Circular Linear Progressions in SWEET

Linus Källberg[*]
School of Innovation, Design and Engineering
Mälardalen University, Sweden

Version 1.1

Abstract

The SWEET tool offers numerous static program analyses based on abstract interpretation, such as value analyses and a variety of flow analyses. For a long time, the main abstract domain used in SWEET to represent sets of concrete integers has been an interval domain that can model the wraparound behavior of fixed-precision arithmetic. Although this domain is made attractive by its simplicity, it cannot abstract sparse value sets with sufficient precision. Sparse value sets in which the elements are separated by a constant stride commonly occur, e.g., when analyzing array accesses to derive the possible byte offsets into the array. To improve the accuracy of SWEET's analyses in such situations, we extended SWEET with a variant of Sen and Srikant's domain of Circular Linear Progressions, which keeps stride information together with the interval bounds. This report describes our implementation, in which we improved the expressiveness of the original domain by lifting some restrictions on the parameters defining the objects of the domain.

[*]linus.kallberg@mdh.se

Contents

1 Introduction **3**

2 Circular Linear Progressions **6**
2.1 CLP canonization 8
2.2 Computing gap lengths 10

3 Abstract operations **12**
3.1 Splitting CLPs into APs 15
3.2 Set operations 18
3.2.1 Union 18
3.2.2 Intersection 19
3.2.3 Subset-of 19
3.3 Arithmetic operations 20
3.3.1 Unary negation 20
3.3.2 Addition 20
3.3.3 Subtraction 20
3.3.4 Signed multiplication 20
3.3.5 Unsigned multiplication 21
3.3.6 Signed division 21
3.3.7 Unsigned division 23
3.3.8 Signed modulo 23
3.3.9 Unsigned modulo 23
3.4 Bitwise operations 23
3.4.1 NOT 23
3.4.2 AND 24
3.4.3 OR 25
3.4.4 XOR 25
3.4.5 Left shift 25
3.4.6 Signed (arithmetic) right shift 25
3.4.7 Unsigned (logical) right shift 26
3.5 Comparison operators 26
3.5.1 Equality/inequality 27
3.5.2 Order relations 27

4 Implementation **29**

5 Evaluation **31**

6 Conclusions **32**

Acknowledgments **33**

References **34**

1 Introduction

SWEET (SWEdish Execution time analysis Tool) is a static program analysis tool developed by the WCET research group at Mälardalen University in Sweden. Its analyses, based on abstract interpretation, can derive numerous types of information, such as possible values of variables at different program points, upper and lower loop bounds, and infeasible execution paths. These analyses in SWEET are performed on programs represented in the intermediate format ALF (ARTIST2 Language for Flow Analysis) [3]. Designed to facilitate translations of programs from a wide range of sources, ALF is a fairly low-level language in many regards, but includes a few high-level language constructs such as switch-case statements—which are also used to model simple two-way branches as well as loops—and subroutines.

ALF's type system defines only a few basic data types: two's-complement signed and unsigned integers, floats, and pointers to data or code. Moreover, while there is a clear semantic distinction between the numerical types and the pointer types, the differentiation between the different numerical types is largely superficial. Similarly to assembly language, numerical values in ALF are in practice merely bitstrings of certain widths, and their interpretation is implicit in each operation applied to them. Thus, ALF defines separate versions of all operators where the result, in terms of its binary representation, depends on the interpretation of the operand bitstrings.

This text is concerned with the accurate and efficient analysis of fixed-precision integer arithmetic, such as that found in ALF and most other languages and computer systems. In this arithmetic, when the result of an operation does not fit in the available bit width, its most significant bits are discarded, resulting in a "wraparound" effect. This makes this arithmetic an instance of modular arithmetic over $\mathbb{Z}/2^w\mathbb{Z}$, where w denotes the bit width. A challenge in analyzing this arithmetic is that its properties are quite different compared to traditional integer arithmetic over $\mathbb{Z}$. For example, if the sum of two positive numbers wraps around, it becomes smaller than both of the summands.

Due to this wraparound behavior, it aids intuition to think of the values representable in w bits as being placed along a circle of circumference 2^w. Throughout this text, we refer to such a circle as a *number circle*, in analogy to the number line. As shown in Figure 1, we visualize a number circle to resemble a clock face, placing the origin at "12 o'clock", and representing positive increments as clockwise rotations around the circle. Points on the left half of the circle represent negative values $\geq -2^{w-1}$ under signed interpretation and positive values ranging from 2^{w-1} up to and including $2^w - 1$ under unsigned interpretation. Note that since we assume two's-complement representation, the signed and unsigned interpretations of a bitstring always appear at the same point on the number circle.

Abstract interpretation is a technique by which sets of possible concrete values are over-approximated by objects from an abstract domain. Previously, the workhorse abstract domain in SWEET has been a type of interval with built-in support to model wraparound as well as dual signed/unsigned interpretations.

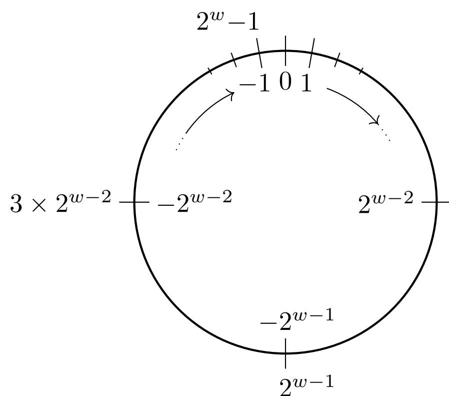


Figure 1: A number circle of circumference 2^w. A double labeling is used for points where the signed and unsigned interpretations differ, with the unsigned label placed on the outside of the circle.

In this domain, a set of w-bit values V is abstracted using lower and upper bounds[1] $l, u \in [0..2^w)$ satisfying $(u - l) \bmod 2^w \geq (v - l) \bmod 2^w$ for all $v \in V$. Note that the Euclidean definition of the modulo operation [1] is adopted here, with which $x \bmod y$ gives the smallest nonnegative value that is congruent to x modulo y, i.e., $0 \leq x \bmod y < |y|$ and $x = x \bmod y + yq$ for some quotient $q \in \mathbb{Z}$. Going back to the number circle analogy, an interval object thus includes all values that are passed over during a clockwise rotation from l to u on the circle. When $l > u$, this means that the interval passes over 0 on the circle. Note also that the bounds are selected from the unsigned range merely for convenience: this has no bearing on which values the interval can represent, and it is trivial to translate the included values between their signed and unsigned interpretations when needed during an analysis. Due to its simplicity, this domain provides very compact abstractions and efficient abstract operations based on a variant of interval arithmetic.

Unfortunately, the interval domain suffers from an inability to abstract sparse value sets with reasonable accuracy. Such value sets commonly result from analyzing, e.g., array accesses, where the possible array offsets are separated by multiples of the width of the elements. Consider, for example, the set of concrete 8-bit values $\{16, 48, 80, 144\}$, as shown laid out along the left-most number circle in Figure 2. The tightest interval abstracting this value set is given by $l = 16$ and $u = 144$. This interval, visualized in the middle in Figure 2, includes all the concrete values $16, 17, \ldots, 144$ (when interpreted as unsigned). Thus, if the value set represents the possible offsets into an array, the elements stored at these offsets will be hugely over-approximated. Furthermore, this over-approximation is quite arbitrary, as any values loaded from the array offsets in-between the multiples of the element widths depend on the endianness

[1] Throughout this report, "..'' is used to indicate integer intervals.

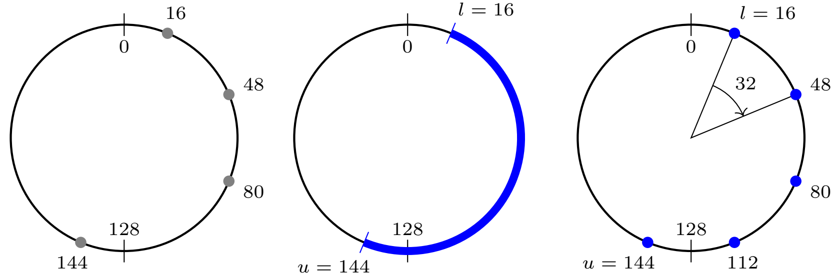


Figure 2: Left: The set of concrete 8-bit values $\{16, 48, 80, 144\}$ laid out along a number circle of circumference $2^8 = 256$. Middle: The tightest interval abstracting the value set. Since $l = 16$ and $u = 144$, this interval includes $144 - 16 + 1 = 129$ concrete values. Right: The same value set abstracted by a Circular Linear Progression. Save for the superfluous value 112, the concrete value set is approximated exactly.

of the modeled architecture.

An earlier version of SWEET included a congruence domain, which could be used in combination with the interval domain to provide stride information. However, this domain was based on mathematical congruences, and did not model wraparound in a safe way. Thus, this domain was abandoned in later versions. Recently, we instead turned to the work by Sen and Srikant [5, 6]. Similarly to our interval domain, their Circular Linear Progression (CLP) domain bounds each concrete value set using lower and upper bounds[2] $l, u \in [0..2^w)$. However, by using an additional stride parameter $s \in [0..2^w)$, they are able to represent a constant dispersion of the values in-between the bounds as well. Assuming an unsigned interpretation, a CLP object thus represents the concrete set of values

$$\{(l + si) \bmod 2^w : i \in \mathbb{Z}, 0 \leq i \leq \lfloor((u - l) \bmod 2^w)/s\rfloor\}. \tag{1}$$

Returning to the earlier example, this domain enables an almost exact abstraction of the concrete value set by setting the stride s to 32. The values represented by such a CLP are visualized on the rightmost number circle in Figure 2.

During the process of implementing the CLP domain in SWEET, we recognized some opportunities for improvements of the domain. Most notably, by lifting some restrictions on the parameters defining a CLP object, we achieve better expressiveness of the domain and improved approximation accuracy. The rest of this report is outlined as follows. Our modified CLP domain is described in Section 2 together with some elementary operations on CLP objects. In Section 3, the full set of abstract operations on CLPs are described. Section 4 gives an overview of the implementation of the domain in SWEET. Then in Section 5,

[2] In fact, they let l and u come from the signed range, but as discussed earlier, this choice is arbitrary.

an evaluation of the implementation is given. Section 6 concludes the report and outlines some directions for future work.

2 Circular Linear Progressions

With Sen and Srikant's original definition, l and u are always selected so as to be no farther apart than $2^w - 1$. Thus, a CLP can never overlap with itself, or "eat its own tail". This is assumed in all their abstract operations, and whenever an operation returns bounds that differ by more than $2^w - 1$ (before wraparound is handled), the result must be "collapsed" into a less precise CLP to uphold this assumption. We remove this restriction altogether to improve the expressiveness of the domain. Instead of using lower and upper bounds to define a CLP, we use a base value $b \in \mathbb{Z}$ and a cardinality $n \in \mathbb{N}_0$ giving the number of concrete values represented. We denote a w-bit CLP by $\mathrm{CLP}_w(b, s, n)$, where $s \in \mathbb{Z}$ gives the stride. Bottom CLPs of width w, which we denote using the shorthand $\bot_w$, are characterized by $n = 0$. Top CLPs, henceforth denoted $\top_w$, have an odd value of s and $n \geq 2^w$, which means they include all values representable in w bits.

We define two separate concretization functions for the signed and unsigned cases. The unsigned concretization function γ_{u} is defined as

$$\gamma_{\mathrm{u}}(\mathrm{CLP}_w(b, s, n)) = \{(b + si) \bmod 2^w : i \in [0..n)\}. \tag{2}$$

Thus, the concrete unsigned values generated by γ_{u} are the elements of an arithmetic progression of length n, with the modulo operation applied to each element. Due to this resemblance to a sequence, we will commonly refer to the variable i as the index. The signed concretization function γ_{s} is similar, but maps any bitstrings in which the sign bit is set to negative values:

$$\gamma_{\mathrm{s}}(\mathrm{CLP}_w(b, s, n)) = \{v_i : i \in [0..n)\}, \text{ where}$$
$$v_i = \begin{cases} (b + si) \bmod 2^w & \text{if } (b + si) \bmod 2^w < 2^{w-1}, \\ (b + si) \bmod 2^w - 2^w & \text{otherwise.} \end{cases} \tag{3}$$

An important fact to note is that these two functions are identical with respect to the bitstring representations of the concrete values they generate. Thus, for two CLPs c_1 and c_2, $\gamma_{\mathrm{s}}(c_1) = \gamma_{\mathrm{s}}(c_2)$ implies $\gamma_{\mathrm{u}}(c_1) = \gamma_{\mathrm{u}}(c_2)$ and vice versa.

Another difference is that our CLP definition allow the parameters b and s to be any values in $\mathbb{Z}$, since the transformation of the concrete values to their wrapped counterparts is implicit in the concretization functions (although some constraints will be imposed on all three CLP parameters in Section 2.1). Sometimes, however, we want to reason about the untransformed values. We therefore define the following additional concretization function that views a CLP as a pure arithmetic progression without wraparound:

$$\gamma(\mathrm{CLP}_w(b, s, n)) = \{b + si : i \in [0..n)\}. \tag{4}$$

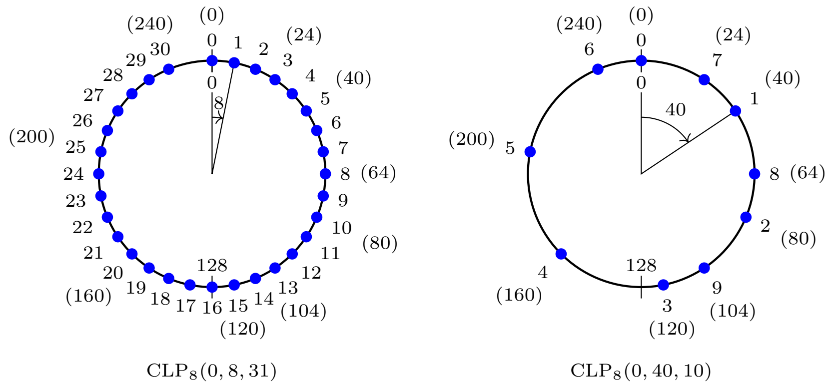


Figure 3: CLPs abstracting the set $\{0, 40, \ldots, 240\} \cup \{24, 64, 104\}$, computed using Sen and Srikant's approach (left) as well as our new approach (right). Shown for each value is the index i; the values included in the original abstracted set are shown in parentheses.

For notational convenience, we also let $c[i]$ stand for the element $b + si$ of $\gamma(\mathrm{CLP}_w(b, s, n))$. It should be pointed out that while $\gamma(c_1) = \gamma(c_2)$ always implies $\gamma_\mathrm{s}(c_1) = \gamma_\mathrm{s}(c_2)$ and $\gamma_\mathrm{u}(c_1) = \gamma_\mathrm{u}(c_2)$, the reverse does not necessarily hold.

As an illustrative example of the benefits of our version of the CLP domain, consider a value analysis performed on the following C code snippet:

```
1  uint8_t x, y;
2  x = user_input();
3  if (x < 10)
4      y = 40 * x;
```

On the line before the branch, the variable x is assigned an unknown value. In the true branch, x must fall in $[0..9]$, and thus $40x$ evaluates to a value in $\{0, 40, \ldots, 360\}$. Since y is 8 bits wide, its contents after the assignment is a value in $\{0, 40, \ldots, 240\} \cup \{24, 64, 104\}$, where the second subset represents the values that are wrapped around because they are larger than $2^8 - 1 = 255$. There is no way to represent the possible contents of y exactly using the definition in [5], since the difference between the first value 0 and last value 360 in the unwrapped concrete set is larger than 255. The resulting CLP with this method is shown to the left in Figure 3 (for convenience, we use our notation for CLPs). Using our CLP definition, on the other hand, the value set can be represented exactly simply by using a stride of 40, as illustrated to the right in Figure 3. Notice in this CLP that the values at $i = 7, 8, 9$ appear in-between values at lower indices.

Although this type of deliberate use of wraparound in a program might be a rare practice, such effects may find their way into the analysis for other reasons, e.g., due to over-approximations earlier in the analysis, or when analyzing partial

code where some variables and results cannot be known. It is also possible for compilers to exploit properties of fixed-precision arithmetic for optimization purposes. Furthermore, the strengths of our modified CLP domain are not limited to the handling of wraparound. On a more general level, its increased expressiveness enables more value sets to be abstracted with high precision, regardless of whether these sets come from overflowing computations or not.

2.1 CLP canonization

To facilitate various operations such as equality comparisons between CLPs, we now impose a few restrictions on the CLP parameters b, s, and n, to ensure that for any pair of CLPs c_1 and c_2 where $\gamma_{\mathrm{s}}(c_1) = \gamma_{\mathrm{s}}(c_2)$ (which also implies $\gamma_{\mathrm{u}}(c_1) = \gamma_{\mathrm{u}}(c_2)$), it holds that $c_1 = c_2$. As noted by Cassé et. al [2], who also employ the CLP domain in their analyses, using the cardinality parameter n in place of an upper bound avoids the problem that several upper bounds may give the same concrete set, since the upper bound is not required to coincide with a value in the set. However, their claim that this measure alone guarantees canonical CLP representations is incorrect, as can be seen below.

First note that if one were to enumerate the values $(b + si) \bmod 2^w$ for $i = 0, 1, \ldots$, eventually a previous value would be revisited, and from this point on the sequence would cycle. Thus, we can calculate a cap k on the cardinality n as the period of this sequence, so that no concrete value is represented more than once in the CLP. Clearly, k satisfies

$$b + si \equiv b + s(i + k) \pmod{2^w},$$

which can be simplified as

$$sk \equiv 0 \pmod{2^w}.$$

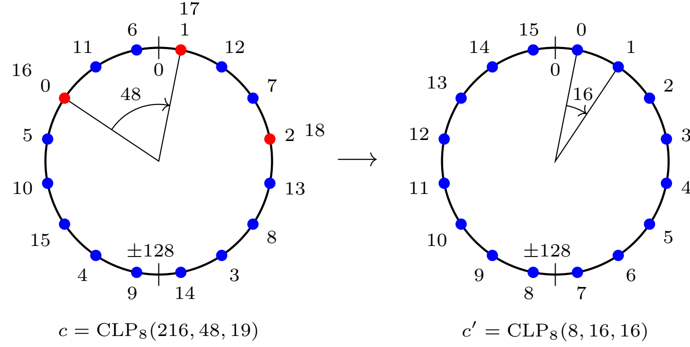
It is straightforward to verify that the smallest positive solution to this congruence is $k = \mathrm{lcm}(|s|, 2^w)/|s|$ if $s \neq 0$, where lcm denotes the least common multiple, and $k = 1$ otherwise. Thus, if $n > k$, at least one concrete value is represented twice by the CLP.

To transform a general CLP object $c = \mathrm{CLP}_w(b, s, n)$ into the canonical CLP $c' = \mathrm{CLP}_w(b', s', n')$, we distinguish between the following cases regarding n and k:

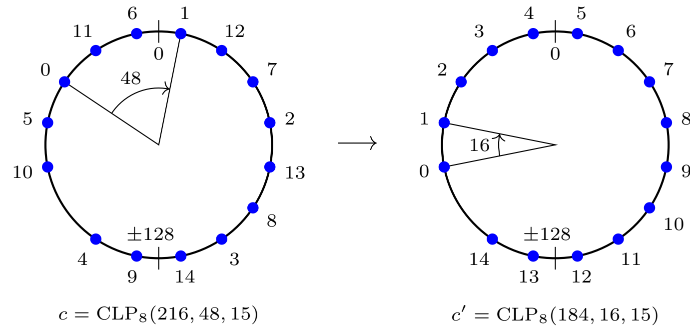
1. If $n = 0$ (the bottom case), we set $b' = s' = n' = 0$.

2. If $n = 1$, we set $b' = b \bmod 2^w$, $s' = 0$, and $n' = 1$.

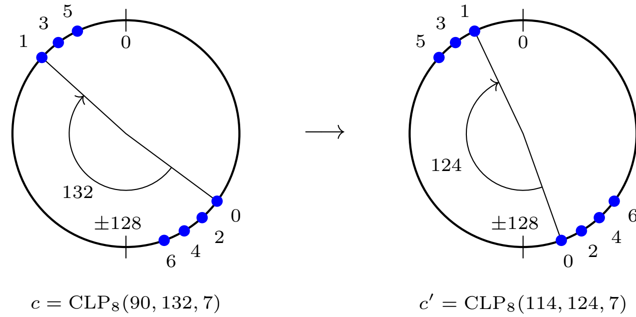
3. If $n \geq k \geq 2$, we set $n' = k$ as per above. Furthermore, in this case the gap length between adjacent concrete values is constant, as illustrated by the example in Figure 4a. We set b' to the smallest unsigned value and s' to the constant gap length, which is given by $\gcd(s, 2^w) \bmod 2^w$. Note that this rule implicitly covers the top case, i.e., where $|\gamma_{\mathrm{u}}(c)| = 2^w$, where it gives $b' = 0$, $s' = 1$, and $n' = 2^w$. It also covers singletons where $n > 1$ and s is a multiple of 2^w, and gives the same result as the previous rule.



(a) Canonization of an 8-bit CLP c with $b = 216$, $s = 48$, and $n = 19$. In this case, $k = \text{lcm}(48, 256)/48 = 16$. Since $n = k + 3$, the last three values overlap with the first three values (shown in red), and the gap length between all adjacent values is constant. The base b' of the canonical CLP c' is set to the smallest unsigned value 8, and the stride s' is set to the constant gap length 16.



(b) The same example as in (a), except $n = k - 1$. This means that there are exactly two gap lengths, and the larger of the two occurs exactly once (between $i = 4$ and $i = 10$ in c). In this case, b' is set to the first value after the larger gap, and s' is set to the smaller gap length.



(c) The CLP to be canonized has $n < k - 1$ and $s \bmod 2^w > 2^{w-1}$. The base b' is set to the last value of c (at $i = 6$) and s' is set to $(-s) \bmod 2^w$. Intuitively, c' is the "reversed" c.

Figure 4: CLP canonization examples. The numbers along the circumference show the index i for each value.

4. If $n = k - 1 \geq 2$, then there are exactly two gap lengths, the larger of which occurs only once, as in the example in Figure 4b. In this case, we set b' to the first value after the larger gap, and s' to the smaller gap length, given by $\gcd(s, 2^w)$. Since $n < k$, we set $n' = n$.

5. In all other cases, there are either three gap lengths, or there are two gap lengths with multiple occurrences of both. In these cases, we restrict b' to the range $[0..2^w)$ and s' to the range $[0..2^{w-1})$. If $s \bmod 2^w < 2^{w-1}$, we simply set $b' = b \bmod 2^w$ and $s' = s \bmod 2^w$. Otherwise, we "reverse" c by setting b' to the last concrete value, i.e., $b' = c[n-1] \bmod 2^w$, and negating the stride: $s' = (-s) \bmod 2^w = 2^w - (s \bmod 2^w)$. An example of this is shown in Figure 4c.

2.2 Computing gap lengths

We are frequently interested in the order in which the elements of a CLP are laid out along the number circle. Since we allow a CLP to span more than one revolution of the number circle, elements from one revolution may be interleaved with elements from previous and/or subsequent revolutions. Thus, the order of the elements in terms of their signed or unsigned interpretations is not necessarily reflected by their indices. A related problem is that of determining the gaps between adjacent elements, which, for the same reason, are not necessarily given by the stride. We solve these two problems by adapting results from the study of fractions to the context of CLPs (see, e.g., [7, 8]).

To be precise, an element $c[j]$ of $c = \mathrm{CLP}_w(b, s, n)$ immediately succeeds another element $c[i]$ in this ordering if the gap length from $c[i]$ to $c[j]$, given by $(c[j] - c[i]) \bmod 2^w$, is smaller than that from $c[i]$ to any other element of c (excluding $c[i]$ itself, of course). Conversely, $c[i]$ then immediately precedes $c[j]$. Intuitively, the gap length can be seen as the clockwise increment needed to get from $c[i]$ to $c[j]$ along the number circle. Thus, the gap length is noncommutative, i.e., the gap length from $c[i]$ to $c[j]$ is not equal to that from $c[j]$ to $c[i]$ in general.

Denote by i_a and i_b the indices of the elements immediately succeeding and preceding the base b, respectively, and let α be the gap length from b to $c[i_\mathrm{a}]$, and let β be the gap length from $c[i_\mathrm{b}]$ to b. Then the Three Gap Theorem, adapted for our purposes, states that the gap length between any two adjacent elements of c is either α, β, or $\alpha + \beta$. Furthermore, it always holds that $i_\mathrm{a} + i_\mathrm{b} \geq n$, and that the gap length α occurs exactly $n - i_\mathrm{a}$ times, β occurs exactly $n - i_\mathrm{b}$ times, and $\alpha + \beta$ occurs exactly $i_\mathrm{a} + i_\mathrm{b} - n$ times. Consequently, in the case $i_\mathrm{a} + i_\mathrm{b} = n$, no gaps of length $\alpha + \beta$ occur at all. Thus, there are at most three gap lengths and at least two, although it may be the case that $\alpha = \beta$.

Given an index $i \in [0..n)$, the following rules give the index of the immediate

successor, $c[\mathrm{succ}(i)]$, and predecessor, $c[\mathrm{pred}(i)]$, of the element $c[i]$:

$$\mathrm{succ}(i) = \begin{cases} i + i_\mathrm{a} & \text{if } i < n - i_\mathrm{a} \text{ (gap length } \alpha\text{)}, \\ i + i_\mathrm{a} - i_\mathrm{b} & \text{if } n - i_\mathrm{a} \leq i < i_\mathrm{b} \text{ (gap length } \alpha + \beta\text{)}, \\ i - i_\mathrm{b} & \text{if } i_\mathrm{b} \leq i \text{ (gap length } \beta\text{)}, \end{cases} \tag{5}$$

$$\mathrm{pred}(i) = \begin{cases} i + i_\mathrm{b} & \text{if } i < n - b \text{ (gap length } \beta\text{)}, \\ i - i_\mathrm{a} + i_\mathrm{b} & \text{if } n - i_\mathrm{b} \leq i < i_\mathrm{a} \text{ (gap length } \alpha + \beta\text{)}, \\ i - i_\mathrm{a} & \text{if } i_\mathrm{a} \leq i \text{ (gap length } \alpha\text{)}. \end{cases} \tag{6}$$

These rules are quite intuitive. For example, since doing i_a steps with stride s from b gives a successor at gap length α from b, doing the same from any element $c[i]$ should give a successor of $c[i]$ at the same gap length. However, such a stepping is valid only if $i < n - i_\mathrm{a}$, because otherwise $\mathrm{succ}(i)$ would give an index outside of c. The same reasoning holds for stepping by $-i_\mathrm{b}$ to end up at a successor at gap length β, which is valid only if $i_\mathrm{b} \leq i$. When $n - i_\mathrm{a} \leq i < i_\mathrm{b}$, neither of these steppings are valid on their own, but stepping by both i_a and $-i_\mathrm{b}$ simultaneously is valid, and gives the gap length $\alpha + \beta$.

Algorithm 1

Input: $c = \mathrm{CLP}_w(b, s, n)$ in canonical form
Output: $i_\mathrm{a} = \mathrm{succ}(0)$, $i_\mathrm{b} = \mathrm{pred}(0)$, α, β

```
 1: if n = 1 then
 2:     return 0, 0, 2^w, 2^w
 3: end if
 4: i_a ← 1; i_b ← 1
 5: α ← s; β ← 2^w − s
 6: while i_a + i_b < n do
 7:     if α < β then
 8:         k ← min(⌊(β − 1)/α⌋, ⌊(n − 1 − i_b)/i_a⌋)
 9:         i_b ← i_b + k·i_a
10:         β ← β − kα
11:     else
12:         k ← min(⌊(α − 1)/β⌋, ⌊(n − 1 − i_a)/i_b⌋)
13:         i_a ← i_a + k·i_b
14:         α ← α − kβ
15:     end if
16: end while
17: return i_a, i_b, α, β
```

While the rules (5) and (6) give constant-time access to the closest neighbors of any element of a CLP, they depend on the indices i_a and i_b being present. Therefore, we now present an algorithm for computing these indices as well as α and β. Interestingly, this algorithm, shown in Algorithm 1, turns out to be very similar to the Euclidean algorithm for computing the greatest common divisor.

On Lines 1–3, the special case that c is a singleton is handled. In fact, this is the only case when any of i_a and i_b become 0. For all other cases, both indices are initialized to 1, and α and β are initialized accordingly (Lines 4 and 5). The task of the loop starting on Line 6 is then to search through the remaining elements of c, in order of increasing indices, to find i_a and i_b. At the beginning of each iteration, the current i_a and i_b give the closest successor and predecessor, respectively, of b from the prefix of c that has been processed so far, the length of which is given by $\max(i_\mathrm{a}, i_\mathrm{b})$. The smallest of the current gap lengths α and β is then used to shrink the other gap length. For example, if $\alpha < \beta$ in an iteration, β is shrunken by α until $\beta \leq \alpha$ by incrementing i_b several times by i_a. The number of such increments is given by k, which is computed in a way that ensures $\beta > 0$ and $i_\mathrm{b} < n$. Since now $\beta \leq \alpha$, the roles are reversed in the next iteration. When $i_\mathrm{a} + i_\mathrm{b} \geq n$, the loop terminates, and the final values of i_a, i_b, α, and β are returned. As mentioned above, this condition always holds for the final values of i_a and i_b. However, intuitively, this condition also means that it is no longer possible to step i_a by i_b or vice versa without stepping beyond $n - 1$, so the process must be finished.

Note that one can apply the Three Gap Theorem to the current prefix of c in each iteration to see that the smallest of α and β must be the smallest gap length in this prefix. Thus, this is the only candidate available to use to shrink the larger of the gaps.

3 Abstract operations

This section defines abstract versions of a majority of the integer operations found in ALF in terms of operations on CLPs. These operations in ALF follow the conventional semantics found in many other languages and architectures, so the definitions presented here should carry over directly also to other program formats than ALF. Similarly to the abstract operations described in [5, 6], some of the operations in this section require that no wraparound occurs in the operand CLPs. Of course, the meaning of wraparound depends on whether the operation in question interprets its operand bitstrings as signed or unsigned. Under signed interpretation, wraparound occurs in a CLP c if the most significant bit (sign bit) flips from 0 to 1 between some element $c[i]$ and the next element $c[i + 1]$; under unsigned interpretation, wraparound occurs if said bit instead flips from 1 to 0.

For clarity, we refer to CLPs in which no wraparound occurs as arithmetic progressions (APs), and denote them using the alias $\mathrm{AP}(b, s, n)$, with analogous meanings of the parameters b, s, and n to the CLP parameters. We say that $c = \mathrm{AP}(b, s, n)$ is *w-bit signed* when $c[i] \in [-2^{w-1}..2^{w-1})$ holds for $i \in [0..n)$, and we call it *w-bit unsigned* when $c[i] \in [0..2^{w})$ holds for $i \in [0..n)$. Equivalently, c is signed if it satisfies $\gamma(c) = \gamma_\mathrm{s}(c)$, and unsigned if it satisfies $\gamma(c) = \gamma_\mathrm{u}(c)$. Of course, an AP can be simultaneously w-bit signed and unsigned, when $c[i] \in [0..2^{w-1})$ holds for $i \in [0..n)$. It should be noted that for signed APs, the base b necessarily falls in the signed range $[-2^{w-1}..2^{w-1})$ as opposed to the unsigned

range $[0..2^w)$, which is a slight deviation from the canonization rules described in Section 2.1.

Whenever wraparound does occur in a w-bit CLP, it must first be split into several w-bit signed or unsigned APs as mandated by the invoked operation. Then the operation is carried out in isolation on each of the resulting APs to produce the same number of results. The final output is then computed by merging these results into one using the union operation to be described in Section 3.2.1. For binary operations, the splitting into APs is done for both operands, and then the partial results are computed by applying the operation to all combinations of APs from the two operands. For all the operations described in this section, this split-compute-merge procedure gives sound results because these operations distribute over the union operation (as discussed in [5, Section 4]). The splitting step is described in detail in the next subsection.

The reason for this no-wraparound policy on operands is to simplify the design of the abstract operations. For a given AP $c = \mathrm{AP}(b, s, n)$, it is known that each $c[i]$ represents the real signed/unsigned concrete value at index i. This further means that the concrete values are monotonically increasing with i, and that they all differ by multiples of the stride s (which, for the reasons discussed in Section 2.2, is not always the case for a general CLP). Furthermore, a proper handling of general CLPs in some of the operations would likely require something similar to the above split-compute-merge strategy anyway, so this can as well be factored out into a separate procedure.

However, while this policy is imposed on all operations except for a few in [5], we note that it is unnecessary in several cases. In other words, for some operations no splitting is required because a sound result is guaranteed even if treating general CLP operands as though they were APs. To motivate this, we first establish more formally the simplifications that are made possible when the operands are APs. Let f_w be a concrete unary operator over bitstrings of width w. Assuming for the moment that f_w is an unsigned operator, its function can be modeled in terms of an underlying mathematical operator f over $\mathbb{Z}$ as

$$f_w(x) = f(x \bmod 2^w) \bmod 2^w. \tag{7}$$

Given a CLP $c = \mathrm{CLP}_w(b, s, n)$, the corresponding abstract operator F_w is sound if and only if it simulates the application of f_w to all elements from $\gamma(c)$. This requirement can be expressed as

$$\gamma_{\mathrm{u}}(F_w(c)) \supseteq \{f_w(c[i]) : i \in [0..n)\},$$

which can be rewritten using (7) as

$$\gamma_{\mathrm{u}}(F_w(c)) \supseteq \{f(c[i] \bmod 2^w) \bmod 2^w : i \in [0..n)\}. \tag{8}$$

If c is in fact a w-bit unsigned AP, we can do away with the modulo operation applied to $c[i]$ in (8), and get the simplified requirement

$$\gamma_{\mathrm{u}}(F_w(c)) \supseteq \{f(c[i]) \bmod 2^w : i \in [0..n)\}. \tag{9}$$

Furthermore, since the modulo operation is now only applied after f, we can exploit the fact that γ_u too applies the modulo operation to each generated element, and simplify one step further:

$$\gamma(F_w(c)) \supseteq \{f(c[i]) : i \in [0..n)\}. \tag{10}$$

Note the use of γ in place of γ_u here.

If f_w is instead a signed operator, its definition in terms of the mathematical operator f becomes

$$f_w(x) = \begin{cases} f(x') \bmod 2^w & \text{if } f(x') \bmod 2^w < 2^{w-1}, \\ f(x') \bmod 2^w - 2^w & \text{otherwise,} \end{cases}$$

where

$$x' = \begin{cases} x \bmod 2^w & \text{if } x \bmod 2^w < 2^{w-1}, \\ x \bmod 2^w - 2^w & \text{otherwise.} \end{cases}$$

This gives a soundness condition for F_w that is analogous to (8), with γ_u replaced by γ_s, and the alternative definition of f_w substituted. It is quite easy to verify that if c is a w-bit signed AP in this case, analogous simplifications can be made to end up with (10).

Our insight is that simplifying (8) into (10) is possible also for general CLPs, if f satisfies, for all $m \in \mathbb{N}$, the following associativity-like property with respect to the modulo operation:

$$f(x) \bmod m = f(x \bmod m) \bmod m. \tag{11}$$

This is the case for, e.g., negation and bitwise NOT, and also the reason why no separate signed and unsigned versions of these operators are needed. The transformation of (8) into (9) then follows directly from (11) for unsigned f_w, and the step from (9) to (10) follows from the definition of γ_u as before. The corresponding simplifications for signed f_w are also valid when (11) is satisfied.

Now consider instead a binary unsigned operator f'_w, defined in terms of a mathematical operator f' over $\mathbb{Z} \times \mathbb{Z}$ as

$$f'_w(x, y) = f'(x \bmod 2^w, y \bmod 2^w) \bmod 2^w.$$

Given two operands $c_1 = \mathrm{CLP}_w(b_1, s_1, n_1)$ and $c_2 = \mathrm{CLP}_w(b_2, s_2, n_2)$, the soundness condition for the corresponding abstract operator F'_w becomes

$$\gamma_\mathrm{u}(F'_w(c_1, c_2)) \supseteq \{f'(c_1[i] \bmod 2^w, c_2[j] \bmod 2^w) \bmod 2^w\}, \tag{12}$$

where $i \in [0..n_1)$ and $j \in [0..n_2)$. If both c_1 and c_2 are w-bit unsigned APs, the same arguments as above simplify this condition into

$$\gamma_\mathrm{u}(F'_w(c_1, c_2)) = \{f'(c_1[i], c_2[j]) \bmod 2^w\}, \tag{13}$$

and then, using the definition of γ_{u}, into

$$\gamma(F'_w(c_1, c_2)) = \{f'(c_1[i], c_2[j])\}. \tag{14}$$

As in the unary case, the same simplifications are valid also for signed f'_w.

For the binary operator f'_w, the "shortcut" for general CLPs described above, is valid if f' is such that the modulo operator distributes over its operands, i.e., if it satisfies

$$f'(x, y) \bmod m = f'(x \bmod m, y \bmod m) \bmod m. \tag{15}$$

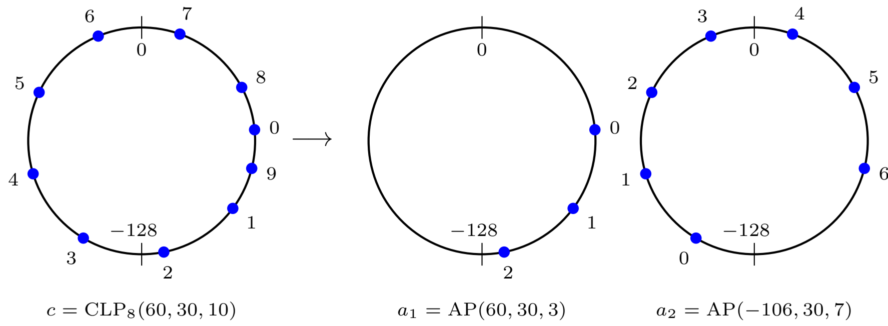
This property holds for, e.g., addition and subtraction, and it is again this property that obviates the need for separate signed and unsigned versions of these operators. Using (15), the simplification of (12) to (13) follows directly, and (14) follows from the definition of γ_{u}. Again, it is straightforward to show that (14) is a sufficient soundness condition also for signed f'_w.

3.1 Splitting CLPs into APs

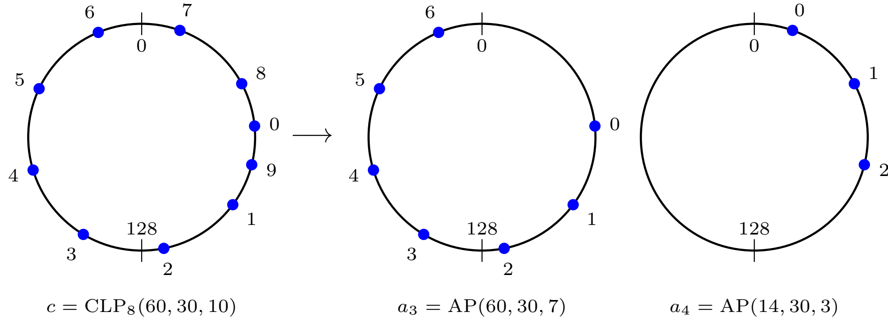
When splitting a CLP c into a number of APs $a_1, a_2, \ldots, a_r$, we want to keep intact as much precision as possible; ideally, we want the resulting APs to satisfy $\bigcup_{j=1}^{r}\{\gamma(a_j)\} = \gamma_{\mathrm{s}}(c)$ (assuming signed a_j; substitute γ_{u} for γ_{s} otherwise). However, at the same time we strive to keep the number of APs low. This owes to the fact that the subsequent operation is then carried out on each of the APs, after which the union of all the partial results must be computed. For binary operations, it is particularly critical to keep the number of APs low, as these operations are invoked for all pairs of APs from both operands. This gives a quadratic time complexity in the number of APs resulting from the splitting of each operand.

In many cases a naïve strategy is sufficient, whereby an "incision" is made at each point on the CLP that passes over the wraparound point, resulting in APs that have the same stride as the original CLP. Figure 5 illustrates this. This is a simple and straightforward strategy that is also used by Sen and Srikant. In their case, at most two APs are generated in any given case. Unfortunately, since we place no restrictions on the number of times a CLP can overlap with itself, for us this strategy generates $O(2^w)$ APs in the worst case. While the canonization rules in Section 2.1 aim to reduce the number of overlaps by keeping the stride below 2^{w-1}, no such rules can reduce this worst case without introducing imprecision.

The worst input CLP in any precision w is arguably one on the format $\mathrm{CLP}_w(b, 2^{w-1} - 1, 2^w - 2)$, where b is either $2^{w-1} - 1$ or $2^w - 1$ depending on whether signed or unsigned APs are to be generated, respectively. Due to the large stride $s = 2^{w-1} - 1$, at most two concrete values occur in each "lap" of such a CLP (as $3s \geq 2^w$). Furthermore, since the cardinality $n = 2^w - 2$ is smaller than $k - 1$ (as $k = \mathrm{lcm}(|s|, 2^w)/|s| = 2^w$) none of the canonization rules reduce the cardinality, and since $s \bmod 2^w < 2^{w-1}$, the last rule does not reduce



(a) Splitting an 8-bit CLP c into 8-bit signed APs. Wraparound occurs after 127, and thus an incision is made in c after $i = 2$, resulting in two APs a_1 and a_2. Note that no imprecision is incurred here, i.e., $\gamma(a_1) \cup \gamma(a_2) = \gamma_{\mathrm{s}}(c)$.



(b) Splitting the CLP from (a) into 8-bit unsigned APs. Since wraparound occurs after 255, c is split after $i = 6$, again resulting in two APs a_3 and a_4 satisfying $\gamma(a_3) \cup \gamma(a_4) = \gamma_{\mathrm{u}}(c)$.

Figure 5: Example CLP where the naïve splitting strategy is sufficient, because only two APs are generated in both the signed and unsigned case.

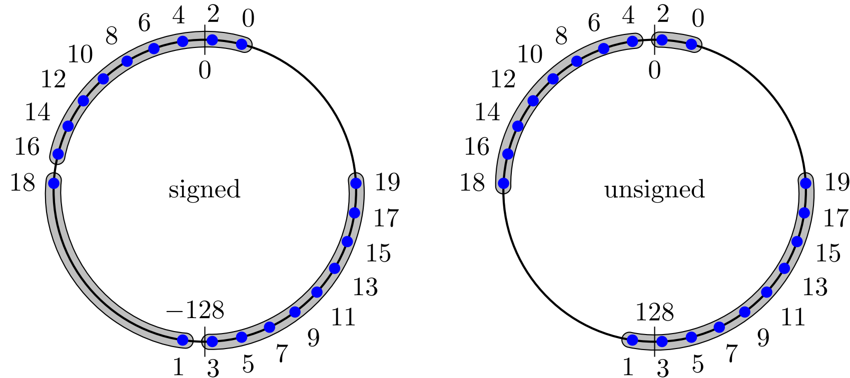


Figure 6: Splitting $\mathrm{CLP}_8(10, 124, 20)$ into signed (left) and unsigned (right) APs using the fallback strategy. The outlines mark which elements are grouped together into distinct APs. In the signed example, we start at the smallest signed value, which is at $i = 1$, and add $i = 18$ to the same group. After $i = 18$, however, the gap length changes, so the current sequence is closed and a new one is started at $i = 16$. This next sequence ends after $i = 0$, when the gap length changes again, and so on. In the unsigned example, we start with the smallest unsigned value at $i = 2$, and then proceed in a similar manner. Using the naïve strategy, 10 APs would result in both cases.

the stride. The above splitting strategy thus produces $O(2^w)$ APs containing at most two concrete values each from such a CLP.

For this reason, we fall back to an alternative splitting strategy for such difficult cases. This strategy utilizes the fact that there typically are sequences of adjacent values that have a common gap length, that can be collected into APs with the stride set to that gap length. With this strategy, a single scan is performed along the circumference of the number circle, and all values are collected into such sequences using a variant of rule (5) from Section 2.2 that allows several steps to be performed in one go. For signed splitting, the scan starts at the smallest signed value, otherwise it starts at the smallest unsigned value. An example is given in Figure 6. The choice between the two strategies is made dynamically for each input case, by first computing in constant time the number of APs that would result with the simple strategy, and then proceeding with a "trial run" using the second strategy. If the number of APs generated ever exceeds the number computed initially, this procedure is terminated prematurely and then the simple splitting procedure is used instead.

As with the naïve strategy, this strategy maintains full precision and takes time proportional to the number of APs generated. Unfortunately, it is harder to analyze to determine its worst-case input in terms of the number of resulting APs. However, investigations of a number of cases suggest that the two strategies complement each other, so that inputs where one strategy is inefficient are handled more successfully by the other strategy. A more rigorous analysis

remains as future work.

In the following subsections, we specify for each operation the expected format of the input CLPs, i.e., whether they are assumed to be general CLPs or signed/unsigned APs. All general CLPs are assumed to be in canonical form, and all APs are assumed to be generated from canonical CLPs. Also, an implicit canonization is performed after each operation, which means that when an operation is defined in terms of other operations, the results from those operations are assumed to be in canonical form as well.

3.2 Set operations

The operations described in this subsection do not correspond to actual concrete operations found in the ALF language, but are needed in many analyses to, e.g., merge abstract program states and to restrict program states according to constraints derived by the analysis.

3.2.1 Union

Input: $c_1 = \mathrm{CLP}_w(b_1, s_1, n_1)$, $c_2 = \mathrm{CLP}_w(b_2, s_2, n_2)$. Let $\cup'$ be a function defined for two CLPs $c_1' = \mathrm{CLP}_w(b_1', s_1', n_1') \neq \bot_w$ and $c_2' = \mathrm{CLP}_w(b_2', s_2', n_2') \neq \bot_w$ as

$$c_1' \cup' c_2' = \mathrm{CLP}_w(b, s, n),$$

where

$$\begin{aligned} b &= \min(b_1', b_2'), \\ s &= \gcd(s_1', s_2', |b_1' - b_2'|), \\ n &= \left\lfloor \frac{\max(c_1'[n_1' - 1], c_2'[n_2' - 1]) - b}{s} \right\rfloor + 1. \end{aligned}$$

Note that $\cup'$ is undefined if $n_1' = n_2' = 1$ and $b_1' = b_2'$, because then $s = 0$ in the definition of n, assuming c_1' and c_2' are in canonical form.

The union operator $\cup$ for CLPs is defined in terms of $\cup'$ as

$$c_1 \cup c_2 = \begin{cases} c_1 & \text{if } c_2 = \bot_w, \\ c_2 & \text{if } c_1 = \bot_w, \\ \mathrm{CLP}_w(\min(b_1, b_2), |b_1 - b_2|, 2) & \text{if } n_1 = n_2 = 1, \\ c_1 \cup' \mathrm{CLP}_w(b_2 + 2^w m, s_2, n_2) & \text{otherwise,} \end{cases} \tag{16}$$

where

$$m = \operatorname*{arg\,min}_{m' \in \mathbb{Z}} |\gamma_{\mathrm{u}}(c_1 \cup' \mathrm{CLP}_w(b_2 + 2^w m', s_2, n_2))|.$$

In other words, m is the value of m' that minimizes the cardinality of $c_1 \cup' \mathrm{CLP}_w(b_2 + 2^w m', s_2, n_2)$.

3.2.2 Intersection

Input: $c_1 = \mathrm{CLP}_w(b_1, s_1, n_1)$, $c_2 = \mathrm{CLP}_w(b_2, s_2, n_2)$. Let

$$
\begin{aligned}
d &= \gcd(s_1, 2^w), \qquad s = \text{the Bézout coefficient of } \gcd(s_1, 2^w) \text{ for } s_1,\\
e &= \gcd(s_2, d), \qquad t = \text{the Bézout coefficient of } \gcd(s_2, d) \text{ for } s_2,\\
j_0 &= t(b_1 - b_2)/e \bmod (d/e),\\
I &= \mathrm{CLP}_{\log_2(2^w/d)}(s(b_2 - b_1 + s_2 j_0)/d, s_2 s/e, \lfloor (n_2 - j_0)/(d/e) \rfloor),
\end{aligned}
$$

and let i_0 be the smallest value of $\gamma_{\mathrm{u}}(I)$ and i_1 the largest value of $\gamma_{\mathrm{u}}(I)$ that is smaller than n_1. An efficient procedure to compute the latter two parameters is given in Section 3.5.2. The intersection operation for CLPs is then defined as

$$
c_1 \cap c_2 = \begin{cases}
\bot_w & \text{if } c_1 = \bot_w \text{ or } c_2 = \bot_w,\\
\bot_w & \text{if } e \text{ does not divide } b_1 - b_2,\\
\bot_w & \text{if } j_0 \geq n_2,\\
\bot_w & \text{if } i_0 \geq n_1,\\
\mathrm{CLP}_w(b, s, n) & \text{otherwise,}
\end{cases} \tag{17}
$$

where

$$
\begin{aligned}
b &= c_1[i_0],\\
s &= s_1 \gcd(|I[k] - I[k']|, \ldots) \text{ for all } k, k' \text{ such that } i_0 \leq I[k], I[k'] \leq i_1,\\
n &= \left\lfloor \frac{c_1[i_1] - c_1[i_0]}{s} \right\rfloor + 1.
\end{aligned}
$$

3.2.3 Subset-of

Input: $c_1 = \mathrm{CLP}_w(b_1, s_1, n_1)$, $c_2 = \mathrm{CLP}_w(b_2, s_2, n_2)$. Let

$$
\begin{aligned}
d &= \gcd(s_2, 2^w),\\
s &= \text{the Bézout coefficient of } \gcd(s_2, 2^w) \text{ for } s_2,\\
J &= \mathrm{CLP}_{\log_2(2^w/d)}(s(b_1 - b_2)/d, s_1 s/d, n_1),\\
j_1 &= \max \gamma_{\mathrm{u}}(J).
\end{aligned}
$$

The parameter j_1 can be computed efficiently using the procedure described in Section 3.5.2. The subset operation for CLPs is then defined as

$$
c_1 \subseteq c_2 = \begin{cases}
\text{true} & \text{if } c_1 = c_2 = \bot_w,\\
\text{false} & \text{if } n_1 > n_2,\\
\text{false} & \text{if } d \text{ does not divide } b_1 - b_2 \text{ and } s_1,\\
\text{false} & \text{if } j_1 \geq n_2,\\
\text{true} & \text{otherwise.}
\end{cases} \tag{18}
$$

3.3 Arithmetic operations

3.3.1 Unary negation

Input: $c_1 = \mathrm{CLP}_w(b_1, s_1, n_1)$.

$$-c_1 = \begin{cases} \bot_w & \text{if } c_1 = \bot_w, \\ \mathrm{CLP}_w(-c_1[n_1 - 1], s_1, n_1) & \text{otherwise.} \end{cases}$$

3.3.2 Addition

Input: $c_1 = \mathrm{CLP}_w(b_1, s_1, n_1)$, $c_2 = \mathrm{CLP}_w(b_2, s_2, n_2)$.

$$c_1 + c_2 = \begin{cases} \bot_w & \text{if } c_1 = \bot_w \text{ or } c_2 = \bot_w, \\ \mathrm{CLP}_w(b, 0, 1) & \text{if } n_1 = n_2 = 1, \\ \mathrm{CLP}_w(b, s, n) & \text{otherwise,} \end{cases}$$

where

$$\begin{aligned} b &= b_1 + b_2, \\ s &= \gcd(s_1, s_2) \\ n &= \left\lfloor \frac{c_1[n_1 - 1] + c_2[n_2 - 1] - b}{s} \right\rfloor + 1. \end{aligned}$$

3.3.3 Subtraction

Input: $c_1 = \mathrm{CLP}_w(b_1, s_1, n_1)$, $c_2 = \mathrm{CLP}_w(b_2, s_2, n_2)$.

$$c_1 - c_2 = c_1 + (-c_2).$$

3.3.4 Signed multiplication

This operator abstracts a concrete multiplication operator where the bit width of the result is the sum of the bit widths of the operands. Although the modulo operator distributes over multiplication, i.e., it holds that $(x \bmod m)(y \bmod m) \bmod m = xy \bmod m$, the operand CLPs need to split into APs for this reason. If one were to define an abstract multiplication operator where the operands and the result have the same precision, the input could be CLPs instead. Also, it would then be unnecessary to have separate signed and unsigned multiplication operators.

Input: w_1-bit signed AP $c_1 = \mathrm{AP}(b_1, s_1, n_1)$, w_2-bit signed AP $c_2 = \mathrm{AP}(b_2, s_2, n_2)$.

$$c_1 \times_{\mathrm{s}} c_2 = \begin{cases} \bot_{w_1+w_2} & \text{if } c_1 = \bot_{w_1} \text{ or } c_2 = \bot_{w_2}, \\ \mathrm{CLP}_{w_1+w_2}(b_1 b_2, 0, 1) & \text{if } n_1 = n_2 = 1, \\ \mathrm{CLP}_{w_1+w_2}(b, s, n) & \text{otherwise,} \end{cases}$$

where

$$
\begin{aligned}
b &= \min(b_1 b_2, b_1 c_2[n_2-1], c_1[n_1-1]b_2, c_1[n_1-1]c_2[n_2-1]),\\
s &= \gcd(|b_1 s_2|, |s_1 b_2|, s_1 s_2),\\
n &= \left\lfloor \frac{\max(b_1 b_2, b_1 c_2[n_2-1], c_1[n_1-1]b_2, c_1[n_1-1]c_2[n_2-1]) - b}{s} \right\rfloor + 1.
\end{aligned}
$$

3.3.5 Unsigned multiplication

Similar remarks about the operands as for the signed operator apply here.

Input: w_1-bit unsigned AP $c_1 = \mathrm{AP}(b_1, s_1, n_1)$, w_2-bit unsigned AP $c_2 = \mathrm{AP}(b_2, s_2, n_2)$.

$$
c_1 \times_{\mathrm{u}} c_2 = \begin{cases} \bot_{w_1+w_2} & \text{if } c_1 = \bot_{w_1} \text{ or } c_2 = \bot_{w_2},\\ \mathrm{CLP}_{w_1+w_2}(b, 0, 1) & \text{if } n_1 = n_2 = 1,\\ \mathrm{CLP}_{w_1+w_2}(b, s, n) & \text{otherwise,} \end{cases}
$$

where

$$
\begin{aligned}
b &= b_1 b_2,\\
s &= \gcd(b_1 s_2, s_1 b_2, s_1 s_2),\\
n &= \left\lfloor \frac{c_1[n_1-1]c_2[n_2-1] - b}{s} \right\rfloor + 1.
\end{aligned}
$$

3.3.6 Signed division

This operator abstracts a concrete integer division operator $\div_{\mathrm{s}}$ that rounds towards 0, i.e., $x \div_{\mathrm{s}} y = \lfloor x/y \rfloor$ when $x/y \geq 0$ and $x \div_{\mathrm{s}} y = \lceil x/y \rceil$ when $x/y < 0$. Let $\div$ denote the mathematical counterpart to this operator.

Input: w-bit signed APs $c_1 = \mathrm{AP}(b_1, s_1, n_1)$, $c_2 = \mathrm{AP}(b_2, s_2, n_2)$, where all $c_2[j]$ have the same sign.

$$
c_1 \div_{\mathrm{s}} c_2 = \begin{cases} \bot_w & \text{if } c_1 = \bot_w \text{ or } c_2 = \bot_w,\\ \top_w & \text{if } 0 \in \gamma(c_2),\\ \mathrm{CLP}_w(b_1 \div b_2, 0, 1) & \text{if } n_1 = n_2 = 1,\\ \mathrm{CLP}_w(b, s, n) & \text{otherwise,} \end{cases}
$$

where

$$b = \min(b_1 \div b_2, b_1 \div c_2[n_2 - 1], c_1[n_1 - 1] \div b_2, c_1[n_1 - 1] \div c_2[n_2 - 1]),$$

$$s = \begin{cases} \gcd(b_1 \div c_2[j] - b, |s_1 \div c_2[j]|, \ldots) \text{ for } j \in [0..n_2) \\ \qquad \text{if all } c_2[j] \text{ divide both } s_1 \text{ and } b_1 \text{, or} \\ \qquad \text{all } c_2[j] \text{ divide } s_1 \text{, and } c_1[n_1 - 1] < 0 \text{ or } 0 < b_1, \\ 1 \qquad \text{otherwise,} \end{cases}$$

$$n = \left\lfloor \frac{\begin{array}{l} \max(b_1 \div b_2, b_1 \div c_2[n_2 - 1], \\ \qquad c_1[n_1 - 1] \div b_2, c_1[n_1 - 1] \div c_2[n_2 - 1]) - b \end{array}}{s} \right\rfloor + 1.$$

When both c_1 and c_2 are singletons, the operation is of course trivial. However, the above definition simplifies in some regards also when only c_2 is a singleton. Firstly, we then have $b_2 = c_2[n_2 - 1]$, which means that the there are only two candidate extrema of the result instead of four, and the definition of b is either $b = b_1 \div b_2$ or $b = c_1[n_1 - 1] \div b_2$. Also, the gcd expression simplifies to $\gcd(b_1 \div b_2 - b, |s_1 \div b_2|)$. If $b = b_1 \div b_2$, the first argument to the gcd becomes 0, so the gcd is simply $|s_1 \div b_2|$. If $b = c_1[n_1 - 1] \div b_2$, the first argument can be rewritten as

$$\begin{aligned} b_1 \div b_2 - b &= b_1 \div b_2 - (c_1[n_1 - 1] \div b_2) \\ &= b_1 \div b_2 - ((b_1 + s_1(n_1 - 1)) \div b_2) \\ &= b_1 \div b_2 - (b_1 \div b_2 + s_1(n_1 - 1) \div b_2) \\ &= -s_1(n_1 - 1) \div b_2. \end{aligned}$$

The third equality follows from the assumption that b_2 divides s_1. Since we assume $s_1 \neq 0$, clearly $|s_1 \div b_2|$ is a divisor of the last form, and consequently also of $b_1 \div b_2 - b$. Thus, for both possible definitions of b, the gcd expression further simplifies to $|s_1 \div b_2|$.

It should also be pointed out that our definition of this operation differs somewhat from that described by Sen and Srikant. Firstly, whereas they pessimistically set the stride of the result to 1 whenever c_2 is not a singleton, our design gives a larger stride when possible. Also, their handling of the case when c_2 is a singleton is unsound, because they always set the stride to $s = |s_1 \div b_2|$, which is only valid under the conditions discussed above. For example, consider the division of $c_1 = \mathrm{CLP}_8(5, 5, 3)$ by $c_2 = \mathrm{CLP}_8(2, 0, 1)$. We have $\gamma_s(c_1) = \{5, 10, 15\}$ and $\gamma_s(c_2) = \{2\}$, and a sound result $c = \mathrm{CLP}_8(b, s, n)$ must satisfy $\gamma_s(c) \supseteq \{2, 5, 7\}$. Their division operator gives $b = 5 \div 2 = 2$, $s = |5 \div 2| = 2$, and an upper bound corresponding to $n = 3$, which gives the incorrect result $\gamma_s(c) = \{2, 4, 6\}$. With our definition, s is instead set to 1, since $b_2 = 2$ does not divide the base of c_1 nor its stride, and n is set to 6.

3.3.7 Unsigned division

Similarly to the operator $\div_\mathrm{s}$, this division operator rounds towards 0.

Input: w-bit unsigned APs $c_1 = \mathrm{AP}(b_1, s_1, n_1)$, $c_2 = \mathrm{AP}(b_2, s_2, n_2)$.

$$c_1 \div_\mathrm{u} c_2 = \begin{cases} \bot_w & \text{if } c_1 = \bot_w \text{ or } c_2 = \bot_w, \\ \top_w & \text{if } 0 \in \gamma(c_2), \\ \mathrm{CLP}_w(b_1 \div b_2, 0, 1) & \text{if } n_1 = n_2 = 1, \\ \mathrm{CLP}_w(b, s, n) & \text{otherwise,} \end{cases}$$

where

$$b = b_1 \div c_2[n_2 - 1]$$

$$s = \begin{cases} \gcd(b_1 \div c_2[j] - b, s_1 \div c_2[j], \ldots) \text{ for } j \in [0..n_2) \\ \qquad \text{if all } c_2[j] \text{ divide } s_1, \\ 1 \qquad \text{otherwise,} \end{cases}$$

$$n = \left\lfloor \frac{c_1[n_1 - 1] \div b_2 - b}{s} \right\rfloor + 1.$$

3.3.8 Signed modulo

This operation abstracts the modulo operation given in § 6.5.5 of the C99 standard, where $x = x \div_\mathrm{s} y \times_\mathrm{s} y + x \,\%_\mathrm{s}\, y$. Thus, the sign of the result is that of the dividend.

Input: w-bit signed APs $c_1 = \mathrm{AP}(b_1, s_1, n_1)$, $c_2 = \mathrm{AP}(b_2, s_2, n_2)$, where all $c_2[j]$ have the same sign.

$$c_1 \,\%_\mathrm{s}\, c_2 = c_1 - c_1 \div_\mathrm{s} c_2 \times_\mathrm{s} c_2.$$

3.3.9 Unsigned modulo

Input: w-bit unsigned APs $c_1 = \mathrm{AP}(b_1, s_1, n_1)$, $c_2 = \mathrm{AP}(b_2, s_2, n_2)$.

$$c_1 \,\%_\mathrm{u}\, c_2 = c_1 - c_1 \div_\mathrm{u} c_2 \times_\mathrm{u} c_2.$$

3.4 Bitwise operations

In the following, the bitwise operators NOT, AND, OR, and XOR are denoted by "$\sim$", "&", "|", and "^", respectively. The left shift, logical right shift, and arithmetic right shift operators are denoted by "$\ll$", "$\gg$", and "$\ggg$", respectively.

3.4.1 NOT

Input: $c_1 = \mathrm{CLP}_w(b_1, s_1, n_2)$.

$$\sim c_1 = \begin{cases} \bot_w & \text{if } c_1 = \bot_w, \\ \mathrm{CLP}_w(\sim c_1[n_1 - 1], s_1, n_1) & \text{otherwise.} \end{cases}$$

3.4.2 AND

Input: w-bit unsigned APs $c_1 = \mathrm{AP}(b_1, s_1, n_1)$, $c_2 = \mathrm{AP}(b_2, s_2, n_2)$.

$$c_1 \,\&\, c_2 = \begin{cases} \bot_w & \text{if } c_1 = \bot_w \text{ or } c_2 = \bot_w, \\ \mathrm{CLP}_w(b_1 \,\&\, b_2, 0, 1) & \text{if } n_1 = n_2 = 1, \\ \mathrm{CLP}_w(b_1 \,\&\, b_2, (b_1 \,\&\, c_2[n_2-1]) - (b_1 \,\&\, b_2), 2) & \text{if } n_1 = 1 \text{ and } n_2 = 2, \\ \mathrm{CLP}_w(b_1 \,\&\, b_2, (c_1[n_1-1] \,\&\, b_2) - (b_1 \,\&\, b_2), 2) & \text{if } n_1 = 2 \text{ and } n_2 = 1, \\ \mathrm{CLP}_w(b, s, n) & \text{otherwise,} \end{cases}$$

where b, s, and n are computed as follows. If $n_1 > 1$, let L_1 be the position of the least significant bit that is set in s_1, and let U_1 be the most significant bit position where b_1 and $c_1[n_1 - 1]$ differ (i.e., the most significant bit set in $b_1 \,\hat{}\, c_1[n_1 - 1]$). The range $[L_1..U_1]$ then gives the bit positions where the values in $\gamma(c_1)$ may differ. Note that $L_1 \le U_1$ is guaranteed to hold. If $n_1 = 1$, simply let $L_1 = w$ and $U_1 = -1$ to represent that all bits are constant. Define L_2 and U_2 analogously for c_2.

If $L_1 < L_2$, then let L be the position of the least significant bit that is set in bits $[L_1..L_2)$ of b_2, or let $L = L_2$ if no such bit exists. Conversely, if $L_2 < L_1$, let L be the position of the least significant bit that is set in bits $[L_2..L_1)$ of b_1, or let $L = L_1$ if there is no such bit. If $L_1 = L_2$, let L be equal to the common value. Note that L gives the least significant bit position where the concrete values of the result may differ. Similarly, we can find the position U of the most significant bit in which the result varies. If $U_1 > U_2$, let U be the most significant bit that is set in bit positions $(U_2..U_1]$ of b_2, or let $U = U_2$ if no such bit exists. Conversely, if $U_2 > U_1$, let U be the most significant bit that is set in bit positions $(U_1..U_2]$ of b_1, or let $U = U_1$ if there is no such bit. If $U_1 = U_2$, let U be equal to the common value. If $L \le U$, the range $[L..U]$ gives the bit positions in which the elements of the result differ. Otherwise, the result is a singleton, so

$$b = b_1 \,\&\, b_2, \qquad s = 0, \qquad n = 1.$$

If $U_1 > U_2$ and $U = U_1$, let U' be the least significant bit in the range $(U_2..U_1]$ such that bits $[U'..U_1]$ of b_2 are all 1. If $U_1 \ne U$, no such bit can exist; then let $U' = U + 1$. The significance of U' is that bits $[U'..U_1]$ of the result are identical to the same bits of c_1 (if $U' < U_1$), and thus increase monotonically with the index of c_1. Furthermore, bits $[L..U')$ of the result vary possibly non-monotonically. Conversely, if $U_2 > U_1$ and $U = U_2$, let U' be the least significant bit in the range $(U_1..U_2]$ such that bits $[U'..U_2]$ of b_1 are all 1s, or let $U' = U + 1$ if $U \ne U_2$. Then bits $[U'..U_2]$ of the result are instead identical to the same bits of c_2.

Define a safe lower bound $l = (b_1 \,\&\, b_2) \,\&\, {\sim}m$ and a safe upper bound $u = \min((c_1[n_1-1] \,\&\, c_2[n_2-1]) \,|\, m, c_1[n_1-1], c_2[n_2-1])$, where m is a bit mask

with 1 in bit positions $[L..U')$ (if $L < U'$) and 0 in all other positions. Then

$$s = \begin{cases} \max(s_1, 2^L) & \text{if } U_1 > U_2,\ U = U_1, \text{ and } U' = L, \\ \max(s_2, 2^L) & \text{if } U_2 > U_1,\ U = U_2, \text{ and } U' = L, \\ 2^L & \text{otherwise,} \end{cases}$$

$$b = (b_1 \mathbin{\&} b_2) + s \left\lceil \frac{l - (b_1 \mathbin{\&} b_2)}{s} \right\rceil,$$

$$n = \left\lfloor \frac{u - b}{s} \right\rfloor + 1.$$

3.4.3 OR

Input: w-bit unsigned APs $c_1 = \mathrm{AP}(b_1, s_1, n_1)$, $c_2 = \mathrm{AP}(b_2, s_2, n_2)$.

$$c_1 \mid c_2 = {\sim}({\sim}c_1 \mathbin{\&} {\sim}c_2).$$

3.4.4 XOR

Input: w-bit unsigned APs $c_1 = \mathrm{AP}(b_1, s_1, n_1)$, $c_2 = \mathrm{AP}(b_2, s_2, n_2)$.

$$c_1 \mathbin{\hat{}} c_2 = ({\sim}c_1 \mathbin{\&} c_2) \mid (c_1 \mathbin{\&} {\sim}c_2).$$

3.4.5 Left shift

Input: $c_1 = \mathrm{CLP}_{w_1}(b_1, s_1, n_1)$, w_2-bit unsigned AP $c_2 = \mathrm{AP}(b_2, s_2, n_2)$.

$$c_1 \ll c_2 = \begin{cases} \bot_{w_1} & \text{if } c_1 = \bot_{w_1} \text{ or } c_2 = \bot_{w_2}, \\ \mathrm{CLP}_{w_1}(b, s, n) & \text{otherwise,} \end{cases}$$

where

$$b = b_1 \ll b_2,$$

$$s = \begin{cases} s_1 \ll b_2 & \text{if } n_2 = 1, \\ \gcd(b_1, s_1) \ll b_2 & \text{otherwise,} \end{cases}$$

$$n = \left\lfloor \frac{(c_1[n_1 - 1] \ll c_2[n_2 - 1]) - b}{s} \right\rfloor + 1.$$

3.4.6 Signed (arithmetic) right shift

Input: w_1-bit signed AP $c_1 = \mathrm{AP}(b_1, s_1, n_1)$, w_2-bit unsigned AP $c_2 = \mathrm{AP}(b_2, s_2, n_2)$.

$$c_1 \gg c_2 = \begin{cases} \bot_{w_1} & \text{if } c_1 = \bot_{w_1} \text{ or } c_2 = \bot_{w_2}, \\ \mathrm{CLP}_{w_1}(b_1 \gg b_2, 0, 1) & \text{if } n_1 = n_2 = 1, \\ \mathrm{CLP}_{w_1}(b, s, n) & \text{otherwise,} \end{cases}$$

where

$$b = \begin{cases} b_1 \gg c_2[n_2-1] & \text{if } b_1 \geq 0, \\ b_1 \gg b_2 & \text{otherwise,} \end{cases}$$

$$s = \begin{cases} \gcd(s_1 \gg c_2[n_2-1], (b_1 \gg (c_2[n_2-1] - s_2)) - (b_1 \gg c_2[n_2-1])) \\ \qquad \text{if } s_1 \text{ is divisible by } 2^{c_2[n_2-1]} \text{ and } n_2 = 1, \text{ or} \\ \qquad \text{both } s_1 \text{ and } b_1 \text{ are divisible by } 2^{c_2[n_2-1]}, \text{ or} \\ \qquad s_1 \text{ is divisible by } 2^{c_2[n_2-1]} \text{ and bits } [0..c_2[n_2-1]) \text{ of } b_1 \text{ are all } 1, \\ 1 \qquad \text{otherwise,} \end{cases}$$

$$n = \begin{cases} \left\lfloor \frac{(c_1[n_1-1] \gg b_2) - b}{s} \right\rfloor + 1 & \text{if } b_1 \geq c_1[n_1-1], \\ \left\lfloor \frac{(c_1[n_1-1] \gg c_2[n_2-1]) - b}{s} \right\rfloor + 1 & \text{otherwise.} \end{cases}$$

3.4.7 Unsigned (logical) right shift

Input: w_1-bit unsigned AP $c_1 = \text{AP}(b_1, s_1, n_1)$, w_2-bit unsigned AP $c_2 = \text{AP}(b_2, s_2, n_2)$.

$$c_1 \gg c_2 = \begin{cases} \perp_{w_1} & \text{if } c_1 = \perp_{w_1} \text{ or } c_2 = \perp_{w_2}, \\ \text{CLP}_{w_1}(b_1 \gg b_2, 0, 1) & \text{if } n_1 = n_2 = 1, \\ \text{CLP}_{w_1}(b, s, n) & \text{otherwise,} \end{cases}$$

where

$$b = b_1 \gg c_2[n_2-1],$$

$$s = \begin{cases} \gcd(s_1 \gg c_2[n_2-1], (b_1 \gg (c_2[n_2-1] - s_2)) - (b_1 \gg c_2[n_2-1])) \\ \qquad \text{if } s_1 \text{ is divisible by } 2^{c_2[n_2-1]} \text{ and } n_2 = 1, \text{ or} \\ \qquad \text{both } s_1 \text{ and } b_1 \text{ are divisible by } 2^{c_2[n_2-1]}, \text{ or} \\ \qquad s_1 \text{ is divisible by } 2^{c_2[n_2-1]} \text{ and bits } [0..c_2[n_2-1]) \text{ of } b_1 \text{ are all } 1, \\ 1 \qquad \text{otherwise,} \end{cases}$$

$$n = \left\lfloor \frac{(c_1[n_1-1] \gg b_2) - b}{s} \right\rfloor + 1.$$

3.5 Comparison operators

All the comparison operators in ALF return the outcome as a 1-bit value, where 0 means false and 1 means true. In the abstract semantics, the result may be both false and true, or neither.

3.5.1 Equality/inequality

Input: $c_1 = \mathrm{CLP}_w(b_1, s_1, n_1)$, $c_2 = \mathrm{CLP}_w(b_2, s_2, n_2)$.

$$(c_1 == c_2) = \begin{cases} \bot_1 & \text{if } c_1 = \bot_w \text{ or } c_2 = \bot_w, \\ \mathrm{CLP}_1(1,0,1) & \text{if } n_1 = n_2 = 1 \text{ and } b_1 = b_2, \\ \mathrm{CLP}_1(0,0,1) & \text{if } c_1 \cap c_2 = \bot_w, \\ \top_1 & \text{otherwise,} \end{cases}$$

$$(c_1 \mathrel{!=} c_2) = {\sim}(c_1 == c_2).$$

3.5.2 Order relations

ALF defines signed as well as unsigned versions of the usual order relations less-than, less-then-or-equal-to, etcetera. For brevity, we describe here only the abstract version of the signed less-than operator, $<_\mathrm{s}$, as the remaining abstract operators are designed using similar principles.

Input: $c_1 = \mathrm{CLP}_w(b_1, s_1, n_1)$, $c_2 = \mathrm{CLP}_w(b_2, s_2, n_2)$.

$$c_1 <_\mathrm{s} c_2 = \begin{cases} \bot_1 & \text{if } c_1 = \bot_w \text{ or } c_2 = \bot_w, \\ \mathrm{CLP}_1(1,0,1) & \text{if } \max \gamma_\mathrm{s}(c_1) < \min \gamma_\mathrm{s}(c_2), \\ \mathrm{CLP}_1(0,0,1) & \text{if } \min \gamma_\mathrm{s}(c_1) \geq \max \gamma_\mathrm{s}(c_2), \\ \top_1 & \text{otherwise.} \end{cases}$$

This definition is quite straightforward, and any of the other order relations can be designed similarly. However, the question remains how to compute the minimum and maximum values of $\gamma_\mathrm{s}(c_1)$ and $\gamma_\mathrm{s}(c_2)$ (or $\gamma_\mathrm{u}(c_1)$ and $\gamma_\mathrm{u}(c_2)$ for the unsigned operators) in an efficient way. As we do not require the operands to be APs, their extrema are not necessarily given by their endpoints b_1, b_2, $c_1[n_1-1]$, and $c_2[n_2-1]$. Furthermore, for the reasons discussed in Section 2.2, iterating through the full sets of concrete values to find the extrema does not suffice in general, as this takes $O(2^w)$ time in the worst case.

It turns out that Algorithm 1 can be modified slightly to help with this. The modified algorithm, shown in Algorithm 2, takes as an additional parameter a point l on the number circle, not necessarily an element of the input CLP c, and returns the index i_c of the element at the smallest gap length ζ from this point. If l coincides with an element of c, the returned i_c is the index of this element and ζ is 0.

The first difference from Algorithm 1 is found in the handling of the case that c is a singleton on Line 2, where also i_c and ζ should be given suitable values. Clearly, b must be the closest element to l, so $i_\mathrm{c} = 0$ is returned as well as $\zeta = (b - l) \bmod 2^w$. On Lines 6–12, i_c is initialized for the general case to either 0 or 1, depending on which of the elements b and $c[1]$ is closest to l, and ζ is assigned accordingly.

The next addition to the algorithm is found on Lines 15–19, where i_c and ζ are updated before i_b and β. Conceptually, i_c is first stepped once by i_b, which

Algorithm 2

Input: $c = \mathrm{CLP}_w(b, s, n)$ in canonical form, $l \in \mathbb{Z}$

Output: $i_\mathrm{a} = \mathrm{succ}(0)$, $i_\mathrm{b} = \mathrm{pred}(0)$, i_c, α, β, ζ

1: **if** $n = 1$ **then**
2:     **return** $0, 0, 0, 2^w, 2^w, (b - l) \bmod 2^w$
3: **end if**
4: $i_\mathrm{a} \leftarrow 1; i_\mathrm{b} \leftarrow 1$
5: $\alpha \leftarrow s; \beta \leftarrow 2^w - s$
6: **if** $(b - l) \bmod 2^w < (c[1] - l) \bmod 2^w$ **then**
7:     $i_\mathrm{c} \leftarrow 0$
8:     $\zeta \leftarrow (b - l) \bmod 2^w$
9: **else**
10:     $i_\mathrm{c} \leftarrow 1$
11:     $\zeta \leftarrow (c[1] - l) \bmod 2^w$
12: **end if**
13: **while** $i_\mathrm{a} + i_\mathrm{b} < n$ **do**
14:     **if** $\alpha < \beta$ **then**
15:         $k' \leftarrow \lceil -(\zeta - \beta)/\alpha \rceil$
16:         **if** $k' \leq \lfloor (n - 1 - (i_\mathrm{c} + i_\mathrm{b}))/i_\mathrm{a} \rfloor$ **and** $-\beta + k'\alpha < 0$ **then**
17:             $i_\mathrm{c} \leftarrow i_\mathrm{c} + i_\mathrm{b} + k' i_\mathrm{a}$
18:             $\zeta \leftarrow \zeta - \beta + k'\alpha$
19:         **end if**
20:         $k \leftarrow \min(\lfloor (\beta - 1)/\alpha \rfloor, \lfloor (n - 1 - i_\mathrm{b})/i_\mathrm{a} \rfloor)$
21:         $i_\mathrm{b} \leftarrow i_\mathrm{b} + k i_\mathrm{a}$
22:         $\beta \leftarrow \beta - k\alpha$
23:     **else**
24:         $k' \leftarrow \min(\lfloor \zeta/\beta \rfloor, \lfloor (n - 1 - i_\mathrm{c})/i_\mathrm{b} \rfloor)$
25:         $i_\mathrm{c} \leftarrow i_\mathrm{c} + k' i_\mathrm{b}$
26:         $\zeta \leftarrow \zeta - k'\beta$
27:         $k \leftarrow \min(\lfloor (\alpha - 1)/\beta \rfloor, \lfloor (n - 1 - i_\mathrm{a})/i_\mathrm{b} \rfloor)$
28:         $i_\mathrm{a} \leftarrow i_\mathrm{a} + k i_\mathrm{b}$
29:         $\alpha \leftarrow \alpha - k\beta$
30:     **end if**
31: **end while**
32: **return** $i_\mathrm{a}, i_\mathrm{b}, i_\mathrm{c}, \alpha, \beta, \zeta$

reduces ζ by β to a negative value, and then stepped by i_a (and ζ increased by α) the minimum number of times to make $\zeta \geq 0$ again. Note that in the beginning of each iteration of the loop, $\zeta \leq \min(\alpha, \beta)$ holds; thus, since $\alpha < \beta$ here, it means that $\zeta < \beta$. The number of steps by α, denoted by k', is computed on Line 15. The update to i_c is thus $i_\mathrm{b} + k' i_\mathrm{a}$ and the update to ζ is $-\beta + k'\alpha$. On Line 16, it is checked whether i_c can in fact be stepped by $i_\mathrm{b} + k' i_a$ without appearing beyond $n-1$ (first condition), and that ζ is actually reduced by the update (second condition), which is not necessarily the case. If not, i_c and ζ are left unchanged.

The next new block of code appears on Lines 24–26. Here i_c and ζ are updated using only i_b and β. The number of increments of i_c by i_b, given by k', is computed very similarly to how k is computed on Line 27. The main difference is that ζ is allowed to become 0 (if l coincides with an element of c), whereas α must be positive.

Now, to get the extreme values of $\gamma_\mathrm{s}(c)$, we call Algorithm 2 with $l = -2^{w-1}$. The returned index i_c then gives

$$\min \gamma_\mathrm{s}(c) = \begin{cases} c[i_\mathrm{c}] \bmod 2^w & \text{if } c[i_\mathrm{c}] \bmod 2^w < 2^{w-1}, \\ c[i_\mathrm{c}] \bmod 2^w - 2^w & \text{otherwise,} \end{cases}$$
$$\max \gamma_\mathrm{s}(c) = \begin{cases} c[\mathrm{pred}(i_\mathrm{c})] \bmod 2^w & \text{if } c[\mathrm{pred}(i_\mathrm{c})] \bmod 2^w < 2^{w-1}, \\ c[\mathrm{pred}(i_\mathrm{c})] \bmod 2^w - 2^w & \text{otherwise.} \end{cases}$$

Similarly, to get the extrema of $\gamma_\mathrm{u}(c)$, we call Algorithm 2 with $l = 0$, and compute

$$\min \gamma_\mathrm{u}(c) = c[i_\mathrm{c}] \bmod 2^w,$$
$$\max \gamma_\mathrm{u}(c) = c[\mathrm{pred}(i_\mathrm{c})] \bmod 2^w.$$

4 Implementation

SWEET is written entirely in C++, and being a research tool, its architecture is designed specifically for extensibility. By exposing only high-level interfaces to the different analysis engines, these are kept largely oblivious of aspects such as the actual abstract domain used. Thus, modifying the existing analyses to make use of the new CLP domain simply amounted to adding implementations of a few of these interfaces. The choice between the previous interval domain and the new CLP domain is made using a command-line parameter to SWEET.

Figure 7 shows a conceptual image of the interaction between a SWEET analysis and these interfaces. During analysis of an ALF program, any constants encountered in the program are abstracted into objects implementing the *Value* interface, by calling the method *CreateValue*() in the interface *ValueFactory*. The type of underlying object returned from this method depends on the current user-selected domain. Furthermore, if a constant occurs as the leaf in a

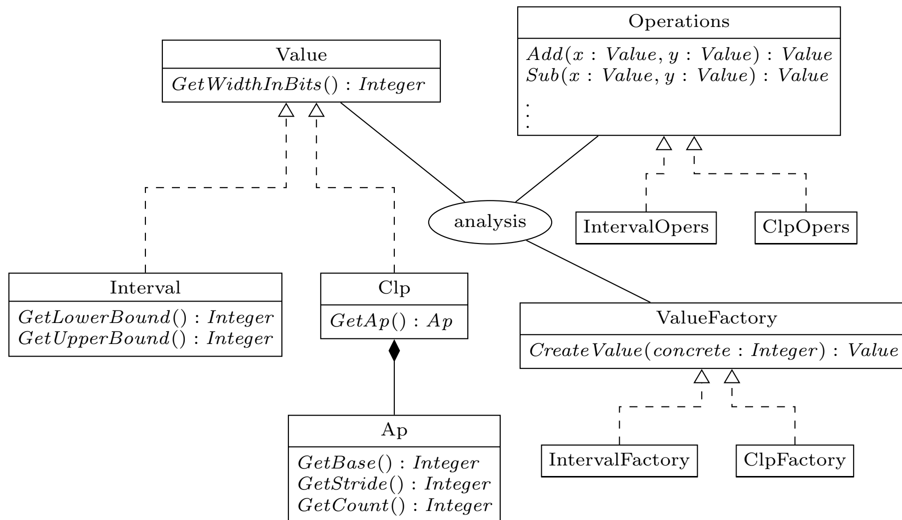


Figure 7: Any analysis in SWEET associates only with high-level interfaces such as *Value*, *Operations*, and *ValueFactory*. This allows the underlying abstract domains to be interchanged as specified by the command-line parameters to SWEET. Each CLP object is represented as an underlying AP, with a width parameter defining the wraparound behavior.

larger expression tree in the code, the (abstract) result of the expression is computed by calling the appropriate methods in the *Operations* interface, starting with the leafs and working upwards to the root. Again, the actual method implementations invoked depends on the current domain setting.

The data structures used to represent the CLP objects, as well as the functions corresponding to the abstract operations, reside in their own files in SWEET's source tree. Each CLP object internally stores an arithmetic progression object as well as a width parameter, the latter corresponding to the w parameter in the CLP definition. The internal AP object, in turn, has data members corresponding to the parameters b, s, and n. Thus, a CLP is in a sense viewed as an AP with the wraparound effects superimposed. All the data members are encapsulated and accessible only through member functions, to ensure that each CLP remains in canonical form at all times, in accordance with the rules in Section 2.1.

Unlike most programming languages, ALF does not limit the possible widths of its data types. Instead, all data types—integers, floats, pointers, etcetera—are parameterized with respect to an arbitrary positive integer specifying their width in bits. Thus, data types with bit widths such as, e.g., 1, 6, 24, or 1024 are supported in ALF. Unfortunately, this renders the basic data types in C++ too inflexible to use for representing any general quantities occurring in an analyzed ALF program. We therefore use an arbitrary-precision integer ("bignum") library to represent the CLP parameters b, s, and n, as well as any

other runtime quantities.

5 Evaluation

All abstract operations and their surrounding functionality were rigorously tested throughout development by adding unit tests to the existing automated testing framework in SWEET. These unit tests, which amount to several hundreds in number, were written such that the abstract result from each tested operation is first compared against an expected abstract result, and then this assumption is itself verified by invoking the concrete computation on all the concrete elements of the operands.

The following simple example C code serves to illustrate both the correctness of the implementation, and the immediate benefits of the new CLP domain compared to the previous interval domain in SWEET.

```
1  volatile int32_t top;
2
3  int32_t main() {
4      int32_t arr[5] = { 110, 140, 100, 130, 120 };
5      int32_t i = top, ret;
6      if (0 <= i && i < 5)
7          ret = arr[i];
8      else
9          ret = 90;
10     return ret;
11 }
```

This code was first translated into ALF through our LLVM-based translator AlfBackend, and then run through the Abstract Execution (AE) analysis in SWEET [4]. Using the output annotations functionality, the possible return values of the program that were derived using the two domains could then be inspected and compared.

On Line 4, the array *arr* is initialized to 5 integers that differ by multiples of 10. The index i is then initialized to $\top_{32}$ by assigning the contents of the volatile variable *top* to it. This simulates a situation where the contents of i depend on, e.g., code that is unavailable to the analysis, or some unknown input. Although the same could be achieved with an auxiliary code annotation setting i to $\top_{32}$, the "trick" used here makes the program self-contained. When analyzing the true branch of the conditional, the AE prunes the possible values of i to $i \in [0..4]$. Thus, although i has an unknown value before the branch, it can be determined to stay within the bounds of *arr* at this point. In the false branch, *ret* is instead assigned the value 90, whose difference to any of the array elements is also a multiple of 10. The program point immediately after the branch is a join point, which means that the AE then merges the abstract values of *ret* from the two branches.

When loading the element *arr*[i] on Line 7, the difference in abstraction

accuracy between the CLP and interval domains manifests itself: $i \in [0..4]$ multiplied by the size of the array elements, which is 4 bytes, is abstracted in the interval domain as $[0..16]$, whereas in the CLP domain its abstraction is exactly $\{0, 4, 8, 12, 16\}$. Thus, with the interval domain, the AE derives the result by loading a 4-byte value from each of the (byte) offsets $0, 1, 2, \ldots, 16$ and then computing the upper bound of these values. The following table shows the concrete values loaded from each of these offsets.

offset	value
0	110
1	−1946157056
2	9175040
3	35840
4	140
5	1677721600
6	6553600
7	25600
8	100
9	−2113929216
10	8519680
11	33280
12	130
13	2013265920
14	7864320
15	30720
16	120

As can be seen, a majority of these values are simply garbage values. Furthermore, the program was translated to assume a little-endian memory model in the ALF program; using big endian would give different garbage values at the offsets in-between the multiples of 4. As expected, the resulting output annotation reported an interval with $l = 1677721600$ and $u = 2156658688$, which represents the signed values $[1677721600..2^{31}) \cup [-2^{31}..9175040]$. With the CLP domain, the corresponding output annotation represented the tightest possible result, namely $\{90, 100, \ldots, 140\}$.

It should also be noted that even if the values in the array were stored as 1-byte integers, in which case the array offsets would be accurately represented by an interval, the possible return values from the program would still be over-approximated as $[90..140]$. Thus, even in this short example code the strengths of the CLP domain show on multiple levels.

6 Conclusions

This report described our version of Sen and Srikant's CLP domain that is currently implemented in the SWEET tool. By removing the restriction that a CLP may span an interval no larger than $2^w - 1$, our CLPs are able to

abstract more value sets with high accuracy. To avoid the issues arising from allowing the CLPs to overlap with themselves an unlimited number of times, we developed new canonization rules (Section 2.1) as well as a procedure that keeps the number of APs low when splitting a CLP (Section 3.1). Furthermore, we are able to skip the splitting step altogether for several operations, thanks to the insight that it becomes unnecessary if the underlying mathematical operation satisfies certain properties with respect to the modulo operation.

The are a number of subjects for future work. For example, the abstract operations in Sections 3.2–3.5 that were defined in terms of other abstract operations would likely improve in terms of accuracy if they were instead given tailor-made definitions, as each added computation step tends to increase the over-approximation of the result. Also, as mentioned in Section 3.1, the splitting strategy based on collecting subsequences with constant gap lengths needs a theoretical analysis to determine its worst-case in terms of the number of APs generated.

Another open question that seems particularly important is how to design a union operation that computes the *optimal* fit of a CLP to the elements of its operands, which the operation described in Section 3.2.1 does not achieve. As the abstract operations that do require the incoming operands to be split into APs rely on this operation to merge their partial results into a final result, their accuracy is tightly connected to the accuracy of the union operation. Furthermore, when invoking the union operation for this purpose, there are typically more than two partial results to be merged, while the union operation described here is binary. In the current solution, the operation is therefore invoked repeatedly on pairs of CLPs in a reduction-style manner until only one CLP remains. However, as the union operation is not associative in general, the accuracy of this reduction is also affected by the order in which it is carried out. The ideal would be a general union operation that takes a set of CLPs and finds an optimal fit to all of them.

Also, it remains to conduct a thorough experimental evaluation on real codes. The short code example in Section 5 gave clear indications of the strengths of the CLP domain compared to the interval domain, and even larger benefits should appear in more large-scale codes where the imprecision of the interval domain risks to propagate further. Such an evaluation could also be used to illustrate the benefits of our modified CLP domain compared to the original domain.

Acknowledgments

This work was supported by the EU FP7 project APARTS, Grant Number 251413. We thank Rathijit Sen for helping to sort out a few questions.

References

[1] Raymond T. Boute. The Euclidean definition of the functions div and mod. *ACM Trans. Program. Lang. Syst.*, 14(2):127–144, April 1992. 4

[2] Hugues Cassé, Florian Birée, and Pascal Sainrat. Multi-architecture Value Analysis for Machine Code. In Claire Maiza, editor, *13th International Workshop on Worst-Case Execution Time Analysis*, volume 30 of *OpenAccess Series in Informatics (OASIcs)*, pages 42–52, Dagstuhl, Germany, 2013. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. 8

[3] Jan Gustafsson, Andreas Ermedahl, Björn Lisper, Christer Sandberg, and Linus Källberg. ALF – a language for WCET flow analysis. In Niklas Holsti, editor, *Proceedings of the 9th International Workshop on Worst-Case Execution Time Analysis (WCET09)*. OCG, June 2009. 3

[4] Jan Gustafsson, Andreas Ermedahl, Christer Sandberg, and Björn Lisper. Automatic derivation of loop bounds and infeasible paths for WCET analysis using abstract execution. In *The 27th IEEE Real-Time Systems Symposium (RTSS 2006)*, December 2006. 31

[5] Rathijit Sen and Y. N. Srikant. Executable analysis using abstract interpretation with circular linear progressions. In *Proceedings of the 5th IEEE/ACM International Conference on Formal Methods and Models for Codesign*, MEMOCODE '07, pages 39–48, Washington, DC, USA, 2007. IEEE Computer Society. 5, 7, 12, 13

[6] Rathijit Sen and Y. N. Srikant. Executable analysis with circular linear progressions. Technical report, Computer Science and Automation, Indian Institute of Science, February 2007. 5, 12

[7] Noel B. Slater. Gaps and steps for the sequence $n\theta \bmod 1$. *Mathematical Proceedings of the Cambridge Philosophical Society*, 63:1115–1123, 10 1967. 10

[8] Tony van Ravenstein. The three gap theorem (Steinhaus conjecture). *Journal of the Australian Mathematical Society (Series A)*, 45:360–370, 12 1988. 10