Taming the Wrapping of Integer Arithmetic

Axel Simon and Andy King

Computing Laboratory, University of Kent, Canterbury, UK
{a.simon,a.m.king}@kent.ac.uk

Abstract. Variables in programs are usually confined to a fixed number of bits and results that require more bits are truncated. Due to the use of 32-bit and 64-bit variables, inadvertent overflows are rare. However, a sound static analysis must reason about overflowing calculations and conversions between unsigned and signed integers; the latter remaining a common source of subtle programming errors. Rather than polluting an analysis with the low-level details of modelling two's complement wrapping behaviour, this paper presents a computationally light-weight solution based on polyhedral analysis which eliminates the need to check for wrapping when evaluating most (particularly linear) assignments.

1 Introduction

Static analysis methods are increasingly used to prove the partial correctness of software [5]. In contrast to formal methods that verify properties of a high-level specification, a static analysis is complicated by low-level details of source code. For instance, while a specification expresses properties over arbitrary integers, variables in a program are usually confined to finite integer types that are deemed to be large enough to hold all values occurring at run-time. On one hand, the use of 32-bit and 64-bit variables make accidental overflows rare and adding checks to each transfer function of the analysis seems to be excessive considering the infrequency of variable overflows. On the other hand, programmers often inadvertently introduce wrapping when converting between signed and unsigned variables and deliberately exploit the wrapping effects of two's complement arithmetic. Thus, wrapping itself should not be considered harmful, particularly when the objective of an analysis is the verification of a different property, such as absence of out-of-bound memory accesses [8,9].

In fact, there is a danger in flagging all wrapping, since any intentional use of wrapping generates a warning message which the developer immediately dismisses as a false positive. The code in Figure 1 illustrates why this is a problem. The purpose of the shown C function is to print how many times each individual character occurs in the given string `*str`. To this end, the elements of `dist` are initialised to zero by the call to `memset`. In the loop that follows, the nth element is incremented each time a character n is encountered in `str`. The `for` loop then prints the distribution of printable ASCII characters.

The shown program is correct on platforms where `char` is unsigned such as Linux on PowerPC. However, for Linux on x86 and MacOS X on PowerPC, `char`

H. Riis Nielson and G. Filé (Eds.): SAS 2007, LNCS 4634, pp. 121–136, 2007.

```
void showDistribution(char* str) {
  int i;
  int dist[256];            /* Table of character counts.*/

  memset(dist, 0, sizeof(dist));             /* Clear table.*/

  while (*str) {
    dist[(unsigned int) *str]++;
    str++;
  };

  for(i=32; i<128; i++)        /* Show dist for printable */
    printf("'%c'␣:␣%i\n", i, dist[i]);       /* characters.*/
}
```

Fig. 1. Example C function that counts the occurrences of each character

is signed. In the latter case, the value `*str` which is used to index into `dist` can take on values in the range $[-128, 127]$. Although the programmer intended to convert the value of `*str` to an unsigned value before the extension to a 4-byte quantity takes place, the C standard [3] dictates that the value of `*str` is first promoted to `int` before the conversion to an unsigned type is performed. Hence, `dist` can be accessed at indices $[2^{32}-128 \ldots 2^{32}-1] \cup [0 \ldots 127]$, of which the first range is out-of-bounds. A static analysis that considers all wrapping to be erroneous would flag this statement as possibly faulty. However, since the programmer expects that wrapping does occur (namely when converting from a `char` to an unsigned quantity), the warning about wrapping at the `unsigned int`-level will be dismissed as a false positive. Hence, the analysis above should flag the out-of-bound array access but treat wrapping itself as intentional.

In this work, we propose a re-interpretation of polyhedra in which the wrapping of integer calculations is reflected in the classic polyhedral domain [6]. In particular, we avoid making cross-cutting changes to all transfer functions but refine the approximation relation such that wrapping is mostly implicit, that is, the need for extra polyhedral operations is largely finessed. For the few cases in which wrapping has to be addressed in the transfer functions, we illustrate how to wrap values within the polyhedral domain and propose an algorithm for doing so. To summarise our contributions, this paper presents:

- an approximation relation that implicitly wraps polyhedral variables;
- an algorithm to perform wrapping in the polyhedral domain;
- a formal description of an analysis that faithfully models wrapping and an accompanying correctness argument.

We commence with the definition of a small language and its concrete semantics. Section 3 introduces polyhedral analysis. Sections 4 and 5 explain how wrapping is supported. Section 6 presents a wrapping-aware polyhedral analysis which is discussed in Section 7. We conclude with the related work in Section 8.

2 A Language Featuring Finite Integer Arithmetic

The function shown in Figure 1 demonstrates that it is important to clarify where wrapping can arise in a program. This is particularly true when arguing the correctness of an analysis. To this end we introduce the language $\mathcal{L}$(ELang) which is a subset of an intermediate language which is used to analyse C programs.

2.1 The Syntax of $\mathcal{L}$(ELang)

$\mathcal{L}$(ELang) features linear expressions and casts between integers. In the following grammar of $\mathcal{L}$(ELang), (T)* denotes the repetition of the non-terminal T.

$$\begin{array}{ll}
\langle\text{ELang}\rangle & :: (\text{Block})^* \\
\langle\text{Block}\rangle & :: l\,\mathbf{:}\ (\langle\text{Stmt}\rangle\ \mathbf{;})^*\ \langle\text{Next}\rangle \\
\langle\text{Next}\rangle & :: \mathbf{jump}\ l\,\mathbf{;} \\
 & \mid\ \mathbf{if}\ \langle\text{Type}\rangle\ v\ \langle\text{Op}\rangle\ \langle\text{Expr}\rangle\ \mathbf{then\ jump}\ l\,\mathbf{;}\ \langle\text{Next}\rangle \\
\langle\text{Op}\rangle & :: < \mid \leq \mid = \mid \neq \mid \geq \mid > \\
\langle\text{Expr}\rangle & :: n \mid n\ \mathbf{*}\ v\ \mathbf{+}\ \langle\text{Expr}\rangle \\
\langle\text{Stmt}\rangle & :: \langle\text{Size}\rangle\ v\ \mathbf{=}\ \langle\text{Expr}\rangle \\
 & \mid\ \langle\text{Size}\rangle\ v\ \mathbf{=}\ \langle\text{Type}\rangle\ v \\
\langle\text{Type}\rangle & :: (\mathbf{uint} \mid \mathbf{int})\ \langle\text{Size}\rangle \\
\langle\text{Size}\rangle & :: \mathbf{1} \mid \mathbf{2} \mid \mathbf{4} \mid \mathbf{8}
\end{array}$$

An ELang program consists of a sequence of basic blocks with execution commencing with the first block. Each basic block consists of a sequence of statements and a list of control-flow instructions. In the sequel, we use *lookupBlock*(l) and *lookupNext*(l) to access the statements and control-flow instructions, respectively, of basic block l. The control-flow of a basic block consists of either a **jump** statement or a conditional which itself is followed by more control flow instructions. The ⟨Stmt⟩ production is restricted to the two statements of interest, namely the assignment of linear expressions to a variable and a type cast. We require that each variable in the program is used with only one size which is always specified in bytes. In particular, the assignment statement and the conditional require that all occurring variables are of the same size. Note, though, that variables may be used as an **uint** (unsigned integer) in one statement and as an **int** (signed integer) in another.

2.2 The Semantics of $\mathcal{L}$(ELang)

In order to specify the semantics of $\mathcal{L}$(ELang), we introduce the following notation. Let $\mathbb{B} = \{0, 1\}$ denote the set of Booleans. A vector $\boldsymbol{b} = \langle b_{w-1}, \ldots b_0\rangle \in \mathbb{B}^w$ is interpreted as unsigned integer by $\mathit{val}^{w,\text{uint}}(\boldsymbol{b}) = \sum_{i=0}^{w-1} b_i 2^i$ and as signed integer by $\mathit{val}^{w,\text{int}}(\boldsymbol{b}) = (\sum_{i=0}^{w-2} b_i 2^i) - b_{w-1}2^{w-1}$. Conversely, $\mathit{bin}^w : \mathbb{Z} \to \mathbb{B}^w$ converts an integer to the lower w bits of its Boolean representation. Formally, $\mathit{bin}^w(v) = \boldsymbol{b}$ iff there exists $\boldsymbol{b}' \in \mathbb{B}^q$ such that $\mathit{val}^{(q+w),\text{int}}(\boldsymbol{b}' \| \boldsymbol{b}) = v$ where $\|$ denotes the concatenation of bit-vectors. For instance, $\mathit{bin}^3(15) = \langle 1, 1, 1\rangle$ since

Basic Blocks.

$$\llbracket\ l : s_1; \ldots s_n;\ \rrbracket^{\natural}_{\text{Block}}\sigma = \llbracket\ \mathit{lookupNext}(l)\ \rrbracket^{\natural}_{\text{Next}}(\llbracket\ s_n\ \rrbracket^{\natural}_{\text{Stmt}}(\ldots(\llbracket\ s_1\ \rrbracket^{\natural}_{\text{Stmt}}\sigma)\ldots))$$

Control Flow.

$$\llbracket\ \textbf{jump}\ l\ \rrbracket^{\natural}_{\text{Next}}\sigma = \llbracket\ \mathit{lookupBlock}(l)\ \rrbracket^{\natural}_{\text{Block}}\sigma$$

$$\llbracket\ \textbf{if}\ t\ s\ v\ \mathit{op}\ \mathit{exp}\ \textbf{then jump}\ l\ ;\ \mathit{nxt}\ \rrbracket^{\natural}_{\text{Next}}\sigma = \begin{cases} \llbracket\ \mathit{lookupBlock}(l)\ \rrbracket^{\natural}_{\text{Block}}\sigma & \text{if } \mathit{val}^{8s,t}(\sigma^{s}(\mathit{addr}^{\natural}(v)))\ \mathit{op}\ \mathit{val}^{8s,t}(\llbracket\ \mathit{exp}\ \rrbracket^{\natural,s}_{\text{Expr}}\sigma) \\ \llbracket\ \mathit{nxt}\ \rrbracket^{\natural}_{\text{Next}}\sigma & \text{otherwise} \end{cases}$$

Expressions.

$$\llbracket\ n\ \rrbracket^{\natural,s}_{\text{Expr}}\sigma = \mathit{bin}^{8s}(n)$$

$$\llbracket\ n * v + \mathit{exp}\ \rrbracket^{\natural,s}_{\text{Expr}}\sigma = \mathit{bin}^{8s}(n) *^{8s} \sigma^{s}(\mathit{addr}^{\natural}(v)) +^{8s} \llbracket\ \mathit{exp}\ \rrbracket^{\natural,s}_{\text{Expr}}\sigma$$

Assignment.

$$\llbracket\ s\ v = \mathit{exp}\ \rrbracket^{\natural}_{\text{Stmt}}\sigma = \sigma[\mathit{addr}^{\natural}(v) \overset{s}{\mapsto} \llbracket\ \mathit{exp}\ \rrbracket^{\natural,s}_{\text{Expr}}\sigma]$$

Type Casts.

$$\llbracket\ s_1\ v_1 = t\ s_2\ v_2\ \rrbracket^{\natural}_{\text{Stmt}}\sigma = \sigma[\mathit{addr}^{\natural}(v_1) \overset{s_1}{\mapsto} \mathit{bin}^{8s_1}(\mathit{val}^{8s_2,t}(\sigma^{s_2}(\mathit{addr}^{\natural}(v_2))))]$$

Fig. 2. Concrete Semantics of $\mathcal{L}$(ELang)

$\mathit{val}^{5,\text{int}}(\langle 0,1,1,1,1\rangle) = 15$. In order to distinguish calculations on Boolean vectors from standard arithmetic, let $+^w, *^w : \mathbb{B}^w \times \mathbb{B}^w \to \mathbb{B}^w$ denote addition and multiplication that truncate the result to the lower w bits, for instance $\langle 1,1,1,1\rangle +^4 \langle 0,0,0,1\rangle = \langle 0,0,0,0\rangle$. Note that the signedness of the arguments of $+^w$ and $*^w$ do not affect the result of these operations.

$\mathcal{L}$(ELang) programs operate in a virtual memory environment which we formalise as a sequence of bytes. Let $\mathcal{B} = \mathbb{B}^8$ denote the set of bytes and $\Sigma = \mathcal{B}^{2^{32}}$ all states of 4 GByte that a program on a 32-bit architecture can take on. Let $\sigma \in \Sigma$ denote a given memory state of a program and let $\sigma^s : [0, 2^{32}-1] \to \mathcal{B}^s$ denote a read access at the given 32-bit address where $s \in \{1,2,4,8\}$ is the number of bytes to be read. A write operation is formalised as a substitution $\sigma[a \overset{s}{\mapsto} v]$. The resulting store $\sigma' = \sigma[a \overset{s}{\mapsto} v]$ satisfies $\sigma'^s(a) = v$ and furthermore $\sigma'^1(b) = \sigma^1(b)$ for all addresses $b \notin \{a, \ldots a+s-1\}$.

Figure 2 presents the concrete semantics (or natural semantics, hence the $\natural$) of the $\mathcal{L}$(ELang) language. These definitions use $\mathit{addr}^{\natural}(v) \in [0, 2^{32}-1]$ which maps the program variable v to its address in memory. We assume that $\mathit{addr}^{\natural}$ maps different variables to non-overlapping memory regions, an assumption that makes $\mathcal{L}$(ELang) independent of the endianness of an architecture.

The concrete semantics manipulates the store mainly by operations on bit-vectors; only in the conditional and in the cast are bit vectors interpreted as numbers. In these cases the signedness of the variables can actually influence the result. In particular, the type t of the cast determines if the source bit-vector is sign-extended (if $t = \text{int}$) or zero-extended (if $t = \text{uint}$) when $s_1 \geq s_2$. We now proceed by abstracting this semantics so as to specify a polyhedral analysis.

3 Polyhedral Analysis of Finite Integers

Two's complement arithmetic exploits the wrapping behaviour of integer variables that are confined to a fixed number of bits. For instance, subtracting 1 from an integer is equivalent to adding the largest representable integer value. In fact, the binary representation of the signed integer -1 is identical to that of the largest, unsigned integer of the same size. In the context of verification, this dichotomy in interpretation cannot be dismissed since $\mathcal{L}(\text{ELang})$ has insufficient information about the signedness of assignments. This omission allows our model to be applied to languages which freely mix signed and unsigned values, e.g. C.

Accessing the same bit sequence as either signed or unsigned integer corresponds to a wrapping behaviour in that the negative range of the signed integer wraps to the upper range of an unsigned integer, see Figure 3. This wrapping behaviour of finite integers creates a mismatch against the infinite range of polyhedral variables. We present our solution to this mismatch in two parts: Section 4 presents a concretisation map between the polyhedral domain and the bit-level representation of variables. This map wraps values of abstract variables implicitly to finite sequences of bits, thereby alleviating the need to check for wrapped values each time a variable is read or written. In contrast, Section 5 details an algorithm that makes the wrapping of program variables explicit in the abstract domain which is important for casts and the conditional statement whose semantics depend on the size and signedness of the operands.

3.1 The Domain of Closed, Convex Polyhedra

In order to make the paper self-contained, this section gives a concise introduction to the notation used in our polyhedral analysis. Let $X = \{x_1, \ldots x_n\}$ be a finite set of variables, let $\boldsymbol{x} = \langle x_1, \ldots x_n \rangle$ and let I be the set of linear inequalities that can be rewritten as $\boldsymbol{c} \cdot \boldsymbol{x} \leq d$ where $\boldsymbol{c} \in \mathbb{Z}^n$ and $d \in \mathbb{Z}$. For brevity we write $e_1 = e_2$ to denote the set of inequalities $\{e_1 \leq e_2, e_2 \leq e_1\}$ where e_i is any linear expression. Furthermore, let $e_1 < e_2$ abbreviate $e_1 \leq e_2 - 1$. Let $\iota \in I$ be an inequality that can be transformed into $\boldsymbol{c} \cdot \boldsymbol{x} \leq d$, then $[\![\iota]\!] = \{\boldsymbol{x} \in \mathbb{R}^n \mid \boldsymbol{c} \cdot \boldsymbol{x} \leq d\}$ denotes the induced half-space. Given a finite set of inequalities $E = \{\iota_1, \ldots \iota_n\} \subseteq I$ the notation $[\![E]\!] = \bigcap_{i=1,\ldots n} [\![\iota_i]\!]$ denotes the induced convex polyhedron. Let

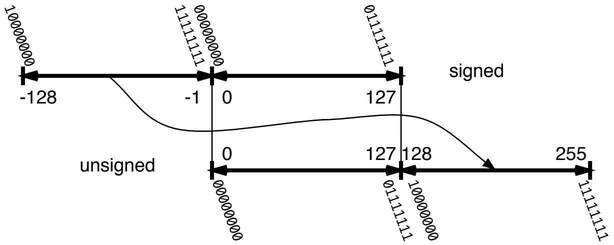

Fig. 3. The difference between a signed and an unsigned access can be interpreted as a wrap of negative values to the upper range in an unsigned access

Poly be the set of all convex polyhedra (polyhedra for short). Given two polyhedra $P_1, P_2 \in \mathit{Poly}$, let $P_1 \sqsubseteq P_2$ iff $P_1 \subseteq P_2$ and let $P_1 \sqcap P_2 = P_1 \cap P_2$. Note that $P_1 \sqcap P_2 = [\![E_1 \cup E_2]\!]$ where $P_i = [\![E_i]\!]$ and $i = 1, 2$. We write $P_1 \sqcup P_2$ to denote the closure of the convex hull [6] of two polyhedra P_1 and P_2 which is defined as the smallest polyhedron P such that $P_1 \cup P_2 \sqsubseteq P$. Let $P \in \mathit{Poly}$ be non-empty and $V_i = \{v_i \mid \langle v_0, \ldots v_n \rangle \in P\}$. We write $P(x) = [l, u]$ where $l = \lceil \min(V_i) \rceil$ if the minimum exists, otherwise $l = -\infty$ and $u = \lfloor \max(V_i) \rfloor$ if the maximum exists, otherwise $u = \infty$. Define $\exists_{x_i} : \mathit{Poly} \to \mathit{Poly}$ such that $\exists_{x_i}(P) = \{\langle v_1, \ldots, v_{i-1}, x, v_{i+1}, \ldots v_n \rangle | \langle v_1, \ldots v_n \rangle \in P \wedge x \in \mathbb{R}\}$. Let T denote an additional set of temporary variables such that $T \cap X = \emptyset$. All operations above lift from X to $T \cup X$. The operation $P \triangleright x := e \in \mathit{Poly}$ denotes an update of a variable $x \in X$ to the linear expression e and is defined as $\exists_t([\![x = t]\!] \sqcap \exists_x([\![t = e]\!] \sqcap P))$ where $t \in T$ does not occur in e and $t \neq x$. The rationale for assigning the result to a temporary variable is that e might contain x. Assigning e to a fresh variable t will retain the value of x before it is projected out. If x does not occur in e, the update above is equivalent to $\exists_x(P) \sqcap [\![x = e]\!]$. In order to argue about updates of polyhedra, observe the following:

Lemma 1. *Let $P \in \mathit{Poly}$ and $P' = P \triangleright x_i := \boldsymbol{c} \cdot \boldsymbol{x} + d$ with $\boldsymbol{c} \in \mathbb{Z}^n$ and $d \in \mathbb{Z}$. Then $P' = \{\langle v_1, \ldots v_{i-1}, v'_i, v_{i+1}, \ldots v_n \rangle \mid \boldsymbol{v} = \langle v_1, \ldots v_n \rangle \in P \wedge v'_i = \boldsymbol{c} \cdot \boldsymbol{v} + d\}$.*

4 Implicit Wrapping of Polyhedral Variables

This section formalises the relationship between polyhedral variables and bit-sequences that constitute the program state. For simplicity, we assume a one-to-one correspondance between the variable names in the program and the polyhedral variables that represent their values. The values of a program variable are merely bit sequences that are prescribed by the possible values of the polyhedral variable. To illustrate, suppose that `x` is of type `char` and $P(x) = [-1, 2]$. The represented bit patterns are 11111111, 00000000, 00000001 and 00000010, no matter whether `x` is signed or unsigned. These bit patterns are given by $\mathit{bin}^{8s}(v)$ which turns a value $v \in [-1, 2]$ into a bit sequence of s bytes. Going further, the function $\mathit{bits}_a^s : \mathbb{Z} \to \mathcal{P}(\Sigma)$ produces all concrete stores in which $8s$ bits at address $a = \mathit{addr}^\natural(\mathtt{x})$ are set to the value corresponding to $v \in P(x)$ as follows:

$$\mathit{bits}_a^s(v) = \{\langle r_{8*2^{32}} \ldots r_{8(a+s)} \rangle \| \; \mathit{bin}^{8s}(v) \; \| \; \langle r_{8a-1} \ldots r_0 \rangle \mid r_i \in \mathbb{B}\}$$

Note that this definition only considers the lower $8s$ bits of the value v. For instance, $\mathit{bits}_a^1(0) = \mathit{bits}_a^1(256)$ since the lower eight bits of 0 and 256 are equal. The mapping bits_a^s can be lifted from the value v of a single variable to the values $\langle v_1, \ldots v_n \rangle \in \mathbb{Z}^n$ of a vector of variables $\langle x_1, \ldots x_n \rangle$, resulting in the stores $\bigcap_{i \in [1,n]} \mathit{bits}_{a_i}^{s_i}(v_i)$. Here $a_i \in [0, 2^{32} - 1]$ denotes the address of the variable x_i in the concrete store and $s_i \in \mathbb{N}$ denotes its size in bytes. Observe that, if variables were allowed to overlap, the above intersection might incorrectly collapse to the empty set for certain vectors $\langle v_1, \ldots v_n \rangle \in \mathbb{Z}^n$. Using this lifting, a polyhedron is now related to a set of stores by $\gamma_{\boldsymbol{a}}^{\boldsymbol{s}} : \mathit{Poly} \to \mathcal{P}(\Sigma)$ which is defined as

$$\gamma_{\boldsymbol{a}}^{\boldsymbol{s}}(P) = \bigcup_{\boldsymbol{v} \in P \cap \mathbb{Z}^n} \left(\bigcap_{i \in [1,n]} bits_{a_i}^{s_i}(v_i) \right)$$

where $\boldsymbol{s} = \langle s_1, \ldots s_n \rangle$, $\boldsymbol{a} = \langle a_1, \ldots a_n \rangle$ and $\boldsymbol{v} = \langle v_1, \ldots v_n \rangle$.

The definition of $\gamma_{\boldsymbol{a}}^{\boldsymbol{s}}$ provides a criterion for judging the correctness of an abstract semantics. In addition, $\gamma_{\boldsymbol{a}}^{\boldsymbol{s}}$ permits linear expressions to be evaluated in the abstract semantics without the need to address overflows since $\gamma_{\boldsymbol{a}}^{\boldsymbol{s}}$ maps the result of calculations in the polyhedral domain to the correctly wrapped result in the actual program. This property is formalised below:

Proposition 1. *Let $e \in \mathcal{L}(Expr)$ and $e \equiv \boldsymbol{c} \cdot \boldsymbol{x} + d$, that is, e is a reformulation of $\boldsymbol{c} \cdot \boldsymbol{x} + d$. If $\sigma \in \gamma_{\boldsymbol{a}}^{\boldsymbol{s}}(P)$ then $\sigma[a_i \overset{s_i}{\mapsto} [\![\, e \,]\!]_{\text{Expr}}^{\natural, s_i}] \in \gamma_{\boldsymbol{a}}^{\boldsymbol{s}}(P \rhd x_i := \boldsymbol{c} \cdot \boldsymbol{x} + d)$.*

Proof. Define $\pi_i(\langle x_1, \ldots x_{i-1}, x_i, x_{i+1}, \ldots x_n \rangle) = x_i$. Since $\sigma \in \gamma_{\boldsymbol{a}}^{\boldsymbol{s}}(P)$ there exists $\boldsymbol{v} \in P \cap \mathbb{Z}^n$ such that $\sigma = \bigcap_{i \in [1,n]} bits_{a_i}^{s_i}(\pi_i(\boldsymbol{v}))$. Let $P' = P \rhd x_i := \boldsymbol{c} \cdot \boldsymbol{x} + d$ for some $\boldsymbol{c} \in \mathbb{Z}^n$ and $d \in \mathbb{Z}$. By Lemma 1, there exists $\boldsymbol{v}' \in P'$ with $\pi_j(\boldsymbol{v}') = \pi_j(\boldsymbol{v})$ for all $j \neq i$. Since $\{a_i, \ldots a_i + s_i - 1\} \cap \{a_j, \ldots a_j + s_j - 1\} = \emptyset$ for all $j \neq i$, there exists $\sigma' \in \gamma_{\boldsymbol{a}}^{\boldsymbol{s}}(P')$ such that $\sigma'^1(a) = \sigma^1(a)$ for $a \in [0, 2^{32} - 1] \setminus \{a_i, \ldots a_i + s_i - 1\}$. Furthermore, the lemma states that $\pi_i(\boldsymbol{v}') = \boldsymbol{c} \cdot \boldsymbol{v} + d$ and, by the definition of $\gamma_{\boldsymbol{a}}^{\boldsymbol{s}}$, it follows that $\sigma'^{s_i}(a_i) = bin^{8s_i}(\boldsymbol{c} \cdot \boldsymbol{v} + d)$. To show that $\sigma'^{s_i}(a_i) = [\![\, e \,]\!]_{\text{Expr}}^{\natural, s_i} \sigma$, we find $\boldsymbol{a} \in \mathbb{Z}^n$, $d \in \mathbb{Z}$ such that $e \equiv \boldsymbol{c} \cdot \boldsymbol{x} + d$ and $[\![\, e \,]\!]_{\text{Expr}}^{\natural, s_i} \sigma = bin^{8s_i}(\boldsymbol{c} \cdot \boldsymbol{v} + d)$ by induction over e:

1. Let $e = n$. By definition of $[\![\cdot]\!]_{\text{Expr}}^{\natural, s}$, $[\![\, n \,]\!]_{\text{Expr}}^{\natural, s_i} \sigma = bin^{8s_i}(n) = bin^{8s_i}(\boldsymbol{c} \cdot \boldsymbol{v} + d)$ where $d = n$ and $\boldsymbol{c} = \langle 0, \ldots 0 \rangle$. Hence $e \equiv \boldsymbol{c} \cdot \boldsymbol{x} + d$.
2. Let $e = n * x_j + e'$. Suppose that $[\![\, e' \,]\!]_{\text{Expr}}^{\natural, s_i} \sigma = bin^{8s_i}(\boldsymbol{c}' \cdot \boldsymbol{v} + d')$ where $e' \equiv \boldsymbol{c}' \cdot \boldsymbol{x} + d'$. By the definition of $[\![\cdot]\!]_{\text{Expr}}^{\natural, s}$, $[\![\, n * x_j + e \,]\!]_{\text{Expr}}^{\natural, s_i} \sigma = bin^{8s}(n) *^{8s_i} \sigma^{s_i}(a_j) +^{8s_i} [\![\, e' \,]\!]_{\text{Expr}}^{\natural, s_i} \sigma$ where $\sigma^{s_i}(a_j) = bin^{8s_i}(v_j)$. By definition of bin^{8s}, $bin^{8s}(n) *^{8s_i} bin^{8s_i}(v_j) = ((n \; mod \; 2^{8s_i}) * (v_j \; mod \; 2^{8s_i})) \; mod \; 2^{8s_i} = (n * v_j) \; mod \; 2^{8s_i}$, c.f. [7, page 42]. Similarly, $(n * v_j) \; mod \; 2^{8s_i} +^{8s_i} [\![\, e' \,]\!]_{\text{Expr}}^{\natural, s_i} \sigma = (n * v_j) \; mod \; 2^{8s_i} +^{8s_i} bin^{8s_i}(\boldsymbol{c}' \cdot \boldsymbol{v} + d') = (n * v_j + \boldsymbol{c}' \cdot \boldsymbol{v} + d') \; mod \; 2^{8s_i}$. Thus, set $d = d'$ and $\langle c_1, \ldots c_n \rangle = \langle c'_1, \ldots c'_{i-1}, c'_i + n, c'_{i+1}, \ldots c'_n \rangle$ where $\boldsymbol{c} = \langle c_1, \ldots c_n \rangle$ and $\boldsymbol{c}' = \langle c'_1, \ldots, c'_n \rangle$. Hence $e \equiv \boldsymbol{c} \cdot \boldsymbol{x} + d$.

The force of the above result is that a linear expression ⟨Expr⟩ over finite integer variables can be interpreted as an expression over polyhedral variables without regard for overflows or evaluation order. A prerequisite for this convenience is that all variables occurring in an expression have the same size s. In contrast, assignments between different sized variables have to revert to a cast statement. In this case, and in the case of conditionals, wrapping has to be made explicit which is the topic of the next section.

5 Explicit Wrapping of Polyhedral Variables

A consequence of the wrapping behaviour of $\gamma_{\boldsymbol{a}}^{\boldsymbol{s}}$ is that the effect of a guard such as `x<=y` cannot be modelled as a transformation from a polyhedron P to

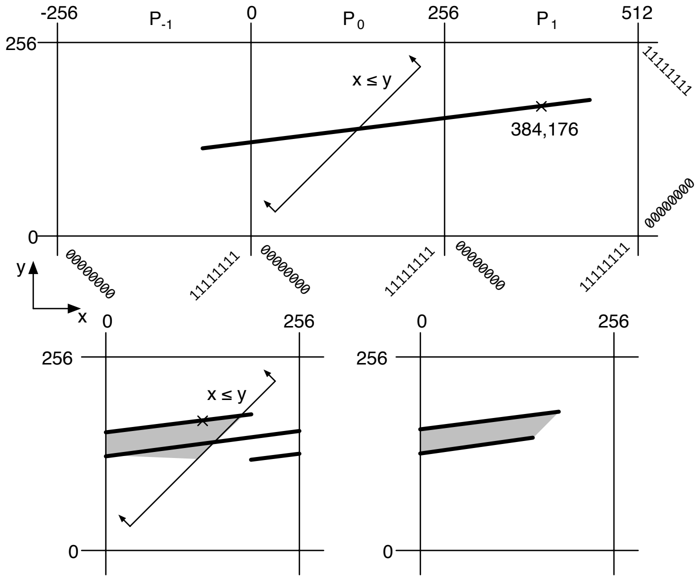

Fig. 4. Explicitly wrapping the possible values of x to its admissible range

$P \sqcap [\![x \leq y]\!]$. This section explains this problem, discusses possible solutions and proposes an efficient wrapping algorithm *wrap*.

5.1 Wrapping Variables with a Finite Range

In order to illustrate the requirements on the *wrap* function, consider Figure 4. The thick line in the upper graph denotes $P = [\![x + 1024 = 8y, -64 \leq x \leq 448]\!]$ which we suppose feeds into the guard `x<=y` where `x` and `y` both represent variables of type **uint8**. In order to illustrate a peculiarity of modelling the guard, consider the point $\langle x, y\rangle = \langle 384, 176\rangle \in P$ and let $\sigma \in \gamma_{\boldsymbol{a}}^{\boldsymbol{s}}(\{\langle 384, 176\rangle\})$. Due to implicit wrapping in $\gamma_{\boldsymbol{a}}^{\boldsymbol{s}}$, the state σ stipulates that $val^{8,\text{uint}}(\sigma^1(addr^\natural(x))) = 128$ and $val^{8,\text{uint}}(\sigma^1(addr^\natural(y))) = 176$. Thus, although `x<=y` is true when interpreting `x` and `y` as **uint8** in σ, the polyhedron $\{\langle 384, 176\rangle\} \sqcap [\![x \leq y]\!]$ is empty. Hence, it is not correct to model the guard in the classic way as $P \sqcap [\![x \leq y]\!]$.

In order to model relational tests correctly, the values of expressions occurring on each side of a relational operator have to be wrapped to the type prescribed in an $\mathcal{L}$(ELang) conditional. In the example, the expression y is already in the required range $[0, 255]$ whereas the range of x impinges on the two neighbouring quadrants as indicated in the upper graph of Figure 4. These quadrants are obtained by partitioning the state P into $P_{-1} = P \sqcap [\![-256 \leq x \leq -1]\!]$, $P_0 = P \sqcap [\![0 \leq x \leq 255]\!]$ and $P_1 = P \sqcap [\![256 \leq x \leq 511]\!]$. The result of wrapping x can now be calculated by translating P_{-1} by 256 units towards positive x-coordinates and P_1 by 256 units towards negative x-coordinates, yielding $P' = P_0 \sqcup (P_{-1} \rhd x := x + 256) \sqcup (P_1 \rhd x := x - 256)$. The contribution of each

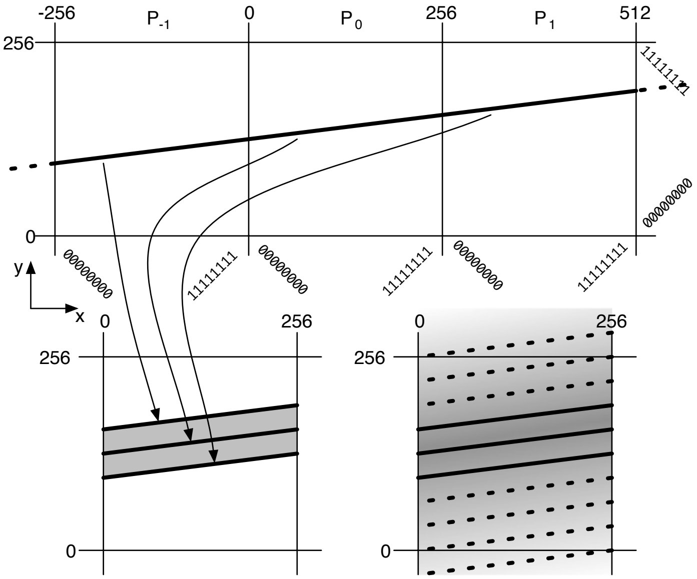

Fig. 5. The quest for an efficient wrapping of unbounded variables

partition is shown as a thick line in the lower left graph and the grey region depicts $P' \sqcap [\![x \leq y]\!]$. Observe that a more precise state P'' can be obtained by intersecting each translated state separately with $[\![x \leq y]\!]$, that is, by calculating $(P_0 \sqcap [\![x \leq y]\!]) \sqcup ((P_{-1} \rhd x := x + 256) \sqcap [\![x \leq y]\!]) \sqcup ((P_1 \rhd x := x - 256) \sqcap [\![x \leq y]\!])$. This state, depicted as the grey area in the lower right graph, is smaller than P' since P_{-1} does not contribute at all. Indeed, this example shows that polyhedra are not meet-distributive, that is, $P \sqcap (P_1 \sqcup P_2) \neq (P \sqcap P_1) \sqcup (P \sqcap P_2)$. In this work, we chose to calculate the equivalent of P' in our wrapping function *wrap* as it simplifies the presentation; implementing the refined model is mere engineering.

5.2 Wrapping Variables with Infinite Ranges

In the given example, it was possible to obtain a wrapped representation of the values of x and y by calculating the join of three constituent state spaces. In general, however, wrapping x and y can require the join of an infinite number of constituent state spaces as depicted in Figure 5. Here, the line in the upper graph depicts $P = [\![x + 1024 = 8y]\!]$, that is, P denotes the same linear relation as before, except that x is unbounded. Translating P by i times the range of **uint8** yields $P_i = (P \rhd x := x + i2^8 \sqcap [\![0 \leq x \leq 255]\!]) \sqcup (P \rhd x := x - i2^8 \sqcap [\![0 \leq x \leq 255]\!])$ for $i \geq 0$. A polyhedron that includes the sequence $P'_j = \bigsqcup_{0 \leq i \leq j} P_i$ can be computed using widening [6], thereby yielding the grey area in the lower right graph. In fact, this region is equivalent to $\exists_x(P) \sqcap [\![0 \leq x \leq 255]\!]$ as it contains neither bounds on x nor relational information between x and other variables. This suggests that, rather than wrapping unbounded variables, it is cheaper and

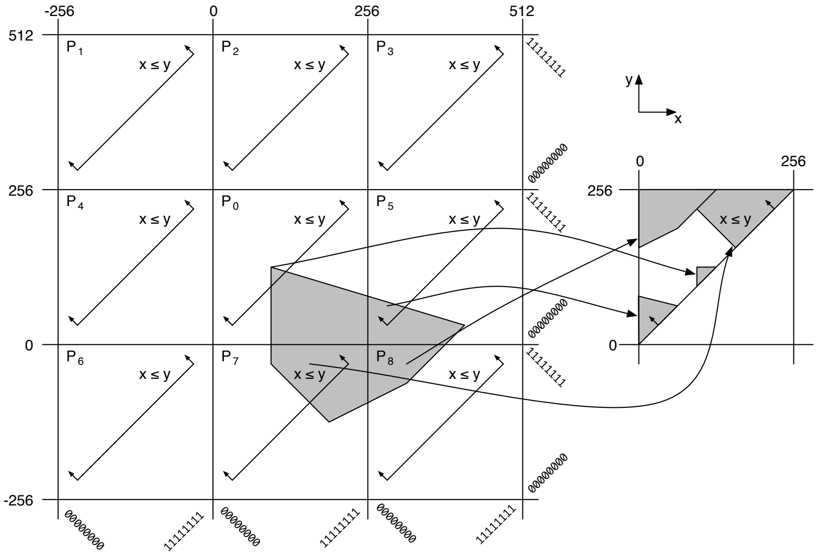

Fig. 6. Precise wrapping of two bounded variables

as precise to set them to the whole range of their type. After wrapping x, it becomes apparent that y is unbounded too, and hence needs wrapping.

5.3 Wrapping Several Variables

Even though the guard x<=y used in the example of Section 5.1 involves two variables, it was only necessary to wrap x to obtain a wrapped representation of both x and y. The example of Section 5.2 hints at the fact that both variables might need wrapping to ensure that both sides of the guard are within range. In particular, it is not possible to translate a guard x<=y to the inequality $x - y \leq 0$ and to merely wrap $x - y$ to $[0, 255]$. To illustrate this, consider the simpler case x<=42 which is satisfied for bit sequences of x that fall within $[0, 42]$. In order to evaluate x<=42, set $x' = x - 42$ and wrap x' such that $0 \leq x' \leq 255$. The intersection with $[\![x' \leq 0]\!]$ constrains x' to 0 which implies $x = 42$ instead of $x \in [0, 42]$. Thus, both arguments to a guard x<=y need to be wrapped independently.

The example in Figure 5 showed how wrapping the unbounded x leaves y unconstrained which thus has to be wrapped as well. Figure 6 shows a potentially more precise solution for bounded variables in which variables are wrapped simultaneously. Here, the bounded state space shown in grey expands beyond the state P_0 that corresponds to the actual range of the variables. The result of translating each neighbouring quadrant and intersecting it with $x \leq y$ is shown in the graph on the right. Note that the join of these four translated spaces retains no information on either x or y. While it is possible that relational information with other variables is retained, wrapping the variables independently has the same precision if one of the variables is within bounds and, in particular,

Algorithm 1. Explicitly wrapping an expression to the range of a type

```
procedure $wrap(P, t\ s, x)$ where $P \neq \emptyset, t \in \{\text{uint}, \text{int}\}$ and $s \in \{1, 2, 4, 8\}$
 1: $b_l \leftarrow 0$
 2: $b_h \leftarrow 2^s$
 3: if $t =$ int then  /* Adjust ranges when wrapping to a signed type. */
 4:     $b_l \leftarrow b_l - 2^{s-1}$
 5:     $b_h \leftarrow b_h - 2^{s-1}$
 6: end if
 7: $[l, u] \leftarrow P(x)$
 8: if $l \neq -\infty \wedge u \neq \infty$ then  /* Calculate quadrant indices. */
 9:     $\langle q_l, q_u \rangle \leftarrow \langle \lfloor (l - b_l)/2^s \rfloor, \lfloor (u - b_l)/2^s \rfloor \rangle$
10: end if
11: if $l = -\infty \vee u = \infty \vee (q_u - q_l) > k$ then  /* Set to full range. */
12:     return $\exists_x(P) \sqcap [\![ b_l \leq x < b_h ]\!]$
13: else  /* Shift and join quadrants $\{q_l, \dots q_u\}$. */
14:     return $\bigsqcup_{q \in [q_l, q_u]}((P \triangleright x := x - q2^s) \sqcap [\![ b_l \leq x < b_h ]\!])$
15: end if
```

if a variable is compared to a constant. In the 3000 LOC program `qmail-smtp` that our analysis targets, 427 out of 522 conditionals test a variable against a constant, which motivates our design choice of wrapping variables independently.

5.4 An Algorithm for Explicit Wrapping

Guided by the observations made in the three examples, Algorithm 1 gives a procedure to wrap a polyhedral variable to the range of a given integer type. Due to the observations in the last section, we only present an algorithm to wrap one variable at a time. Thus, both sides of a guard have to be wrapped individually.

The algorithm commences by calculating the bounds of the type $t\ s$. A **uint8** type, for instance, will set $b_l = 0$ and $b_h = 2^8 = 256$ while an **int8** type results in the bounds $b_l = 0 - 2^{8-1} = -128$ and $b_h = 2^8 - 2^{8-1} = 128$. Line 7 calculates the bounds of x in P. If one of these bounds is infinite, line 12 removes all information on x and restrains x to $[b_l, b_h - 1]$. In case of finite bounds, line 9 calculates the smallest and largest quadrant into which the values of x impinge. For instance, in the example of Figure 4, these numbers are $q_l = -1$ (for the quadrant $[-256, -1]$) and $q_h = 1$ (for $[256, 511]$). Line 11 ensures that the linear expression is simply set to its maximum bounds if more than k quadrants have to be transposed and joined, where k is a limit that can be tuned to the required precision. Line 14 transposes each quadrant and restricts it to the bounds of the type. The correctness of *wrap* is asserted below:

Proposition 2. *Given $P \neq \emptyset$ and $P' = wrap(P, t\ s, x_i)$, the interval $P'(x_i)$ lies in the range of the type $t\ s$. Furthermore $\gamma_{\boldsymbol{a}}^{\boldsymbol{s}}(P) \subseteq \gamma_{\boldsymbol{a}}^{\boldsymbol{s}}(P')$.*

Proof. Upon return from lines 12 and 14, x_i is restricted to lie between the bounds b_l and $b_h - 1$ of the type $t\ s$, hence $P'(x_i)$ lies in the range of type $t\ s$.

Suppose $\boldsymbol{a} = \langle a_1, \ldots a_n \rangle$, $\boldsymbol{s} = \langle s_1, \ldots s_n \rangle$ where $a_i = addr^{\natural}(x_i)$ and s_i denotes the size of x_i in bytes. Let $\sigma \in \gamma_{\boldsymbol{a}}^{\boldsymbol{s}}(P)$. Then there exists $\langle v_1, \ldots v_n \rangle \in P \cap \mathbb{Z}^n$ such that $\sigma \in \gamma_{\boldsymbol{a}}^{\boldsymbol{s}}(\{\boldsymbol{v}\})$. We consider two behaviours of *wrap*:

- Suppose that *wrap* is exited at line 12. Observe that for any $\boldsymbol{b} \in \mathbb{B}^{8s_i}$ there exists $v \in \{b_l, \ldots, b_h - 1\}$ such that $bin^{8s_i}(v) = \boldsymbol{b}$. Hence, there exists $\boldsymbol{v'} = \langle v_1, \ldots v'_i, \ldots v_n \rangle$ with $v'_i \in [b_l, b_h - 1] \cap \mathbb{Z}$ and $bin^{8s_i}(v'_i) = bin^{8s_i}(v_i)$. Observe that $\boldsymbol{v'} \in P' = \exists_x(P) \sqcap [\![b_l \leq x < b_h]\!]$ and $\boldsymbol{v'} \in \mathbb{Z}^n$. Hence $bits_{\boldsymbol{a}}^{\boldsymbol{s}}(v_j) = bits_{\boldsymbol{a}}^{\boldsymbol{s}}(v'_j)$ for all $j \in [1, n]$ and it follows that $\sigma \in \gamma_{\boldsymbol{a}}^{\boldsymbol{s}}(P')$.
- Suppose now that *wrap* exits at line 14. Observe that $v_i \in [l, u]$ hence there exists $q \in [q_l, q_u]$ such that $v_i - q2^{s_1} \in [b_l, b_h - 1]$. Hence, there exists $\boldsymbol{v'} = \langle v_1, \ldots v'_i, \ldots v_n \rangle \in P'$ such that $v'_i = v_i - q2^{s_i}$. Since $bin^{8s_i}(q2^{s_i}) = \boldsymbol{0}$ it follows that $bin^{8s_i}(v'_i) = bin^{8s_i}(v_i - q2^{s_i}) = bin^{8s_i}(v_i)$. Thus $\sigma \in \gamma_{\boldsymbol{a}}^{\boldsymbol{s}}(P')$.

Note the translation of quadrants using $P \rhd x := x + q2^s$ can be implemented by a potentially cheaper affine transformation [2]. However, the shown solution can be readily implemented using other polyhedral domains [15,17] that do not directly support affine translations.

6 An Abstract Semantics for $\mathcal{L}$(ELang)

This section defines the abstract semantics of $\mathcal{L}$(ELang) in which a single polyhedron P is calculated for each label l where each label marks the beginning of a basic block. Starting with the unrestricted polyhedron $\mathbb{R}^{|X|}$ for the first basic block and with the empty polyhedron $\emptyset \subseteq \mathbb{R}^{|X|}$ for all others, the semantic function of the basic blocks are repeatedly evaluated until a (post-)fixpoint is reached [4]. We omit a formal definition of this fixpoint for simplicity. Once a fixpoint is reached, each state σ that may arise in the concrete program at l satisfies $\sigma \in \gamma_{\boldsymbol{a}}^{\boldsymbol{s}}(P)$ where P is the polyhedron associated with the label l.

The first rule in Figure 7 specifies how the evaluation of statements feeds into the evaluation of control-flow statements. Specifically, $[\![\; lookupNext(l) \;]\!]^{\sharp}_{\text{Next}} P_l$ yields tuples such as $\langle P'_{l'}, l' \rangle$ indicating that $P'_{l'}$ must be joined with the existing state $P_{l'}$ at l'. For instance, **jump** l merely returns the current state paired with the target label. The conditional calculates two new polyhedra P^{then} (which is returned for the label l) and P^{else} (which is used to evaluate other control-flow instructions). The calculation of P^{else} makes use of a function *neg* which negates a relational operator, for example, $neg('<') = '\geq'$. The auxiliary function *cond* wraps the two arguments of the relational operator *op*. Like *wrap*, this function can only wrap single polyhedral variables which requires that *exp* is assigned to a temporary variable y which is projected out once the guard is applied.

Observe that enforcing the guard by intersecting with $[\![x \; op \; y]\!]$ has the same effect as wrapping the expression *exp* itself since $y = exp$ holds in P'. However, if *wrap* returns from line 12 in Algorithm 1, the variable y is merely set to the bounds of the type. In this case *wrap* discards the relational information between y and *exp* and the intersection with $[\![x \; op \; y]\!]$ has no effect on P'', thereby ignoring the condition. An alternative treatment for expressions *exp* that exceed

k quadrants would be to discard any previous information on variables in *exp* using projection and to modify *wrap* to intersect P with $[\![b_l \leq exp < b_h]\!]$. In this case the information in the guard could be retained by intersecting with $[\![x\ op\ exp]\!]$ at the cost of discarding any previous bounds on the variables of *exp*.

The following Proposition states the correctness of the conditional:

Proposition 3. *If* $\sigma \in \gamma_{\boldsymbol{a}}^{\boldsymbol{s}}(P)$ *and* $val^{8s,t}(\sigma^s(addr^\natural(x_i)))\ op\ val^{8s,t}([\![\ exp\ ]\!]_{\text{Expr}}^{\natural,s}\sigma)$ *then* $\langle P', l\rangle \in [\![\ \textbf{if}\ t\ s\ x_i\ op\ exp\ \textbf{then jump}\ l\ ;\ nxt\ ]\!]_{\text{Next}}^\sharp P$ *and* $\sigma \in \gamma_{\boldsymbol{a}}^{\boldsymbol{s}}(P')$.

Proof. Since $\sigma \in \gamma_{\boldsymbol{a}}^{\boldsymbol{s}}(P)$ there exists $\boldsymbol{v} \in P \cap \mathbb{Z}^n$ such that $\sigma \in \gamma_{\boldsymbol{a}}^{\boldsymbol{s}}(\{\boldsymbol{v}\})$. Let $\hat{P} = cond(\bar{P}, t\ s, x_i, y, op)$ where $\bar{P} = P \triangleright y := [\![\ exp\ ]\!]_{\text{Expr}}^\sharp$. Then $\langle v_1, \ldots v_n, \hat{v}\rangle \in \bar{P} \cap \mathbb{Z}^{n+1}$ where $\langle v_1, \ldots v_n\rangle = \boldsymbol{v}$, $\hat{v} = \boldsymbol{c} \cdot \boldsymbol{v} + d$ and $exp \equiv \boldsymbol{c} \cdot \boldsymbol{x} + d$. By Proposition 2, there exists $\boldsymbol{v}' = \langle v_1, \ldots v_{i-1}, v_i', v_{i+1}, \ldots v_n, \hat{v}'\rangle \in \hat{P} \cap \mathbb{Z}^{n+1}$ such that $bin^{8s}(v_i) = bin^{8s}(v_i') = \sigma^{8s}(addr^\natural(x_i))$. By following Proposition 1, $bin^{8s}(\hat{v}) = bin^{8s}(\hat{v}') = [\![\ exp\ ]\!]_{\text{Expr}}^{\natural,s}\sigma$. Furthermore, v_i' and $\hat{v}'$ lie in the range of $t\ s$ and thus $val^{8s,t}(\sigma^{8s}(addr^\natural(x_i))) = v_i'$ and $val^{8s,t}([\![\ exp\ ]\!]_{\text{Expr}}^{\natural,s}\sigma) = \hat{v}'$. Hence $\boldsymbol{v}' \in \hat{P} \sqcap [\![x\ op\ y]\!]$ for $op \notin \{\neq\}$. With $P' = \exists_y(\hat{P} \sqcap [\![x\, op\, y]\!])$ it follows that $\sigma \in \gamma_{\boldsymbol{a}}^{\boldsymbol{s}}(P')$. The argument is similar for $op \in \{\neq\}$.

The fall-through case can be shown correct by a similar argument.

Due to the modulo nature of $\gamma_{\boldsymbol{a}}^{\boldsymbol{s}}$ the evaluation of linear expressions and assignments resembles that of classic polyhedral analysis in that linear expressions in the program are simply re-interpreted as expressions over polyhedra variables. This holds true even for casts between different sized variables as long as the target variable is smaller. Assigning smaller variables to larger, on the contrary, requires that wrapping is made explicit since a value that exceeds the range of the smaller source variable would wrap in the actual program whereas it might not exceed the range of the larger target variable.

We conclude this section with a correctness argument for the cast statement:

Proposition 4. *Suppose* $\sigma \in \gamma_{\boldsymbol{a}}^{\boldsymbol{s}}(P)$, $\sigma' = [\![\ s_1\ x_i = t\ s_2\ x_j\ ]\!]_{\text{Stmt}}^\natural\sigma$ *and let* $P' = [\![\ s_1\ x_i = t\ s_2\ x_j\ ]\!]_{\text{Stmt}}^\sharp P$. *Then* $\sigma' \in \gamma_{\boldsymbol{a}}^{\boldsymbol{s}}(P')$.

Proof. Since $\sigma \in \gamma_{\boldsymbol{a}}^{\boldsymbol{s}}(P)$ there exists $\boldsymbol{v} = \langle v_1, \ldots v_n\rangle \in P$ such that $\sigma \in \gamma_{\boldsymbol{a}}^{\boldsymbol{s}}(\{\boldsymbol{v}\})$. Let $\langle v_1', \ldots v_n'\rangle \in P \triangleright x_i := x_j$ where $v_i' = v_j'$ and $v_k' = v_k$ for all $k \neq i$. By Lemma 1, $\sigma'^{s_k}(addr^\natural(x_k)) = \sigma^{s_k}(addr^\natural(x_k))$ for all $k \neq i$. By definition of $[\![\ \cdot\ ]\!]_{\text{Stmt}}^\natural$, we need to show that $\sigma'^{s_1}(addr^\natural(x_i)) = bin^{8s_1}(val^{8s_2,t}(\sigma^{s_2}(addr^\natural(x_j))))$.

- Suppose $s_1 \leq s_2$. Then $bin^{8s_1}(x) = bin^{8s_1}(val^{8s_2,t}(bin^{8s_2}(x)))$. But $bin^{8s_1}(v_i') = \sigma'^{s_1}(addr^\natural(x_i))$ and $bin^{8s_2}(v_j) = \sigma^{s_2}(addr^\natural(x_j))$, thus $\sigma' \in \gamma_{\boldsymbol{a}}^{\boldsymbol{s}}(P')$ follows.
- Suppose now that $s_1 > s_2$. By Proposition 2, there exists $\langle v_1, \ldots \hat{v}_i, \ldots v_n\rangle \in P'$ such that $bin^{8s_2}(\hat{v}_i) = bin^{8s_2}(v_j)$ and $\hat{v}_i$ lies in the range of $t\ s_2$, that is, $val^{8s_2,t}(bin^{8s_2}(\hat{v}_i)) = \hat{v}_i$. But since $bin^{8s_2}(\hat{v}_i) = bin^{8s_2}(v_j) = \sigma'^{s_1}(addr^\natural(x_j))$ it follows that $\sigma'^{s_1}(addr^\natural(x_i)) = bin^{8s_1}(\hat{v}_i)$ as required.

Basic Blocks.

$$[\![\ l : s_1; \dots s_n;\ ]\!]^\sharp_{\text{Block}} P = [\![\ \mathit{lookupNext}(l)\ ]\!]^\sharp_{\text{Next}}([\![\ s_n\ ]\!]^\sharp_{\text{Stmt}}(\dots [\![\ s_1\ ]\!]^\sharp_{\text{Stmt}} P \dots))$$

Control Flow.

$$[\![\ \textbf{jump}\ l\ ]\!]^\sharp_{\text{Next}} P = \{\langle P, l\rangle\}$$

$$[\![\ \textbf{if}\ t\ s\ v\ \mathit{op}\ \mathit{exp}\ \textbf{then jump}\ l\ ;\ \mathit{nxt}\ ]\!]^\sharp_{\text{Next}} P = \{\langle P^{then}, l\rangle\} \cup [\![\ \mathit{nxt}\ ]\!]^\sharp_{\text{Next}} P^{else}$$

$$\begin{aligned} \text{where } P' &= P \rhd y := [\![\ \mathit{exp}\ ]\!]^\sharp_{\text{Expr}} \text{ where } y \in T \text{ fresh} \\ P^{then} &= \exists_y(\mathit{cond}(P', t\ s, v, y, \mathit{op})) \\ P^{else} &= \exists_y(\mathit{cond}(P', t\ s, v, y, \mathit{neg}(\mathit{op}))) \end{aligned}$$

$$\mathit{cond}(P', t\ s, x, y, \mathit{op}) = \begin{cases} (P'' \sqcap [\![x < y]\!]) \sqcup (P'' \sqcap [\![x > y]\!]) & \text{if } \mathit{op} \in \{\neq\} \\ (P'' \sqcap [\![x\ \mathit{op}\ y]\!]) & \text{otherwise} \end{cases}$$

$$\text{where } P'' = \mathit{wrap}(\mathit{wrap}(P', t\ s, x), t\ s, y)$$

Expressions.

$$[\![\ n\ ]\!]^\sharp_{\text{Expr}} = n$$

$$[\![\ n * v + \mathit{exp}\ ]\!]^\sharp_{\text{Expr}} = n\ v + [\![\ \mathit{exp}\ ]\!]^\sharp_{\text{Expr}}$$

Assignments.

$$[\![\ s\ v = \mathit{exp}\ ]\!]^\sharp_{\text{Stmt}} P = P \rhd v := [\![\ \mathit{exp}\ ]\!]^\sharp_{\text{Expr}}$$

Type Casts.

$$[\![\ s_1\ v_1 = t\ s_2\ v_2\ ]\!]^\sharp_{\text{Stmt}} P = \begin{cases} P' & \text{if } s_1 \leq s_2 \\ \mathit{wrap}(P', t\ s_2, v_1) & \text{otherwise} \end{cases}$$

$$\text{where } P' = P \rhd v_1 := v_2$$

Fig. 7. Abstract semantics of $\mathcal{L}$(ELang)

7 Discussion

The existence of a concretisation map $\gamma^{\boldsymbol{s}}_{\boldsymbol{a}}$ begs the question of whether an abstraction map can be defined. For classic polyhedral analysis [6], it is well-known that no best abstraction exists for certain shapes such as a disc [4]. In the context of our analysis, the set of concrete states Σ is finite. However, a given set of states still has no best abstraction. Consider $\sigma \in \Sigma$ with $\sigma^1(\mathit{addr}^\natural(\mathtt{x})) = 11111111$, $P_1 = [\![x = -1]\!]$ and $P_2 = [\![x = 255]\!]$. Although $\sigma \in \gamma^{\boldsymbol{s}}_{\boldsymbol{a}}(P_1) = \gamma^{\boldsymbol{s}}_{\boldsymbol{a}}(P_2)$, P_1 and P_2 are incomparable. As a consequence, the meet operation can only be applied after *wrap* has expressed the polyhedra in the same quadrant and thereby made them comparable. Termination is not compromised as *wrap* is monotonic.

Since different polyhedra can describe the same set of concrete states, care is needed when applying join. Suppose that the loop in Figure 8 is entered with $P = [\![x = -1]\!]$: the largest value an unsigned variable can take. As the loop invariant $x \geq 42$ mentions x, $Q = P \sqcup U$ is wrapped to $R = \mathit{wrap}(Q, \textbf{uint8}, x) = [\![x = 255]\!]$. A precision loss occurs when P and $U = [\![x = 254]\!]$ are joined to obtain $[\![-1 \leq x \leq 254]\!]$ as x cannot fall below 42. One solution to this particular problem is to unroll the loop once, which avoids the join of different representatives.

Observe that *wrap* is idempotent and, as such, is the identity if the variable is in range. An important consequence is that our solution is as precise as classic polyhedral analysis if all variables remain within the range of their types.

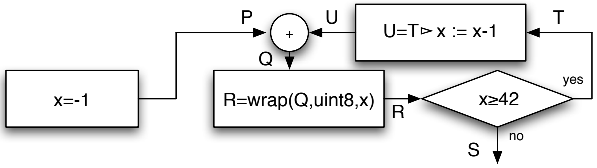

Fig. 8. Precision loss incurred by joining flow paths

An interesting benefit of $\gamma^{s}_{\boldsymbol{a}}$ is that the possible values of a byte x can be represented as either $[\![-128 \leq x \leq 127]\!]$ or $[\![0 \leq x \leq 255]\!]$. For example, an analysis of C string buffers [14] does not model individual array elements but tracks a single NUL position (a character with value zero) within the array. Even though `char` is often signed, the range $[0, 255]$ (rather than $[-128, 127]$) can be returned to indicate an arbitrary value when reading a byte from the array. The unsigned range can then be refined using the NUL position to $[1, 255]$ whenever the access lies in front of the NUL position (see [16] for an example). If a signed range $[-128, 127]$ had to be returned, it would include the NUL character since $[-128, -1] \cup [1, 127] = [-128, 127]$ is the best convex approximation. Without this tactic, it can be difficult to prove termination of loops that iterate over strings.

8 Related Work and Conclusion

A sound analysis must reason about overflowing calculations and correctly model conversions between unsigned and signed integers. Rather than contaminating an analysis with the low-level details of two's complement wrapping behaviour, we presented a computationally light-weight solution based on a novel concretization map for polyhedra which eliminates the need to check for wrapping in assignments of linear expression and conversions to smaller integers. Conditionals and other casts require an implementation using the presented algorithm *wrap*. We proved the presented analysis correct and argued the precision is no worse than that of classic polyhedral analysis that warns about every wrapping.

Although a number of works have addressed the modulo properties of congruences [1,11,12,13], little work exists for polyhedral analyses. Blanchet *et al.* use a two-tier approach [5]: For signed integers, any wrapping is erroneous. In this case, each time a variable is set, its range is checked for overflows. Overflows of unsigned integers are assumed to be intentional as wrapping may result from bit-level operations. This approach requires a separation of signed and unsigned variables which incurs false warnings for many programs, such as the C program in our introduction. Indeed, an analysis of our example would lead to the misleading warning about converting from a signed to an unsigned integer.

Further afield is work on tracing the propagation of rounding errors in floating point calculations. In this context, Goubault et al. [10] treat wrapping of integers as a rounding error that is as large as the range of the integer variable.

Acknowledgements. This work is supported by EPSRC project EP/C015517.

References

1. Bagnara, R., Dobson, K., Hill, P.M., Mundell, M., Zaffanella, E.: Grids: A Domain for Analyzing the Distribution of Numerical values. In: Puebla, G. (ed.) LOPSTR 2006. LNCS, vol. 4407, pp. 219–235. Springer, Heidelberg (2007)
2. Bagnara, R., Ricci, E., Zaffanella, E., Hill, P.M.: Possibly Not Closed Convex Polyhedra and the Parma Polyhedra Library. In: Hermenegildo, M.V., Puebla, G. (eds.) SAS 2002. LNCS, vol. 2477, pp. 213–229. Springer, Heidelberg (2002)
3. Joint Technical Committee.: International Standard ISO/IEC of C 98/99 (1999)
4. Cousot, P., Cousot, R.: Comparing the Galois Connection and Widening/Narrowing Approaches to Abstract Interpretation. In: Bruynooghe, M., Wirsing, M. (eds.) PLILP 1992. LNCS, vol. 631, pp. 269–295. Springer, Heidelberg (1992)
5. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: The ASTRÉE Analyzer. In: European Symposium on Programming, pp. 21–30. Springer, Edinburgh, Scotland (2005)
6. Cousot, P., Halbwachs, N.: Automatic Discovery of Linear Constraints among Variables of a Program. In: Principles of Programming Languages, pp. 84–97. ACM Press, Tucson, Arizona (1978)
7. Davenport, H.: The Higher Arithmetic, 7th edn. Cambridge University Press, Cambridge (1952)
8. Dor, N., Rodeh, M., Sagiv, M.: Cleanness Checking of String Manipulations in C Programs via Integer Analysis. In: Cousot, P. (ed.) SAS 2001. LNCS, vol. 2126, pp. 194–212. Springer, Heidelberg (2001)
9. Dor, N., Rodeh, M., Sagiv, M.: CSSV: Towards a Realistic Tool for Statically Detecting All Buffer Overflows in C. In: Gupta, R. (ed.) Programming Language Design and Implementation, pp. 155–167. ACM Press, San Diego, USA (2003)
10. Goubault, E., Putot, S., Beaufreton, P., Gassino, J.: Static Analysis of the Accuracy in Control Systems: Principles and Experiments. In: FMICS 2007. 12th International Workshop on Formal Methods for Industrial Critical systems, Springer, Heidelberg (2007)
11. Granger, P.: Static Analysis of Arithmetic Congruences. International Journal of Computer Mathematics 30, 165–199 (1989)
12. Granger, P.: Static Analyses of Congruence Properties on Rational Numbers (Extended Abstract). In: Symposium on Static Analysis, pp. 278–292. Springer, London, UK (1997)
13. Müller-Olm, M., Seidl, H.: Analysis of Modular Arithmetic. In: Sagiv, M. (ed.) ESOP 2005. LNCS, vol. 3444, pp. 46–60. Springer, Heidelberg (2005)
14. Simon, A., King, A.: Analyzing String Buffers in C. In: Kirchner, H., Ringeissen, C. (eds.) AMAST 2002. LNCS, vol. 2422, pp. 365–379. Springer, Heidelberg (2002)
15. Simon, A., King, A.: Exploiting Sparsity in Polyhedral Analysis. In: Hankin, C., Siveroni, I. (eds.) SAS 2005. LNCS, vol. 3672, pp. 336–351. Springer, Heidelberg (2005)
16. Simon, A., King, A.: Widening Polyhedra with Landmarks. In: Kobayashi, N. (ed.) APLAS 2006. LNCS, vol. 4279, pp. 166–182. Springer, Heidelberg (2006)
17. Simon, A., King, A., Howe, J.M.: Two Variables per Linear Inequality as an Abstract Domain. In: Leuschel, M.A. (ed.) LOPSTR 2002. LNCS, vol. 2664, pp. 71–89. Springer, Heidelberg (2003)