Implementing Value Abstractions in BinCaml: The Known Bits Domain

Sneha Zazen

January 2026

Introduction

The Known Bits domain (TNum) represents sets of bitvectors by tracking which bits are known (0 or 1) and which are unknown (can be either 0 or 1). The implementation uses two bitvectors: `value` contains the known bit values, and `mask` indicates which bits are unknown (1 = unknown, 0 = known).

The domain is defined as:

$$t ::= \text{Bot} \mid \text{TNum of } \{\text{value} : \text{Bitvec.t}, \text{mask} : \text{Bitvec.t}\} \mid \text{Top}$$

Examples

1. **Exactly 5 (binary: 0101): all bits known**
   `value: 0101, mask: 0000`
2. **All even numbers (last bit is 0 - $\mu\mu\mu$0):**
   `value: 0000, mask: 1110` (only last bit known)
3. **All values where bits 0 and 2 are set: μ1μ0**
   `value: 0100, mask: 1010`

Invariant 1. *The core invariant for tnums is: $\texttt{value} \wedge \texttt{mask} = 0$. A bit is either known or unknown (μ): a known bit is represented by its value in `value` with the corresponding `mask` bit set to 0, while an unknown bit is represented by `value` = 0 and `mask` = 1. At no point should both `value` and `mask` have a 1 at the same bit position.*

The constructor `tnum` enforces this invariant by asserting that the bitwise AND of `value` and `mask` is zero, and verifying that both bitvectors are of equal size. Every tnum produced by an operation is returned through `tnum`, ensuring uniform construction throughout the implementation. Intermediate steps in each operation may extract the `value` and `mask` bitvectors separately, but the final result is always reconstructed via `tnum` to guarantee the invariant holds.

Lattice Structure

The Known Bits domain forms a complete lattice $(L, \sqsubseteq, \bot, \top, \sqcup)$, allowing for sound abstract interpretation.

1. Partial Order

The partial order $a \sqsubseteq b$ represents "a is atleast as precise as b" or equivalently "a represents a subset of b's values." Our implementation of the `compare` function gives this ordering by checking two conditions for TNum elements as given in the reference material[3].

First, we verify that a's unknown bits form a subset of b's unknown bits, by checking that $\texttt{mask}_a \wedge \neg\texttt{mask}_b = 0$, ensuring every unknown bit in a is also unknown in b. Second, for all known bits (where $\texttt{mask}_b = 0$), the values must agree. We compute $\texttt{shared\_known} = \texttt{mask}_a \vee \neg\texttt{mask}_b$, which identifies all bit positions that are known in b, and verify that $\texttt{value}_a \wedge \texttt{shared\_known} = \texttt{value}_b \wedge \texttt{shared\_known}$.

2. Join Operation

The join operation $a \sqcup b$ computes the least upper bound, representing the union of value sets. For two TNum elements, the new mask identifies bits that differ between operands or are already unknown in either: $\text{mask}_{result} = (\text{value}_a \oplus \text{value}_b) \vee \text{mask}_a \vee \text{mask}_b$. The new value retains only bits that are known and agree in both operands: $\text{value}_{result} = \neg\text{mask}_{result} \wedge (\text{value}_a \wedge \text{value}_b)$.

widening: It is identical to join. This works because the Known Bits lattice has finite height (bounded by bitvector size), ensuring all ascending chains stabilize without requiring a separate widening operator. For domains with infinite ascending chains, widening would accelerate convergence by jumping to a safe overapproximation. Figure 1 illustrates the lattice structure for a single bit in the Known Bits domain.

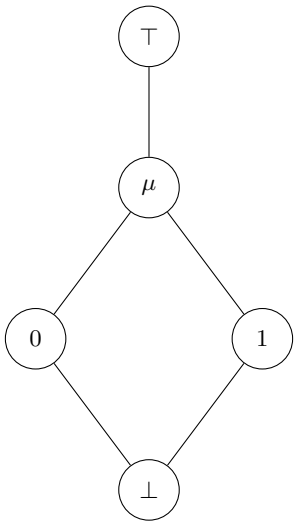

Figure 1: Lattice structure for Known Bits domain.

- Lifted Top ($\top$): The universal top element, representing the absence of any size information — used before a width is established.
- Domain Top (μ): The unknown element of a fixed width, where all mask bits are set to 1 and all value bits are 0 (`value: 0, mask: 1`). The width is inferred from context, e.g. `unknown ~width` constructs this as `tnum (zero ~size:width) (ones ~size:width)`.
- Left node: The certain value 0 at that position (`value: 0, mask: 0`).
- Right node: The certain value 1 at that position (`value: 1, mask: 0`).
- Bottom ($\bot$): The bottom element, representing an unreachable or contradictory state.

Operations on Abstract Values

The implementation uses higher-order functions to abstract common patterns, keeping it consistent and reducing redundancy. The `bind1` combinator lifts unary operations on bitvector pairs (value, mask) to operations on the abstract domain. Similarly, `bind2` handles binary operations. The helper functions `bind1` and `bind2` are used to destructure so that we can extract components and apply operations, handling the lattice structure uniformly. The design and implementation of operations are informed by existing formulations of the `tnum` abstraction in the Linux kernel and by the Known Bits domain implementation in BASIL. The logic for mask and value propagation closely follows the semantics described in the Linux kernel's `tnum` implementation [1] and the Scala-based Known Bits analysis in BASIL [2].

1. Bitwise Operations

Bitwise operations have straightforward implementations in the Known Bits domain. Extension operations handle both zero-extension and sign-extension.

Zero-extension simply extends both value and mask with zeros—the extended bits are known.

Sign-extension extends the value and mask by replicating the sign bit. Bit extraction slices both the value and mask at the specified range.

Bitwise NOT, we flip all known bits while preserving unknown bits unchanged—the mask remains identical.

Bitwise AND propagates known zeros: if either operand has a known 0 bit, the result has a known 0; otherwise the bit may be unknown. The implementation computes the result value as the AND of input values, and the result mask as the OR of input masks.

Bitwise OR dually propagates known ones: if either operand has a known 1 bit, the result has a known 1. The implementation sets result bits to 1 where either input has 1, then marks as unknown only those bits that were unknown in both operands but didn't produce a known 1.

Bitwise XOR can be computed exactly for known bits, but produces unknown results whenever either input bit is unknown.

Shift operations share a common structure: if the shift amount b is unknown (i.e. $\texttt{mask}_b \neq 0$), the result is fully unknown, since we cannot determine how far the bits have moved. If the shift amount is known (i.e. $\texttt{mask}_b = 0$), we apply the shift directly to both `value` and `mask` by the concrete amount $\texttt{value}_b$, preserving the known/unknown structure of a.

Logical left shift (`tnum_shl`) shifts both `value` and `mask` left by $\texttt{value}_b$, filling the vacated low bits with known zeros.

Logical right shift (`tnum_lshr`) shifts both right by $\texttt{value}_b$, filling the vacated high bits with known zeros.

Arithmetic right shift (`tnum_ashr`) shifts both right by $\texttt{value}_b$, replicating the sign bit of `value` and `mask` respectively into the vacated high bits, preserving sign information where it is known.

2. Arithmetic Operations

Arithmetic operations are more complex due to carry and borrow propagation. Addition computes carry chains by analyzing where unknown bits might generate carries. The implementation sums the masks (potential variation), adds this to the sum of values, and uses XOR to identify bits affected by carries. These affected bits, combined with originally unknown bits, form the result mask.

Subtraction similarly tracks borrow propagation using bitwise operations to identify which bits might be affected. Negation is implemented as subtraction from zero. Using simple bitwise operations gives us computational efficiency and accuracy, the logic is blatantly taken from reference papers to avoid silly developer bugs.

3. Multiplication

The multiplication implementation was a little tricky but with generous support it was implemented effectively. The implementation is based on the soundness reasoning given in [3] and closely follows the kernel implementation as of February 2026 [1], adapted to be easier to reason about in a functional context.

Tnum Multiplication

Require: $a = (\texttt{av}, \texttt{am})$, $b = (\texttt{bv}, \texttt{bm})$ are tnums of width w
Ensure: Returns $a \times b$ as a tnum

```
1: acc ← known(0_w)
2: procedure TnumMulAux(acc, a, b)
3:     if av ∨ am = 0 then                          ▷ all remaining bits of a are known zero
4:         return acc
5:     end if
6:     if av ∧ 1 ≠ 0 then                           ▷ LSB of a is a known 1
7:         acc' ← tnum_add(acc, b)
8:     else if am ∧ 1 ≠ 0 then                      ▷ LSB of a is unknown
9:         acc' ← join(acc, tnum_add(acc, b))
10:    else                                         ▷ LSB of a is a known 0
11:        acc' ← acc
12:    end if
13:    a ← tnum_lshr(a, 1)                          ▷ advance to next bit of a
14:    b ← tnum_shl(b, 1)                           ▷ shift b up by one position
15:    return TnumMulAux(acc', a, b)
16: end procedure
17: return TnumMulAux(acc, a, b)
```

The idea is supported by a value-mask decomposition that separates certain and uncertain bit contributions, using `tnum_add` and `join` to compute the final result iteratively. At each step, we take the least significant bit (LSB) of a and multiply it by b, accumulating the result:

- If LSB(a) is a *known* 0: the current accumulator is unchanged.
- If LSB(a) is a *known* 1: b is added to the current accumulator via `tnum_add`.
- If LSB(a) is *unknown*: we join the current accumulator with the sum of the accumulator and b, reflecting that the bit may or may not contribute.

We iterate through the bits of a to build the product incrementally. At each stage, both a is logically shifted to the right reducing size of a at each step and b are shifted to left to emulate multiplication as series of addition collected in the accumulator as at each bit position. An early termination condition fires when $\texttt{value}_a \vee \texttt{mask}_a = 0$, i.e. all remaining bits of a are known zeros, allowing the recursion to stop before processing all bits.

Value Abstraction

The value abstraction evaluate an expression from the IR in abstract space.

1. **Constants** The `eval_const` function abstracts constant values. Concrete bitvectors become fully-known TNum elements using the `known` constructor. Boolean values map to single-bit bitvectors (1 for true, 0 for false). All other constant types that don't have natural bitvector representations return Top.

2. **Unary Operations** The `eval_unop` function builds on the operation type. Supported operations include bitwise NOT, zero-extension, sign-extension, bit extraction, and negation—each delegating to the corresponding domain operation. Unsupported unary operations conservatively return Top to maintain soundness.

3. **Binary Operations** The `eval_binop` function similarly uses binary operations. The implementation supports arithmetic operations (addition, subtraction), bitwise operations (AND, OR, XOR, NAND), and shift operations (logical and arithmetic shifts). As of now complex operations like division and modulo return Top, to keep it sound.

Code Structure

The implementation uses a modular architecture with nested modules. `KnownBitLattice` defines the core lattice structure. `KnownBitsOps` extends this with bitvector-specific operations. `KnownBitsValueAbstraction` adds the expression language interface. Finally, `KnownValueAbstractionBasil` instantiates the analysis for BinCaml's expression language, and the `EasyForwardAnalysisPack` functor integrates it into the dataflow framework.

The Known Bits domain gave developers a great opportunity to work together and learn. This domain was used alongside Wrapped interval to get a more precise value in reduce product domain analysis.

References

[1] Linux Kernel Developers. tnum: Tracking numbers with unknown bits. `https://github.com/torvalds/linux/blob/master/kernel/bpf/tnum.c`, 2026. Accessed January 2026.

[2] UQ-PAC Research Group. Knownbits.scala in basil. `https://github.com/UQ-PAC/BASIL/blob/main/src/main/scala/analysis/KnownBits.scala`, 2026. Accessed January 2026.

[3] Harishankar Vishwanathan, Matan Shachnai, Srinivas Narayana, and Santosh Nagarakatte. Sound, precise, and fast abstract interpretation with tristate numbers. In *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 254–265. IEEE, 2022.