

# A Numerical Abstract Domain based on *Expression Abstraction* and *Max Operator* with Application in Timing Analysis

Bhargav S. Gulavani<sup>1</sup> and Sumit Gulwani<sup>2</sup>

<sup>1</sup> Indian Institute of Technology, Bombay, India [bhargav@cse.iitb.ac.in](mailto:bhargav@cse.iitb.ac.in)

<sup>2</sup> Microsoft Research, Redmond, US [sumitg@microsoft.com](mailto:sumitg@microsoft.com)

**Abstract.** This paper describes a precise numerical abstract domain for use in timing analysis. The numerical abstract domain is parameterized by a linear abstract domain and is constructed by means of two domain lifting operations. One domain lifting operation is based on the principle of *expression abstraction* (which involves defining a set of expressions and specifying their semantics using a collection of directed inference rules) and has a more general applicability. It lifts any given abstract domain to include reasoning about a given set of expressions whose semantics is abstracted using a set of axioms. The other domain lifting operation incorporates disjunctive reasoning into a given linear relational abstract domain via introduction of `max` expressions. We present experimental results demonstrating the potential of the new numerical abstract domain to discover a wide variety of timing bounds (including polynomial, disjunctive, logarithmic, exponential, etc.) for small C programs.

## 1 Introduction

The oldest trick for proving termination of loops has been that of finding a ranking function [25]. A ranking function for a loop is a function (over loop variables) whose value decreases in each loop iteration and is bounded below by some finite quantity. Earlier work on proving termination of loops focused on synthesizing *linear* ranking functions [11, 22]. However, not all programs have linear ranking functions (e.g., Figure 1(a) and 2(a)). This led to more sophisticated proposals for proving termination like the principle of disjunctive well-foundedness of ranking functions (which can handle Figure 1(a)), and work on a richer class of ranking functions like lexicographic linear ranking functions [7] and polyranking functions [8] (which can handle Figure 2(a)).

In contrast to this recent literature on multiple methodologies for proving termination, we present a numerical abstract domain that can be used to uniformly prove the termination of a large class of programs (including the ones in Figure 1 and Figure 2), and more importantly establish precise timing bounds, a richer piece of information than simply establishing termination. Computing timing bounds is much more useful than simply proving termination in several settings such as embedded systems and performance critical software. The basic

<pre> Disjunction(int x<sub>0</sub>, y, z<sub>0</sub>)   x := x<sub>0</sub>; z := z<sub>0</sub>; i := 0;   while (x &lt; y) do     i := i + 1;     if (z &gt; x) {       x := x + 1;     }     else       z := z + 1;   (a) </pre>	<pre> Sequential(int n, m)   x := 0; i := 0;   while (x &lt; n) do     i := i + 1;     x := x + 1;   while (x &lt; m) do     i := i + 1;     x := x + 1;   (b) </pre>	<pre> Simple(int x<sub>0</sub>, n)   x := x<sub>0</sub>; i := 0;   while (x &lt; n) do     i := i + 1;     x := x + 1;   (c) </pre>
--	---	---

**Fig. 1.** Examples that illustrate the importance of the max operator in our numerical abstract domain for both representing timing bounds as well as computing the invariants required to establish timing bounds. The examples have been instrumented with the monitor variable  $i$ . For (a) (taken from [12] which uses disjunctive well-foundedness to prove termination), our tool computes the bound  $\max(0, y - x_0) + \max(0, y - z_0)$  on the monitor variable  $i$  after establishing the inductive loop invariant  $i = (x - x_0) + (z - z_0) \wedge z \leq \max(z_0, y) \wedge x \leq \max(x_0, y)$ . For (b), and (c) our tool computes the bounds  $\max(0, n, m)$  and  $\max(0, n - x_0)$  respectively on the monitor variable  $i$ .

idea of our methodology is to instrument the program with a monitor variable that increases in each loop iteration and then establish bounds on the monitor variable using abstract interpretation over a numerical abstract domain. Essentially, we have instrumented a candidate ranking function  $-i$  inside the loop since the value of  $-i$  always decreases and establishing an upper bound  $u$  on  $i$  will imply a lower bound on  $-i$ , thereby making  $-i$  a ranking function and implying termination, but more importantly yielding a timing bound of  $u$ .

One key feature of our numerical abstract domain is that it incorporates (some level of) disjunctive reasoning by use of max operator (which returns the largest of its arguments). This allows our abstract domain to naturally express bounds for loops with complex control flow inside, as is the case for the program in Figure 1(a). (Observe that even proving termination of this program is non-trivial; one way to prove termination of this program is to use the principle of disjunctive well-foundedness [23], which involves splitting the termination argument into multiple ranking functions corresponding to different branches in the program.) However, it is important to note that the presence of max operator in our abstract domain not only caters to handling loops with complex control structure, but it is equally important to express bounds for programs even with simple loops (which entail linear ranking functions, a simpler termination argument), like the ones in Figure 1(b) and (c).

Another key feature of our numerical abstract domain is that it incorporates reasoning about other operators (like multiplication, exponentiation, logarithm, square-root etc) whose semantics is specified using some set of inference rules. This allows our abstract domain to naturally represent bounds for loops with inherent non-linear behavior as is the case for the program in Figure 2(a). (Observe that even proving termination of this program is non-trivial; one way to prove termination is to use lexicographic polyranking functions [7, 8].) However,

<pre> NonLinear(int y0, n)   assume(n &gt; 0);   x := 0; y := y0;   i := 0;   while (x &lt; n) do     i := i + 1;     y := y + 1;     x := x + y;     (a) </pre>	<pre> ModularMultiply(int n)   i := 0;   assume(n &gt; 0);   for j = 1 to n     for k = 1 to n       i := i + 1;       ...     (b) </pre>	<pre> ModularSquare(int n)   i := 0;   assume(n &gt; 0);   for j = 1 to n     for k = 1 to j       i := i + 1;       ...     (c) </pre>
--	---	---

**Fig. 2.** Examples that illustrate the importance of using non-linear operators like multiplication and square root in our numerical abstract domain for both representing timing bounds as well as computing the invariants required to establish timing bounds. The examples have been instrumented with the monitor variable  $i$ . For (a) (taken from [6] which uses the principle of second-order differences to establish a lexicographic polyranking function for proving termination), our tool computes the bound  $\sqrt{2n} + \max(0, -2y_0) + 1$  on the monitor variable  $i$  after establishing the inductive loop invariant  $i = y - y_0 \wedge y^2 \leq y_0^2 + 2x$ . For (b) and (c), our tool computes the bound  $n^2$  and  $n(n + 1)/2$  respectively on the monitor variable  $i$ .

it is important to note that the flexibility of using arbitrary expressions (from a given set) allows our abstract domain to precisely represent precise bounds of programs with simple termination arguments. For example, we can prove that mergesort has a complexity of  $n \log n$  and that Fibonacci has a complexity of  $2^n$ . It is important to note that our technique not only aims to find precise computational complexity, but also precise constant factors. For example, it can precisely establish the  $n^2$  complexity of the doubly nested loop used for modular multiplication and  $n(n + 1)/2$  complexity of the doubly nested loop used for modular squaring as shown in Figure 2(b) and (c). This difference was the source of a real timing attack on implementations of RSA protocol [19].

Our numerical abstract domain is parameterized by a linear arithmetic numerical domain like intervals, difference constraints, or polyhedron domain [13]. We present two domain lifting operations that extend the base linear arithmetic domain to reason about the max operator and other operators whose semantics is specified using a set of inference rules. One of the domain lifting operation extends the linear arithmetic domain to represent linear relationships over variables as well as max-expressions (an expression of the form  $\max(e_1, \dots, e_n)$  where  $e_i$ 's are linear expressions). Another domain lifting operation lifts an abstract domain to represent constraints not only over program variables, but also over expressions from a given finite set of expressions  $S$ . The semantics of the operators used in constructing expressions in  $S$  is specified as a set of inference rules. (Our abstract domain retains efficiency by treating these expressions just like any other variable, while relying on the inference rules to achieve precision.) The rationale behind this choice is that it is easy to specify or heuristically infer the set of base expressions, but specifying linear combinations of those expressions and specifying which subsets of those linear combinations should be grouped under a max operator is a cumbersome process. Fortunately, the latter process

can be automated by means of a domain lifting operation that we describe in this paper.

This paper has three main technical contributions:

- We introduce a domain constructor operation based on the notion of expression abstraction. Given a base abstract domain  $A$  and a set of expressions  $S$  whose semantics is specified using a set of rewrite rules, we show how to construct a more precise abstract domain  $A_S$  (Section 3).
- We introduce another domain constructor operation for linear arithmetic domains. Given a linear arithmetic abstract domain  $A$ , we show how to construct a new arithmetic domain  $\tilde{A}$  that can represent linear relations over *max* expressions (Section 4).
- Given a linear arithmetic domain  $A$ , and a set of expressions  $S$ , we use our domain constructor operations to construct the numerical domain  $\tilde{A}_S$  (Section 5), and show how it can be used to compute precise bounds for a wide variety of programs (Section 6). We discuss preliminary experimental results in Section 7.

We start with a description of the operations that need to be supported by an abstract domain for performing abstract interpretation in Section 2.

## 2 Preliminaries

An abstract domain  $A$  needs to be equipped with four operators (or transfer functions),  $\text{Join}_A$ ,  $\text{Widen}_A$ ,  $\text{Eliminate}_A$ , and  $\text{PostPredicate}_A$  to enable (forward) abstract interpretation over the flowchart nodes of a program.

The join operator  $\text{Join}_A$  for an abstract domain  $A$  takes two abstract elements  $E_1$  and  $E_2$  and computes the least upper bound of  $E_1$  and  $E_2$  in the abstract domain  $A$ . In other words  $\text{Join}_A(E_1, E_2)$  denotes the most precise element  $E$  in the abstract domain  $A$  such that  $E_1 \Rightarrow E$  and  $E_2 \Rightarrow E$ . The join operator is used to obtain the abstract element after a join node by merging the abstract elements before the join node.

The widen operator  $\text{Widen}_A$  for an abstract domain  $A$  takes two abstract elements  $E_1$  and  $E_2$  such that  $E_1 \Rightarrow E_2$  and computes another element  $E$  such that  $E_2 \Rightarrow E$ . The sequence of widen operations converges in a bounded number of steps, i.e., for any strictly increasing sequence  $E_0, E_1, \dots$  (such that  $E_i \Rightarrow E_{i+1}$  for all  $i$ ), if we define  $E'_0 = E_0$ ,  $E'_1 = \text{Widen}_A(E'_0, E_1)$ ,  $E'_2 = \text{Widen}_A(E'_1, E_2), \dots$ , then there exists  $i \geq 0$  such that  $E'_j = E'_i$  for all  $j > i$ .

The postpredicate operator  $\text{PostPredicate}_A$  takes as input an abstract element  $E'$  and a predicate  $p$  and computes the most precise element  $E$  expressible in the domain  $A$  such that  $E' \wedge p \Rightarrow E$  (meaning that  $\gamma(E') \cap S_p \subseteq \gamma(E)$ , where  $S_p$  is the set of states that satisfy predicate  $p$ ). The postpredicate operator is used to incorporate the information provided by the predicate inside an assume statement or a conditional guard.

The existential elimination operator  $\text{Eliminate}_A$  takes as input an abstract element  $E'$  and a variable  $x$  and computes the most precise element  $E$  expressible

in the domain  $A$  such that  $E' \Rightarrow E$  and  $E$  does not refer to variable  $x$ . The existential elimination operator is used to transform the abstract element  $E'$  before an assignment statement  $x := e$ <sup>3</sup> to the element  $E$  as follows (assuming that  $x$  does not occur in  $e$ <sup>4</sup>):  $E = \text{PostPredicate}_A(\text{Eliminate}_A(E', x), x = e)$ .

In Section 3 and Section 4 below, we show how to construct the transfer functions for the richer abstract domains  $A_S$  and  $\tilde{A}$  from the transfer functions of the base abstract domain  $A$ .

### 3 Domain Lifting using Expression Abstraction

In this section, we introduce the notion of *expression abstraction* and use it to define a more precise abstract domain given any base abstract domain.

The process of *expression abstraction* involves defining a set of expressions  $S$  over program variables using some operators, and defining the abstract semantics of those operators using a set of directed inference rules  $R$ . We additionally assume that the set  $S$  is closed under sub-expressions. (See Section 5 for an example of  $S$  and  $R$ .) For every expression  $e \in S$ , we introduce a fresh variable denoted by  $Z_e$ . The elements of the abstract domain  $A_S$  represent the same kind of constraints as the abstract domain  $A$  but over an extended set of variables that includes  $Z_e$ 's. The transfer functions for the abstract domain  $A_S$  (defined in Section 3.2) make use of the **Saturate** operator, which we define next.

#### 3.1 The Saturate Operator

The saturate operator **Saturate** takes as input an abstract element  $E$  and a set of expressions  $S$  and returns another abstract element  $E'$  that contains constraints from  $E$  as well as includes constraints over expressions from  $S$  obtained by applying the rewrite rules that define the semantics of the expressions in  $S$ . The pseudo-code for the saturate function is shown below.

```

Saturate( $E, S$ ) =
1   $E_{old} := \perp$ ;
2  while ( $E \neq E_{old}$ ) do
3     $E_{old} := E$ ;
4    foreach instantiation of an inference rule:  $P_1 \Rightarrow P_2$ 
5      If  $E \Rightarrow P_1$ , then  $E := \text{PostPredicate}(E, P_2)$ ;
6  return  $E$ ;

```

The for loop in Line 4 considers all instantiations of an inference rule  $P_1 \Rightarrow P_2$  such that all terms that occur in both  $P_1$  and  $P_2$  are from the given set of expressions  $S$ . If the set of inference rules  $R$  has the property that the number

<sup>3</sup> Without loss of generality, we can assume that all assignment statements are of the form  $x := e$ . Memory reads and writes can be modeled using select and update expressions, without losing any precision.

<sup>4</sup> Without loss of generality, we can assume that  $x$  does not occur inside  $e$  since an arbitrary assignment  $x := e$  can be split into two assignments  $t := e; x := t$  with this property, where  $t$  is a fresh variable.

of applications of the inference rules is bounded in any context (i.e., given any context, the number of applications of the inference rules that yield a predicate not implied by the context and other derived predicates is bounded), then the while loop in Line 2 is terminating. If not, then we simply use the heuristic of iterating a bounded number of times.

*Example 1.* Let  $A$  be the polyhedron abstract domain. Let  $R$  consist of the following useful inference rule for reasoning about the product operator. The rule multiplies both sides of an equality by some term  $u$ .

$$\left(\sum_i a_i x_i = a\right) \Rightarrow \left(\sum_i a_i Z_{x_i u} = Z_{au}\right)$$

Let  $S$  be the set of expressions  $\{y^2, y_0^2, yy_0\}$ . Let  $E_1$  be  $y = y_0 \wedge x = 0$  and  $E_2$  be  $y = y_0 + 1 \wedge x = y_0 + 1$ .  $E_1$  and  $E_2$  denote (part of) the abstract elements at the loop entry and at the loop back-edge after one loop iteration for the example in Figure 2(a). Then,

$$\begin{aligned} \text{Saturate}(E_1, S) &= (y = y_0 \wedge x = 0 \wedge Z_{y^2} = Z_{yy_0} \wedge Z_{yy_0} = Z_{y_0^2}) \\ \text{Saturate}(E_2, S) &= (y = y_0 + 1 \wedge x = y_0 + 1 \wedge Z_{yy_0} = Z_{y_0^2} + y_0 \wedge Z_{y^2} = Z_{yy_0} + y) \end{aligned}$$

### 3.2 Transfer Functions

In this section, we describe how to construct the transfer functions for the abstract domain  $A_S$  using the transfer functions for the base domain  $A$ . The key idea in the construction is to simply saturate the input abstract elements using the **Saturate** algorithm (described in Section 3.1) and then apply the corresponding transfer function from the base abstract domain.

- Join Operator.

$$\text{Join}_{A_S}(E_1, E_2) = \text{Join}_A(\text{Saturate}(E_1, S), \text{Saturate}(E_2, S))$$

*Example 2.* Let  $E_1$  and  $E_2$  be the abstract elements as in Example 1. Then,

$$\text{Join}_{A_S}(E_1, E_2) = (y_0 \leq y \leq y_0 + 1 \wedge Z_{y^2} \leq Z_{y_0^2} + 2x)$$

Observe that the above join operation gives us one of the desired inductive invariants  $y^2 \leq y_0^2 + 2x$  required for proving bounds in Figure 2(a).  $\square$

- PostPredicate Operator.

$$\text{PostPredicate}_{A_S}(E, p) = \text{PostPredicate}_A(\text{Saturate}(E, S), p)$$

- Eliminate Operator.

The eliminate operator for the new domain  $A_S$  involves saturating the input abstract element and then eliminating not only the given variable  $x$ , but also all the variables corresponding to expressions from  $S$  that involve  $x$ .

$$\text{Eliminate}_{A_S}(E, x) = \text{Eliminate}_A(\text{Saturate}(E, S), V_x)$$

where  $V_x = \{x\} \cup \{Z_e \mid e \in S \text{ and change to } x \text{ results in change to } e\}$ .

- Widen Operator.

$$\text{Widen}_{A_S}(E_1, E_2) = \text{Widen}_A(\text{Saturate}(E_1, S), \text{Saturate}(E_2, S))$$

## 4 Linear Domain Lifting using Max Operator

In this section, we define a domain lifting operation that takes a linear arithmetic abstract domain  $A$  that represents linear constraints over some set of variables  $V$ , and a subset  $U$  of  $V$ , and produces a domain  $\tilde{A}$  that can represent linear constraints over  $V - U$  as allowed by  $A$ , but allowing for a richer constant term - one that is constructed using linear combinations of max-linear expressions over  $U$ . A *max-linear expression* over  $U$  is of the form  $\mathbf{max}(e_1, \dots, e_n)$ , where each  $e_i$  is some linear expression over  $U$ . For eg., if  $A$  is the difference constraints domain, then the domain  $\tilde{A}$  can represent constraints like  $v_1 - v_2 \leq \mathbf{max}(u_1, 2u_2 + u_3)$ , where  $v_1, v_2 \in V$  and  $u_1, u_2, u_3 \in U$ .

The transfer functions for the abstract domain  $\tilde{A}$  (defined in Section 4.2) make use of the **Witness** operator, which we define next.

### 4.1 Witness Coefficients

**Lemma 1.** (*Farkas Lemma*) *Let  $e, e_i$  be some linear arithmetic expressions without the constant term. If  $\left(\bigwedge_{i=1}^n (e_i \leq 0)\right) \Rightarrow e \leq 0$ , then it must be the case that there exist non-negative  $\lambda$ 's such that  $e \equiv \sum_{i=1}^n \lambda_i e_i$ .*

For an implication  $\left(\bigwedge_{i=1}^n (e_i \leq 0)\right) \Rightarrow e \leq 0$ , we define  $\mathbf{Witness}\left(\bigwedge_{i=1}^n (e_i \leq 0), e \leq 0\right)$  to be  $(\lambda_1, \dots, \lambda_n)$ . Note that there may exist multiple witnesses but any single witness is sufficient for soundness of the transfer functions described below.

*Example 3.*

$$\begin{aligned} \mathbf{Witness}(i \leq 0 \wedge -x \leq 0, i \leq 0) &= (1, 0) \\ \mathbf{Witness}(i - x \leq 0 \wedge x \leq 0, i \leq 0) &= (1, 1) \end{aligned}$$

### 4.2 Transfer Functions

In this section, we describe how to construct the transfer functions for the abstract domain  $\tilde{A}$  using the transfer functions for the base domain  $A$ . The key idea in the construction is to remove the constant and the part corresponding to variables in  $U$  from each inequality in the input(s), and then apply the corresponding transfer function from the base domain, and then add back an appropriate symbolic max expression to each inequality in the result.

– Join Operator.

```

Join $\bar{A}$ ( $E_1, E_2$ ) =
1 Let  $E_1$  be  $\bigwedge_i (e_i \leq f_i)$  and let  $E_2$  be  $\bigwedge_i (e'_i \leq f'_i)$ ;
   (where  $e_i, e'_i$  are linear over  $V-U$  and  $f_i, f'_i$  are max-linear over  $U$ )
2  $E := \top$ ;  $E' := \text{Join}_A(\bigwedge_i (e_i \leq 0), \bigwedge_i (e'_i \leq 0))$ ;
3 Foreach inequality  $e \leq 0 \in E'$ ,
4    $(\lambda_i) := \text{Witness}(\bigwedge_i (e_i \leq 0), e \leq 0)$ ;  $(\lambda'_i) := \text{Witness}(\bigwedge_i (e'_i \leq 0), e \leq 0)$ ;
5    $f := \sum_i \lambda_i f_i$ ;  $f' := \sum_i \lambda'_i f'_i$ ;
6    $f'' := \text{ComputeMax}(f, f')$ ;
7   if  $f'' \neq \top$ ,  $E := E \wedge (e \leq f'')$ ;
8 return  $E$ ;

```

The function  $\text{ComputeMax}(f, f')$  either returns  $\top$  denoting that there are too many arguments to the max function, or returns a possibly max function.

*Example 4.* Let  $V = \{i, x, x_0, n\}$  and  $U = \{x_0, n\}$ . Let  $E_1$  be  $i \leq 0 \wedge -x \leq -x_0$  and  $E_2$  be  $i - x \leq -x_0 \wedge x \leq n$ .  $E_1$  and  $E_2$  denote (part of) the abstract elements at the loop entry and at the loop back-edge after one loop iteration for the example in Figure 1(c.) Then, using the result of the witness functions from Example 3, we obtain

$$\text{Join}_{\bar{A}}(E_1, E_2) = i \leq \max(0, n - x_0)$$

Observe that the above join operation provides us with the invariant  $i \leq \max(0, n - x_0)$  required for computing bounds in Figure 1(a).  $\square$

– PostPredicate Operator.

$$\text{PostPredicate}_{\bar{A}}(E, p) = \text{PostPredicate}_A(E, p)$$

– Eliminate Operator.

```

Eliminate $\bar{A}$ ( $E, x$ ) =
1 Let  $E$  be  $\bigwedge_i (e_i \leq f_i)$ ;
   (where  $e_i$  are linear over  $V-U$  and  $f_i$  are max-linear over  $U$ )
2  $E_1 := \text{Eliminate}_A(\bigwedge_i (e_i \leq 0))$ ;  $E_2 := \top$ ;
3 Foreach inequality  $e \leq 0 \in E_1$ :
4    $(\lambda_i) := \text{Witness}(\bigwedge_i (e_i \leq 0), e \leq 0)$ ;
5    $f := \sum_i \lambda_i f_i$ ;
6    $E_2 := E_2 \wedge (e \leq f)$ ;
7 return  $E_2$ ;

```

Note that we require the variables to be eliminated be from the set  $V - U$ . This puts the restriction that the variables in set  $U$  are never modified in the program.

*Example 5.* Let  $V = \{i, x, x_0, n\}$  and  $U = \{x_0, n\}$ . Let  $E$  be  $i - x \leq -x_0 \wedge x \leq n$ . Consider computing  $\text{Eliminate}_{\bar{A}}(E, x)$ . Line 2 of the above algorithm computes  $E_1$  as  $i \leq 0$ . Using the witness  $(1, 1)$ , we obtain the result  $i \leq -x_0 + n$ .

– Implication Operator.

```

Implies $\bar{A}(E_1, E_2) =$ 
1  Let  $\bar{E}_1$  be  $\bigwedge_i e_i \leq f_i$  and let  $E_2$  be  $\bigwedge_j e'_j \leq f'_j$ ;
    (where  $e_i, e'_j$  are linear over  $V-U$  and  $f_i, f'_j$  are max-linear over  $U$ )
2  Let result := true;
3  Foreach inequality  $e'_j \leq f'_j \in E_2$ :
4     $(\lambda_i) := \text{Witness}(\bigwedge_i e_i \leq 0, e'_j \leq 0)$ ;  $f := \sum_i \lambda_i f_i$ ;
5    if not LessEq( $f, f'_j$ ), result := false;
6  return result;

```

Let  $f = \max(e_1, \dots, e_n)$  and  $f' = \max(e'_1, \dots, e'_n)$  be max-linear expressions. **LessEq**( $f, f'$ ) returns true iff for each  $e_i$  there exists  $e'_j$  such that  $e_i \leq e'_j$  is valid. For example **LessEq**( $\max(x, n), \max(x+1, n)$ ) returns true, whereas **LessEq**( $\max(x, n), \max(x+1, n-1)$ ) returns false.

– Widen Operator.

The widen operator **Widen** $\bar{A}(E_1, E_2)$  simply returns the conjunction of those constraints from  $E_1$  that are also implied by  $E_2$ .

## 5 A New Numerical Abstract Domain

In this section, we discuss the design choices made while applying the domain lifting operators for obtaining the numerical abstract domain that we use for timing analysis. We pick any linear relational abstract domain  $A$  and lift it using the domain lifting operation based on expression abstraction (as described in Section 3). We use the operators multiplication ( $x \times y$ ), logarithm ( $\lceil \log x \rceil$ ), square-root ( $\lceil \sqrt{x} \rceil$ ), and exponentiation ( $2^x$ ) to construct the set  $S$  of expressions. The set  $S$  of expressions involving these operators can either be provided by the programmer<sup>5</sup> (since they may have a better idea of what kinds of bounds the program may entail) or it can be constructed automatically using some initial heuristic such as we can apply the unary operators (logarithm, square-root, exponentiation) to all program variables once and then apply the only binary operator (multiplication) to all pairs of resulting expressions. This will allow our abstract domain to represent linear relationships over expressions like  $n \times m$ ,  $n \lceil \log m \rceil$ , etc. We use the following inference rules  $R$  to reason about these operators. (These specific rules were chosen because they appear to capture the reasoning required to compute bounds over the chosen set of non-linear operators for a large class of programs.) For simplicity, we overload the notation  $Z_x$  to denote  $x$ , if  $x$  is a program variable. (Recall that the notation  $Z_x$  normally denotes the special variable associated with an expression  $x \in S$ ).

<sup>5</sup> Note that we are not requiring the programmer to provide the exact bound. We are simply requiring the programmer to provide the base expressions and the bounds would be automatically computed by taking linear combinations of these expressions and additionally the expressions obtained by applying max operator. Furthermore, computation of bounds requires establishing the inductive invariants, which are usually much harder than the bound itself.

1.  $(Z_x \leq c) \Rightarrow (Z_{2^x} \leq 2^c)$
2.  $(Z_x \leq Z_y + c) \Rightarrow (Z_{2^x} \leq 2^c \times Z_{2^y})$
3.  $(Z_x \leq c) \wedge (Z_x > 0) \Rightarrow (Z_{\lceil \log x \rceil} \leq \lceil \log c \rceil)$
4.  $(Z_x \leq c \times Z_y) \wedge (Z_x > 0) \wedge (Z_y > 0) \Rightarrow (Z_{\lceil \log x \rceil} \leq \lceil \log c \rceil + Z_{\lceil \log y \rceil})$
5.  $(\sum_i a_i Z_{x_i} \geq a) \wedge (Z_y \geq 0) \Rightarrow (\sum_i a_i Z_{x_i y} \geq ay)$
6.  $(\sum_i a_i Z_{x_i} = a) \Rightarrow (\sum_i a_i Z_{x_i y} = aZ_y)$
7.  $\text{true} \Rightarrow Z_{\sqrt{x^2}} = \max(Z_x, -Z_x)$
8.  $\text{true} \Rightarrow Z_{x^2} \geq 0$
9.  $(Z_{x^2} \leq \sum_i a_i Z_{x_i}) \wedge \bigwedge_i a_i \geq 0 \wedge \bigwedge_i Z_{x_i} \geq 0 \Rightarrow (Z_x \leq \sum_i \sqrt{a_i} Z_{\sqrt{x_i}})$

This rule is useful to compute an upper bound on a variable if an upper bound has been computed on its square.

The application of the above rules requires querying the abstract domain for constraints between a specific set of variables. These queries can be performed by existential elimination of all variables other than the specific set of variables.

We then apply the domain construction discussed in Section 4 on the domain  $A_S$  obtained above to obtain the domain  $\tilde{A}_S$ . For this purpose, we choose  $U$  to be the set of the input variables since we are ultimately interested in finding bounds with possibly max expressions over only the input variables. (Recall that  $U$  was the set of variables over which the abstract domain computes max expressions.)

## 6 Timing Analysis

In this section, we consider the problem of computing upper bounds on the time complexity of a program expressed as a function of the program inputs. We assume that each atomic statement is annotated with the units of time that it takes to execute. (To reduce cluttering in our examples, we assume that a recursive procedure call instruction and a backward jump instruction takes unit amount of time, while all other instructions take zero time. In other words, we estimate a bound on the total number of loop iterations and total number of recursive procedure call invocations).

Given a program  $P$ , we instrument the program with a monitor variable  $i$  that keeps track of the time consumed by the program. The monitor variable  $i$  is initialized to 0 at the beginning of the program, and is incremented by  $t$  units after execution of any instruction that takes  $t$  units of time to execute.

*Claim.* Let  $u_1, \dots, u_n$  be the upper bounds on the instrumented monitor variable  $i$  at different locations where  $i$  is incremented, expressed as a function of program inputs. Then,  $\max(0, u_1, \dots, u_n)$  denotes an upper bound on the timing complexity of the program.<sup>6</sup>

Note that we simply cannot compute an upper bound on the monitor variable  $i$  at the end of the program to obtain an upper bound on the timing complexity

<sup>6</sup> This assumes that the input variables are not modified in the program. (If they are, then we can create their copies and modify them instead.)

Program	Time (s)	$S$	Upper Bound
Disjunction (Fig1)	0.030	-	$\max(0, y - x_0) + \max(0, y - z_0)$
Sequential (Fig1)	0.010	-	$\max(0, n, m)$
NonLinear (Fig2)	0.018	$y^2, y_0^2, \sqrt{n}$	$\sqrt{2n} + \max(0, -2y_0) + 1$
ModularMultiply (Fig2)	0.105	$jn, n^2$	$n^2$
ModularSquare (Fig2)	0.098	$j^2, n^2$	$n(n+1)/2$
Log	0.084	$\lceil \log n \rceil, \lceil \log x \rceil$	$\lceil \log n \rceil$
Fibonacci	0.138	$2^n$	$2^n$
MergeSort	0.065	$n \lceil \log n \rceil$	$n \lceil \log n \rceil$
p1	0.141	-	41
p2	0.022	-	$\max(0, z_0)$
p3	0.215	$\lceil \log i \rceil, \lceil \log max \rceil, \lceil \log size \rceil$	$\lceil \log size \rceil$
p4	0.163	$s^2, bs, ts$	$s^2$
p5	0.025	-	$m + 1$
p6	0.031	-	$N$

**Table 1.** Experimental Results: Column 4 describes the upper bounds computed by our tool on the number of loop iterations and recursive procedure call invocations. Column 3 gives the expression set used for computing the bounds shown in Column 4.

of the program. For example, consider a program with a non-terminating loop. 0 is a valid upper bound on  $i$  at the end of the loop (since at an unreachable program location, any fact holds), but does not describe an upper bound on the timing complexity of the program.

We compute upper bounds on the monitor variable  $i$  at different program locations by performing abstract interpretation of the program over the numerical domain  $A_S$  described in Section 5, which provides us invariants at different program locations. An appropriate upper bound on  $i$  at a program location  $\pi$  is then obtained by considering the invariant  $I$  at  $\pi$  and existentially quantifying out all variables except  $i$  and the input variables from  $I$ . We can use a similar strategy for computing lower bounds on  $i$ . (An advantage of computing lower bounds is that they can be used as a measure of precision of our analysis for computing upper bounds.)

## 7 Experiments

We have implemented a prototype of our numerical abstract domain on top of the APRON [1] numerical abstract domains library. We have used this abstract domain in an abstract interpreter to bound the total number of loop iterations and the total number of recursive procedure calls invocations in several C programs <sup>7</sup>. Our abstract interpreter is implemented in `ocaml` and uses the CIL infrastructure to parse input C programs. We summarize the results of running our tool on a set of benchmarks in Table 1.

The programs  $p^*$  are taken from some benchmarks (originally from Octagon library distribution) presented in a paper that describes and compares some state-of-the-art techniques for proving program termination[10]. Most of the remaining programs are presented in Section 1. `Fibonacci` and `MergeSort` are

<sup>7</sup> Available at <http://www.cfdvs.iitb.ac.in/~bhargav/timing.html>

recursive programs. `Log` uses a multiplicative counter for loop iteration. The programs were analyzed using (cartesian-product) combination of polyhedra and octagon abstract domains lifted with *expression abstraction* and interval domain lifted with *max operator*.

For most of the programs shown in Table 1, the computed upper bounds are precise (i.e., they match the lower bounds computed by our tool on the monitor variable). These benchmarks include programs whose termination cannot be established by simple linear ranking functions [22], but requires more sophisticated techniques as in [23, 7, 8, 10]. This shows that not only our abstract domain is precise enough to express exact upper bounds but also that the abstract operations are precise enough to compute these bounds. The number of required input expressions is relatively small and simple heuristics (like the one described in Section 5) can be used to infer these automatically.

Our analyzer takes for each program a set of interesting expressions to track during the analysis. Although currently we provide these expressions manually, we intend to use some heuristics in the future to automatically infer these expressions.

## 8 Related Work

Inference-rule based reasoning has often been used for building efficient (but incomplete) decision procedures for otherwise intractable logics [18, 4]. The idea is to partially axiomatize the semantics of the underlying operators such that it leads to efficient reasoning as well as is precise enough to capture the reasoning required in the common case. In this paper, we show how to apply inference-rule based reasoning in the context of abstract interpreters as opposed to simply decision procedures. Secondly, we focus on a different domain, one that involves numerical operators.

There has been work on extending linear arithmetic abstract domains to also represent linear constraints over expressions constructed using uninterpreted functions [9, 14]. In contrast, our domain constructor allows extension to expressions with arbitrary operators (as opposed to only uninterpreted functions), but represents linear constraints only over the given set of expressions.

There has been work on discovering restricted form of quadratic inequalities [5], polynomial inequality invariants [2], and equality invariants [24, 21] of bounded degree. In contrast, our domain lifting operation allows for discovering arbitrary polynomial inequalities as well as non-polynomial inequalities in a uniform setting, but over a given set of expressions.

[20] describes a technique based on solving recurrences for computing a non-negative constant that represents the number of loop iterations required for reaching a particular error state. In contrast, we produce symbolic bounds.

There is a large body of work on estimating worst case execution time (WCET) in the embedded and real-time systems community [26, 15, 16, 3, 17]. The WCET research is more orthogonally focused on distinguishing between the complexity of different codepaths and low-level modeling of architectural

features such as caches, branch prediction, instruction pipelines. For establishing loop bounds, the WCET techniques either require user annotation, or use simple techniques based on pattern matching or a simple interval analysis. In contrast, we present a path-insensitive analysis, but one that automatically estimates precise (non-linear and disjunctive) bounds on loop iterations.

For example, the AiT-WCET tool uses combination of interval-based abstract interpretation and pattern matching. Loop bound analysis of BoundT-WCET tool is based on Presburger arithmetics. Both these are much less precise than our abstract domain which is not only relational but can also represent non-linear bounds. [15] describes an interval analysis based approach (as opposed to our more precise relational linear analysis) for automatic computation of loop bounds. However, it analyzes single-path executions of programs (i.e., using input data corresponding to one execution). Hence, their bounds are in real seconds, while our bounds are symbolic and functions of inputs. The analysis described in [16] is aimed at synchronous programs and linear hybrid systems. The only similarity is that they model delays in such programs using simple counters. We also use counter instrumentation; however our abstract domain construction allows us to compute disjunctive and non-linear bounds using a base linear abstract domain. Bagnara and Zaccagnini describe how to solve a class of recursive equations that are used to express complexity measures in several systems [3]. However, the class of equations that they handle are far apart from cost relations generated from real programs.

## 9 Conclusion

We have presented two domain lifting operations to make linear numerical abstract domains more precise. Domain lifting via expression abstraction enables computation of non-linear invariants. Domain lifting by max expressions provides a compact representation for disjunctive bounds. The importance of these domain lifting operations is reflected by the fact that we have been able to automatically compute precise timing bounds for several benchmark programs that have recently been used by the state-of-the-art techniques for proving termination of programs.

**Acknowledgments.** The first author was supported by Microsoft Corporation and Microsoft Research India under the Microsoft Research India PhD Fellowship Award.

## References

1. APRON. Numerical abstract domain library. <http://apron.cri.enscm.fr/library>, 2007.
2. R. Bagnara, E. Rodríguez-Carbonell, and E. Zaffanella. Generation of basic semi-algebraic invariants using convex polyhedra. In *SAS, LNCS*, 2005.
3. R. Bagnara and A. Zaccagnini. Checking and bounding the solutions of some recurrence relations. Quaderno 344, Università di Parma, Italy, 2004.

4. J. D. Bingham and Z. Rakamaric. A logic and decision procedure for predicate abstraction of heap-manipulating programs. In *VMCAI*, pages 207–221, 2006.
5. B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software. LNCS 2566. Oct. 2002.
6. A. Bradley, Z. Manna, and H. Sipma. Termination of polynomial programs. In *VMCAI*, 2005.
7. A. R. Bradley, Z. Manna, and H. B. Sipma. Linear ranking with reachability. In *CAV*, pages 491–504, 2005.
8. A. R. Bradley, Z. Manna, and H. B. Sipma. The polyranking principle. In *ICALP*, volume 3580 of *LNCS*, pages 1349–1361, 2005.
9. B.-Y. E. Chang and K. R. M. Leino. Abstract interpretation with alien expressions and heap structures. In *VMCAI*, pages 147–163, 2005.
10. A. Chawdhary, B. Cook, S. Gulwani, M. Sagiv, and H. Yang. Ranking abstractions. In *ESOP*, 2008.
11. M. Colón and H. Sipma. Practical methods for proving program termination. In *CAV*, pages 442–454, London, UK, 2002. Springer-Verlag.
12. B. Cook, A. Podelski, and A. Rybalchenko. Termination proofs for systems code. In *PLDI*, pages 415–426, 2006.
13. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL*, pages 84–97, 1978.
14. S. Gulwani and A. Tiwari. Combining abstract interpreters. In *PLDI*, pages 376–386, 2006.
15. J. Gustafsson, A. Ermedahl, C. Sandberg, and B. Lisper. Automatic derivation of loop bounds and infeasible paths for wcet analysis using abstract execution. In *RTSS*, pages 57–66, 2006.
16. N. Halbwachs, Y.-E. Proy, and P. Roumanoff. Verification of real-time systems using linear relation analysis. *Form. Methods Syst. Des.*, 11(2):157–185, 1997.
17. C. A. Healy, M. Sjodin, V. Rustagi, D. B. Whalley, and R. van Engelen. Supporting timing analysis by automatic bounding of loop iterations. *Real-Time Systems*, 18(2/3):129–156, 2000.
18. R. Jhala and K. McMillan. Array abstractions from proofs. In *CAV*, pages 193–206, 2007.
19. P. C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *CRYPTO*, pages 104–113, 1996.
20. D. Kroening and G. Weissenbacher. Counterexamples with loops for predicate abstraction. In *CAV*, pages 152–165, 2006.
21. M. Müller-Olm and H. Seidl. Precise interprocedural analysis through linear algebra. In *POPL*, pages 330–341, 2004.
22. A. Podelski and A. Rybalchenko. A complete method for the synthesis of linear ranking functions. In *VMCAI*, pages 239–251, 2004.
23. A. Podelski and A. Rybalchenko. Transition invariants. In *LICS*, pages 32–41. IEEE, July 2004.
24. S. Sankaranarayanan, H. Sipma, and Z. Manna. Non-linear loop invariant generation using gröbner bases. In *POPL*, pages 318–329, 2004.
25. A. Turing. Checking a large routine. pages 70–72, 1989.
26. R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The Determination of Worst-Case Execution Times—Overview of the Methods and Survey of Tools. In *ACM Transactions on Embedded Computing Systems (TECS)*, 2007.