Automatic Repair of Overflowing Expressions with Abstract Interpretation

Francesco Logozzo
Microsoft Research, Redmond, WA, USA

Matthieu Martel
Université de Perpignan Via Domitia, DALI, Perpignan, France
Université Montpellier II & CNRS, LIRMM, UMR 5506, Montpellier, France

We consider the problem of synthesizing provably non-overflowing *integer* arithmetic expressions or Boolean relations among integer arithmetic expressions. First we use a numerical abstract domain to infer numerical properties among program variables. Then we check if those properties guarantee that a given expression does not overflow. If this is not the case, we synthesize an equivalent, yet not-overflowing expression, or we report that such an expression does not exists.

The synthesis of a non-overflowing expression depends on three, orthogonal factors: the input expression (*e.g.*, is it linear, polynomial, ...?), the output expression (*e.g.*, are case splits allowed?), and the underlying numerical abstract domain – the more precise the abstract domain is, the more correct expressions can be synthesized.

We consider three common cases: (i) linear expressions with integer coefficients and intervals; (ii) Boolean expressions of linear expressions; and (iii) linear expressions with templates. In the first case we prove there exists a complete and polynomial algorithm to solve the problem. In the second case, we have an incomplete yet polynomial algorithm, whereas in the third we have a complete yet worst-case exponential algorithm.

1 Introduction

Unwarranted integer overflows are a common source of bugs even for the most experienced programmers. Programmers have the tendency of forgetting that *machine* integers behave differently than *mathematical* integers and that apparently innocuous expressions may lead to hard to debug bugs in the program. Consider for instance the statement below, where `y` is a negative 32-bits integer

```
x = -y;
```
(1)

One may then expect that `x` is always a positive value, and also that $\texttt{x} \neq \texttt{y}$. However, this is false. When `y` is the minimum value -2^{31}, then $\texttt{x} = -2^{31} = \texttt{y}$, *i.e.*, the result of the negation of a negative integer is negative! This is not an artificial example. The `Math.abs` function in the standard Java library implements the absolute function value function according to the common *mathematical* definition (if the input is non-negative return it, otherwise return its negation). As a consequence, `Math.abs` may return a negative value, very likely breaking most callers relying on the (somehow obvious) fact that the absolute value is always non-negative.

The mismatch between the mathematical interpretation and the machine interpretation comes by the fact that the expression evaluation may originate in an *arithmetic overflow*: the result of the expression may be too large (or too small) to be exactly represented on b bits.

Most abstract interpretation-based static analysis tools like `cccheck` [5] or `Astree` [1] can detect potential arithmetic overflows. They first analyze the program using some numerical abstract domain

A. Banerjee, O. Danvy, K.-G. Doh, J. Hatcliff (Eds):
David A. Schmidt's 60th Birthday Festschrift
EPTCS 129, 2013, pp. 341–357, doi:10.4204/EPTCS.129.21

(*e.g.*, Intervals [3], Pentagons [9], Octagons [12]) to infer ranges and relations among program variables at each program point. Then, they use such information to prove that the evaluation of an arithmetic expression may never result into an overflow. If they cannot prove it, they emit a warning. For instance, in the example above `cccheck` warns about the possible negation of the `MinValue`.

In this paper, we want to push it a step further. We envision static analysis tools not only reporting possible arithmetic overflows, but *also* suggesting fixes for them. The suggested fixes are *verified repairs* in the sense of [8]. A verified repair is an expression which is equivalent to the original one when interpreted over the mathematical integers $\mathbb{Z}$, but which does not overflow when evaluated according to the given programming language semantics.

In general, the repair depends on three orthogonal factors:

(i) the input expression language, $\mathscr{I}$,

(ii) the output expression language, $\mathscr{O}$,

(iii) the available semantic information, $\mathscr{S}$, *i.e.*, the underlying abstract domain and the abstract state inferred by the analyzer.

For instance, let us consider the repairing of (1) where the input and output expression language are arithmetic expressions with *only* the 4 operations. The semantic information is

$$\mathscr{S} = \{\mathtt{y} \in [-2^{31}, -1]\}.$$

Under these conditions, there is no way to repair the expression. Our algorithm in Sec. 4 will prove that there is no way to fix the expression (1). Nevertheless, if we change the hypotheses, allowing the output language $\mathscr{O}$ to include expression casting, then `-((long) y)` is a verified repair for `-y` under $\mathscr{S}$. The arithmetic overflow disappears as 2^{31} is exactly representable in 64 bits[1], and the semantics coincides with that over $\mathbb{Z}$. Alternatively, we can imagine adding the conditional expression to $\mathscr{O}$. Then

$$\mathtt{y} == \mathtt{MinValue}\,?\,\mathtt{0} : -\mathtt{y}$$

is a repair in that the expression is guaranteed to not overflow, but it is *not* a verified repair as the $\mathbb{Z}$ interpretation and the machine one disagree.

Example 1 Let us consider the code in Fig. 1, returning a sub-array of `arr` made up of `count` elements from index `start`. The careful programmer added preconditions (using CodeContracts, [4]) to protect its code against buffer overruns: the starting index and the count should be non-negative (to avoid buffer underflows) and the subsegment to extract should be included in the original array (to avoid buffer overflows). However, when `start` and `count` are very large, `start + count` may result into an arithmetic overflow, *i.e.*, it evaluates to a negative value. As a consequence, the third precondition is trivially satisfied (an array length is always non-negative), but the program will probably go wrong, because of some buffer underflow later in the execution.

Again, sound static analysis tools like `cccheck` will spot the possible arithmetic overflow in the example above. Our goal here is to synthesize an arithmetic expression matching the intent of the programmer (*i.e.*, the semantics over $\mathbb{Z}$). The algorithm we introduce in Sec. 5 will synthesize the expression:

```
start <= arr.Length - count                    (2)
```

[1] Note that this repair may require changing the nominal type of `x` if it is not defined as a `long`.

```
int[] GetSubArray(int[] arr, int start, int count)
{
  Contract.Requires(0 <= start);
  Contract.Requires(0 <= count);
  Contract.Requires(start + count <= arr.Length);
  // ... rest of the code omitted ...
}
```

Figure 1: A parameter validation incorrect because of overflows.

which is guaranteed to be arithmetic overflows-free. The reason for that is that the abstract element

$$\mathscr{S} = \{\texttt{start} \in [0, 2^{31}-1], \texttt{count} \in [0, 2^{31}-1], \texttt{arr.Length} \in [0, 2^{31}-1]\}$$

implies that

$$\texttt{arr.Length} - \texttt{count}$$

can never be too small to cause an arithmetic overflow.

In general, on machine arithmetics (2) *is not* equivalent to the third postcondition of Fig. 1. Therefore, the repair cannot be done in a purely syntactic way, but it requires some semantic knowledge.

Example 2 Let us consider $\mathscr{I} = \mathscr{O}$ to be the language of arithmetic additions. For simplicity, let us assume to have 4 bits signed integers. As a consequence -8 and 7 are respectively the smallest and the largest representable natural numbers. The sum

$$\mathtt{x}_1 + \mathtt{x}_2 + \mathtt{x}_3 + \mathtt{x}_4 + \mathtt{x}_5 + \mathtt{x}_6 + \mathtt{x}_7 \tag{3}$$

with the semantic knowledge

$$\mathscr{S} = \{\mathtt{x}_1 \in [-2,3], \mathtt{x}_2 \in [-1,0], \mathtt{x}_3 \in [1,2], \mathtt{x}_4 \in [-3,-1], \mathtt{x}_5 \in [-3,-2], \mathtt{x}_6 \in [-1,1], \mathtt{x}_7 \in [2,4]\}$$

may overflow when adding $\mathtt{x}_6$ to the partial sum — assuming left-to-right evaluation order.

Our algorithm in Sec. 4 will automatically synthesize the following sum:

$$\mathtt{x}_3 + \mathtt{x}_7 + \mathtt{x}_4 + \mathtt{x}_5 + \mathtt{x}_2 + \mathtt{x}_6 + \mathtt{x}_1 \tag{4}$$

which is a verified repair, as (3) and (4) coincide with the $\mathbb{Z}$ interpretation, but (4) does not overflow.

Main Results. First, we will define the problem of the automatic repairing of overflowing expressions starting from the semantic knowledge inferred by a static analyzer (Sec. 3). Next, we will focus our attention on the common case of linear expressions with integer coefficients. We show that when the underlying abstract domain is Intervals (Sec. 4) then there exists a *complete and polynomial* algorithm to repair possibly overflowing expressions – this result was quite surprising for us. When the input language is widened to consider relations among those linear expressions, then the algorithm is still polynomial, but incomplete (Sec. 5). On the other hand, when the underlying abstract domain is refined to Octagons, we have a complete yet worst-case exponential algorithm — this result can be easily generalized to template-based numerical domains (Sec. 6). We will conclude evaluating a prototype implementation on a set of benchmarks (Sec. 7).

2 Preliminaries

Syntax. Without any loss of generality we assume a strongly typed while-language W with expressions E. Expressions are either *arithmetic* expressions (*e.g.*, $5 * \mathtt{x} + 1$) or *relational* expressions (*e.g.*, $\mathtt{x} + 2 < 3 * \mathtt{y}$). Variables are declared to belong to some integral type M_b. Expressions are well typed, too.

An integral type can be a signed or unsigned integer of b bits. The smallest representable unsigned value over b bits is 0 and the largest is $2^b - 1$. The smallest representable signed is -2^{b-1} and the largest one is $2^{b-1} - 1$. We denote by $\underline{M}_b$ and $\overline{M}_b$ the smallest and the largest values of $M_\mathtt{b}$. When clear from the context, we will omit the subscript b.

Semantics. A program state either denotes an error or maps variables to values, *i.e.*,

$$\Sigma = \{\mathtt{err}\} \cup (\mathtt{Vars} \to \mathbb{Z}).$$

The only way to reach the error state err is by means of an arithmetic overflow — for the sake of simplicity we treat division by zero as a particular case of overflow. We assume a semantic evaluation function

$$\mathsf{eval} \in \mathtt{E} \times \Sigma \to \mathbb{Z} \cup \{\mathtt{err}\}.$$

The function eval respects a fixed left-to-right evaluation of arithmetic expressions (like C# or Java, but unlike C).

If an arithmetic overflow is encountered during the evaluation of $\mathtt{e} \in \mathtt{E}$ in $\sigma \in \Sigma$ then $\mathsf{eval}(\mathtt{e}, \sigma) = \mathtt{err}$.

We define the relation $\models_{\mathit{notOverflow}} \in \wp(\wp(\Sigma) \times \mathtt{E})$ as

$$S \models_{\mathit{notOverflow}} \mathtt{e} \iff \forall \sigma \in S.\ \mathsf{eval}(\mathtt{e}, \sigma) \neq \mathtt{err}.$$

The *concrete* semantics of a program P associates each program point $\mathtt{pc} \in \mathtt{PP}$ with the set of reachable states $\subseteq \Sigma$. Formally: $[\![\mathtt{P}]\!] \in \mathtt{PP} \to \wp(\Sigma)$.

An abstract domain $\langle \mathscr{A}, \sqsubseteq \rangle$ over-approximates sets of states. In the abstract interpretation framework, this is formalized by a Galois connection, *i.e.*, by two monotonic functions α, γ such that

$$\forall S \in \wp(\Sigma).\ S \subseteq \gamma \circ \alpha(S) \text{ and } \forall a \in \mathscr{A}.\ \alpha \circ \gamma(a) \sqsubseteq a.$$

The *abstract* semantics of P associates each program point with a *sound* approximation of the reachable states. Formally, $[\![\mathtt{P}]\!]^a \in \mathtt{PP} \to \mathscr{A}$ is such that

$$\forall \mathtt{pc} \in \mathtt{PP}.\ [\![\mathtt{P}]\!](\mathtt{pc}) \subseteq \gamma([\![\mathtt{P}]\!]^a(\mathtt{pc})).$$

3 The Problem

In order to define the problem of repairing (or synthesis) of non-overflowing expressions, we should make all the underlying assumptions explicit.

First, we assume the program P is analyzed using a sound static analyzer with an underlying abstract domain $\mathscr{A}$ [3, 19].

Second, we denote the set of potential arithmetic overflows by $\mathbb{A} = \{\langle \mathtt{pp}, \mathtt{e} \rangle \mid \mathtt{pc} \in \mathtt{PP}\}$.

Third, we define $\mathbb{O}$, the subset of $\mathbb{A}$ for which we cannot prove the absence of overflows, *using* the abstract domain $\mathscr{A}$:

$$\mathbb{O} = \{\langle \mathtt{pp}, \mathtt{e} \rangle \in \mathbb{A} \mid \gamma([\![\mathtt{P}]\!]^a(\mathtt{pp})) \not\models_{\mathit{notOverflow}} \mathtt{e}\}.$$

Usually, static analyzers stop here: They simply report the assertions $\mathbb{O}$ as possible overflows. We want to move it a step further.

We want to propose a verified repair for the expressions in $\mathbb{O}$. In general, we cannot hope to repair arbitrary arithmetic expressions, so we focus our attention on a given set of input expressions $\mathscr{I}$. Examples of $\mathscr{I}$ are sums with unary coefficients $\mathtt{U} = \{\mathtt{e} \mid \mathtt{e} = \sum_i \mathtt{x}_i\}$ and linear arithmetic with integer coefficients $\mathtt{L} = \{\mathtt{e} \mid \mathtt{e} = \sum_i a_i \cdot \mathtt{x}_i, a_i \in \mathbb{Z}\}$. The repaired, or non-overflowing output expression belongs to a set of expressions $\mathscr{O}$.

We require that $\mathscr{I} \subseteq \mathscr{O} \subseteq \mathtt{W}$, *i.e.*, the output expressions are at least as expressive as the input ones, and both of them are expressible in our programming language. We assume that the expressions in $\mathscr{I}$ and $\mathscr{O}$ are side-effect free.

Our problem is to find for each expression $\mathtt{e}$ in $\mathbb{O}$ an equivalent expression in $\mathscr{O}$ which is provably non-overflowing according to the semantic information inferred by the static analyzer at program point $\mathtt{pp}$. Formally, we want an algorithmic characterization of the set of repairs:

$$\mathbb{R}_{\langle \mathscr{I},\mathscr{O},\mathscr{A}\rangle} = \{\langle \mathtt{pp}, \mathtt{e}'\rangle \mid \langle \mathtt{pp}, \mathtt{e}\rangle \in \mathbb{O}, \mathtt{e} \in \mathscr{I}, \mathtt{e}' \in \mathscr{O}, \mathtt{e} \equiv_{\mathbb{Z}} \mathtt{e}', \gamma([\![\mathtt{P}]\!]^a(\mathtt{pp})) \models_{\mathit{notOverflow}} \mathtt{e}'\}$$

where $\equiv_{\mathbb{Z}}$ denotes the equivalence of expressions when they are interpreted over natural numbers and *not* machine numbers. In practice, we are interested in non-trivial solutions to $\mathbb{R}$, or in non-trivial under-approximations of $\mathbb{R}$. An example of a trivial solution is the brute-force generation of all the possible parsings $\mathtt{e}'$ of an expression $\mathtt{e} \in \mathscr{I}$ followed by testing $\mathtt{e}'$ whether or not may overflow. This solution is exponential in the size of $\mathtt{e}$, a situation we want to avoid. An example of a trivial under-approximation is the empty set — *i.e.*, no expression is repaired.

4 Linear Expressions with Integer Coefficients

We begin by focusing our attention on the very common case when $\mathscr{I} = \mathscr{O} = \mathtt{L}$, *i.e.*, the input and the output languages are linear expression with *integer* coefficients. We assume the underlying abstract domain $\mathscr{A}$ to be the abstract domain Intv of intervals [3]. In this section we show that there exists a *complete* and *polynomial* algorithm for $\mathbb{R}_{\langle \mathtt{L},\mathtt{L},\mathsf{Intv}\rangle}$ – Algorithm 1.

The first step of Algorithm 1 is a pre-processing step to get rid of multiplications of an integer constant to a variable. Given an expression $\mathtt{e} \in \mathtt{L}$, *i.e.*,

$$\mathtt{e} = \sum_{i=1}^{n} a_i \cdot \mathtt{x}_i \qquad \text{with } a_i \in \mathbb{Z} \text{ and } 1 \leq i \leq n$$

each term of the sum $a_i \cdot \mathtt{x}_i$ is replaced by the corresponding sum:

$$a_i \cdot \mathtt{x}_i = \underbrace{\mathtt{x}_i + \ldots + \mathtt{x}_i}_{a_i \text{ times}}.$$

Without loss of generality, we assume that all the a_i are positive. Otherwise:

(i) if a_i is zero, then it is trivial to remove $\mathtt{x}_i$,

(ii) if a_i is negative, then we record it by negating the interval $\mathscr{S}(\mathtt{x}_i)$.

The pre-processed expression is then in the form of $\sum_{i=1}^{k} \mathtt{y}_i$ where $k = \sum_{i=1}^{n} a_i$ and $\mathtt{y}_i$ are variables appearing in $\mathtt{e}$.

Algorithm 1 Sum Refactoring

Require: An expression $\mathtt{e} \in \mathtt{L}$ and a map $\mathscr{S} \in \mathtt{Vars} \to \mathbb{Z} \times \mathbb{Z}$

Ensure: A permutation $\pi \in [1,k] \to [1,k]$ indicating in which order to add the elements $\mathtt{y}_i$ to avoid overflows, or fail if it does not exist.

1: Transform $\mathtt{e}$ into a sum such that $\mathtt{e} = \sum_{i=1}^{k} \mathtt{y}_i$
2: Create $Y_{>0}$, $Y_{<0}$ and $Y_\top$
3: $R \leftarrow Y_{>0} \cup Y_{<0}$
4: $s \leftarrow [0,0]$; $i \leftarrow 1$; $\pi = \lambda\, j.j$
5: **while** $R \neq \emptyset$ **do**
6: **if** $\exists\ j \in R$ such that $s + \mathscr{S}(\mathtt{y}_j) \subseteq M$ **then**
7: $s \leftarrow s + \mathscr{S}(\mathtt{y}_j)$; $R \leftarrow R \setminus j$
8: $\pi(i) \leftarrow j$; $i \leftarrow i+1$
9: **else**
10: Fail
11: **end if**
12: **end while**
13: **for all** $j \in Y_\top$ **do**
14: $s \leftarrow s + \mathscr{S}(\mathtt{y}_j)$; $\pi(i) \leftarrow j$; $i \leftarrow i+1$
15: **if** $s \not\subseteq M$ **then**
16: Fail
17: **end if**
18: **end for**

After pre-processing, Algorithm 1 partitions the (indexes of the) variables appearing in the expression into three sets: the positive variables $Y_{>0}$, the negative variables $Y_{<0}$, and the variables that can be positive, negative, or zero $Y_\top$. Let $\underline{x}$ and $\overline{x}$ denote respectively the lower and upper bounds of an interval x. Then we can formally define the above sets as:

$$Y_{>0} = \{i \mid 1 \leq i \leq k,\ \underline{\mathtt{y}_i} > 0\}, Y_{<0} = \{i \mid 1 \leq i \leq k,\ \overline{\mathtt{y}_i} < 0\}, Y_\top = \{i \mid 1 \leq i \leq k,\ \underline{\mathtt{y}_i} \leq 0 \ \wedge\ \overline{\mathtt{y}_i} \geq 0\}.$$

We use the shortcut $R = Y_{>0} \cup Y_{<0}$ for the set of variables that do contain zero.

The main loop of Algorithm 1 constructs a permutation of terms in the input sum such that the partial sum s is kept as far as possible from $\underline{M}$ and $\overline{M}$. Initially, the partial interval sum s is set to $[0,0]$ and π is set to the identity function. The algorithm iteratively selects an element $\mathtt{y}_i$ with $i \in R$ such that $s + \mathtt{y}_i \subseteq M$ (while loop of Line 5). The index i is removed from R and the value of $\mathtt{y}_i$ is added to partial (interval) sum s. The permutation π records which term is selected at each iteration. If, at some iteration, there exists no term $\mathtt{y}_i$ with $i \in R$ such that $s + \mathtt{y}_i \in M$ then the algorithm fails. We will see in Property 4 that, in this case, there exists no solution. Once all the elements with indexes in R have been selected, the algorithm selects the elements whose indexes belong to $Y_\top$ in any order (for loop of Line 13). If the sum is not included in M then the algorithm fails.

Example 3 Let us consider the Ex. 1, with $b = 4$. In this case $M_4 = [-8,7]$ and an overflow arises when the term $\mathtt{x}_6 \in [-1,1]$ is added since the partial sum evaluates to $[-9,3] \not\subseteq M_4$. In Algorithm 1, $Y_{>0} = \{3,7\}$, $Y_{<0} = \{4,5\}$ and $Y_\top = \{1,2,6\}$. One output of the algorithm is (the permutation corresponding to) the sum: $\mathtt{x}_3 + \mathtt{x}_7 + \mathtt{x}_4 + \mathtt{x}_5 + \mathtt{x}_2 + \mathtt{x}_6 + \mathtt{x}_1$.

The Fig. 2 provides a graphical intuition of the way in which the Algorithm 1 works. It starts with

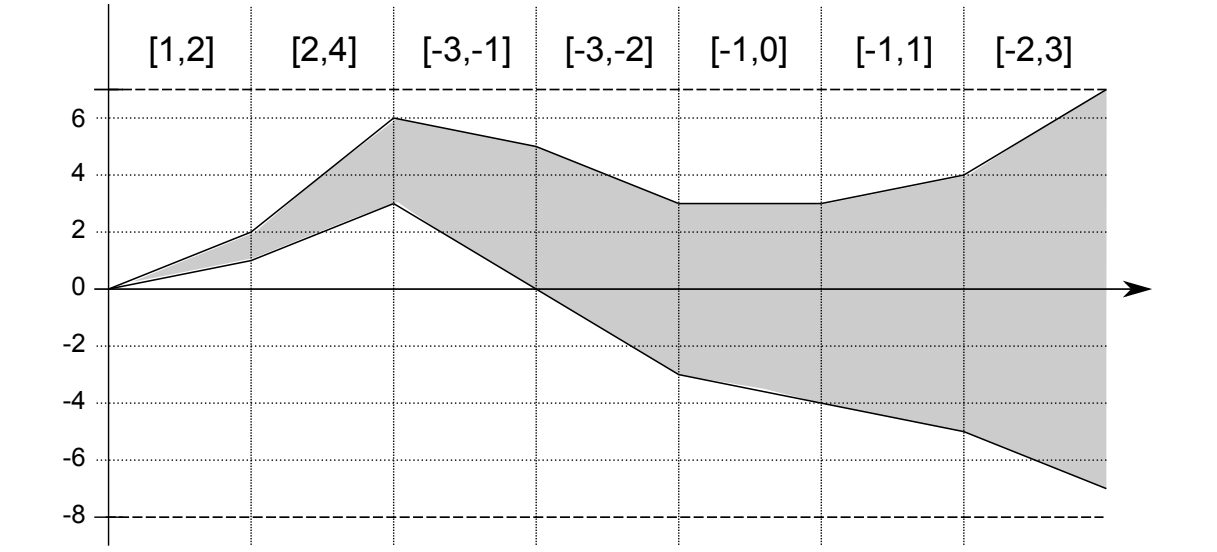

Figure 2: The graphical evolution of the partial sums of the repaired expression in Ex. 4.

the interval $[0,0]$, and it greedily selects variables such that the current partial sum gets closer, *e.g.*, to $\overline{M}$. When any value in $Y_{>0}$ causes the partial sum to go above $\overline{M}$, then elements from $Y_{<0}$ are selected. Graphically this means that the interval is heading toward $\underline{M}$. The algorithm continues until R_0 is empty, or it is not possible to "correct" the partial sum. An interesting yet unexpected result is the following one, stating that the algorithm is complete, *i.e.*, if there exists a solution to $\mathbb{R}_{\langle \mathtt{L},\mathtt{L},\mathsf{Intv}\rangle}$ then our algorithm finds it. If it fails, there is no way to repair $\mathtt{e}$ using $\mathscr{S}$.

Property 1 *(Completeness)* Let $[\![\mathtt{P}]\!]^a(\mathtt{pp}) = \mathscr{S}$ be an abstract element belonging to Intv. If Algorithm 1 fails then there exists no permutation $\pi \in [1,k] \to [1,k]$ of the elements of the sum such that the partial sums of $\sum_{i=1}^{n} \mathtt{y}_{\pi(i)}$ do not overflow, i.e. there is no $\mathtt{e'} \in \mathtt{L}$ such that $\mathtt{e} \equiv_{\mathbb{Z}} \mathtt{e'}$ and $\gamma(\mathscr{S}) \models_{\mathit{notOverflow}} \mathtt{e'}$.

Proof First, let us focus on the while loop of Line 5. We show that if Algorithm 1 fails at Line 10 then any parsing of the sum leads to an overflow. More precisely, if $\exists Y'$ and Y'' such that $Y' \cup Y'' = [1,k]$, $Y' \cap Y'' = \emptyset$, $s = \sum_{i \in Y'} \mathtt{y}_i \subseteq M$ and $\forall i \in Y''$, $s + \mathtt{y}_i \not\subseteq M$ then $\sum_{i=1}^{k} \mathtt{y}_i \not\subseteq M$.

Let $Y'' = \{b_1, \ldots, b_m\}$ and let us focus on the upper bound. If $\forall i,\ 1 \le i \le m,\ \overline{s} + \overline{\mathtt{y}}_{b_i} > \overline{M}$ then

$$\begin{array}{rcl} \overline{s} + \overline{\mathtt{y}}_{b_1} & > & \overline{M} \\ \overline{s} + \overline{\mathtt{y}}_{b_2} & > & \overline{M} \\ \vdots & & \vdots \\ \overline{s} + \overline{\mathtt{y}}_{b_m} & > & \overline{M}. \end{array}$$

Their sum implies that

$$m\overline{s} + \sum_{b \in Y''} \overline{\mathtt{y}}_b > m\overline{M}. \tag{5}$$

Using the fact that $s = \sum_{i \in Y'} \mathtt{y}_i$ and that $Y' \cup Y'' = [1,k]$, Equation (5) can be transformed as follows:

$$\begin{array}{rcl} & & m\overline{s} + \sum_{b \in Y''} \overline{\mathtt{y}}_b > m\overline{M} \\ \Leftrightarrow & & (m-1)\overline{s} + \overline{s} + \sum_{b \in Y''} \overline{\mathtt{y}}_b > m\overline{M} \\ \Leftrightarrow & & (m-1)\overline{s} + \sum_{a \in Y'} \overline{\mathtt{y}}_a + \sum_{b \in Y''} \overline{\mathtt{y}}_b > m\overline{M} \\ \Leftrightarrow & & \sum_{i=1}^{k} \overline{\mathtt{y}}_i > m\overline{M} - (m-1)\overline{s}. \end{array}$$

Now, let us assume that $\overline{s} \leq \overline{M}$. From the previous equation it follows that

$$\sum_{i=1}^{k} \overline{\mathtt{y}}_i > m\overline{M} - (m-1)\overline{s} \geq m\overline{M} - (m-1)\overline{M}.$$

Therefore, $\sum_{i=1}^{k} \overline{\mathtt{y}}_i > \overline{M}$, which is a contradiction. The same arguments holds for the lower bound $\underline{M}$.

Concerning the for loop of Line 13, any element $\mathtt{y}_i$ with $i \in Y_\top$ makes the bounds of s grow (except for the trivial case of adding [0, 0]). Consequently, if we cannot fix the variables in R, there is no way we can fix the initial set of variables. The interval sum is deterministic, so the partial sum after the while loop of Line 5 is always the same interval, no matter which parsing is inferred. In the loop of Line 13, the width of the interval for the partial sum can only grow at each step with one of the two bounds strictly moving toward $\overline{M}$ or $\underline{M}$. As a consequence, if some parsing fails then all the parsings fail.

Property 2 *(Complexity)* For a linear expression with $O(k)$ variables, Algorithm 1 performs $O(k^2)$ operations.

Proof Constants are bounded and, consequently, the preprocessing may only add a constant multiplicative factor. So the number of terms in the sums given to Algorithm 1 is asymptotic linear in the number of variables of the source expressions. The while loop of Line 5 is executed $O(k)$ times and the selection of j at Line 6 is done in $O(k)$. The for loop of Line 13 being linear in k, the global complexity of the algorithm is $O(k^2)$.

5 Relations among Linear Expressions

We extend the results of the previous section when the considered expressions are comparisons of linear arithmetic expressions with unary coefficients. We consider the set

$$\mathtt{B} = \{\mathtt{e}_1 \lozenge \mathtt{e}_2 \mid \mathtt{e}_1, \mathtt{e}_2 \in \mathtt{L}, \lozenge \in \{=, \neq, <, >, \leq, \geq\}\}$$

for input and output expressions. We use the abstract domain Intv for the semantic knowledge. The Algorithm 2 computes an *under*-approximation to $\mathbb{R}_{\langle \mathtt{B},\mathtt{B},\mathsf{Intv}\rangle}$ in *polynomial* time.

We first need to define some functions used by the algorithm. The (distance from zero) function $d \in \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$ is such that

$$d([\underline{x}, \overline{x}]) = \max\left(|\underline{x}|, |\overline{x}|\right).$$

We let $d(\mathtt{e})$ denote $d(\mathsf{eval}_{\mathsf{Intv}}(\mathtt{e}, [\![\mathtt{P}]\!]^a(\mathtt{pp}))$, where $\mathsf{eval}_{\mathsf{Intv}}$ is the *most* precise evaluation of $\mathtt{e}$ using *unbounded* interval arithmetics [13].

The function

$$\mathsf{select}(X) \in \wp(\mathbb{Z} \times \mathbb{Z}) \to (\mathbb{Z} \times \mathbb{Z}) \cup \{\mathtt{fail}\}$$

selects an interval $x \in X$ such that $\underline{M} \notin x$ and $\forall y \in X. d(y) \leq d(x)$. If such an interval does not exist, then $\mathsf{select}(X) = \mathtt{fail}$.

Given an (abstract) environment mapping variables to intervals $\mathscr{S} \in \mathtt{Vars} \to (\mathbb{Z} \times \mathbb{Z})$, we assume, without losing generality, that $\mathscr{S}$ is injective — the Algorithm can be easily extended otherwise.

Finally, we let $\phi_{\mathscr{S}_0,\mathscr{S}} \in \mathtt{Vars} \to \mathtt{E}$ denote the sign of a variable. It is defined as $\phi_{\mathscr{S}_0,\mathscr{S}}(\mathtt{x}) = \mathtt{x}$ if $\mathscr{S}_0(\mathtt{x}) = \mathscr{S}(\mathtt{x})$; $\phi_{\mathscr{S}_0,\mathscr{S}}(\mathtt{x}) = -1 * \mathtt{x}$ if $\mathscr{S}_0(\mathtt{x}) = -\mathscr{S}(\mathtt{x})$ and undefined otherwise.

The idea of Algorithm 2 is to move terms to the left or the right of the $\lozenge$ relation, and then check whether they overflow. If it cannot pick any variable, then it simply fails. The input of the algorithm are two sets of variables, A and B, appearing as sum on the left and on the right of $\lozenge$. The set $\mathscr{M}$ remembers

Algorithm 2 Boolean Expressions Refactoring

Require: Two sets of variables A, B, a Boolean expression $\mathtt{b} = \mathtt{e}_1 \lozenge \mathtt{e}_2$ such that $\mathtt{e}_1 = \sum_{\mathtt{a}\in A} \mathtt{a}$ and $\mathtt{e}_2 = \sum_{\mathtt{b}\in B} \mathtt{b}$ and an interval map $\mathscr{S}_0$.

Ensure: A solution to the overflowing problem, or fail if it does not exist

1: $\mathscr{M} \leftarrow \emptyset$; $\mathtt{e}'_1 \leftarrow \mathtt{e}_1$; $\mathtt{e}'_2 \leftarrow \mathtt{e}_2$; $\mathscr{S} \leftarrow \mathscr{S}_0$
2: **while** $\neg$Algorithm1$(\mathtt{e}'_1, \mathscr{S}) \vee \neg$Algorithm1$(\mathtt{e}'_2, \mathscr{S})$ **do**
3: **if** $d(\mathtt{e}'_1) \geq d(\mathtt{e}'_2)$ **then**
4: $\alpha \leftarrow \mathsf{select}(\mathscr{S}(A \setminus \mathscr{M}))$
5: **if** α = `fail` **then**
6: Fail
7: **end if**
8: $\mathtt{y} \leftarrow \mathscr{S}^{-1}(\alpha)$
9: $A \leftarrow A \setminus \{\mathtt{y}\}; B \leftarrow B \cup \{\mathtt{y}\};$
10: **else**
11: $\beta \leftarrow \mathsf{select}(\mathscr{S}(B \setminus \mathscr{M}))$
12: **if** β = `fail` **then**
13: Fail
14: **end if**
15: $\mathtt{y} \leftarrow \mathscr{S}^{-1}(\beta)$
16: $A \leftarrow A \cup \{\mathtt{y}\}; B \leftarrow B \setminus \{\mathtt{y}\}$
17: **end if**
18: $\mathtt{e}'_1 \leftarrow \sum_{\mathtt{a}\in A} \mathtt{a}$; $\mathtt{e}'_2 \rightarrow \sum_{\mathtt{b}\in B} \mathtt{b}$
19: $\mathscr{M} \leftarrow \mathscr{M} \cup \{\mathtt{y}\}$
20: $\mathscr{S} \rightarrow \mathscr{S}[\mathtt{y} \mapsto -\mathscr{S}]$
21: **end while**
22: **return** $\sum_{\mathtt{a}\in A} \phi_{\mathscr{S}_0,\mathscr{S}}(\mathtt{a}) \lozenge \sum_{\mathtt{b}\in B} \phi_{\mathscr{S}_0,\mathscr{S}}(\mathtt{b})$

which variables have been moved on the left or right side. On entry it is trivially the empty set. The arithmetic expressions $\mathtt{e}'_1, \mathtt{e}'_2$ are the current approximations for the solution, and $\mathscr{S}$ records the ranges for variables. On entry, those variables are set to the input parameters.

The Algorithm 2 iterates as long as at least one of the expressions $\mathtt{e}'_1$ or $\mathtt{e}'_2$ overflows — we use Algorithm 1 to check it. It first selects the expression which is further away from zero — roughly, the expression which causes the largest overflow. Then, it selects the addendum that contributes most to the overflow, and moves it on the other side of $\lozenge$. This is reflected by the updates to A and B. The memoization set $\mathscr{M}$ remembers which terms have already been moved. This forbids the same term to be moved twice. Finally, $\mathscr{S}$ is updated to remember that y is negated. The algorithm fails if select fails, that is if no more term can be moved to the other side of the relation. Otherwise, a new expression is returned, where variables moved to one side or another of $\lozenge$ are negated.

Example 4 Let us consider the evaluation of the Boolean expression $\mathtt{x}_1 + \mathtt{x}_2 \leq \mathtt{x}_3 + \mathtt{x}_4 + \mathtt{x}_5$ with simplified 4 bits integers and with the semantic information:

$$\mathscr{S}_0 = \mathscr{S} = \{\mathtt{x}_1 \in [-1,1], \mathtt{x}_2 \in [-2,0], \mathtt{x}_3 \in [1,2], \mathtt{x}_4 \in [2,3], \mathtt{x}_5 \in [5,6]\}.$$

An overflow arises in the evaluation of the right hand side of the relation since:

$$\mathtt{x}_3 + \mathtt{x}_4 + \mathtt{x}_5 \in [8,11] \not\subseteq [-8,7].$$

Let us apply Algorithm 2. At the first step, $B = \{\mathtt{x}_3, \mathtt{x}_4, \mathtt{x}_5\}$, $\mathscr{M} = \emptyset$. Therefore $\mathsf{select}(\mathscr{S}(B \setminus \mathscr{M})) = [5,6]$ since $d(\mathtt{x}_5) = 6$. The memoization set is updated to $\mathscr{M} = \{\mathtt{x}_5\}$ and the relation is transformed into $\mathtt{x}_1 + \mathtt{x}_2 + \mathtt{x}_5 \leq \mathtt{x}_3 + \mathtt{x}_4$. We record the fact that $\mathtt{x}_5$ appears in a negated context by updating the semantic information:

$$\mathscr{S} = \{\mathtt{x}_1 \in [-1,1], \mathtt{x}_2 \in [-2,0], \mathtt{x}_3 \in [1,2], \mathtt{x}_4 \in [2,3], \mathtt{x}_{\mathbf{5}} \in [-\mathbf{6}, -\mathbf{5}]\}.$$

Now, $\mathtt{x}_1 + \mathtt{x}_2 + \mathtt{x}_5 \in [-9,-4] \not\subseteq [-8,7]$. Therefore, the evaluation of the left operand may lead to an overflow, and so the loop is entered again. The function $\mathsf{select}(\mathscr{S}(A \setminus \mathscr{M}))$ selects $[-2,0]$, that is the interval corresponding to $\mathtt{x}_2$. The new expression is $\mathtt{x}_1 + \mathtt{x}_5 \leq \mathtt{x}_3 + \mathtt{x}_4 + \mathtt{x}_2$, with

$$\mathscr{S} = \{\mathtt{x}_1 \in [-1,1], \mathtt{x}_{\mathbf{2}} \in [\mathbf{0}, -\mathbf{2}], \mathtt{x}_3 \in [1,2], \mathtt{x}_4 \in [2,3], \mathtt{x}_5 \in [-6,-5]\}.$$

The algorithm stops since the expression does not overflow. The expression $\mathtt{x}_1 - \mathtt{x}_5 \leq \mathtt{x}_3 + \mathtt{x}_4 - \mathtt{x}_2$ is then returned to the caller.

Property 3 For a relation between two linear expressions with $O(n)$ variables, Algorithm 2 performs $O(n^3)$ steps.

Proof Each variable is moved at most once from one side of the relation to the other. Then the while loop of Line 2 is repeated $O(n)$ times. The select function is $O(n)$ but the Algorithm 1 is $O(n^2)$. The global complexity is then $O(n^3)$.

6 Exploiting Relational Information

We extend the results of Sect. 4 by using Oct, the abstract domain of Octagons [12] as the underlying semantic knowledge. Intuitively, having a more precise abstract domain enables the inference of more relations among variables and as a consequence more arithmetic expressions can be repaired. On the other hand, it also makes the search space for the problem larger. We show that there exists an algorithm to solve $\mathbb{R}_{\langle \mathscr{I}, \mathscr{O}, \mathsf{Oct} \rangle}$ exactly, *i.e.*, if an expression can be repaired, then our algorithm finds it.

With Oct, constraints are either in the form $a \leq \mathtt{x} \leq b$ (non-relational constraints) or $a \leq \mathtt{x} \pm \mathtt{y} \leq b$ (relational constraints). In the following we will use an interval notation to denote those constraints: $\mathscr{S}(\mathtt{x}) = [a,b]$ and $\mathscr{S}(\mathtt{x+y}) = [a,b]$.

Example 5 Let us consider variables `x, y, z` of type M_4, the sum `x+y+z`, and the semantic information $\mathscr{S}$:

$$\{\mathtt{x} \in [-2,2], \mathtt{y} \in [-1,3], \mathtt{z} \in [-1,4], \mathtt{x+y} \in [-2,3], \mathtt{y+z} \in [-2,4], \mathtt{x+z} \in [-1,5]\}.$$

If we ignore the relational constraints then we cannot propose a fix for the expression. In fact, Algorithm 1 will fail because:

$$[-2,2] + [-1,3] + [-1,4] \not\subseteq [-8,7].$$

We assume the application of the same preprocessing steps of the previous sections (multiplications by constants are expanded into sums, variables are only added – subtraction is captured in $\mathscr{S}$). Furthermore, for the sake of simplicity, we also assume that all the variables are different — variables with multiple occurrences are renamed.

Our repairing algorithm performs a depth-first visit of a weighted directed graph G. Intuitively, the graph indicates in which order we should add variables or pairs of variables for which we have some

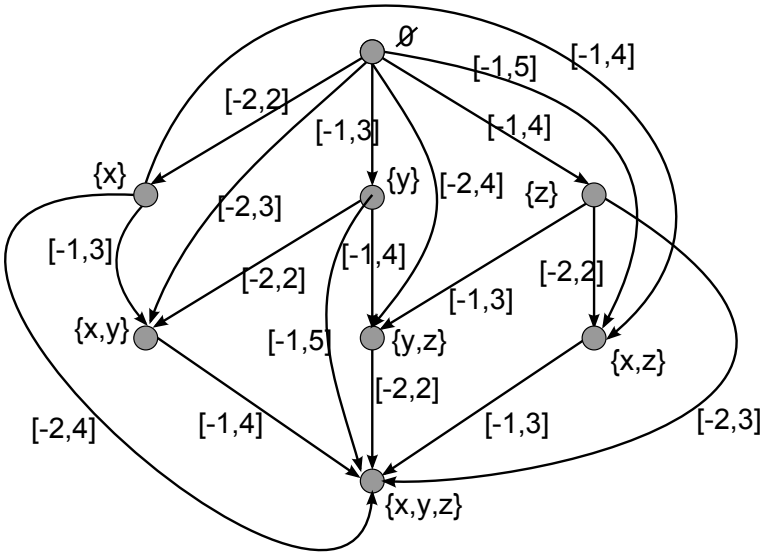

Figure 3: Graph describing the ways of summing x, y, z with the information of Ex. 6.

relational information in $\mathscr{S}$. We do not explicitly build the graph — it is of exponential size. Instead, we memorize only the current path corresponding to a non overflowing partial sum. The nodes correspond to partial sums of variables. The weight (an interval) attached to an edge e indicates the value which must be added to the partial sum when one or two variables are added. In general, there exists several paths from a source node to a destination node depending on which constraints we use to add the variables. For instance, in Ex. 6, we have a path corresponding to $\mathtt{x}+(\mathtt{y}+\mathtt{z})$ and another path corresponding to $(\mathtt{x}+\mathtt{y})+\mathtt{z}$ whose weights are $[-4,6]$ and $[-3,7]$, respectively.

Formally, given an input set of variables $X = \{\mathtt{x}_1, \ldots \mathtt{x}_n\}$ and the semantic information $\mathscr{S}$, we define the graph $G = (V,E)$ as follows:

- $V = \wp(X)$: the nodes of G are sets of variables,
- the entry node is $\emptyset$ and the exit node is X,
- $(v_1, v_2) \in E$ if $v_2 = v_1 \cup \{\mathtt{x}_i\}$ for some $v_1 \subseteq X$ and some variable $\mathtt{x}_i \in X$. In this case, $\mathscr{W}\big((v_1, v_2)\big) = \mathscr{S}(\mathtt{x}_i)$,
- $(v_1, v_2) \in E$ if $v_2 = v_1 \cup \{\mathtt{x}_i, \mathtt{x}_j\}$ for some $v_1 \subseteq X$ and some pair of variables $\mathtt{x}_i$ and $\mathtt{x}_j$, with $i \neq j$. In this case, $\mathscr{W}\big((v_1, v_2)\big) = \mathscr{S}(\mathtt{x}_i + \mathtt{x}_j)$.

Example 6 Let us consider the arithmetic expression and the semantic information of the Ex. 6. The Figure 3 shows the corresponding graph G. With G, we can derive the non-overflowing expressions `(x+y)+z` or `x+(y+z)`. The expression `y+(x+z)` is not a repair – it may overflow under the knowledge encoded by $\mathscr{S}$.

Let $v_1, \ldots v_k$ be a sequence of vertices in G. Then, $\overrightarrow{v} = (v_1, \ldots v_k)$ is a path if $v_1 = \emptyset$ and for all i such that $1 \leq i < k$, $(v_i, v_{i+1}) \in E$; $\overrightarrow{v} \curvearrowright v'$ is the path extended with vertex v'; $dest(\overrightarrow{v}) = v_k$ is the last vertex in a non-empty path; $succ(v)$ are the successors of vertex v, $succ(\overrightarrow{v}) = succ(dest(\overrightarrow{v}_k))$, are the successors of a (non-empty) path.

The function Repair, Algorithm 3, performs a depth-first search in G. The input to Repair is path $\overrightarrow{v}$ in G. Intuitively, $\overrightarrow{v}$ describes a way of summing the variables in v_k. The weight of $\mathscr{W}(\overrightarrow{v})$ is the

Algorithm 3 Sum Refactoring with Relational Information

Require: A path $\overrightarrow{v}$ on G and an octagon $\mathscr{S}$

Ensure: A path indicating how to compute the total sum without overflows, or fail if it does not exists

1: **Function** Repair($\overrightarrow{v}$)
2: $\mathscr{N} \leftarrow succ(\overrightarrow{v})$; $\overrightarrow{r} \leftarrow \overrightarrow{v}$
3: **while** $\mathscr{N} \neq \emptyset \wedge dest(\overrightarrow{r}) \neq X$ **do**
4: $v' \leftarrow pickOne(\mathscr{N})$
5: **if** $\mathscr{S} \models_{notOverflow} \sum_{\mathtt{v} \in \overrightarrow{v} \frown v'} \mathtt{x}$ **then**
6: $\overrightarrow{r} \leftarrow \mathsf{Repair}(\overrightarrow{v} \frown v')$
7: **end if**
8: $\mathscr{N} \leftarrow \mathscr{N} \setminus \{v'\}$
9: **end while**
10: **if** $dest(\overrightarrow{r}) = X$ **then**
11: **return** $\overrightarrow{r}$
12: **else**
13: Fail
14: **end if**

unbounded interval sum of the weights of the edges:

$$\mathscr{W}(\overrightarrow{x}) = \sum_{j=1}^{k-1} \mathscr{W}\big((v_j, v_{j+1})\big).$$

Algorithm 3 visits G without explicitly building it. The set $\mathscr{N}$ is initialized to the set of successor nodes of the current path. The algorithm iterates until $\mathscr{N}$ is empty (in which case the algorithm fails) or the exit node X is reached (in which case the algorithm succeeds). In the main loop, a new node from the unexplored successors $\mathscr{N}$ is picked via the function pickOne. Heuristically, we want pickOne to return the node v' minimizing $\mathscr{W}\big((x_n, x')\big)$. If the new partial sum corresponding to the path $\overrightarrow{v} \frown v'$ does not overflow, then the exploration continues from such a path. Otherwise, the exploration continues with another successor in $\mathscr{N}$.

Example 7 Let us apply Algorithm 3 to Ex. 6. The initial value for Repair is $\emptyset$, *i.e.* the path corresponding to the entry node. We have

$$\mathscr{N} = \big\{\{\mathtt{x}\}, \{\mathtt{y}\}, \{\mathtt{z}\}, \{\mathtt{x},\mathtt{y}\}, \{\mathtt{x},\mathtt{z}\}, \{\mathtt{y},\mathtt{z}\}\big\}.$$

The function pickOne selects $\{\mathtt{x}\}$ since $d((\emptyset, \{\mathtt{x}\})) = 2$ is minimal. Therefore, it invokes Repair on the path $(\emptyset, \{\mathtt{x}\})$. It choses $\{\mathtt{x},\mathtt{y}\}$ as successor. At the next recursive call, when it adds $\mathtt{z}$ to the path, it detects an overflow since

$$[-2,2] + [-1,3] + [-1,4] \not\subseteq [-8,7].$$

The algorithm backtracks and finds the path

$$(\emptyset, \{\mathtt{x}\}, \{\mathtt{x},\mathtt{y},\mathtt{z}\})$$

whose weight is $[-2,2] + [-2,4] \subseteq [-8,7]$.

Property 4 *(Completeness)* Algorithm 3 is complete.

Proof Algorithm 3 performs a search into a graph G which describes all the ways of summing the variables of the expression. Then, if a solution exists, it is included in G and the algorithm necessarily finds it.

The Algorithm 3 can be generalized to any template-based numerical abstract domain [18]. In this case, we have additional constraints of form

$$a \leq a_1 \cdot \mathtt{x}_1 + \ldots + a_n \cdot \mathtt{x}_n \leq b \tag{6}$$

where the $\mathtt{x}_i$, $1 \leq i \leq n$ are variables of the program and where a, b and the coefficients a_i, $1 \leq i \leq n$ are constants. The preprocessing transforms each term $a_i \cdot x_i$ into $\mathtt{x}_i^{(1)} + \mathtt{x}_i^{(2)} + \ldots + \mathtt{x}_i^{(a_i)}$ and the graph used by Algorithm 3 can be extended with new edges (v_1, v_2) such that

$$v_2 = v_1 \cup \bigcup_{1 \leq i \leq n} \left(\bigcup_{1 \leq j \leq a_i} \mathtt{x}_i^{(j)} \right)$$

for some $v_1 \subseteq X$ and for the constraint of Equation (6). In this case, the weight associated to the new edge (v_1, v_2) is $\mathscr{S}(\mathtt{x}_1 + \ldots + a_n \cdot \mathtt{x}_n)$.

7 Experimental Results

We have a prototype OCAML implementation that we used to validate and to experiment with the algorithms presented in the previous sections. We use 32 bits signed integers, *i.e.*, $M_{32} \in [-2^{31}, 2^{31} - 1]$.

For the algorithms 1 and 2, we use the variables $\mathtt{x}_1, \ldots, \mathtt{x}_6$ with

$$\mathscr{S} = \{\mathtt{x}_1 \in [0, 2^{29}], \mathtt{x}_2 \in [-2^{29}, 0], \mathtt{x}_3 \in [-2^{29}, 2^{29}], \mathtt{x}_4 \in [1,1], \mathtt{x}_5 \in [-1,-1], \mathtt{x}_6 \in [-1,1]\}.$$

There is nothing special about $\mathscr{S}$, we generated it randomly.

To evaluate Algorithm 1, we consider *all* the expressions in the form of $\mathtt{e} = \sum_{i=1}^{6} a_i \cdot \mathtt{x}_i$ and $a_i \in \{0,1,2,4\}$ for $1 \leq i \leq 6$. Overall, there are 3402 of such expressions. Under $\mathscr{S}$, 1093 expressions do not overflow, 2268 cannot be repaired, and 43 can be repaired.

Algorithm 1 proves it in a very little time: The *overall* time to analyze, suggest a repair or prove that such repair does not exists is of 120*ms*. We report some of the repairs discovered by the algorithm in Fig. 4.

In order to evaluate Algorithm 2, we consider all the inequalities in the form of $\sum_{i=1}^{5} a_i \cdot \mathtt{x}_i < \sum_{i=1}^{5} a_i \cdot \mathtt{x}_i$ such that:

(i) $a_i \in \{0,1,2,4\}$, for $1 \leq i \leq 5$,

(ii) a given variable $\mathtt{x}_i$ appears at most in one side of the equation,

(iii) each side of the equation owns at least one term with coefficient $a_i \neq 0$.

There exist 7290 of such expressions, and only 213 are not repaired by Algorithm 2. The *overall* time to explore all such expressions and to suggest a repair, if any, is of 202*ms*. Example of repairs are in Figure 4.

To evaluate Algorithm 3, we add to $\mathscr{S}$ the relational information:

$$\mathscr{S}(\mathtt{x}_1 + \mathtt{x}_3) = \mathscr{S}(\mathtt{x}_1 + \mathtt{x}_2) = [-2^{29}, 2^{29}].$$

With this information, we can repair some more expressions than with Algorithm 1 — with a negligible additional cost. Such cases are in Fig. 4.

Overall, in our randomly generated tests, the Algorithms seem to perform extremely well.

Algorithm 1

$$2\cdot x_1+2\cdot x_3+2\cdot x_5+x_6 \longrightarrow 2\cdot x_5+2\cdot x_1+x_6+2\cdot x_3$$
$$2\cdot x_1+2\cdot x_3+x_4+2\cdot x_5 \longrightarrow 2\cdot +x_5+x_4+2\cdot x_1+2\cdot x_3$$
$$4\cdot x_1+2\cdot x_2+x_4+2\cdot x_5 \longrightarrow 2\cdot +x_5+x_4+2\cdot x_2+4\cdot x_1$$
$$4\cdot x_1+2\cdot x_2+x_4+4\cdot x_5+x_6 \longrightarrow 4\cdot x_5+x_4+2\cdot x_2+4\cdot x_1+x_6$$
$$4\cdot x_1+2\cdot x_2+2\cdot x_4+4\cdot x_5 \longrightarrow 4\cdot x_5+2\cdot x_4+2\cdot x_2+4\cdot x_1$$
$$2\cdot x_1+x_2+2\cdot x_3+2\cdot x_4+4\cdot x_5+x_6 \longrightarrow 4\cdot x_5+2\cdot x_4+x_2+2\cdot x_1+x_6+2\cdot x_3$$

Algorithm 2

$$x_4+x_5<4\cdot x_1+4\cdot x_2+4\cdot x_3 \longrightarrow 4\cdot x_3+4\cdot x_1+x_4+x_5<4\cdot x_2$$
$$4\cdot x_3+4\cdot x_5<4\cdot x_1+4\cdot x_2+4\cdot x_4 \longrightarrow 3\cdot x_3+4\cdot x_5<x_3+4\cdot x_1+4\cdot x_2+4\cdot x_4$$
$$2\cdot x_3+4\cdot x_5<4\cdot x_1+2\cdot x_2+4\cdot x_4 \longrightarrow x_1+3\cdot x_4+2\cdot x_3+4\cdot x_5<3\cdot x_1+2\cdot x_2$$
$$x_4<4\cdot x_1+x_2+2\cdot x_3+4\cdot x_5 \longrightarrow 2\cdot x_3+x_2+x_4<4\cdot x_5+4\cdot x_1$$
$$2\cdot x_1+4\cdot x_3+x_4<4\cdot x_2+4\cdot x_5 \longrightarrow 3\cdot x_3<2\cdot x_1+x_4+x_3+4\cdot x_2+4\cdot x_5$$
$$4\cdot x_2+2\cdot x_4+x_5<x_1+4\cdot x_3 \longrightarrow x_3+x_1+4\cdot x_2+2\cdot x_4+x_5<3\cdot x_3$$

Algorithm 3

$$2\cdot x_2+2\cdot x_3+x_6 \longrightarrow x_6+(x_2+x_3)+x_3+x_2$$
$$2\cdot x_1+2\cdot x_3+x_5+x_6 \longrightarrow x_5+x_6+x_3+(x_1+x_3)+x_1$$
$$2\cdot x_1+2\cdot x_2+2\cdot x_3+x_4+x_5 \longrightarrow x_4+x_5+x_2+(x_1+x_3)+x_3+x_1+x_2$$

Figure 4: Some of the repairs produced by the implementation of our algorithms. For Algorithm 3, more repairs are enabled because of a more refined semantic knowledge.

8 Related Work

The dynamic analysis community has been interested for a while to the problem of repairing faulty programs, *i.e.* programs failing some test cases [16]. Their approaches can be abstracted as follows. When a bug is found in a program `P`, a modified program `P'` is generated such that `P'` behaves as `P` on a certain set of tests but it does not manifest the bug. The repaired program `P'` is generated searching a program in the set of program transformations/mutations of `P` –*e.g.*, applying a set of templates [15, 21, 7, 17]. This approach has many drawbacks. First, it requires running the program, first to find the bug in `P` and then to generate and check the candidates for `P'`. This may not be practical because the test suite can be very large – in real software up to several hours. As a consequence the search for `P'` is *de facto* impossible for a large `P`. Second, the whole program may not be available – this is the case for instance when developing libraries. Third, the repair is as good as the test suite itself: *e.g.*, how can we be sure that an arithmetic overflow has been removed for all the possible inputs and not just for the particular test case? Fourth, the generated repairs are often counter-intuitive, and not realistic, in that they perturb the semantics of the original program in the "good" runs [20]. We differentiate from those approaches because our approach is completely static, it exploits the semantic information inferred by the static analyzer, and the repair is guaranteed to only improve the good executions, while removing overflows ("bad" executions).

Orthogonally, a recent paper from Coker and Hafiz [2], introduces a set of refactorings to fix overflows in C programs to be applied in a IDE. However, unlike us, they do not propose any way to auto-

matically generate such fixes.

The bases for our paper are [8] and [10, 11]. In [8] we introduced the concept of verified repair, the idea is that one can generate a program repair from the semantic information inferred by the static analyzer, and verify that it improves the program by removing bad behaviors while increasing good ones, up to some level of abstraction. We proposed some fixes for common bugs, including a greedy incomplete algorithm to fix overflowing expressions. Here we improve over such an algorithm, by first making clear the concept of input and output language, and that of the underlying semantic information used to repair the overflowing expression. Furthermore, in the present paper we present three algorithms for repairing expressions in common cases, and we formally study their properties (complexity and completeness). In [10, 11] we considered the problem of generating (an approximation of) the most precise arithmetic expression over floating-point values [14]. The problem of improving a floating-point expression presents a slightly different challenge than the one considered here in that, in general, the expressions cannot be entirely repaired. Because the floating-point arithmetic introduces roundoff errors, it is in general impossible to find an equivalent expression which computes the exact result, *i.e.*, the result that we would obtain with real numbers. A verified repair then is an expression which is equivalent to the original one when interpreted over the real numbers and which is more accurate than the original expression in the sense that it returns a value closer to the mathematical result when evaluated with floating-point numbers.

9 Conclusions

It is our firm belief that we need to bring static analysis tools to the next level, to make them more practical. They should not limit themselves to find (or prove the absence of) bugs, but they should actively help the programmers by providing suggestions to improve their program, by removing the bug or by proposing better design choices.

In this paper, we considered the particular yet relevant problem of repairing integer expressions starting from the warnings and the invariants inferred by an abstract interpretation-based static analyzer. We characterized the three elements in the problem: (i) the input language for expressions ("which expression should I repair?"); (ii) the output language for the expressions ("how am I allowed to repair an expression?"); and (iii) the semantic information ("what do I know about the values of the expression?"). Then we focused our attention on three common cases, the repairing of: (i) linear expressions with intervals; (ii) Boolean expressions containing linear ones with intervals; and (iii) linear expressions with templates. We showed that in the first case, quite surprisingly, there exists a complete polynomial algorithm to solve the problem, whereas in the second we have a polynomial yet incomplete one, and in the third we have a complete but worst-case exponential one.

References

[1] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux & Xavier Rival (2003): *A static analyzer for large safety-critical software*. In Ron Cytron & Rajiv Gupta, editors: *PLDI*, ACM, pp. 196–207. Available at `http://doi.acm.org/10.1145/781131.781153`.

[2] Zack Coker & Munawar Hafiz (2013): *Program transformations to fix C integers*. In David Notkin, Betty H. C. Cheng & Klaus Pohl, editors: *ICSE*, IEEE / ACM, pp. 792–801. Available at `http://dl.acm.org/citation.cfm?id=2486892`.

[3] Patrick Cousot & Radhia Cousot (1977): *Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints*. In Robert M. Graham, Michael A. Harrison & Ravi Sethi, editors: *POPL*, ACM, pp. 238–252. Available at `http://doi.acm.org/10.1145/512950.512973`.

[4] Manuel Fähndrich, Michael Barnett & Francesco Logozzo (2010): *Embedded contract languages*. In Sung Y. Shin, Sascha Ossowski, Michael Schumacher, Mathew J. Palakal & Chih-Cheng Hung, editors: *SAC*, ACM, pp. 2103–2110. Available at `http://doi.acm.org/10.1145/1774088.1774531`.

[5] Manuel Fähndrich & Francesco Logozzo (2010): *Static Contract Checking with Abstract Interpretation*. In Bernhard Beckert & Claude Marché, editors: *FoVeOOS*, *Lecture Notes in Computer Science* 6528, Springer, pp. 10–30. Available at `http://dx.doi.org/10.1007/978-3-642-18070-5_2`.

[6] Martin Glinz, Gail C. Murphy & Mauro Pezzè, editors (2012): *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*. IEEE. Available at `http://ieeexplore.ieee.org/xpl/mostRecentIssue.jsp?punumber=6218989`.

[7] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest & Westley Weimer (2012): *A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each*. In Glinz et al. [6], pp. 3–13. Available at `http://dx.doi.org/10.1109/ICSE.2012.6227211`.

[8] Francesco Logozzo & Thomas Ball (2012): *Modular and verified automatic program repair*. In Gary T. Leavens & Matthew B. Dwyer, editors: *OOPSLA*, ACM, pp. 133–146. Available at `http://doi.acm.org/10.1145/2384616.2384626`.

[9] Francesco Logozzo & Manuel Fähndrich (2008): *Pentagons: a weakly relational abstract domain for the efficient validation of array accesses*. In Roger L. Wainwright & Hisham Haddad, editors: *SAC*, ACM, pp. 184–188. Available at `http://doi.acm.org/10.1145/1363686.1363736`.

[10] Matthieu Martel (2007): *Semantics-Based Transformation of Arithmetic Expressions*. In Hanne Riis Nielson & Gilberto Filé, editors: *SAS*, *Lecture Notes in Computer Science* 4634, Springer, pp. 298–314. Available at `http://dx.doi.org/10.1007/978-3-540-74061-2_19`.

[11] Matthieu Martel (2009): *Program transformation for numerical precision*. In Germán Puebla & Germán Vidal, editors: *PEPM*, ACM, pp. 101–110. Available at `http://doi.acm.org/10.1145/1480945.1480960`.

[12] Antoine Miné (2006): *The octagon abstract domain*. *Higher-Order and Symbolic Computation* 19(1), pp. 31–100. Available at `http://dx.doi.org/10.1007/s10990-006-8609-1`.

[13] Ramon E. Moore, R. Baker Kearfott & Michael J. Cloud (2009): *Introduction to Interval Analysis*. SIAM, doi:10.1137/1.9780898717716.

[14] Jean-Michel Muller, Nicolas Brisebarre, Florent de Dinechin, Claude-Pierre Jeannerod, Vincent Lefèvre, Guillaume Melquiond, Nathalie Revol, Damien Stehlé & Serge Torres (2010): *Handbook of Floating-Point Arithmetic*. Birkhäuser. Available at `http://dx.doi.org/10.1007/978-0-8176-4705-6`.

[15] Jeff H. Perkins, Sunghun Kim, Samuel Larsen, Saman P. Amarasinghe, Jonathan Bachrach, Michael Carbin, Carlos Pacheco, Frank Sherwood, Stelios Sidiroglou, Greg Sullivan, Weng-Fai Wong, Yoav Zibin, Michael D. Ernst & Martin C. Rinard (2009): *Automatically patching errors in deployed software*. In Jeanna Neefe Matthews & Thomas E. Anderson, editors: *SOSP*, ACM, pp. 87–102. Available at `http://doi.acm.org/10.1145/1629575.1629585`.

[16] Mauro Pezzè, Martin C. Rinard, Westley Weimer & Andreas Zeller (2011): *Self-Repairing Programs (Dagstuhl Seminar 11062)*. *Dagstuhl Reports* 1(2), pp. 16–29. Available at `http://dx.doi.org/10.4230/DagRep.1.2.16`.

[17] Hesam Samimi, Max Schäfer, Shay Artzi, Todd D. Millstein, Frank Tip & Laurie J. Hendren (2012): *Automated repair of HTML generation errors in PHP applications using string constraint solving*. In Glinz et al. [6], pp. 277–287. Available at `http://dx.doi.org/10.1109/ICSE.2012.6227186`.

[18] Sriram Sankaranarayanan, Henny B. Sipma & Zohar Manna (2005): *Scalable Analysis of Linear Systems Using Mathematical Programming*. In Radhia Cousot, editor: *VMCAI*, *Lecture Notes in Computer Science* 3385, Springer, pp. 25–41. Available at `http://dx.doi.org/10.1007/978-3-540-30579-8_2`.

[19] David A. Schmidt (2009): *Abstract Interpretation From a Denotational-semantics Perspective*. *Electr. Notes Theor. Comput. Sci.* 249, pp. 19–37, doi:10.1007/978-3-642-04164-8-14.

[20] Yi Wei, Yu Pei, Carlo A. Furia, Lucas Serpa Silva, Stefan Buchholz, Bertrand Meyer & Andreas Zeller (2010): *Automated fixing of programs with contracts*. In Paolo Tonella & Alessandro Orso, editors: *ISSTA*, ACM, pp. 61–72. Available at `http://doi.acm.org/10.1145/1831708.1831716`.

[21] Westley Weimer, ThanhVu Nguyen, Claire Le Goues & Stephanie Forrest (2009): *Automatically finding patches using genetic programming*. In: *ICSE*, IEEE, pp. 364–374. Available at `http://dx.doi.org/10.1109/ICSE.2009.5070536`.