we can therefore reason about

[1] If we abuse some notation slightly in order to conflate an interval with the set it represents
[2] https://github.com/UQ-PAC/BASIL

variables as n-bit bitvectors and glean information about their (signed or unsigned) value[3]. We want to be able to make a statement about which bits of a variable we know to be 1, which we know to be 0, and which we can't know for certain.

In order to do this, we need a third value μ to represent this unknown state, meaning that we must model each variable not as a vector of bits (elements in $\{0, 1\}$) but as a vector of trits (elements in $\{0, 1, \mu\}$). For example, if our analysis can deduce that the second bit of the eight-byte variable X is 1 and the third is zero, but nothing about the others, we would write that abstract tritvector as $X = [\mu 01 \mu\mu\mu\mu\mu\mu]$.

5 Maths time

I've been talking about domains this whole time, but what is a domain? I won't give a formal definition (see [1] or [2] for that), but I'll give enough examples to hopefully provide some intuition.

5.1 Order theory and lattices

Lattices are pretty fundamental to the field of static analysis. If we have some set (let's say the integers $\mathbb{Z}$) and a partial ordering[4] on that set (let's say $\leq$), then together they form a partially ordered set (written as the pair $(\mathbb{Z}, \leq)$), or poset for short.

Let's look at another poset $(\mathcal{P}, \subseteq)$. This uses the operator $\subseteq$ to construct an ordering over every possible subset of the integers.

We use these orderings in a static analysis to represent the *specificity* of the information we know about the program. If we have two things a and b, then $a \sqsubseteq b$ iff a tells us more information than b. Additionally, we define an element of our set denoted $\top$, loosely defined as $\forall s \in S, s \sqsubseteq \top$[5].

5.2 Theory is boring hurry up

Okay! So we've got a concrete domain $(C, \leq)$ and an abstract domain $(A, \sqsubseteq)$. We can define a *concretisation function* $\gamma : A \to C$ that turns a value in the *abstract* domain into a value in the *concrete* domain. If our concrete domain is the powerset of integers under inclusion as before, then the concretisation function γ gives us the set of integers that the abstract representation represents. For the interval domain, this happens to be $\gamma([a, b]) = \{x \in \mathbb{Z} \mid a \leq x \leq b\}$.

We can also define the opposite operator (although it isn't guaranteed to exist!), an abstraction function $\alpha : C \to A$, which gives the optimal representation in the abstract domain of some element of the concrete domain. For our interval domain again, we can define this as $\alpha(Z) = [\min(Z), \max(Z)]$. I'll point out here that, as is always the case in static analysis, this is an approximation. Given the set $\{1, 3, 4\}$ we are forced to represent it as the interval $[1, 4]$, despite the fact that this includes 2 while our original set did not[6].

If we get lucky, γ and α together form a Galois connection, which lets us take an abstract value on a trip through the concrete world $\alpha(\gamma(a))$ (and vice versa) and, if we repeat this more than once, not have it lose any information past that which is lost on the first trip (i.e. $\alpha \circ \gamma$ and $\gamma \circ \alpha$ are idempotent). We denote this Galois connection

---

[3] And a number of other things - congruency being prominent among them.

[4] Exactly what a partial ordering is isn't super important, but it's worth noting that the ordering *doesn't* need to be defined on every pair of elements in the set. If a and b are elements in my poset $(S, \lhd)$, then it's possible that neither of $a \lhd b$ and $a \rhd b$ are true.

[5] In our example of $(\mathcal{P}(\mathbb{Z}), \subseteq)$ from before, $\top = \mathbb{Z}$

[6] This is a tradeoff between accuracy and computability. We could always use the perfectly accurate method of simulating every possible execution of a program, but this is usually infeasible[citation needed], so we are forced to make approximations for the sake of being able to compute an answer in a reasonable amount of time. The trick lies in making the *right* approximations.

$$(C, \subseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (A, \sqsubseteq)$$

6 Reduced product. Finally!

Shut up. So we've got two domains: $\mathcal{I}$ of intervals and $\mathcal{T}$ of tnums. We want to find some way of using the information from one domain to improve the accuracy of the other. Since it happens to be the case that $\gamma_{\mathcal{I}}$ and $\gamma_{\mathcal{T}}$ share a codomain (namely $\mathcal{P}(\mathbb{Z})$), we can take an interval $[a, b]$ and a tnum x and compute both $\gamma_{\mathcal{I}}([a, b])$ and $\gamma_{\mathcal{T}}(x)$ to obtain two sets of integers. Since both of our domains are themselves *sound*,[7] we can take the intersection of these two sets in order to produce a new, more specific set of possible values. Let's call that function $\gamma_{\mathcal{I}\times\mathcal{T}}$, and define

$$\gamma_{\mathcal{I}\times\mathcal{T}}([a, b], x) = \gamma_{\mathcal{I}}([a, b]) \cap \gamma_{\mathcal{T}}(x)$$

Now that we've refined our result, we need to move it back into our abstract domains, so that the analysis can continue. Luckily, we've got functions do that for us! $\alpha_{\mathcal{I}} : \mathcal{P}(\mathbb{Z}) \to \mathcal{I}$ and $\alpha_{\mathcal{T}} : \mathcal{P}(\mathbb{Z}) \to \mathcal{T}$ will do exactly what we want, so we can construct the *optimal reduction* ρ as

$$\rho(p, q) = (\alpha_1(\gamma_{1\times 2}(p, q)), \alpha_2(\gamma_{1\times 2}(p, q)))$$

which in our case looks like

$$\rho([a, b], x) = (\alpha_{\mathcal{I}}(\gamma_{\mathcal{I}\times\mathcal{T}}([a, b], x)), \alpha_{\mathcal{T}}(\gamma_{\mathcal{I}\times\mathcal{T}}([a, b], x)))$$

Yay!

What next?

7 Computers aren't maths, dummy[8]

Writing that down was easy! Computing that is less easy!

One option is to go for the simple but sub-optimal approach:

$$\rho([a, b], x) = ([a', b], x')$$

where

$$\begin{aligned} a' &= \max(a, \lfloor x \rfloor) \\ b' &= \min(b, \lceil x \rceil) \\ x' &= x_{i>\text{width}(b)} \end{aligned}$$

I've pulled some notation out of thin air there, so let's go through it:

- $\lfloor x \rfloor$ and $\lceil x \rceil$ denote the minimum and maximum possible values of the tritvector x, i.e. replace every μ with a 0 in the case of $\lfloor x \rfloor$ and with a 1 for $\lceil x \rceil$[9].
- $x_{i\geq n}$ is the tritvector x, except with all the trits past the n$^{\text{th}}$ place replaced with zeros (i.e. bounding the value of x above). Bounding it below is more difficult; all you can say is "At least one of the trits past the n$^{\text{th}}$ place are 1", which is less useful.

[7] Not that I've defined what being sound is. Put simply, soundness says that I will only ever produce an overapproximation of the true result, and never an underapproximation. A sound analysis will never fail to include a possible value in its result.

[8] Citation needed.

[9] This is assuming an unsigned representation. The signed case *should* have approximately the same properties with different mechanics, but I haven't thought about it yet.

This approach is sound, and easy to compute, but is not optimal. Consider an example where we arrive at a point in the program with an interval of $[6, 10]$ and a tnum of $\mu 00\mu$. $\lfloor \mu 00\mu \rfloor = 0$, which isn't much use and leaves $a' = 6$, but $\lceil \mu 00\mu \rceil = 1001 = 9$, which lets us give $b' = 9$, a slight improvement. We also aren't able to improve x' at all using this method; it remains as $x' = \mu 00\mu$.

7.1 Let's get clever

We want to be able to make better refinements to our interval and tnum than just the simple maximums and minimums above. The two algorithms in Algorithm 2 define a much more precise refinement. We perform a binary search[10] over the unknown bits in the tnum to find the smallest value representable by the tnum which is greater than or equal to the original interval bound (and vice versa for the upper bound). For example, given the interval $[6, 10]$ and the tnum $\mu 00\mu$, we can refine our lower bound to 8, since neither of 6 or 7 have zeros in the middle two bits.

```
1 def refineLowerBound(a, x)
2 |  a' ← ⌈x⌉
3 |  for i ∈ [width(x) − 1, 0] s.t. x[i] = μ do
4 |  |  a'[i] ← 0
5 |  |  if a' < a then
6 |  |  |  a'[i] ← 1     # Went too far
7 |  |  endif
8 |  done
9 |  return a'
```

(a) Algorithm to refine an lower bound

```
1 def refineUpperBound(b, x)
2 |  b' ← ⌊x⌋
3 |  for i ∈ [width(x) − 1, 0] s.t. x[i] = μ do
4 |  |  b'[i] ← 1
5 |  |  if b' > b then
6 |  |  |  b'[i] ← 0     # Went too far
7 |  |  endif
8 |  done
9 |  return b'
```

(b) Algorithm to refine an upper bound

Algorithm 2: Refine an interval with a tnum.

In theory, Algorithm 2 is $\mathcal{O}(n)$ where n is the number of unknown bits in the tnum, but in most practical implementations of a tnum it is probably $\mathcal{O}(\text{width}(x))$.

We also wish to refine our tnum. This is a trickier operation. We can use our interval to deduce some number (possibly zero) of the high bits of the tnum. For example, consider an interval representing an n-bit value. If the lower bound of the interval is greater than or equal to 2^{n-1} (and we require that $a \leq b$ in our interval $[a, b]$), then we can conclude that the high bit must be set (or unset, if the interval is contained within the lower half of the value's range). In bitwise terms, this is equivalent to finding the largest sequence of the high bits that are common to a and b. Algorithm 3 performs this operation.

[10] Although it doesn't look like it at first glance!

```
1  def refineTnum([a, b], x)
2      mask ← ~(a ⊕ b)    # Bitwise exclusive-or
3      for each i ∈ [width(x) − 1, 0] do
4          if mask[i] = 0 then
5              break
6          assert a[i] = b[i] # Will always be true
7          if ¬(x[i] = μ ∨ x[i] = a[i]) then
8              return ⊥
9          x[i] ← a[i]
10     return x
```

Algorithm 3: Algorithm to refine the lower bound of an interval

Now that we have the component parts of our reduction, we can actually compute our reduced product. It should be sufficient to simply refine our interval bounds, then refine our tnum based on the new interval.[11] This produces the remarkably simple Algorithm 4, although it is only simple because all the difficult parts have been sub-contracted to the other functions.

```
1  def reduce([a, b], x)
2      a ← refineLowerBound(a, x)
3      b ← refineUpperBound(b, x)
4      x ← refineTnum([a, b], x)
5      return [a, b], x
```

Algorithm 4: Fully reduce an interval-tnum pair

Things I read and you should read too

[1] A. Miné, "Tutorial on Static Inference of Numeric Invariants by Abstract Interpretation," *Foundations and Trends® in Programming Languages*, vol. 4, no. 3–4, pp. 120–372, 2017, doi: 10.1561/2500000034.

[2] A. Møller and M. I. Schwartzbach, "Static Program Analysis." Oct. 2018.

[11] I think this is sufficient, but I'm not sure. I dont think that the interval to tnum refinement can introduce any more information than already exists in the interval, so further iterations of refinement would be redundant.