Reduced Product of Known Bits and Wrapped Intervals

1 Introduction

Reduced product domains allow us to use information from differing representations to increase the precision of the other. The wrapped interval domain works best for arithmetic operations, whilst tristate numbers provide more detail for bitwise/logical operations. For wrapped intervals, introducing information from the known bits analysis allows us to have gaps in our intervals, which we can exploit to shorten their width. For tristate numbers, wrapping intervals can be used to resolve unknown bits of high significance.

2 Conversions

2.1 Tristate Numbers to Wrapped Intervals

To ensure the soundness and precision of this conversion, the following rules must hold:

$$\mathsf{tnum}[i] \in \{0,1\} \implies \mathsf{lower}[i] = \mathsf{upper}[i] = \mathsf{tnum}[i]$$

$$\mathsf{tnum}[i] = \mu \implies \mathsf{lower}[i] = 0 \land \mathsf{upper}[i] = 1$$

A conceptual method to ensure this holds is to convert each bit into a single bit wrapped interval, and then concatenate those intervals together.

$$0 \mapsto (\!|0,0|\!) \quad 1 \mapsto (\!|1,1|\!) \quad \mu \mapsto (\!|0,1|\!) = \top_1$$

$$01\mu1 \mapsto \mathsf{concat}((\!|0,0|\!), (\!|1,1|\!), (\!|0,1|\!), (\!|1,1|\!)) = (\!|0101, 0111|\!)$$

However, this method is inefficient since it needs to $\mathsf{extract}$ each bit and then concat. Instead, we can perform the mapping in constant time using bitwise operations on the value and mask. We do this by redefining our rules on value and mask, we get the following:

$$m[i] = 0 \implies \mathsf{lower}[i] = \mathsf{upper}[i] = v[i]$$

$$m[i] = 1 \implies \mathsf{lower}[i] = 0 \land \mathsf{upper}[i] = 1$$

From here, we can use the specified function table to derive our bitwise operations.

$$\{v; m\} \mapsto (\!|v \mathbin{\&} {\sim}m, v \mid m|\!)$$

2.2 Wrapped Intervals to Tristate Numbers

When converting intervals, we must be careful of crossing the south pole $\mathsf{sp} = (\!|1^w, 0^w|\!)$ as relying on only the bounds obscures some possible values, similar to the concat operation.

$$\mathsf{sp} \sqsubseteq (\!|a,b|\!) \implies (\!|a,b|\!) \mapsto \mu^w$$

Based on similar principle, any bits after the first bit where the lower and upper bound must also be considered as unknown. For example, a naive conversion of the interval $(\!|001, 011|\!)$ would produce $0\mu1$. However, this result would be considered unsound as the interval contains 010 which is not included in $0\mu1$.

For an interval which does not cross the south pole, we find k, the minimum number of required to represent lower ^ upper. This is performed with the ZArith library by using the `Z.numbits` function. From here, we are able to construct the mask $0^{w-k}1^k$ using k and the value $\mathsf{lower} \mid {\sim}m$ (either lower or upper can be used, since the differing bits just get masked away).

```ocaml
(* Psuedocode for OCaml impl *)
let diff = Bitvec.bitxor lower upper in
let k = Z.numbits @@ Bitvec.to_unsigned_bigint diff in
let m = Bitvec.(concat (zero ~size:(w - k)) (ones ~size:k)) in
let v = Bitvec.(bitand lower @@ bitnot m) in
tnum v m
```

To finish off, the implementation of the known bits domain uses a lifted lattice to account for the lack of width argument in `Lattice.bottom` and `ValueAbstraction.top`. $\top$ is handled by the current implementation since it crosses the south pole. As for the empty interval $\bot$, we simply map it to the bottom value of the known bits lattice.

3 Reductions

3.1 Meet Reduction

The reduction function ρ takes the pair of abstract representations and uses them to refine each other.

In the case of tristate numbers, we are able to use known bits in one result to determine the unknown bits of another. We assume that given the soundness of the analyses and conversion functions, known bits in the same position will not differ between arguments. We are able to determine the meet using the following function:

$$a \sqcap b = \begin{cases} \bot & \text{if } a_v \mathbin{\&} \mathord{\sim} a_m \neq b_v \mathbin{\&} \mathord{\sim} b_m \\ \{v = (a_v \mid b_v) \mathbin{\&} \mathord{\sim} m;\ m = a_m \mathbin{\&} b_m\} & \text{otherwise} \end{cases}$$

A similar trick can be used by taking the join of the intersection of two intervals. This will not be the exact meet but rather a pseudo meet similar to the join on wrapped intervals.

$$s \mathbin{\tilde{\sqcap}} t = \tilde{\bigsqcup} s \cap t$$

Finally we can combine these for the reduction function:

$$\rho((\mathsf{tnum}, \mathsf{wint})) = \left(\mathsf{tnum} \sqcap \mathsf{into}_{\mathsf{tnum}}(\mathsf{wint}), \mathsf{wint} \mathbin{\tilde{\sqcap}} \mathsf{into}_{\mathsf{wint}}(\mathsf{tnum})\right)$$

Whilst in most cases it is possible to apply the reduction function multiple times until a fixed point is reached, due to the lack of guaranteed termination for join on wrapped intervals this will not work. Using a widening operator for this process would be counterintuitive.

3.2 Arthurian Reduction

Unlike tristate numbers, the meet $\tilde{\sqcap}$ is not the most precise reduction for intervals. For this, we adapt two algorithms from Alex Arthur's implementation of the known bits and unsigned interval reduced product. This method uses a binary search on the unknown bits to truncate the bounds of the interval.

Since the algorithm uses unsigned comparisons, it does not work for wrapped intervals out of the box. Instead we apply the ssplit trick to operate on a set of unsigned intervals, of which we take the least upper bound $\tilde{\bigsqcup}$. Arthur's algorithm for reducing the tristate number is equivalent to $\mathsf{tnum} \sqcap \mathsf{into}_{\mathsf{tnum}}(\mathsf{wint}')$ where wint' is wint after reduction.

Below we show the imperative pseudocode of the interval reduction, with both bounds being computed simultaneously. The implemented version involves recursion on a single bit mask and logical right shifting.

```
def $\rho_{\mathsf{wint}}(\mathsf{wint}, \mathsf{tnum})$
  $\llparenthesis a, b \rrparenthesis, \llparenthesis b', a' \rrparenthesis \leftarrow \mathsf{wint}, \mathsf{into}_{\mathsf{wint}}(\mathsf{tnum})$
  for $i \in [w - 1, 0]$
    if $\mathsf{tnum}[i] = \mu$ then
      $a'[i], b'[i] \leftarrow 0, 1$
      if $a' < a$ then $a'[i] \leftarrow 1$ endif
      if $b' > b$ then $b'[i] \leftarrow 0$ endif
    endif
  endfor
  return $\llparenthesis a', b' \rrparenthesis$
```

Using this method, we derive the full reduction function:

$$\rho((\mathsf{tnum}, \mathsf{wint})) = \left(\mathsf{tnum} \sqcap \mathsf{into}_{\mathsf{tnum}}(\mathsf{wint}'), \widetilde{\bigsqcup}\{\rho_{\mathsf{wint}}(w, \mathsf{tnum}) \mid w \in \mathsf{ssplit}(v), v \in \mathsf{wint} \cap \mathsf{into}_{\mathsf{wint}}(\mathsf{tnum})\}\right)$$

3.3 Coughlin Method

The method shown in this section is equivalent to the one described above, but with the advantage of executing in constant time by using bitwise operations instead of an iterative approach. This method has also been verified in Boogie to be sound and signedness-agnostic (i.e. handles intervals which cross the south pole), meaning we also avoid taking the ssplit and then joining the individually reduced intervals. As such, the complete reduction is now the following:

$$\rho((\mathsf{tnum}, \mathsf{wint})) = (\mathsf{tnum} \sqcap \mathsf{into}_{\mathsf{tnum}}(\mathsf{wint}'), \rho_{\mathsf{wint}}(\mathsf{wint}, \mathsf{tnum}))$$

First, some primitive defintions:

- $\mathsf{mssb}(x)$ returns a bitvector where only the most significant set bit of x is set. We are able to construct this using the `Z.numbits` trick presented earlier.
- $\mathsf{lssb}(x) = x \mathbin{\&} -x$ returns a bitvector where only the least significant set bit of x is set.
- $\mathsf{above}(p) = \mathord{\sim}(p \mid (p - 1))$ takes a bitvector with a single set bit and returns a bitvector where all bits more significant than it are set.
- $\mathsf{below}(p) = p - 1$ takes a bitvector with a single set bit and returns a beitvector where all bits less significant than it set.
- $\mathsf{mergeon}(a, b, p) = (a \mathbin{\&} \mathsf{above}(p)) \mid (b \mathbin{\&} \mathsf{below}(p))$ takes two bitvectors and a bitvector with a single set bit and constructs such that bits more significant that the set bit come from a less significant bits come from b. This is equivalent to $\mathsf{concat}(\mathsf{extract}_{w,k+1}(a), 0, \mathsf{extract}_{k,0}(b))$ where k is the set bit in p.

```
def $\rho_{\mathsf{wint}_{\text{lower}}}(a, \mathsf{tnum})$
  $\text{diff} \leftarrow \mathsf{mssb}((a \mathbin{\char`\^} \mathsf{tnum}_v) \mathbin{\&} \mathord{\sim}\mathsf{tnum}_m)$
  $\llparenthesis \text{tmin}, \_ \rrparenthesis \leftarrow \mathsf{into}_{\mathsf{wint}}(\mathsf{tnum})$
  if $\text{diff} = 0$ then
    return $a$
  else if $a \mathbin{\&} \text{diff} = 0$ then
    return $\text{diff} \mid \mathsf{mergeon}(a, \text{tmin}, \text{diff})$
  else
    $\text{carry} \leftarrow \mathsf{lssb}(\mathsf{above}(\text{diff}) \mathbin{\&} \mathord{\sim}a \mathbin{\&} \mathsf{tnum}_m)$
    return $\text{carry} \mid \mathsf{mergeon}(a, \text{tmin}, \text{carry})$
  endif
```

```
def $\rho_{\mathsf{wint}_{\text{upper}}}(b, \mathsf{tnum})$
  $\text{diff} \leftarrow \mathsf{mssb}((b \mathbin{\char`\^} \mathsf{tnum}_v) \mathbin{\&} \mathord{\sim}\mathsf{tnum}_m)$
  $\llparenthesis \_, \text{tmax} \rrparenthesis \leftarrow \mathsf{into}_{\mathsf{wint}}(\mathsf{tnum})$
  if $\text{diff} = 0$ then
    return $b$
  else if $b \mathbin{\&} \text{diff} \neq 0$ then
    return $\mathsf{mergeon}(b, \text{tmax}, \text{diff})$
  else
    $\text{borrow} \leftarrow \mathsf{lssb}(\mathsf{above}(\text{diff}) \mathbin{\&} b \mathbin{\&} \mathsf{tnum}_m)$
    return $\mathsf{mergeon}(b, \text{tmax}, \text{borrow})$
  endif
```

$$\rho_{\mathsf{wint}}(\mathsf{wint}, \mathsf{tnum}) = \begin{cases} \bot & \text{if } \mathsf{wint} = \bot \\ \mathsf{into}_{\mathsf{wint}}(\mathsf{tnum}) & \text{if } \mathsf{wint} = \top \\ \llparenthesis \rho_{\mathsf{wint}_{\text{lower}}}(a, \mathsf{tnum}), \rho_{\mathsf{wint}_{\text{upper}}}(b, \mathsf{tnum}) \rrparenthesis & \text{if } \mathsf{wint} = \llparenthesis a, b \rrparenthesis \end{cases}$$

4 Transfer Function

In Bincaml, we define the `Domain` module over the `StateAbstraction` $V \rightarrow (\mathsf{tnum}, \mathsf{wint})$, but still want to the use the transfer functions for $V \rightarrow \mathsf{tnum}$ and $V \rightarrow \mathsf{wint}$ to take advantage of the conditional semantics provided for wrapped intervals. As an implementation detail, `Known_bits.Domain` and `Wrapped_intervals.Domain` supply a function `tf` which accepts a `read` function and the evaluated and abstract statement and returns an iterator of updates to the state.

In the transfer function of the reduced product, we define `read` functions of the form `fun k -> (read k dom).wint` to only read the appropriate part of the state, without needing to construct separate maps. To update the state of the reduced product, we first apply only the tnum updates, then the wint updates. This avoids having to zip maps with differing keys and a rather large footgun related to module sealing.

If the statement is an `assert` or `guard`, we collect the variables updated by either analysis and run the reduction on only those variables to propagate the effects of conditional semantics. All evaluation functions in the `ValueAbstraction` (excluding `eval_const`) run reduce on the final result. `eval_intrin` does not perform reduction in the intermediate steps.