Implementation Notes for Wrapped Intervals in Bincaml

1 Background

In low-level representations, values lose typing information and are simply treated as bitvectors. These representations do not specify signedness as a property of a value, as we would in a high-level context, but as part of the semantics of an operation. When defining interval semantics for bitvectors, selecting between signed or unsigned semantics will always result in a loss of precision when needing to interpret as the other signedness.

Wrapped intervals use a circular model of the integers to allow changing signedness depending on the context. We divide the circle into two equal hemispheres, $(\!|0^w, 01^{w-1}|\!)$ and $(\!|10^{w-1}, 1^w|\!)$, creating a north and south pole.

$$\mathsf{np} = (\!|01^{w-1}, 10^{w-1}|\!) \qquad \mathsf{sp} = (\!|1^w, 0^w|\!)$$

By cutting the circle at the north pole and stretching it out into a line, we are able to interpret our intervals as signed, whilst making a cut at the south pole creates the unsigned number line.

Our lattice consists of the empty interval $\bot$, the full interval $\top$ and the delimited interval $(\!|x, y|\!)$. To prevent having multiple representations for the full interval, any delimited interval $(\!|y +_w 1, y|\!)$ is normalised to $\top$. For the partial order, $s \sqsubseteq t$ if and only if s is contained fully within t.

Delimited intervals are defined as containing all the values encountered when moving clockwise on the circle from x to y, including x and y themselves. As such $(\!|001, 011|\!) \neq (\!|011, 001|\!)$, with $(\!|001, 011|\!)$ being the shorter path between the endpoints.

2 Known Widths

> **NOTE**
>
> This section is now obsolete after merging #26 which provides types to the `ValueAbstraction` and #40 which uses these types instead of inference. However, we will keep it around for historical reference.

Multiple functions specified in the paper [1] rely on knowing the length of the bitvector being operated upon, even if said bitvector has an abstract state of $\top$ or $\bot$. Such functions include cardinality (written as $\#(\top)$ and $\#(\!|x, y|\!)$) and the operations which split an interval at either pole $\mathsf{nsplit}(s)$ and $\mathsf{ssplit}(s)$, and their combined version $\mathsf{cut}(s)$.

$$\#(\top) = 2^w$$

$$\mathsf{nsplit}(\top) = \mathsf{ssplit}(\top) = \mathsf{cut}(\top) = \{(\!|0^w, 01^{w-1}|\!), (\!|10^{w-1}, 1^w|\!)\}$$

Whilst these cases only require a known width when the provided input is $\top$, due to the existence of the complement operator, a width must be stored for $\bot$ as well. If $\overline{\top_w} = \bot$ then applying the complement would cause us to lose information, preventing us from satisfying the double negation law $\overline{\overline{\top_w}} = \top_w$.

We choose to represent this in OCaml using a record pairing the abstract interpretation with the bitwidth. Even though it is redundant to store a bitwidth for the interval case, as this can be derived from the bitvectors, we still store it to make pattern matching the elements of lattices nicer to read.

```OCaml
(* Objectively worse version *)
type t =
  | Top of int option
  | Interval of { lower : Bitvec.t; upper : Bitvec.t }
  | Bot of int option
```

```OCaml
(* Some redundant information, but easier to work with *)
type l = Top | Interval of { lower : Bitvec.t; upper : Bitvec.t } | Bot
type t = { w : int option; v : l }
```

As for prior art, the Crab static analysis library [2] implements this data structure using two bitvectors of equal length and a boolean flag for the bottom case. Top is represented by any interval where the cardinality is 2^w and `is_bottom` $=$ `false`. This approach works well for C++ with it's lack of tagged unions and pattern matching, however, the version provided above is more natural for functional programming.

Operations such as the partial order $\sqsubseteq$ and equality are defined on the lattice type `l` and deferred to by the versions for `t` to satisfy the module definition.

2.1 Propagation and Inference

The signature of `module Lattice` in Bincaml requires a constant `val bottom : t` to initialise all variables with prior to determining fixpoints. Since the width of the variable is not provided as an input, we fall back to `{ w = None; v = Bot }`.

Whilst most variables will have a known width as soon as they are assigned a constant or expression containing a constant, it is still necessary to propagate width information to `Bot` and `Top` during analysis. To ensure the precision of signedness-agnostic multiplication, the `cut` operation is used, which requires a known width if $s_v = \top$. A similar situation exists for the bitwise logic, extension, truncation and shifting operators, which frequently use `nsplit` and `ssplit`.

$\mathsf{infer}(s, t)$ attempts to resolve the bitwidth using the width of another object. This relies on the assumption that binary operations will only be performed on bitvectors of the same length (enforced by the type checker). Lattice operations such as join and widening operate on their inputs after inference. This is also the case for the evaluation semantics of binary operations.

$$\mathsf{infer}(s,t) = \begin{cases} (s,t) & \text{if } s_w = t_w \\ (\{\mathsf{Some}\ w; s_v\}, \{\mathsf{Some}\ w; t_v\}) & \text{if } s_w = \mathsf{Some}\ w \vee t_w = \mathsf{Some}\ w \\ \mathsf{fail} & \text{if } s_w = \mathsf{Some}\ w_1 \wedge t_w = \mathsf{Some}\ w_2 \wedge w_1 \neq w_2 \end{cases}$$

3 Least Upper Bound

The join operation provided by the paper is imperfect. It is not associative, nor monotone and biased towards a lexicographically smaller lower bound when joining two disjoint intervals of equal size. Due to the large impact application order has on precision when joining multiple intervals, the paper provides an alternative join implementation $\widetilde{\bigsqcup} S$ for a set of intervals which exploits this bias.

The method consists of two iterations over a set of intervals sorted lexicographically by lower bound. The first iteration finds the least upper bound of all intervals which cross the South Pole (where $\{s \in S \mid (\!|1^w, 0^w|\!) \sqsubseteq s\}$). The second iteration finds the largest uncovered gap in S, the complement of which is the least upper bound of the set.

Figure 3 in the paper mentions an operation $\mathsf{extend}(s,t)$ which is "identical to $\widetilde{\sqcup}$, except that the last cases are omitted". The specific cases to eliminate are not mentioned and the Crab implementation does not bother implementing $\widetilde{\bigsqcup}$, instead opting to left fold using $\widetilde{\sqcup}$. Due to this ambiguity and the fact that it does not break things whilst testing, we assume $\mathsf{extend}(s,t) = s \mathbin{\widetilde{\sqcup}} t$.

A best guess to why this is mentioned is that the check for which interval has the lowest lower bound in the fifth case is made redundant by sorting the set prior to iteration, and can be removed to reduce time spent computing conditions.

4 Concat

The concat operation in Basil does not have an equivalent in the paper or Crab and had to be derived from scratch, through much trial and error. Even though the variant found in the IR takes multiple arguments, we have presented the function as a binary operation for clarity.

$$\mathsf{concat}(s,t) = \begin{cases} \bot & \text{if } s = \bot \vee t = \bot \\ (\!|\mathsf{concat}(a,0^w), \mathsf{concat}(b,1^w)|\!) & \text{if } s = (\!|a,b|\!) \wedge \mathsf{sp} \sqsubseteq t \\ (\!|\mathsf{concat}(a,c), \mathsf{concat}(b,d)|\!) & \text{if } s = (\!|a,b|\!) \wedge \mathsf{sp} \not\sqsubseteq t \wedge t = (\!|c,d|\!) \\ (\!|\mathsf{concat}(0^w,c), \mathsf{concat}(1^w,d)|\!) & \text{if } s = \top \wedge \mathsf{sp} \not\sqsubseteq t \wedge t = (\!|c,d|\!) \\ \top & \text{otherwise} \end{cases}$$

When t crosses the south pole, it becomes possible for the values $\mathsf{concat}(x, 0^{w_t})$ and $\mathsf{concat}(x, 1^{w_t})$ (where $x \in s$) to exist in the interval $\mathsf{concat}(s,t)$. This explains why $(\!|\mathsf{concat}(a, \min(c,d)), \mathsf{concat}(b, \max(c,d))|\!)$ is insufficient, as when $\mathsf{sp} \sqsubseteq t$ we fail to include values in $(\!|\mathsf{concat}(a, 0^{w_t}), \mathsf{concat}(a, \min(c,d))|\!)$ and $(\!|\mathsf{concat}(b, \max(c,d)), \mathsf{concat}(b, 1^{w_t})|\!)$.

This is handled in the code by replacing `t` with `{ w = t.w; v = Top }` if it crosses the south pole and using $s = (\!|a,b|\!) \wedge t = \top$ as the condition for the second case, mostly so that everything can be done in a single pattern match without any nested ifs.

5 Conditional Semantics

Conditional semantics allow us to refine our intervals based on assertions and assumptions in the program. We choose to refine when the condition is a binary operation where at least one of the arguments is a variable. The notation $\rho_{s\theta t}(s)$ to denotes the reduction of s when $s\theta t$ holds where $\theta \in \{=, \neq, \leq, <, >, \geq\}$.

5.1 Equalities and Inequalities

The equalities are trivially defined using meet and the complement. The equivalent reduction for t can be derived by the commutativity of $=$ and $\neq$.

$$\rho_{s=t}(s) = s \mathbin{\widetilde{\sqcap}} t \qquad \rho_{s\neq t}(s) = s \mathbin{\widetilde{\sqcap}} \bar{t}$$

For conditions involving inequalities, we require different variants for signed and unsigned. We will only define the unsigned versions here as the signed versions can be derived by replacing umin and umax with smin and smax.

$$\rho_{s\leq t}(s) = \begin{cases} \bot & \text{if } t = \bot \\ s & \text{if } \mathsf{umax} \in t \\ s \mathbin{\widetilde{\sqcap}} (\!|\mathsf{umin}, b|\!) & \text{if } t = (\!|a,b|\!) \end{cases}$$

The second case is used to account for crossing the south pole; if $\mathsf{umax} \in t$ then every value in s is less than or equal to t.

$$\rho_{s<t}(s) = \begin{cases} \bot & \text{if } t = \bot \\ s \mathbin{\widetilde{\sqcap}} (\!|\mathsf{umin}, \mathsf{umax} - 1|\!) & \text{if } \mathsf{umax} \in t \\ \bot & \text{if } t = (\!|a, b|\!) \wedge b = \mathsf{umin} \\ s \mathbin{\widetilde{\sqcap}} (\!|\mathsf{umin}, b - 1|\!) & \text{if } t = (\!|a, b|\!) \wedge b \neq \mathsf{umin} \end{cases}$$

For strictly less than, umax will not ever be included in the reduced interval as there is no value strictly greater than umax. We factor this into the second branch to improve precision. The third case handles the interval $(\!|\mathsf{umin}, \mathsf{umin}|\!)$ (t is guaranteed not to cross the south pole), where applying the final case would produce $\top$ rather than reducing to the empty interval.

The reductions for $\geq$ and $>$ are handled using similar principles:

$$\rho_{s\geq t}(s) = \begin{cases} \bot & \text{if } t = \bot \\ s & \text{if } \mathsf{umin} \in t \\ s \mathbin{\widetilde{\sqcap}} (\!|a, \mathsf{umax}|\!) & \text{if } t = (\!|a, b|\!) \end{cases} \qquad \rho_{s>t}(s) = \begin{cases} \bot & \text{if } t = \bot \\ s \mathbin{\widetilde{\sqcap}} (\!|\mathsf{umin} + 1, \mathsf{umax}|\!) & \text{if } \mathsf{umin} \in t \\ \bot & \text{if } t = (\!|a, b|\!) \wedge a = \mathsf{umax} \\ s \mathbin{\widetilde{\sqcap}} (\!|a + 1, \mathsf{umax}|\!) & \text{if } t = (\!|a, b|\!) \wedge b \neq \mathsf{umax} \end{cases}$$

The reductions for t can be found by flipping the inequality whilst preserving the strictness, such that interval being reduced is on the left side of the condition. This is why we choose to explicitly define $\geq$ and $>$ even though they cannot be passed to `ValueAbstraction.eval_binop` and `Domain.transfer`.

$$\rho_{s\leq t}(t) = \rho_{t\geq s}(t)$$

5.2 Composition

Since `AND` and `OR` are intrinsics (can a variable number of arguments), we need a precise way to find the greatest lower bound of a set of intervals. By using $\widetilde{\bigsqcup}$ and the complement, we construct this operation via De Morgan's laws:

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

For conjunction, for each variable refined we take the meet of their refinements:

$$\rho_{P_1 \wedge \cdots \wedge P_n}(s) = \widetilde{\bigsqcap} \{\rho_p(s) \mid p \in P\}$$

If the predicate P_i cannot be used to refine s then $\rho_{P_i}(s) = s$. This allows us to make the observation that since disjunction uses join, it is only able to refine a variable if every predicate produces a strictly more precise result than s. In the implementation, we use this to avoid computing the least upper bound for a variable by checking if the number of reductions it appears in is equal to the number of predicates. Given the runaway nature of join, we still need to incorporate our original information after the reduction to ensure our result after reduction is at least as precise as before reduction.

$$\rho_{P_1 \vee \cdots \vee P_n}(s) = s \mathbin{\widetilde{\sqcap}} \widetilde{\bigsqcup} \{\rho_p(s) \mid p \in P\}$$

If a predicate is surrounded by a `BoolNOT`, we use logical laws (mostly De Morgan's) to rearrange so that the top-level expression is either a `BinaryExpr` or `ApplyIntrin`.

We directly define implies and its negations to avoid additional recursive calls for simplification.

$$\rho_{P_1 \Longrightarrow P_2} = \rho_{\neg P_1 \vee P_2} \qquad \rho_{\neg(P_1 \Longrightarrow P_2)} = \rho_{P_1 \wedge \neg P_2}$$

5.3 Limitations and Future Work

Currently, we are unable to propagate information back to variables used as sub-expressions of the arguments. A common case is assertions of the form $s \leq \mathsf{extract}_{32,0}(t)$. Since s is an `RVar` and $\mathsf{extract}_{32,0}(t)$ is a `UnaryExpr`, we are only able to perform reduction on s.

This could be resolved using techniques similar to section 4.6 of the Miné tutorial [3] by defining backward abstract evaluation functions. These backwards operators take the original arguments and produce refined arguments based on the new information derived from the condition. The advantage of this method is that we are able to also propagate the effect of assertions to variables which are not directly contained in the assertion (neither argument is required to be `RVar`).

In our current implementation of this analysis, we operate on the control flow graph rather than the data flow graph, as due to the DFG's structure, chaotic iteration will never call `transfer` on `assert`, `assume` or `guard` statements. The alternative is to use Single Static Information (SSI) form [4] to encode the reductions as new assignments which can be later unified with phi nodes. This allows us to retain the performance benefits of using the DFG whilst still being able to use the information in assertions for precision.

6 Miscellanea

6.1 Intersection Issue

The intersection/exact meet function in the paper is incorrectly specified. The correct version was recovered from Crab and is supplied below.

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

6.2 Signed Multiplication

In the conditions of signed multiplication, we check if the multiplication will produce an interval with a cardinality less than 2^w. However, the paper fails to account for this occurring when the interval increases counter-clockwise. Corrected version below.

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

6.3 Division Semantics

Adapted from Crab since the original paper does not specify these. In the Crab implementation, unsigned division uses nsplit which is incorrect. ssplit should be used to interpret both intervals as unsigned. This is likely a typo as nsplit is named `signed_split` in the code, whilst ssplit is named `unsigned_split`.

$$\mathsf{trim\_zero}(s) = \begin{cases} \emptyset & \text{if } s = \bot \\ \{(\!|1, 1^w|\!)\} & \text{if } s = \top \\ \{(\!|1,b|\!)\} & \text{if } s = (\!|a,b|\!) \wedge a = 0 \\ \{(\!|a, 1^w|\!)\} & \text{if } s = (\!|a,b|\!) \wedge b = 0 \\ \{(\!|a, 1^w|\!), (\!|1,b|\!)\} & \text{if } s = (\!|a,b|\!) \wedge 0 \in s \\ \emptyset & \text{otherwise} \end{cases}$$

$$s \mathbin{\tilde{\div}_u} t = \begin{cases} (\!| a \div_{w_u} d, b \div_{w_u} c |\!) & \text{if } s = (\!| a, b |\!) \land t = (\!| c, d |\!) \\ \top & \text{otherwise} \end{cases}$$

$$(\!| a, b |\!) \mathbin{\tilde{\div}_s} (\!| c, d |\!) = \begin{cases} (\!| b \div_{w_s} c, a \div_{w_s} d |\!) & \text{if } \mathsf{msb}(a) = \mathsf{msb}(c) = 1 \\ & \quad \land (b \neq \mathsf{smin} \lor c \neq -1) \land (a \neq \mathsf{smin} \lor d \neq -1) \\ (\!| a \div_{w_s} d, b \div_{w_s} c |\!) & \text{if } \mathsf{msb}(a) = \mathsf{msb}(c) = 0 \\ & \quad \land (a \neq \mathsf{smin} \lor d \neq -1) \land (b \neq \mathsf{smin} \lor c \neq -1) \\ (\!| a \div_{w_s} c, b \div_{w_s} d |\!) & \text{if } \mathsf{msb}(a) = 1 \land \mathsf{msb}(c) = 0 \\ & \quad \land (a \neq \mathsf{smin} \lor c \neq -1) \land (b \neq \mathsf{smin} \lor d \neq -1) \\ (\!| b \div_{w_s} d, a \div_{w_s} c |\!) & \text{if } \mathsf{msb}(a) = 0 \land \mathsf{msb}(c) = 1 \\ & \quad \land (b \neq \mathsf{smin} \lor d \neq -1) \land (a \neq \mathsf{smin} \lor c \neq -1) \\ \top & \text{otherwise} \end{cases}$$

$$s \div_u t = \tilde{\bigsqcup} \{ u \mathbin{\tilde{\div}_u} w \mid u \in \mathsf{nsplit}(s), v \in \mathsf{ssplit}(t), w \in \mathsf{trim\_zero}(v) \}$$

$$s \div_s t = \tilde{\bigsqcup} \{ u \mathbin{\tilde{\div}_s} w \mid u \in \mathsf{cut}(s), v \in \mathsf{cut}(t), w \in \mathsf{trim\_zero}(v) \}$$

6.4 Logical Operations

The bitwise logical operations use the same trick as multiplication and division. We split all intervals at the south pole to get unsigned intervals and collect them using $\tilde{\bigsqcup}$ after applying the unsigned version of the algorithm. The algorithms presented in Chapter 4.3 of Hacker's Delight [5] have been adapted to work with tail recursion.

Crab does not implement these for some reason.

6.5 Shifts

The definitions of shifting operations provided by the paper rely on a constant shift amount. They also state that this can be extended to handle a variable shift amount by computing for every value in the interval and taking the join of the results.

$$s \mathbin{\texttt{<<}} t = \tilde{\bigsqcup} \{ s \mathbin{\texttt{<<}} k \mid k \in t \}$$

Due to the sort in $\tilde{\bigsqcup}$, this method has $O(n \log n)$ time complexity where n is the cardinality of t (which could be at most 2^w). Since it would be computational costly to do otherwise, we only compute shifts for singleton intervals. This method is also used by Crab.

$$s \mathbin{\texttt{<<}} t = \begin{cases} s \mathbin{\texttt{<<}} k & \text{if } t = (\!| k, k |\!) \\ \top & \text{otherwise} \end{cases}$$

A strategy that uses the cardinality as a threshold or clamping the interval to $t \cap (\!| 0, w |\!)$ was briefly considered. However, these methods are not necessary as variable shifts do not occur particularly often in Basil IR after simplification.

6.6 Extract

The extract operation in Basil is not provided by the paper, but can be derived by combining $\mathsf{trunc}_k(s)$ and logical right shift `>>l`. Zero-length extracts are special cased. Truncate and extract take constant integer values for their indices because doing this with an interval would be a nightmare.

$$\mathsf{extract}_{\mathrm{hi},\,\mathrm{lo}}(s) = \mathsf{trunc}_{\mathrm{hi}\,-\,\mathrm{lo}}(s \mathbin{\texttt{>>l}} \mathrm{lo}) \text{ if } \mathrm{hi} \neq \mathrm{lo}$$

References

[1] J. A. Navas, P. Schachte, H. Søndergaard, and P. J. Stuckey, "Signedness-Agnostic Program Analysis: Precise Integer Bounds for Low-Level Code," *Programming Languages and Systems*, vol. 7705. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 115–130, 2012. doi: 10.1007/978-3-642-35182-2_9.

[2] Seahorn, "Crab: A C++ Library for Building Program Static Analyses."

[3] 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, Dec. 2017, doi: 10.1561/2500000034.

[4] C. S. Ananian, "The Static Single Information Form," Master's thesis, 1999.

[5] H. S. Warren, *Hacker's Delight*, 2nd ed. Upper Saddle River, N.J: Addison-Wesley, 2013.