A Class of Polynomially Solvable Range Constraints for Interval Analysis without Widenings and Narrowings

Zhendong Su[1] and David Wagner[2]

[1] Department of Computer Science, UC Davis, e-mail: su@cs.ucdavis.edu
[2] EECS Department, UC Berkeley, e-mail: daw@cs.berkeley.edu

Abstract. In this paper, we study the problem of solving integer range constraints that arise in many static program analysis problems. In particular, we present the first polynomial time algorithm for a general class of integer range constraints. In contrast with abstract interpretation techniques based on widenings and narrowings, our algorithm computes, in polynomial time, the *optimal solution* of the arising fixpoint equations. Our result implies that "precise" range analysis can be performed in polynomial time without widening and narrowing operations.

1 Introduction

Many program analysis and verification algorithms and tools have the need to solve linear integer constraints or its extensions, such as for checking array bounds to ensure memory safety [15, 16, 35, 38] and for detecting buffer overruns for security applications [36], and for array dependency analysis for parallel compilers [6, 7, 29, 31–33]. However, solving integer linear constraints is a difficult problem [22], and only very special cases have efficient algorithms [3, 30].

In this paper, we study constraints over *integer ranges*, *e.g.*, the set $\{-1, 0, 1\}$, represented as $[-1, 1]$. These constraints can be used to express many interesting program analyses [9, 10, 36]. Furthermore, we show that these constraints can be solved for their optimal solution efficiently. A key property that makes integer range constraints efficiently solvable is its simple join operation in the lattice of ranges. The join of two ranges is defined as $[l, u] \sqcup [l', u'] = [\inf\{l, l'\}, \sup\{u, u'\}]$ (where $\inf$ and $\sup$ compute the minimum and maximum of two numbers) instead of the union of the two ranges. (This does not consider $\perp$, the smallest range. See Section 2 for a complete definition of the join operator.) This use of $\sqcup$ is not as precise as the standard union. However, it is sufficient for many analysis problems [10, 36] that need lower and upper bounds information on values of integer variables.

For readers familiar with interval constraints for floating-point computation [4, 18, 19, 28] based on *interval arithmetic* [25], integer range constraints are different. Such work deals primarily with rounding errors in real numbers, and the goal is to get an approximate real interval that includes all solutions to the original constraints. Range constraints deal with integer ranges, and the goal is to find the least solution, *i.e.*, the smallest ranges that satisfy all the constraints.

Our algorithm is based on a graph formulation similar to that used by Pratt [30] and Shostak [34]. We use fixpoint computations to find the least solution. Our techniques are closely related to those used in integer programming [20], especially those targeted at program analysis and verification. We next survey some closely related work.

Tractable Linear Integer Constraints Pratt [30] considers the simple form of linear constraints $x \leq y + k$, where k is an integer, and gives a polynomial time algorithm based on detecting negative cycles in weighted directed graphs. The graph representation we use in this paper borrows from Pratt's method. Shostak [34] considers a slightly more general problem $ax + by \leq k$, where a, b, and k are integer constants. A worst case exponential time algorithm is given for this kind of constraints by so-called "loop residues." Nelson [26] considers the same fragment and also gives an exponential time algorithm. Aspvall and Shiloach [3] refine Shostak's "loop residue" method and give a polynomial time algorithm for the fragment with two variables. Because constraints with three variables are NP-hard [22], this may be the most general class one can hope for a polynomial time algorithm.

General Linear Integer Constraints General linear integer constraints are also considered in the literature. Some provers use the Fourier-Motzkin variable elimination method [31], the Sup-Inf method of Bledsoe [5], or Nelson's method based on Simplex [27]. However, all the algorithms considered for integer programming have either very high complexity or treat only special cases. In contrast, because of the special structure of the range lattice and properties of affine functions, we are able to design polynomial time algorithms for some common and rather expressive class of range constraints.

Dataflow and Fixpoint Equations Also related is work on dataflow equations in program analysis [21, 23], and lattice constraints in abstract interpretation [10–13], and fixpoint computations in general [1,2,14]. There are some key differences. In this paper, the lattice we consider is an infinite height lattice. For most work in dataflow analysis, the lattices used are of finite height, in which case, termination with exact least solution is guaranteed. For abstract interpretation and general fixpoint computation, although infinite lattices are used in many cases, termination is not guaranteed, and sometimes cannot be guaranteed. Techniques such as widening and narrowing are used to control the termination of the analysis. In this work, we exploit an important property of ranges and affine functions to achieve efficient termination. For example, Cousot and Cousot's interval analysis [10] is quite efficient in practice but may lose precision due to its use of widenings (see the last part of Section 2 for an example); in comparison, our algorithm efficiently finds the *exact* least fixpoint by exploiting the structure of affine constraints, but only applies to a less general class of transfer functions. In fact, the class of constraints we consider resembles the fixpoint equations in [10]. In [36], the authors consider a simpler form of constraints than what is considered in this paper and give a worst case exponential time algorithm.

We summarize here the contributions of the paper: (i) It describes a quadratic time algorithm for solving a general class of affine range constraints (Section 3); (ii) It shows, for the first time, that precise interval analysis can be performed in polynomial time; (iii) It presents hardness and decidability results for satisfiability of some natural extensions of our constraint language (Section 4); and (iv) Our techniques might be useful for solving constraints in other lattices.

2 Preliminaries

Let $\mathbb{Z}$ denote the set of integers. The lattice of ranges is given by:

$$L \stackrel{\text{def}}{=} \{\bot\} \cup \{[l, u] \mid l \in \mathbb{Z} \cup \{-\infty\} \wedge u \in \mathbb{Z} \cup \{+\infty\} \wedge l \leq u\}$$

ordered by $\sqsubseteq$, such that $\bot \sqsubseteq r$ for any $r \in L$ and $[l_1, u_1] \sqsubseteq [l_2, u_2]$ if $l_2 \leq l_1 \leq u_1 \leq u_2$. In the lattice, $\bot$ (the empty range) is the smallest range, and $[-\infty, +\infty]$ is the largest range, also denoted by $\top$. The meet $\sqcap$ and join $\sqcup$ are defined as follows:

- $\bot \sqcap r = \bot \ \wedge \ [l_1, u_1] \sqcap [l_2, u_2] = [l = \sup\{l_1, l_2\}, u = \inf\{u_1, u_2\}]$ ($\bot$ if $l > u$);
- $\bot \sqcup r = r \ \wedge \ [l_1, u_1] \sqcup [l_2, u_2] = [\inf\{l_1, l_2\}, \sup\{u_1, u_2\}]$.

for any range $r \in L$. We select the lower bound and upper bound of a non-empty range $r = [l, u]$ by $\mathsf{lb}(r) = l$ and $\mathsf{ub}(r) = u$.

The range expressions are given by $E ::= r \mid X \mid n \times X \mid E + E$, where $r \in L$ denotes a range constant, X is a range variable, $n \times X$ denotes scalar multiplication by $n \in \mathbb{Z}$, and $E + E$ denotes range addition. A *range constraint* has the form $E \sqcap r \sqsubseteq X$. When $r = \top$, we simply write the constraint as $E \sqsubseteq X$. Notice that we require the right-hand side of a range constraint to be a variable, which is related to "definite set constraints" [17]. We give some examples using these constraints below. Readers interested in more information on the connection between range constraints and program analysis may wish to consult, for example [9, 10].

Let $\mathcal{V}$ denote the set of range variables. A *valuation* ρ is a mapping from $\mathcal{V}$ to L, the lattice of ranges. We extend ρ on variables to work on range expressions inductively, such that, $\rho([l, u]) = [l, u]$, $\rho(n \times X) = n \times \rho(X)$, and $\rho(E_1 + E_2) = \rho(E_1) + \rho(E_2)$, where $n \times [l, u] = [\inf\{nl, nu\}, \sup\{nl, nu\}]$ and $[l_1, u_1] + [l_2, u_2] = [l_1 + l_2, u_1 + u_2]$.

We say a valuation ρ *satisfies* a constraint $E \sqcap r \sqsubseteq X$ if $\rho(E) \sqcap r \sqsubseteq \rho(X)$. A valuation satisfies a set of constraints if it satisfies each one of the constraints. Such a valuation is called a *solution* of the constraints.

Proposition 1 *When $f(X) = aX + b$ denotes an affine function, we have $f([l, u] \sqcap [l', u']) = f([l, u]) \sqcap f([l', u'])$ and $f([l, u] \sqcup [l', u']) = f([l, u]) \sqcup f([l', u'])$.*

Definition 1 (Range Saturation). *A valuation ρ* saturates *a constraint $f(X) \sqcap [c, d] \sqsubseteq Y$ if $[c, d] \sqsubseteq \rho(f(X))$. It* partially saturates *the constraint if $l = c$ or $u = d$, where $[l, u] = \rho(f(X)) \sqcap [c, d]$.*

A set of constraints can have many solutions. For most static program analyses, we are interested in the *least solution*, if it exists, because such a solution gives us the most information. For the range constraints we consider, every set of constraints is satisfiable and has a least solution.

Proposition 2 (Existence of Least Solution) *Any set of range constraints has a least solution.*

Our goal is to compute such a least solution effectively. We denote by least_C the least solution of the constraints C. We use $\mathsf{least}(X)$ for the least solution for a range variable X if the underlying constraints are clear from the context. Next, we give some example constraint systems, which may come from an interval analysis similar to [10] of some small C program fragments. We give examples for both a flow-insensitive analysis and a flow-sensitive analysis. Notice that the interval analysis in [10] is traditionally specified as a flow-sensitive one. A constraint-based formulation sometimes can allow a more natural integration of flow-sensitivity and flow-insensitivity.

Example 1. Consider the constraints $\{[0,0] \sqsubseteq X, X+1 \sqsubseteq X\}$ (with least solution $[0,+\infty]$) from the analysis of the following C program fragment:

```
int i = 0;  /* yields the constraint [0,0] <= X */
while (i < 10) {
   ...
   i = i+1; /* yields the constraint X + 1 <= X */
}
```

Notice that this is a flow-insensitive analysis. [1]

Example 2. Consider the constraints $\{[10,10] \sqsubseteq X, (-2) \times X \sqsubseteq X\}$ (with least solution $[-\infty,+\infty]$), which come from the analysis of the following fragment:

```
int i = 10;    /* yields [10,10] <= X */
while (...) {
   ...
   i = -2*i;   /* yields (-2)*X <= X   */
}
```

Let us go back to Example 1. Notice that although the program ensures $\texttt{i} \leq \texttt{10}$, the range we get says its value can be unbounded. To address this imprecision, we can generate more precise constraints to use non-trivial intersection constants r, in $E \sqcap r \sqsubseteq X$. This use is motivated by the goal to provide more precise analysis of ranges by modeling conditionals in `while` and `if` statements. In Example 1, we expect to say that X has the range $[0,10]$. We can model the example more precisely with the constraints $[0,0] \sqsubseteq X$ and $(X+1) \sqcap [-\infty,10] \sqsubseteq X$. Notice that the least solution of these constraints is indeed $[0,10]$ and is what we expect.

Consider another program fragment:

```
int i = 0;
while (i < n) {
   ...
   i = i+1;
}
```

[1] For comparison, the following constraints may be used for a flow-sensitive analysis:

$$\{[0,0] \sqsubseteq X_0, X_0 \sqsubseteq X_1, X_3 \sqsubseteq X_1, X_1 \sqcap [-\infty,9] \sqsubseteq X_2, X_2+1 \sqsubseteq X_3\}$$

where X_i's denote the variable instances at different program points. See [10] for more.

```
L0
     z = x − y − 3;
L1
     if(z > 0){
L2
          x = z + y + 3;
L3
          y = x − z − 3;
L4
          ...
     }
```

(a) Program after transformation.

$$\begin{array}{ll} \mathtt{L1}: & X_0 - Y_0 - 3 \sqsubseteq Z_1 \\ \mathtt{L2}: & Z_1 \sqcap [1, +\infty] \sqsubseteq Z_2 \\ \mathtt{L3}: & Z_2 + Y_0 + 3 \sqsubseteq X_3 \\ \mathtt{L4}: & X_3 - Z_2 - 3 \sqsubseteq Y_4 \end{array}$$

(b) Constraints.

Fig. 1. An example of how to analyze relationships between variables.

We would want to express the constraints $\{[0,0] \sqsubseteq X, (X+1) \sqcap [-\infty, \mathsf{ub}(Y)] \sqsubseteq X\}$, where X and Y are the range variables for the program variables `i` and `n` respectively, and $\mathsf{ub}(Y)$ denotes the upper bound of Y. The constraint $(X+1) \sqcap [-\infty, \mathsf{ub}(Y)] \sqsubseteq X$ is equivalent to $\{[-\infty, \mathsf{ub}(Y)] \sqsubseteq Z, (X+1) \sqcap Z \sqsubseteq X\}$, where Z is a fresh range variable. In practice, we can restrict the meet operation to be with a range constant in most cases, because the range variables X and Y usually do not belong to the same strongly connected component and can be stratified (see Section 3).

Alternatively, it is sufficient to consider conditions for `while` and `if` statements of the form $x \geq 0$ (or $x > 0$) after some semantics-preserving transformations on the original code. As an example, consider the following program fragment:

```
if (x > y + 3) {
    ...
}
```

We can transform it to the code fragment in Figure 1a, where `z` is a temporary variable for storing intermediate results. We give labels for the few program locations. The generated constraints are given in Figure 1b. In the constraints, we use program location labels on the range variables to distinguish the instances, *i.e.*, the underlying analysis is flow-sensitive. Essentially, we use range constraints to "project" the relevant information from the condition $\mathtt{z} \geq 0$ onto `x` and `y`. It is perhaps interesting to notice that ranges are extremely weak in relating variables.

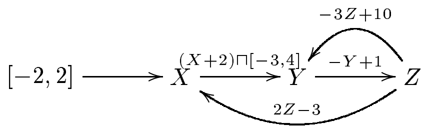
For illustration, we provide here two simple examples to show that the standard widening and narrowing techniques [10] may not give the optimal solution even restricted to affine functions.

Example 3. Consider the following program fragment:

```
int i = 0;
while (...) {
   if (...) { i = 1; }
}
```

$$
\begin{array}{ll}
[-2,2] \sqsubseteq X & (X+2) \sqcap [-3,4] \sqsubseteq Y \\
-Y+1 \sqsubseteq Z & 2Z-3 \sqsubseteq X \\
-3Z+10 \sqsubseteq Y &
\end{array}
$$

(a) Example constraints.



(b) Graph representation.

Fig. 2. Graph representation of constraints.

We obtain the constraints $\{[0,0] \sqsubseteq X, [1,1] \sqsubseteq X\}$. The optimal solution for X is $[0,1]$. However, with widenings and narrowings at program back-edges, we get $[0,+\infty]$. In general, if we have a widening operation at the variable i and if i occurs in two loops of affine constraints with different fixpoints, then widening and narrowing will give an imprecise answer.

Example 4. Consider the constraints $\{[0,0] \sqsubseteq X, (-X+1) \sqsubseteq X\}$. The optimal solution for X is $[0,1]$, however, with widenings and narrowings, we get $[0,+\infty]$.

3 An Algorithm for Solving Range Constraints

Our algorithm is based on *chaotic iteration* [11]. We start by assigning each variable $\perp$, and then iterate through the constraints using the current valuation ρ. For each constraint $E \sqcap r \sqsubseteq X$, if $\rho(E) \sqcap r \not\sqsubseteq \rho(X)$, we set $\rho(X) := (\rho(E) \sqcap r) \sqcup \rho(X)$. This process repeats until all the constraints are satisfied. Although this process always converges for any finite height lattices, it may not converge for infinite height lattices, *e.g.*, consider the constraints $X+1 \sqsubseteq X$ and $[0,0] \sqsubseteq X$. Our approach is to extend chaotic iteration with strategies to handle this kind of cyclic constraints.

We have a natural representation of constraints as graphs. Each vertex in the graph represents a variable (or in some cases, a range constant r), and an edge from X to Y labelled $f(X) \sqcap r$ represents the constraint $f(X) \sqcap r \sqsubseteq Y$. A constraint $[l,u] \sqsubseteq X$ is represented as an edge from a node representing the range constant $[l,u]$ to the node X. Some example constraints and their graph representation are shown in Figure 2.

As mentioned above, our approach is to adapt chaotic iteration to propagate information along edges of the graph until we reach a fixpoint. This fixpoint is the least one. If the graph is acyclic, then we can simply propagate the constraints in its topologically sorted order. In the rest of the section, we consider possibly cyclic graphs. We start with a simple loop (Section 3.2), a multi-loop (Section 3.3), a strongly connected component (Section 3.4), and finally a general graph (Section 3.5).

3.1 Constraint Transformation

Although possible, it is complicated to solve directly constraints with negative coefficients. For a simpler presentation, we first describe a constraint transformation to make all constraints have positive coefficients.

1. Set $\rho(X) := [l, u]$.
2. If $[l', u'] = f(\rho(X)) \sqcap [c, d] \sqsubseteq \rho(X)$, then $\mathsf{least} = \rho$.
3. Otherwise, we have a few cases:
   (a) If $l' < l$ and $u' > u$, then $\mathsf{least}(X) = [c, d]$.
   (b) If $l' < l$ only, then $\mathsf{least}(X) = [c, u]$.
   (c) If $u' > u$ only, then $\mathsf{least}(X) = [l, d]$.

Fig. 3. An algorithm for solving a simple loop.

Lemma 1. *Any system of constraints can be effectively transformed to an equivalent system where all constraints have positive coefficients.*

Proof. For each variable X in the original system, create two variables X^+ and X^-. The variable X^+ corresponds to X, and X^- corresponds to $-X$. We then apply the following transformations on the original constraints:

– Replace each *initial constraint* $[l, u] \sqsubseteq X$ with two constraints:

$$\{[l, u] \sqsubseteq X^+, [-u, -l] \sqsubseteq X^-\}$$

– Replace each constraint of the form $(aX + b) \sqcap [l, u] \sqsubseteq Y$, where $a > 0$, with two constraints:

$$\{(aX^+ + b) \sqcap [l, u] \sqsubseteq Y^+, (aX^- - b) \sqcap [-u, -l] \sqsubseteq Y^-\}$$

– Replace each constraint of the form $(aX + b) \sqcap [l, u] \sqsubseteq Y$, where $a < 0$, with two constraints:

$$\{(-aX^- + b) \sqcap [l, u] \sqsubseteq Y^+, (-aX^+ - b) \sqcap [-u, -l] \sqsubseteq Y^-\}$$

One can verify that the two systems of constraints have the same solutions on the corresponding X and X^+, and in particular, they have the same least solution. In addition, the transformation is linear time and produces a new system of constraints linear in the size of the original system.

Notice that this transformation also applies to affine functions with more than one variables. Hence, in the rest of the paper, we consider only constraints defined over positive affine functions.

3.2 A Simple Loop

Consider a loop with the constraints $[l, u] \sqsubseteq X$ and $f(X) \sqcap [c, d] = (aX + b) \sqcap [c, d] \sqsubseteq X$, where $a > 0$. We give an algorithm in Figure 3 to find its least solution. The algorithm is similar to the widening operator defined on ranges [10].

Lemma 2. *The algorithm in Figure 3 computes the least solution of a simple loop.*

1. Set $\rho(X) = [l, u]$.
2. Pick any constraint $f_i(X) \sqcap [c_i, d_i] \sqsubseteq X$ not satisfied by ρ, find its least solution ρ' with the initial constraint $\rho(X) \sqsubseteq X$ (cf. Figure 3).
3. Set $\rho = \rho'$.
4. Go to step 2 until all constraints are satisfied.

Fig. 4. An algorithm for solving a multi-loop.

Proof. If $[l', u'] \sqsubseteq [l, u]$, then clearly we have reached the least fixpoint, so we have $\mathsf{least}(X) = [l, u]$. Otherwise, we have three cases to consider. (1) If $l' < l$ and $u' > u$, since $f(X) = aX + b$ is a positive affine function, $\mathsf{lb}(f^n([l, u]))$ forms a strictly decreasing sequence and $\mathsf{ub}(f^n([l, u]))$ forms a strictly increasing sequence. However, the lower bound can reach as low as c and the upper bound can reach as high as d. Thus, we have $\mathsf{least}(X) = [c, d]$. The other two cases are similar.

3.3 A Multi-Loop

We call constraints with more than one simple self loop a *multi-loop*. In particular, assume we have the constraints $[l, u] \sqsubseteq X$ and $f_i(X) \sqcap [c_i, d_i] \sqsubseteq X$, for $1 \le i \le n$. A multi-loop is considered because the solution to it hints at the basic idea for solving the more complicated cases. Basically, to solve a multi-loop, we start with X assigned the value $[l, u]$. Pick any constraint $f_i(X) \sqcap [c_i, d_i]$ not satisfied by this valuation. We find its least solution with $[l, u] \sqsubseteq X$ as the initial constraint and update the current assignment to this least solution. This process repeats until all constraints are satisfied. The algorithm is shown in Figure 4.

Lemma 3. *The algorithm in Figure 4 computes the least solution to a multi-loop in quadratic time.*

Proof. It is obvious that the algorithm outputs the least solution when it terminates. Thus, it remains to argue its time complexity. We show that step 2 is executed no more than $2n$ times, *i.e.*, the number of intersection bounds c_i and d_i. Each activation of step 2 causes the current valuation to partially saturate (cf. Definition 1) the particular constraint in question, *i.e.*, at least one lb or ub of the constraint (c_i's or d_i's) is saturated. Because a bound cannot be saturated twice, step 2 is activated at most $2n$ times. Thus, we have shown the algorithm runs in quadratic time in the size of the input constraints.

3.4 A Strongly Connected Component

In this part, we show how to handle a strongly connected component, which forms the core of our algorithm. The main observation is that one can view a strongly connected component as a mutually recursive set of equations working on the set of range variables in the component simultaneously. Let $X_1, \ldots, X_n$ be the set of variables in a component C. We view C as a set of equations working on $X_1, \ldots, X_n$ simultaneously and use the same basic idea for a multi-loop.

Multiple Initial Constraints First, in dealing with a strongly connected component, we need to consider the case where there are multiple initial constraints $[l, u] \sqsubseteq X$ because there may be more than one incoming edges to a component, and each one corresponds to an initial constraint. To simplify our presentation, we apply another graph transformation on a strongly connected component to convert it to an equivalent one with a single initial constraint.

Lemma 4. *In a constraint graph, a strongly connected component with multiple initial constraints can be effectively transformed to an equivalent strongly connected component with a single initial constraint (in linear time and space).*

Proof. Let C be the original component. The transformation works as follows:

- Add a *fresh* range variable X^* with the initial constraint $[1, 1] \sqsubseteq X^*$.
- Replace each initial constraint $[l, u] \sqsubseteq X \in C$, where $l, u \in \mathbb{Z}$, with two constraints $\{lX^* \sqsubseteq X, uX^* \sqsubseteq X\}$.
- Replace each initial constraint $[-\infty, u] \sqsubseteq X \in C$, where $u \in \mathbb{Z}$ with two constraints $\{uX^* \sqsubseteq X, X - 1 \sqsubseteq X\}$.
- Replace each initial constraint $[l, +\infty] \sqsubseteq X \in C$, where $l \in \mathbb{Z}$ with two constraints $\{lX^* \sqsubseteq X, X + 1 \sqsubseteq X\}$.
- Replace each initial constraint $[-\infty, +\infty] \sqsubseteq X \in C$ with three constraints $\{X^* \sqsubseteq X, X + 1 \sqsubseteq X, X - 1 \sqsubseteq X\}$.
- Finally, to make the new graph strongly connected, we add the following constraint from any variable, say Y, to X^*:

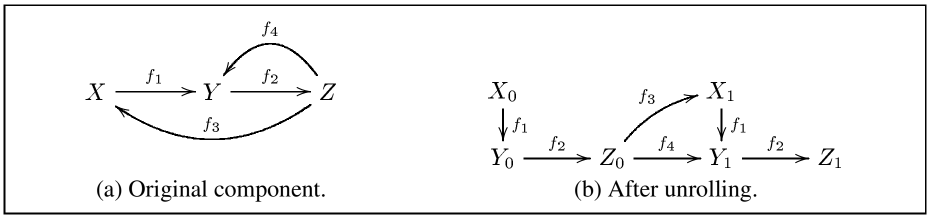
$$Y \sqcap [1, 1] \sqsubseteq X^*$$

One can verify that the new strongly connected component is equivalent to the original component. The running time of the transformation is linear time, and it generates a new constraint system of size linear in $|C|$.

Non-distributivity of Ranges One additional issue is with the non-distributivity of ranges. One can easily verify that $\sqcap$ does not distribute over $\sqcup$, *i.e.*, in general, $(r_1 \sqcup r_2) \sqcap r_3 \neq (r_1 \sqcap r_3) \sqcup (r_2 \sqcap r_3)$. For example, $[2, 2] = ([0, 1] \sqcup [3, 4]) \sqcap [2, 2] \neq ([0, 1] \sqcap [2, 2]) \sqcup ([3, 4] \sqcap [2, 2]) = \perp$. We show, however, this can be remedied to have a slightly altered lemma of distribution of $\sqcap$ over $\sqcup$.

Lemma 5 (Distributivity Lemma). *If $r_1 \sqcap r_3 \neq \perp$ and $r_2 \sqcap r_3 \neq \perp$, then $(r_1 \sqcup r_2) \sqcap r_3 = (r_1 \sqcap r_3) \sqcup (r_2 \sqcap r_3)$.*

Proof. It suffices to show that $(r_1 \sqcup r_2) \sqcap r_3 \sqsubseteq (r_1 \sqcap r_3) \sqcup (r_2 \sqcap r_3)$ because $(r_1 \sqcap r_3) \sqcup (r_2 \sqcap r_3) \sqsubseteq (r_1 \sqcup r_2) \sqcap r_3$. Consider any $a \in (r_1 \sqcup r_2) \sqcap r_3$. We have $a \in (r_1 \sqcup r_2)$ and $a \in r_3$. If $a \in r_1$ or $a \in r_2$, then $a \in (r_1 \sqcap r_3)$ or $a \in (r_2 \sqcap r_3)$. Thus, it follows that $a \in (r_1 \sqcap r_3) \sqcup (r_2 \sqcap r_3)$. Now consider the case where $a \notin r_1$ and $a \notin r_2$. Because $a \in (r_1 \sqcup r_2)$, we have $r_1 \sqcap r_2 = \perp$, and a must lie in the gap of r_1 and r_2. The conditions $r_1 \sqcap r_3 \neq \perp$ and $r_2 \sqcap r_3 \neq \perp$ then guarantees that $a \in (r_1 \sqcap r_3) \sqcup (r_2 \sqcap r_3)$.



(a) Original component. (b) After unrolling.

Fig. 5. An example of graph unrolling.

Lemma 6 (Saturation Lemma). *For any given r_1, r_2, and $r_3 = [a, b]$, either $(r_1 \sqcup r_2) \sqcap r_3 = (r_1 \sqcap r_3) \sqcup (r_2 \sqcap r_3)$ or it holds that $l = a$ or $u = b$, where $[l, u] = (r_1 \sqcup r_2) \sqcap r_3$.*

Proof. If $(r_1 \sqcup r_2) \sqcap r_3 = \bot$, then clearly $(r_1 \sqcup r_2) \sqcap r_3 = (r_1 \sqcap r_3) \sqcup (r_2 \sqcap r_3)$. Otherwise, let $[l, u] = (r_1 \sqcup r_2) \sqcap r_3) \neq \bot$. Assume that $l \neq a$ and $u \neq b$. We must have $a < l \leq u < b$. Then $(r_1 \sqcup r_2) = [l, u]$, which implies $r_1 \sqsubseteq [a, b]$ and $r_2 \sqsubseteq [a, b]$. Thus, $(r_1 \sqcup r_2) \sqcap [a, b] = (r_1 \sqcup r_2) = (r_1 \sqcap [a, b]) \sqcup (r_2 \sqcap [a, b])$.

Graph Unrolling and Constraint Paths A strongly connected component can be viewed in the following sense as a set of functions. We unroll the component starting from X_1 with, for example, a depth first search algorithm. Each time a back-edge is encountered, we create a new instance of the target (if it has not been created). For a variable X_i in a strongly connected component, we use X_{i0} and X_{i1} to denote its first and second instances in its unrolling. We give an example in Figure 5, where Figure 5a shows the original component and Figure 5b is the result after unrolling. Essentially, we are building the depth-first tree (but we also consider the cross edges and back edges). Notice that a depth-first tree with its cross edges is a directed acyclic graph, *i.e.*, a dag. Notice also in the unrolling for a strongly connected component with variables $X_1, \cdots, X_n$, the set of back-edges are exactly those edges between the subgraph induced by $X_{10}, \cdots, X_{n0}$ and the one induced by $X_{11}, \cdots, X_{n1}$.

To solve a strongly connected component, we want to summarize all paths from X_{j0} to X_{j1} by $F_j(X_j) \sqsubseteq X_j$, where

$$F_j(r) \stackrel{\text{def}}{=} r \sqcup \bigsqcup_{X_{j0} \xrightarrow{f_1} \cdots \xrightarrow{f_k} X_{j1}} (f_k \circ \cdots \circ f_1)(r).$$

Note that, even though there may be exponentially many terms in the definition of F_j, nonetheless the output $F_j(r)$ can be computed efficiently for any input r by propagating information in topological sorted order along the edges of the unrolled graph (as done for a dag).

For a strongly connected component, we need to consider a path of constraints. We define formally its semantics.

Definition 2 (Path Constraints). *A path from X_0 to X_n is a path in the constraint graph for C. The function for a path p is the affine function obtained by composing all the functions along the edges on the path. More formally*

1. Transform the component to have a single initial constraint, and let X^* be the new node added by the transformation.
2. Unroll the strongly connected component starting from the node X^*.
3. Compute the least solution ρ for the induced subgraph G_0 with vertices ${X_1}_0, \cdots, {X_n}_0$ and ${X^*}_0$.
4. If ρ satisfies every back-edge ${X_i}_0 \xrightarrow{f \sqcap [c_i, d_i]} {X_j}_1$, then return ρ as the least solution.
5. Otherwise, update ρ through all the back-edges.
   (a) If there is a *new* partially saturated constraint, then go to step 1.
   (b) Otherwise, an unsatisfying cyclic path can be traced to apply the algorithm for a simple loop. Update ρ and go to step 1.

Fig. 6. An algorithm for solving a strongly connected component.

– $\mathsf{pf}(X) = \mathsf{id} \sqcap [-\infty, +\infty]$, *where* id *is the identity function* $\mathsf{id}(X) = X$.
– $\mathsf{pf}(p \xrightarrow{f \sqcap [c,d]} X) = f(\mathsf{pf}(p)) \sqcap [c, d]$.

Notice that for a path $p = X_0 \to \cdots \to X_n$, $\mathsf{pf}(p)$ is of the form $f(X_0) \sqcap [c, d]$, where f is an affine function and $[c, d]$ is a constant range (by Proposition 1).

We apply the same basic idea as that for a multi-loop and Lemma 6 in our algorithm for solving a strongly connected component, which is shown in Figure 6.

Lemma 7. *The algorithm in Figure 6 solves a strongly connected component of the constraint graph in quadratic time in the size of the component.*

Proof. Correctness is again easy to establish since all the constraints are satisfied and every step clearly preserves the least solution. We need to argue the time complexity of the algorithm. The proof technique is similar to that for the algorithm in Figure 4. Again, we argue that the body of the loop executes at most $2n$ times, where n is the number of constraints. Lemma 6 guarantees that if step 5a is not activated, then the previous least solution computation on G_0 distributes. Thus, we can trace, in linear time, an unsatisfying path, which can be converted to a simple loop for its least solution. In that case, at least one bounds must be saturated. Because there are $2n$ bounds, the body of the loops terminates in at most $2n$ steps. Putting everything together, the total running time is quadratic in the size of the component.

3.5 A General Graph

Now it is easy to put everything together to solve an arbitrary set of affine, single-variable range constraints. The idea is to first compute the strongly connected component graph of the original graph representation of the constraints, and then process each component in their topological sorted order. The total running time is quadratic.

Theorem 1. *The least solution for a given system of range constraints can be computed in quadratic time.*

3.6 Affine Functions with More Than One Variables

In this part, we consider affine functions with more than one variable. They are needed for modeling program statements such as `x = y + z + 1` and for precise modeling of loops and conditionals with statements of the above form.

We first consider constraints of the form $a_0 + a_1X_1 + \ldots + a_nX_n \sqsubseteq X$ and then extend the algorithm to the general case, where we allow intersections with constant ranges. First notice it suffices to consider constraints of the form $X + Y \sqsubseteq Z$ along with constraints of the base form, namely, $f(X) \sqsubseteq Y$ and $[l, u] \sqsubseteq X$, where $f(X) = aX + b$ is an affine function over X.

We modify our graph representation of constraints to account for this new type of constraint. The constraint $X + Y \sqsubseteq Z$ can be represented in a hyper-graph setting, with a hyper-edge from X to the node for $+$ and a hyper-edge from Y to $+$. We also have a normal directed edge from $+$ to Z labelled with the identity function. Graphically we have

$$\begin{tikzcd} X \arrow[dr, dashed] & & \\ & + \arrow[r, "\mathrm{id}"] & Z \\ Y \arrow[ur, dashed] & & \end{tikzcd}$$

With this modified graph representation of constraints, we again use the same framework for solving range constraints. The interesting case as before is how to handle a strongly connected component of such a graph. The basic idea for the complete algorithm is the same as before: we compute the strongly connected component graph (using both $\rightarrow$ and $\dashrightarrow$ edges) and process each component in a topological sorted order of the variables nodes.

Here is how we deal with a strongly connected component. The idea is to reduce it to a system of basic constraints (single variable affine functions). Then we apply our algorithm for solving the basic system. We first describe how we reduce a constraint $X+Y \sqsubseteq Z$ to a set of basic constraints. We assume that X and Y have non-empty initial ranges $[l_x, u_x]$ and $[l_y, u_y]$. The constraint is reduced to the following basic constraints

$$\begin{array}{rr} X + [l_y, u_y] \sqsubseteq Z & Y + [l_x, u_x] \sqsubseteq Z \\ [l_x, u_x] \sqsubseteq X & [l_y, u_y] \sqsubseteq Y \end{array}$$

For a strongly connected component in the original graph, we first need to get some initial values for all the variables in the component. This can be easily done by propagating the values in a breath-first fashion starting from the variables with initial ranges. Assume every variable has a non-empty initial value. Otherwise these variables must be the empty range and the constraints can be simplified and solved again. Then for each constraint of the form $X + Y \sqsubseteq Z$, we perform the transformation to basic constraints described above with their current initial ranges. The constraint representation of the obtained constraints is still strongly connected. We then solve for its least solution. We use that to obtain another transformed constraint system. If the current least solution satisfies these new constraints, then we have found the least solution for the original general constraints. If not, we solve for the least solution of the transformed constraints. We repeat this process until the least solution is found. The algorithm is shown in Figure 7.

1. Scan a strongly connected component once to compute an initial valuation ρ that is non-empty on all the variables.
2. Reduce the constraints to basic constraints, solve for the least solution ρ' of the basic constraints subject to $\rho \sqsubseteq \rho'$, and then set $\rho = \rho'$.
3. If all constraints are satisfied, return ρ. Otherwise, go to Step 2.

Fig. 7. An algorithm for solving a strongly connected component with general affine functions.

Theorem 2. *Range constraints over multivariate affine functions are solvable for their least solution in quadratic time.*

Proof. Correctness is easy. We argue that the algorithm for a strongly connected component terminates in linear time. We can simply argue that step 2 is repeated at most three times. Each time step 2 is repeated, it means for one variable, there is unsatisfied self-loop. At least one bound (either lb or ub) reaches $-\infty$ or $+\infty$. With another application of step 2, we must have either reached the least solution, or one variable reaches $[-\infty, +\infty]$, thus the least solution $[-\infty, +\infty]$ for every variable. Because the transformation to basic constraints is quadratic and produces a quadratic size system, the total running time of our algorithm for solving constraints over multivariate affine functions is quadratic.

Finally, we consider constraints with multivariate affine functions and intersections with constant ranges. The constraints are of the general form $f(X_1, \cdots, X_n) \sqcap [c, d] \sqsubseteq X$. We essentially combine the algorithms for multivariate affine functions and intersection constraints to obtain an algorithm for this class of constraints. The interesting case is, as usual, that for a strongly connected component graph. The algorithm, in this case, is exactly the same as the one shown in Figure 7, except the constraints are reduced to basic constraints with intersections. The complexity analysis is based on the same idea as that for basic intersection constraints: with a repeated invocation of Step 2, one lb or ub must be reached. The new system resulted from transformation to basic constraints has a linear number of intersection bounds. Thus we only repeat the loop a linear number of times. However, the size of the new system may be quadratic in the original system. Thus, as the main result of the paper, we obtain a cubic time algorithm for intersection constraints over multivariate affine functions.

Theorem 3 (Main). *The system of constraints $f_i(X_1, \cdots, X_m) \sqcap [c_i, d_i] \sqsubseteq Y_i$, for $1 \le i \le n$, can be solved for their least solution in cubic time.*

4 Decidability and Hardness Results

One might ask whether we can lift the restriction, made earlier, that the right-hand sides be variables. We can thus consider constraints of the form $E_1 \sqsubseteq E_2$, where E_1 and E_2 are range expressions. The interesting question is to ask whether such a system of constraints is satisfiable.

We can show that deciding satisfiability for linear range constraints is NP-hard. The proof is via a reduction from integer linear programming, which is NP-hard [22].

Theorem 4. *The satisfiability problem for general range constraints of the form $E_1 \sqsubseteq E_2$ is NP-hard.*

Proof. We reduce integer linear programming to the satisfiability of range constraints. We simply need to express that a range has to be a singleton, *i.e.*, $[n, n]$ for some integer constant n. This can be easily expressed with the constraint $-Y_i + Y_i = [0, 0]$. One can verify that Y_i is a singleton if and only if this constraint is satisfied.

Let X be an integer linear programming instance. We have m range variables $Y_1, \ldots, Y_m$. For each $(\overline{x}, b) \in X$, we create a range constraint $x_1 Y_1 + \ldots + x_m Y_m \sqsubseteq [b, +\infty]$. We also add constraints of the form $-Y_i + Y_i = [0, 0]$ to ensure that each Y_i is a singleton. It is then straightforward to verify that X has a solution if and only if the constructed range constraints have a solution.

Analogous to Presburger arithmetic, we can consider the first-order theory of range constraints, which we call *Presburger range arithmetic*. By adapting the automata-theoretic proof of the decidability of Presburger arithmetic [8,37], we can easily demonstrate the decidability of Presburger range arithmetic.

Theorem 5. *Presburger range arithmetic is decidable.*

If non-linear range constraints are allowed, the satisfiability problem becomes undecidable via a reduction from Hilbert's 10th Problem [24].

Theorem 6. *The satisfiability problem for non-linear range constraints is undecidable.*

5 Conclusions and Future Work

We have presented the first polynomial time algorithm for finding the optimal solution of constraints for a general class of integer range constraints with applications in program analysis and verification. The algorithm is based on a graph representation of the constraints. Because of the special structure of the range lattice, we are able to guarantee termination with the optimal solution in polynomial time. It is usually difficult to reason about the efficiency and precision of abstract interpretation-based techniques in general because of widenings and narrowings. Through a specialized algorithm, this work shows, for the first time, that "precise" range analysis (w.r.t. the constraints) is achievable in polynomial time. We suspect our techniques for treating non-distributive lattices to be of independent interest and may be adapted to design efficient algorithms for other constraint problems. Future work includes the handling of non-affine functions and floating point computations, and the application of the algorithm to detect buffer overruns and runtime exceptions such as overflows and underflows. It may be also interesting to extend this work to allow symbolic constants in the constraints. Finally, it is interesting to compare the efficiency and precision of an implementation of our algorithm with those algorithms based on widenings and narrowings.

Acknowledgments

We thank the anonymous reviewers of early versions of the paper for their helpful comments.

References

1. K.R. Apt. The essence of constraint propagation. *Theoretical Computer Science*, 221(1–2):179–210, June 1999.
2. K.R. Apt. The role of commutativity in constraint propagation algorithms. *ACM Transactions on Programming Languages and Systems*, 22(6):1002–1036, 2000.
3. B. Aspvall and Y. Shiloach. A polynomial time algorithm for solving systems of linear inequalities with two variables per inequality. *SIAM Journal on Computing*, 9(4):827–845, 1980.
4. F. Benhamou and W.J. Older. Applying interval arithmetic to real, integer, and Boolean constraints. *Journal of Logic Programming*, 32(1):1–24, July 1997.
5. W.W. Bledsoe. The Sup-Inf method in Presburger arithmetic. Technical report, University of Texas Math Dept., December 1974.
6. W. Blume and R. Eigenmann. The range test: A dependence test for symbolic, non-linear expressions. In *Supercomputing '94*. IEEE Computer Society, 1994.
7. W. Blume and R. Eigenmann. Symbolic range propagation. In *Proceedings of the 9th International Symposium on Parallel Processing (IPPS'95*, pages 357–363, Los Alamitos, CA, USA, April 1995. IEEE Computer Society Press.
8. A. Boudet and H. Comon. Diophantine equations, Presburger arithmetic and finite automata. In *Proceedings of Trees in Algebra and Programming (CAAP'96)*, volume 1059 of *Lecture Notes in Computer Science*, pages 30–43. Springer-Verlag, 1996.
9. F. Bourdoncle. Abstract debugging of higher-order imperative languages. In *SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 46–55, 1993.
10. P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *Proceedings of the 2nd International Symposium on Programming*, pages 106–130, 1976.
11. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM Symposium on Principles of Programming Languages*, pages 234–252, 1977.
12. P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, 1992.
13. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th ACM Symposium on Principles of Programming Languages*, pages 84–97, 1978.
14. C. Fecht and H. Seidl. A faster solver for general systems of equations. *Science of Computer Programming*, 35(2–3):137–161, November 1999.
15. R. Gupta. A fresh look at optimizing array bound checking. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 272–282, 1990.
16. R. Gupta. Optimizing array bound checks using flow analysis. *ACM Letters on Programming Languages and Systems*, 1(4):135–150, March-December 1994.
17. N. Heintze and J. Jaffar. A decision procedure for a class of Herbrand set constraints. In *Proceedings of the 5th Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 42–51, June 1990.
18. T.J. Hickey. Analytic constraint solving and interval arithmetic. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POLP-00)*, pages 338–351, N.Y., January 19–21 2000.

19. T.J. Hickey, M.H. van Emden, and H. Wu. A unified framework for interval constraints and interval arithmetic. In *Principles and Practice of Constraint Programming*, pages 250–264, 1998.
20. J.P. Ignizio and T.M. Cavalier. *Introduction to Linear Programming*. Prentice-Hall, 1994.
21. J.B. Kam and J.D. Ullman. Global data flow analysis and iterative algorithms. *Journal of the ACM*, 23(1):158–171, January 1976.
22. R.M. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, New York, 1975.
23. G.A. Kildall. A unified approach to global program optimization. In *Conference Record of the ACM Symposium on Principles of Programming Languages*, pages 194–206. ACM SIGACT and SIGPLAN, 1973.
24. Y.V. Matijasevič. On recursive unsolvability of Hilbert's Tenth Problem. In Patrick Suppes et al., editors, *Logic, Methodology and Philosophy of Science IV*, volume 74 of *Studies in Logic and the Foundations of Mathematics*, pages 89–110, Amsterdam, 1973. North-Holland.
25. R.E. Moore. *Interval Analysis*. Prentice-Hall, Englewood Cliffs, NJ, New York, 1963.
26. G. Nelson. An $n^{\log n}$ algorithm for the two-variable-per-constraint linear programming satisfiability problem. Technical Report STAN-CS-78-689, Stanford University, 1978.
27. G. Nelson. Techniques for program verification. Technical Report CSL-81-10, Xerox Palo Alto Research Center, 1981.
28. W.J. Older and A. Velino. Constraint arithmetic on real intervals. In Frédéric Benhamou and Alain Colmerauer, editors, *Constraint Logic Programming: Selected Research*, pages 175–196. MIT Press, 1993.
29. Y. Paek, J. Hoeflinger, and D.A. Padua. Efficient and precise array access analysis. *ACM Transactions on Programming Languages and Systems*, 24(1):65–109, 2002.
30. V.R. Pratt. Two easy theories whose combination is hard. Unpublished manuscript, 1977.
31. W. Pugh. The omega test: a fast and practical integer programming algorithm for dependence analysis. *Communications of the ACM*, 35(8):102–114, August 1992.
32. W. Pugh. Constraint-based array dependence analysis. *ACM Transactions on Programming Languages and Systems*, 20(3):635–678, May 1998.
33. R. Seater and D. Wonnacott. Polynomial time array dataflow analysis. In *Languages and Compilers for Parallel Computing (LCPC)*, 2001.
34. R. Shostak. Deciding linear inequalities by computing loop residues. *Journal of the ACM*, 28(4):769–779, October 1981.
35. N. Suzuki and K. Ishihata. Implementation of an array bound checker. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages*, pages 132–143, Los Angeles, California, January 17–19, 1977. ACM SIGACT-SIGPLAN.
36. D. Wagner, J.S. Foster, E.A. Brewer, and A. Aiken. A first step towards automated detection of buffer overrun vulnerabilities. In *Symposium on Network and Distributed Systems Security (NDSS '00)*, pages 3–17, San Diego, CA, February 2000. Internet Society.
37. P. Wolper and B. Boigelot. An automata-theoretic approach to presburger arithmetic constraints. In *Static Analysis Symposium*, pages 21–32, 1995.
38. H. Xi and F. Pfenning. Eliminating array bound checking through dependent types. In *Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation (PLDI)*, pages 249–257, Montreal, Canada, 17–19 June 1998.