SubPolyhedra: A (more) scalable approach to infer linear inequalities

Vincent Laviron[1] Francesco Logozzo[2]

[1] École Normale Supérieure, 45, rue d'Ulm, Paris (France)
Vincent.Laviron@ens.fr
[2] Microsoft Research, Redmond, WA (USA)
logozzo@microsoft.com

Abstract. We introduce Subpolyhedra (SubPoly) a new numerical abstract domain to infer and propagate linear inequalities. SubPoly is as expressive as Polyhedra, but it drops some of the deductive power to achieve scalability. SubPoly is based on the insight that the reduced product of linear equalities and intervals produces powerful yet scalable analyses. Precision can be recovered using hints. Hints can be automatically generated or provided by the user in the form of annotations.
We implemented SubPoly on the top of Clousot, a generic abstract interpreter for .Net. Clousot with SubPoly analyzes very large and complex code bases in few minutes. SubPoly can efficiently capture linear inequalities among hundreds of variables, a result well-beyond state-of-the-art implementations of Polyhedra.

1 Introduction

The goal of an abstract interpretation-based static analyzer is to statically infer properties of the execution of a program that can be used to check its specification. The specification usually includes the absence of runtime exceptions (division by zero, integer overflow, array index out of bounds ...) and programmer annotations in the form of preconditions, postconditions, object invariants and assertions ("contracts" [23]). Proving that a piece of code satisfies its specification often requires discovering numerical invariants on program variables.

The concept of abstract domain is central in the design and the implementation of a static analyzer [9]. Abstract domains capture the properties of interest on programs. In particular *numerical* abstract domains are used to infer numerical relationships among program variables. Cousot and Halbwachs introduced the Polyhedra numerical abstract domain (Poly) in [11]. Poly infers all the linear inequalities on the program variables. The application and scalability of Poly has been severely limited by its performance which is worst-case exponential (easily attained in practice). To overcome this shortcoming and to achieve scalability, new numerical abstract domains have been designed either considering only inequalities of a particular shape (weakly relational domains) or fixing *ahead* of the analysis the maximum number of linear inequalities to be considered (bounded domains). The first class includes Octagons (which capture properties in the form

```
class StringBuilder {
  int ChunkLen; char[] ChunkChars;

  public void Append(int wb, int count) {
    Contract.Requires(wb >= 2 * count);
    if (count + ChunkLen > ChunkChars.Length)
(*)   CopyChars(wb, ChunkChars.Length - ChunkLen);
    // ... }

  private void CopyChars(int wb, int len) {
    Contract.Requires(wb >= 2 * len);
  // ...  }
```

Fig. 1. An example extracted from `mscorlib.dll`. `Contract.Requires`$(\dots)$ expresses method preconditions. Proving the precondition of `CopyChars` requires propagating an invariant involving three variables and non-unary coefficients.

$\pm \mathtt{x} \pm \mathtt{y} \leq c$) [24], TVPI ($a \cdot \mathtt{x} + b \cdot \mathtt{y} \leq c$) [29], Pentagons ($\mathtt{x} \leq \mathtt{y} \wedge a \leq \mathtt{x} \leq b$) [22], Stripes ($\mathtt{x} + a \cdot (\mathtt{y} + \mathtt{z}) > b$) [14] and Octahedra ($\pm \mathtt{x}_0 \cdots \pm \mathtt{x}_n \leq c$) [7]. The latter includes constraint template matrices (which capture at most m linear inequalities) [28] and methods to generate polynomial invariants *e.g.* [25, 26].

Although impressive results have been achieved using weakly relational and bounded abstract domains, we experienced situations where the full *expressive* power of Poly is required. As an example, let us consider the code snippet of Fig. 1, extracted from `mscorlib.dll`, the main library of the .Net framework. Checking the precondition at the call site ($*$) involves (i) *propagating* the constraints $\mathtt{wb} \geq 2 \cdot \mathtt{count}$ and $\mathtt{count} + \mathtt{ChunkLen} > \mathtt{ChunkChars.Length}$; and (ii) *deducing* that $\mathtt{wb} \geq 2 \cdot (\mathtt{ChunkChars.Length} - \mathtt{ChunkLen})$. The aforementioned weakly relational domains cannot be used to check the precondition: Octahedra do not capture the first constraint (it involves a constraint with a non-unary coefficient); TVPI do not propagate the second constraint (it involves three variables); Pentagons and Octagons cannot represent any of the two constraints; Stripes can propagate both constraints, but because of the incomplete closure it cannot deduce the precondition. Bounded domains do the job, provided we fix *before* the analysis the template of the constraints. This is inadequate for our purposes: The analysis of a *single* method in `mscorlib.dll` may involve hundreds of constraints, whose shape cannot be fixed ahead of the analysis, *e.g.* by a textual inspection. Poly easily propagates the constraints. However, in the general case the price to pay for using Poly is too high: the analysis will be limited to few dozens of variables.

Subpolyhedra We propose a new abstract domain, Subpolyhedra (SubPoly), which has the same *expressive* power as Poly, but it drops some inference power to achieve scalability: SubPoly exactly represents and propagates linear inequalities containing hundreds of variables and constraints. SubPoly is based on the fundamental insight that the reduced product of linear equalities, LinEq [17], and intervals, Intv [9], can produce very powerful yet efficient program analyses.

SubPoly can represent linear inequalities using slack variables, *e.g.* $\texttt{wb} \geq 2 \cdot \texttt{count}$ is represented in SubPoly by $\texttt{wb} - 2 \cdot \texttt{count} = \beta \wedge \beta \in [0, +\infty]$. As a consequence, SubPoly easily proves that the precondition for `CopyChars` is satisfied at the call site ($*$). In general the join of SubPoly is less precise than the one on Poly, so that it may not infer *all* the linear inequalities. Hints, either automatically generated or provided by the user, help recover precision.

Cardinal operations for SubPoly are: (i) the reduction, which propagates the information between LinEq and Intv; (ii) the join, which derives a compact yet precise upper approximation of two incoming abstract states; and (iii) the hint generator, which recovers information lost at join points.

```
void Foo(int i, int j) {
  int x = i, y = j;
  if (x <= 0) return;
  while (x > 0) { x--; y--; }
  if (y == 0) Assert(i == j); }
```

Fig. 2. An example from [27]. SubPoly infers the loop invariant $\texttt{x} - \texttt{i} = \texttt{y} - \texttt{j} \wedge \texttt{x} \geq 0$, propagates it and proves the assertion.

Reduction Let us consider the example in Fig. 2, taken from [27]. The program contains operations and predicates that can be exactly represented with Octagons. Proving that the assertion is not violated requires discovering the loop invariant $\texttt{x} - \texttt{y} = \texttt{i} - \texttt{j} \wedge \texttt{x} \geq 0$. The loop invariant cannot be fully represented in Octagons: it involves a relation on four variables. Bounded numerical domains are unlikely to help here as there is no way to syntactically figure out the required template. The LinEq component of SubPoly infers the relation $\texttt{x} - \texttt{y} = \texttt{i} - \texttt{j}$. The Intv component of SubPoly infers the loop invariant $\texttt{x} \in [0, +\infty]$, which in conjunction with the negation of the guard implies that $\texttt{x} \in [0, 0]$. The reduction of SubPoly propagates the interval, refining the linear constraint to $\texttt{y} = \texttt{j} - \texttt{i}$. This is enough to prove the assertion (in conjunction with the if-statement guard). It is worth noting that unlike [27] SubPoly does not require any hypothesis on the order of variables to prove the assertion.

Join and Hints Let us consider the code in Fig. 3, taken from [15]. The loop invariant required to prove that the assertion is unreached (and hence that the program is correct) is $\texttt{x} \leq \texttt{y} \leq 100 \cdot \texttt{x} \wedge \texttt{z} = 10 \cdot \texttt{w}$. Without hints, SubPoly can only infer $\texttt{z} = 10 \cdot \texttt{w}$. *Template* hints, inspired by [28], are used to recover linear inequalities that are dropped by the imprecision of the join: In the example the template is $\texttt{x} - \texttt{y} \leq b$, and the analysis automatically figures out that $b = 0$. *Planar Convex hull* hints, inspired by [29], are used to introduce at join points linear inequalities derived by a planar convex hull: In the example it helps the analysis figure out that $\texttt{y} \leq 100 \cdot \texttt{x}$. It is worth noting that SubPoly does not need any of the techniques of [15] to infer the loop invariant.

2 Abstract Interpretation Background

We assume the concrete domain to be the complete Boolean lattice of environments, *i.e.* $\mathsf{C} = \langle \mathcal{P}(\Sigma), \subseteq, \emptyset, \Sigma, \cup, \cap \rangle$, where $\Sigma = [\texttt{Vars} \to \mathbb{Z}]$. An abstract domain A is a tuple $\langle \mathsf{D}, \gamma, \sqsubseteq, \bot, \top, \sqcup, \sqcap, \triangledown, \rho \rangle$. The set of abstract elements

```
int x = 0, y = 0, w = 0, z = 0;
while (...) {
  if (...) { x++; y += 100; }
  else if (...) { if (x >= 4) { x++; y++; } }
  else if (y > 10 * w && z >= 100 * x) { y = -y; }

  w++; z += 10; }
if (x >= 4 && y <= 2) Assert(false);
```

Fig. 3. An example from [15]. SubPoly infers the loop invariant $\mathtt{x} \leq \mathtt{y} \leq 100{\cdot}\mathtt{x} \wedge \mathtt{z} = 10{\cdot}\mathtt{w}$, propagates it out of the loop, and proves that the assertion is unreached.

D is related to the concrete domain by a *monotonic* concretization function $\gamma \in [\mathsf{D} \to \mathsf{C}]$. With an abuse of notation, we will not distinguish between an abstract domain and the set of its elements. The approximation order $\sqsubseteq$ soundly approximates the concrete order: $\forall \mathsf{d}_0, \mathsf{d}_1 \in \mathsf{D}.\ \mathsf{d}_0 \sqsubseteq \mathsf{d}_1 \implies \gamma(\mathsf{d}_0) \subseteq \gamma(\mathsf{d}_1)$. The smallest element is $\bot$, the largest element is $\top$. The join operator $\sqcup$ satisfies $\forall \mathsf{d}_0, \mathsf{d}_1 \in \mathsf{D}.\ \mathsf{d}_0 \sqsubseteq \mathsf{d}_0 \sqcup \mathsf{d}_1 \wedge \mathsf{d}_1 \sqsubseteq \mathsf{d}_0 \sqcup \mathsf{d}_1$. The meet operator $\sqcap$ satisfies $\forall \mathsf{d}_0, \mathsf{d}_1 \in \mathsf{D}.\ \mathsf{d}_0 \sqcap \mathsf{d}_1 \sqsubseteq \mathsf{d}_0 \wedge \mathsf{d}_0 \sqcap \mathsf{d}_1 \sqsubseteq \mathsf{d}_1$. The widening $\triangledown$ ensures the convergence of the fixpoint iterations, *i.e.* it satisfies: (i) $\forall \mathsf{d}_0, \mathsf{d}_1 \in \mathsf{D}.\ \mathsf{d}_0 \sqsubseteq \mathsf{d}_0 \triangledown \mathsf{d}_1 \wedge \mathsf{d}_1 \sqsubseteq \mathsf{d}_0 \triangledown \mathsf{d}_1$; and (ii) for each sequence of abstract elements $\mathsf{d}_0, \mathsf{d}_1, \ldots \mathsf{d}_k$ the sequence defined by $\mathsf{d}_0^\triangledown = \mathsf{d}_0, \mathsf{d}_1^\triangledown = \mathsf{d}_0^\triangledown \triangledown \mathsf{d}_1 \ldots \mathsf{d}_k^\triangledown = \mathsf{d}_{k-1}^\triangledown \triangledown \mathsf{d}_k$ is ultimately stationary. In general, we do not require abstract elements to be in some canonical or closed form, *i.e.* there may exist $\mathsf{d}_0, \mathsf{d}_1 \in \mathsf{D}$, such that $\mathsf{d}_0 \neq \mathsf{d}_1$, but $\gamma(\mathsf{d}_0) = \gamma(\mathsf{d}_1)$. The *reduction* operator $\rho \in [\mathsf{D} \to \mathsf{D}]$ puts an abstract element into a (pseudo-)canonical form without adding or losing any information: $\forall \mathsf{d}.\ \gamma(\rho(\mathsf{d})) = \gamma(\mathsf{d}) \wedge \rho(\mathsf{d}) \sqsubseteq \mathsf{d}$. We do not require ρ to be idempotent. New abstract domains can be systematically derived by cartesian composition or functional lifting [10]. Following [8], we use the dot-notation to denote point wise extensions.

Intervals The abstract domain of interval environments is $\langle \mathsf{Intv}, \gamma_{\mathsf{Intv}}, \sqsubseteq_{\mathsf{Intv}}, \bot_{\mathsf{Intv}}, \top_{\mathsf{Intv}}, \sqcup_{\mathsf{Intv}}, \sqcap_{\mathsf{Intv}}, \triangledown_{\mathsf{Intv}} \rangle$. The abstract elements are maps from program variables to open intervals. The concretization of an interval environment i is $\gamma_{\mathsf{Intv}}(\mathsf{i}) = \{s \in \Sigma \mid \forall \mathtt{x} \in \mathrm{dom}(\mathsf{i}).\ \mathsf{i}(\mathtt{x}) = [a, b] \wedge a \leq s(\mathtt{x}) \leq b\}$. The order is interval inclusion, the bottom element is the empty interval, the top is the interval $[-\infty, +\infty]$, the join is the smallest interval which contains the two arguments, the meet is interval intersection, and the widening keeps the stable bounds. The reduction is the identity function. All the domain operations can be implemented to take linear time.

Linear Equalities The abstract domain of linear *equalities* is $\langle \mathsf{LinEq}, \gamma_{\mathsf{LinEq}}, \sqsubseteq_{\mathsf{LinEq}}, \bot_{\mathsf{LinEq}}, \top_{\mathsf{LinEq}}, \sqcup_{\mathsf{LinEq}}, \sqcap_{\mathsf{LinEq}} \rangle$. The elements are sets of linear equalities, their meaning is given by the set of concrete states which satisfy the constraints, *i.e.* $\gamma_{\mathsf{LinEq}} = \lambda \mathsf{l}.\ \{s \in \Sigma \mid \forall (\sum a_i \cdot \mathtt{x}_i = b) \in \mathsf{l}.\ \sum a_i \cdot s(\mathtt{x}_i) = b\}$. The order is sub-space inclusion, the bottom is the empty space, the top is the whole space, the join is the smallest space which contains the two arguments, the meet is space intersection. LinEq has finite height, so the join suffices to ensure analysis termination. The reduction is Gaussian elimination. The complexity of

the domain operations is subsumed by the complexity of Gaussian elimination, which is cubic.

Polyhedra The abstract domain of linear *inequalities* is $\langle \mathsf{Poly}, \gamma_{\mathsf{Poly}}, \sqsubseteq_{\mathsf{Poly}}, \bot_{\mathsf{Poly}}, \top_{\mathsf{Poly}}, \sqcup_{\mathsf{Poly}}, \sqcap_{\mathsf{Poly}}, \triangledown_{\mathsf{Poly}} \rangle$. The elements are sets of linear inequalities, the concretization is the set of concrete states which satisfy the constraints *i.e.* $\gamma_{\mathsf{Poly}} = \lambda \mathsf{p}.\ \{s \in \Sigma \mid \forall (\sum a_i \cdot \mathtt{x}_i \leq b) \in \mathsf{p}.\ \sum a_i \cdot s(\mathtt{x}_i) \leq b\}$, the order is the polyhedron inclusion, the bottom is the empty polyhedron, the top is the whole space, the join is the convex hull, the meet is just the union of the set of constraints, and the widening preserves the inequalities stable among two successive iterations. The reduction infers the set of generators and removes the redundant inequalities. The cost of the Poly operations is subsumed by the cost of the conversion between the algebraic representation (set of inequalities) and the geometric representation (set of generators) used in the implementation [1]. In fact, some operations require the algebraic representation (*e.g.* $\sqcap_{\mathsf{Poly}}$), some require the geometrical representation (*e.g.* $\sqcup_{\mathsf{Poly}}$), and some others require both (*e.g.* $\sqsubseteq_{\mathsf{Poly}}$). The conversion between the two representations is exponential in the number of variables, and it cannot be done better [18].

3 Subpolyhedra

We introduce the numerical abstract domain of Subpolyhedra, SubPoly. The main idea of SubPoly is to combine Intv and LinEq to capture complex linear *inequalities*. Slack variables are introduced to replace inequality constraints with equalities.

Variables A variable $\mathtt{v} \in \mathtt{Vars}$ can either be a *program* variable ($\mathtt{x} \in \mathtt{Var_P}$) or a *slack* variable ($\beta \in \mathtt{Var_S}$). A slack variable β has associated information, denoted by $\mathtt{info}(\beta)$, which is a linear form $a_1 \cdot \mathtt{v}_1 + \cdots + a_k \cdot \mathtt{v}_k$. Let $\kappa \equiv \sum a_i \cdot \mathtt{x}_i + \sum b_j \cdot \beta_j = c$ be a linear equality: $\mathtt{s}_\kappa = \sum_{\mathtt{x}_i \in \mathtt{Var_P}} a_i \cdot \mathtt{x}_i$ denotes the partial sum of the monomials involving just program variables; $\mathtt{Var_P}(\kappa) = \{\mathtt{x}_i \mid a_i \cdot \mathtt{x}_i \in \kappa, a_i \neq 0\}$ and $\mathtt{Var_S}(\kappa) = \{\beta_j \mid b_j \cdot \beta_j \in \kappa, b_j \neq 0\}$ denote respectively the program variables and the slack variables in κ. The generalization to inequalities and sets of equalities and inequalities is straightforward.

Elements The elements of SubPoly belong to the reduced product LinEq $\otimes$ Intv [10]. Inequalities are represented in SubPoly with slack variables: $\sum a_i \cdot \mathtt{x}_i \leq c \iff \sum a_i \cdot \mathtt{x}_i - c = \beta \wedge \beta \in [-\infty, 0]$ (β is a fresh slack variable with the associated information $\mathtt{info}(\beta) = \sum a_i \cdot \mathtt{x}_i$).

Concretization A subpolyhedron can be interpreted as a polyhedron by projecting out the slack variables: $\gamma_S^{\mathsf{Poly}} \in [\mathsf{SubPoly} \to \mathsf{Poly}]$ is $\gamma_S^{\mathsf{Poly}} = \lambda \langle \mathsf{I};\ \mathsf{i} \rangle.\ \pi_{\mathtt{Vars}}(\mathsf{I} \cup \{a \leq \mathtt{v} \leq b \mid \mathsf{i}(\mathtt{v}) = [a, b]\})$, where π denotes the projection of variables in Poly. The concretization $\gamma_S \in [\mathsf{SubPoly} \to \mathcal{P}(\Sigma)]$ is then $\gamma_S = \gamma_{\mathsf{Poly}} \circ \gamma_S^{\mathsf{Poly}}$.

Approximation Order The order on SubPoly may be defined in terms of order over Poly. Given two subpolyhedra $\mathsf{s}_0, \mathsf{s}_1$, the most precise order relation $\sqsubseteq_S^*$ is $\mathsf{s}_0 \sqsubseteq_S^* \mathsf{s}_1 \iff \gamma_S^{\mathsf{Poly}}(\mathsf{s}_0) \sqsubseteq_{\mathsf{Poly}} \gamma_S^{\mathsf{Poly}}(\mathsf{s}_1)$. However, $\sqsubseteq_S^*$ may be too expensive to compute: it involves mapping subpolyhedra in the dual representation of Poly.

```
if(...)                          if(...)
 { assume x - y <= 0; }           { assume x == y; assume y <= z; }
else                             else
 { assume x - y <= 5; }           { assume x <= y; assume y == z; }
```

(a) (b)

Fig. 4. Examples illustrating the need for Step 1 in the join algorithm

This can easily cause an exponential blow up. We define a weaker approximation order relation which first tries to find a renaming θ for the slack variables, and then checks the pairwise order. Formally ($\cdot \xrightarrow{\text{inj}} \cdot$ denotes an injective function):

$$\langle \mathsf{l}_0;\ \mathsf{i}_0\rangle \sqsubseteq_S \langle \mathsf{l}_1;\ \mathsf{i}_1\rangle \iff \exists\theta.\ \mathtt{Var_S}(\langle \mathsf{l}_0;\ \mathsf{i}_0\rangle) \xrightarrow{\text{inj}} \mathtt{Var_S}(\langle \mathsf{l}_1;\ \mathsf{i}_1\rangle).$$
$$\forall \beta \in \mathtt{Var_S}(\langle \mathsf{l}_0;\ \mathsf{i}_0\rangle).\ \mathtt{info}(\beta) = \mathtt{info}(\theta(\beta)) \wedge \theta(\langle \mathsf{l}_0;\ \mathsf{i}_0\rangle) \dot{\sqsubseteq} \langle \mathsf{l}_1;\ \mathsf{i}_1\rangle.$$

In general $\sqsubseteq_S \subsetneq \sqsubseteq_S^*$. In practice, $\sqsubseteq_S$ is used to check if a fixpoint has been reached. A weaker order relation means that the analysis may perform some extra widening steps, which may introduce precision loss. However, we found the definition of $\sqsubseteq_S$ satisfactory in our experience.

Bottom A subpolyhedron is equivalent to bottom if after a reduction one of the two components is bottom: $\mathsf{s} = \bot_S \iff \rho(\mathsf{s}) = \langle \mathsf{l}, \mathsf{i}\rangle \wedge (\mathsf{i} = \dot{\bot}_{\mathsf{Intv}} \vee \mathsf{l} = \bot_{\mathsf{LinEq}})$.

Top A subpolyhedron is top if both components are top: $\mathsf{s} = \top_S \iff \mathsf{s} = \langle \mathsf{l}, \mathsf{i}\rangle \wedge \mathsf{i} = \top_{\mathsf{Intv}} \wedge \mathsf{l} = \top_{\mathsf{LinEq}}$.

Linear form evaluation Let $\mathtt{s}$ be a linear form: $[\![\mathtt{s}]\!] \in [\mathsf{SubPoly} \to \mathsf{Intv}]$ denotes the evaluation of $\mathtt{s}$ in a subpolyhedron after the reduction has inferred the tightest bounds: $[\![\sum a_i \cdot \mathtt{v}_i]\!]\ \langle \mathsf{l};\ \mathsf{i}\rangle = \textbf{let}\ \langle \mathsf{l}^*;\ \mathsf{i}^*\rangle = \rho(\langle \mathsf{l};\ \mathsf{i}\rangle)\ \textbf{in} \sum a_i \cdot \mathsf{i}^*(\mathtt{v}_i)$.

Join The join $\sqcup_S$ is computed in three steps. First, inject the information of the slack variables into the abstract elements. Second, perform the pairwise join on the saturated arguments. Third, add the constraints that are implied by the two operands of the join, but that were not preserved by the previous step. The join, parameterized by the reduction ρ, is defined by the Algorithm 1 (We let $\underline{0} = 1$, $\underline{1} = 0$). We illustrate the join with examples. We postpone the discussion of the reduction to Sect. 4.

Example 1 (Steps 1 & 2). Let us consider the code in Fig. 4(a). After the assumption, the abstract states on the true branch and the false branch are respectively: $\mathsf{s}_0 = \langle \mathtt{x} - \mathtt{y} = \beta_0;\ \beta_0 \in [-\infty, 0]\rangle$ and $\mathsf{s}_1 = \langle \mathtt{x} - \mathtt{y} = \beta_1;\ \beta_1 \in [-\infty, 5]\rangle$. The information associated with the slack variables is $\mathtt{info}(\beta_0) = \mathtt{info}(\beta_1) = \mathtt{x} - \mathtt{y}$. At the join point we apply Algorithm 1. Step 1 refines the abstract states by introducing the information associated with the slack variables: $\mathsf{s}'_0 = \langle \mathtt{x} - \mathtt{y} = \beta_0 = \beta_1;\ \beta_0 \in [-\infty, 0]\rangle$ and $\mathsf{s}'_1 = \langle \mathtt{x} - \mathtt{y} = \beta_1 = \beta_0;\ \beta_1 \in [-\infty, 5]\rangle$. Step 2 requires the reduction of the operands. The interval for β_1 (resp. β_0) in s'_0 (resp. s'_1) is refined: $\rho(\mathsf{s}'_0) = \langle \mathtt{x} - \mathtt{y} = \beta_0 = \beta_1;\ \beta_0 \in [-\infty, 0], \beta_1 \in [-\infty, 0]\rangle$ and $\rho(\mathsf{s}'_1) = \langle \mathtt{x} - \mathtt{y} = \beta_1 = \beta_0;\ \beta_0 \in [-\infty, 5], \beta_1 \in [-\infty, 5]\rangle$. The pairwise join gets the expected invariant: $\mathsf{s}_\sqcup = \rho(\mathsf{s}'_0) \dot{\sqcup} \rho(\mathsf{s}'_1) = \langle \mathtt{x} - \mathtt{y} = \beta_0 = \beta_1;\ \beta_0 \in [-\infty, 5], \beta_1 \in [-\infty, 5]\rangle$. □

Algorithm 1 The join $\sqcup_S$ on Subpolyhedra

input $\langle \mathsf{l}_i;\ \mathsf{i}_i \rangle \in \mathsf{SubPoly}$, $i \in \{0,1\}$

let $\langle \mathsf{l}_i^{'};\ \mathsf{i}_i^{'} \rangle = \langle \mathsf{l}_i;\ \mathsf{i}_i \rangle$

{Step 1. Propagate the information of the slack variables}
for all $\beta \in \mathtt{Var_S}(\mathsf{l}_i) \setminus \mathtt{Var_S}(\mathsf{l}_{\underline{i}})$ **do**
  $\langle \mathsf{l}_{\underline{i}}^{'};\ \mathsf{i}_{\underline{i}}^{'} \rangle := \langle \mathsf{l}'_{\underline{i}} \sqcap_{\mathsf{LinEq}} \{\beta = \mathtt{info}(\beta)\};\ \mathsf{i}'_{\underline{i}} \rangle$

{Step 2. Perform the point-wise join on the saturated operands}
let $\langle \mathsf{l}_\sqcup;\ \mathsf{i}_\sqcup \rangle = \rho(\langle \mathsf{l}_0^{'};\ \mathsf{i}_0^{'} \rangle) \dot{\sqcup} \rho(\langle \mathsf{l}_1^{'};\ \mathsf{i}_1^{'} \rangle)$

{Step 3. Recover the lost information }
let D_i be the linear equalities dropped from l'_i at the previous step
for all $\kappa \in D_i$ **do**
  let $\mathsf{i}_{s_\kappa} = [\![\mathtt{s}_\kappa]\!] \langle \mathsf{l}_{\underline{i}}^{'};\ \mathsf{i}_{\underline{i}}^{'} \rangle$
  if κ contains no slack variable **then**
    if $\mathsf{i}_{s_\kappa} \neq \top_{\mathsf{Intv}}$ **then**
      let β be a fresh slack variable
      $\langle \mathsf{l}_\sqcup;\ \mathsf{i}_\sqcup \rangle := \langle \mathsf{l}_\sqcup \sqcap_{\mathsf{LinEq}} \{\beta = \kappa\};\ \mathsf{i}_\sqcup \sqcap_{\mathsf{Intv}} \{\beta = \mathsf{i}_{s_\kappa} \sqcup_{\mathsf{Intv}} [0,0]\} \rangle$
  else if κ contains exactly one slack variable β **then**
    if $\mathsf{i}_{s_\kappa} \neq \top_{\mathsf{Intv}}$ **then**
      $\langle \mathsf{l}_\sqcup;\ \mathsf{i}_\sqcup \rangle := \langle \mathsf{l}_\sqcup \sqcap_{\mathsf{LinEq}} \{\kappa\}; \mathsf{i}_\sqcup \sqcap_{\mathsf{Intv}} \{\beta = \mathsf{i}_{s_\kappa} \sqcup_{\mathsf{Intv}} \mathsf{i}_i(\beta)\} \rangle$
return $\langle \mathsf{l}_\sqcup;\ \mathsf{i}_\sqcup \rangle$

Example 2 (Non-trivial information for slack variables). Let us consider the code snippet in Fig. 4(b). The abstract states to be joined are $\langle \mathtt{x}-\mathtt{y}=0, \mathtt{y}-\mathtt{z} = \beta_0; \beta_0 \in [-\infty, 0] \rangle$ and $\langle \mathtt{y}-\mathtt{z}=0, \mathtt{x}-\mathtt{y} = \beta_1; \beta_1 \in [-\infty, 0] \rangle$. The associated information are $\mathtt{info}(\beta_0) = \mathtt{y}-\mathtt{z}$ and $\mathtt{info}(\beta_1) = \mathtt{x}-\mathtt{y}$. Step 1 allows to refine the abstract states with the slack variable information, and hence to infer that after the join $\mathtt{x} \leq \mathtt{y}$ and $\mathtt{y} \leq \mathtt{z}$. □

The two examples above show the importance of introducing the information associated with slack variables in Step 1 and the reduction in Step 2. Without those, the relation between the slack variables and the program point where they were introduced would have been lost.

The join of LinEq is *precise* in that if a linear equality is implied by both operands, then it is implied by the result too. The same for the join of Intv. The pairwise join in $\mathsf{LinEq} \otimes \mathsf{Intv}$ may drop some inequalities. Some of those can be recovered by the refinement step. The next example illustrates it.

Example 3 (Step 3). Let us consider the code in Fig. 5(a). The analysis of the two branches of the conditional produces the abstract states: $\mathsf{s}_0 = \langle \mathtt{x}-3\cdot\mathtt{y}=0;\ \top_{\mathsf{Intv}} \rangle$ and $\mathsf{s}_1 = \langle \mathtt{x}=0, \mathtt{y}=1;\ \mathtt{x} \in [0,0], \mathtt{y} \in [1,1] \rangle$. The reduction ρ does not refine the states (we already have the tightest bounds). The point-wise join produces the abstract state $\top_S$. Step 3 identifies the dropped constraints: $D_0 = \{\mathtt{x}-3\cdot\mathtt{y}=0\}$ and $D_1 = \{\mathtt{x}=0, \mathtt{y}=1\}$. The algorithm inspects them to check if they are satisfied by the "other" branch. The constraint in D_0 is also satisfied in the false branch: $[\![\mathtt{x}-3\cdot\mathtt{y}]\!](\mathsf{s}_1) = [-3,-3]$ $(\neq \top_{\mathsf{Intv}})$. Therefore it can be safely added to

Algorithm 2 The widening $\triangledown_S$ on Subpolyhedra

input $\langle \mathsf{l}_i;\ \mathsf{i}_i\rangle \in \mathsf{SubPoly}$, $i \in \{0,1\}$

let $\langle \mathsf{l}_i^{'};\ \mathsf{i}_i^{'}\rangle = \langle \mathsf{l}_i;\ \mathsf{i}_i\rangle$

{Step 1. Propagate the information of the slack variables}
for all $\beta \in \mathtt{Vars}(\mathsf{l}_0) \setminus \mathtt{Vars}(\mathsf{l}_1)$ **do**
$\langle \mathsf{l}_0^{'};\ \mathsf{i}_0^{'}\rangle := \langle \mathsf{l}_0' \sqcap_{\mathsf{LinEq}} \{\beta = \mathtt{info}(\beta)\};\ \mathsf{i}_0'\rangle$

{Step 2. Perform the point-wise widening}
let $\langle \mathsf{l}_\triangledown;\ \mathsf{i}_\triangledown\rangle = \langle \mathsf{l}_0^{'};\ \mathsf{i}_0^{'}\rangle \dot{\triangledown} \rho(\langle \mathsf{l}_1^{'};\ \mathsf{i}_1^{'}\rangle)$

{Step 3. Recover the lost information }
let D_0 be the linear equalities dropped from l_0' at the previous step
for all $\kappa \in D_0$ **do**
let $\mathsf{i}_{s_\kappa} = [\![\mathtt{s}_\kappa]\!]\langle \mathsf{l}_1^{'};\ \mathsf{i}_1^{'}\rangle$
if κ contains no slack variables **then**
if $\mathsf{i}_{s_\kappa} \neq \top_{\mathsf{Intv}}$ **then**
let β be a fresh slack variable
$\langle \mathsf{l}_\triangledown;\ \mathsf{i}_\triangledown\rangle := \langle \mathsf{l}_\triangledown \sqcap_{\mathsf{LinEq}} \{\beta = \kappa\};\ \mathsf{i}_\triangledown \sqcap_{\mathsf{Intv}} \{\beta = [0,0] \triangledown \mathsf{i}_{s_\kappa}\}\rangle$
else if κ contains exactly one slack variable β **then**
if $\mathsf{i}_{s_\kappa} \neq \top_{\mathsf{Intv}}$ **then**
$\langle \mathsf{l}_\triangledown;\ \mathsf{i}_\triangledown\rangle := \langle \mathsf{l}_\triangledown \sqcap_{\mathsf{LinEq}} \{\kappa\};\ \mathsf{i}_\triangledown \sqcap_{\mathsf{Intv}} \{\beta = \mathsf{i}_0(\mathtt{v}) \triangledown \mathsf{i}_{s_\kappa}\}\rangle$

return $\langle \mathsf{l}_\triangledown;\ \mathsf{i}_\triangledown\rangle$

the result. The constraints of D_2 do not hold on the left branch and they are discarded. The abstract state after the join is $\mathtt{s}_\sqcup = \langle \mathtt{x} - 3\cdot\mathtt{y} = \beta;\ \beta \in [-3,0]\rangle$. □

Meet The meet $\sqcap_S$ is simply the pairwise meet on $\mathsf{LinEq} \otimes \mathsf{Intv}$.

Widening The definition of the widening (Algorithm 2) is similar to the join, with the main differences that: (i) the information associated to slack variables is propagated only in one direction; (ii) only the right argument is saturated; and (iii) the recovery step is applied only to one of the operands. Those hypotheses avoid the well-known problems of interaction between reduction, refinement and convergence of the iterations [24].

Example 4 (Refinement step for the widening). Let us consider the code snippet in Fig. 5(b). The entry state to the loop is $\mathtt{s}_0 = \langle \mathtt{i} - \mathtt{k} = 0;\ \top_{\mathsf{Intv}}\rangle$. The state after one iteration is $\mathtt{s}_1 = \langle \mathtt{i} - \mathtt{k} = 1;\ \top_{\mathsf{Intv}}\rangle$. We apply the widening operator. Step 1 does not refine the states as there are no slack variables. The pairwise widening of Step 2 loses all the information. Step 3 recovers the constraint $\mathtt{k} \leq \mathtt{i}$: $D_0 = \{\mathtt{i} - \mathtt{k} = 0\}$ contains no slack variables and $[\![\mathtt{i} - \mathtt{k}]\!](\mathtt{s}_1) = [1,1]$ so that $\mathtt{s}_\triangledown = \langle \mathtt{i} - \mathtt{k} = \beta;\ \beta \in [0, +\infty]\rangle$. □

Theorem 1 (Fixpoint convergence). *The operator defined in Algorithm 2 is a widening. Moreover, $\sqsubseteq_S$ can be used to check that the fixpoint iterations eventually stabilize.*

```
if(...) { assume x == 3 * y; }
else    { x = 0; y = 1; }
```
(a)

```
i := k;
while(...) i++;
assert i >= k;
```
(b)

Fig. 5. Examples illustrating the need for the Step 3 in the join and the widening.

4 Reduction for Subpolyhedra

The reduction in SubPoly infers tighter bounds on linear forms and hence on program variables. Reduction is cardinal to fine tuning the precision/cost ratio. We propose two reduction algorithms, one based on linear programming, ρ_{LP}, and the other on basis exploration, ρ_{BE}. Both of them have been implemented in `Clousot`, our abstract interpretation-based static analyzer for `.Net` [4].

Linear programming-based reduction A linear programming problem is the problem of maximizing (or minimizing) a linear function subject to a finite number of linear constraints. We consider *upper bounding* linear problems (UBLP) [6], *i.e.* problems in the form (n is the number of variables, m is the number of equations):

$$\begin{array}{ll} \text{maximize} & c \cdot \mathtt{v}_k \qquad k \in 1 \ldots n, c \in \{-1, +1\} \\ \text{subject to} & \sum_{j=1}^{n} a_{ij} \cdot \mathtt{v}_j = b_j \quad (i = 1, \ldots m) \quad \text{and} \quad l_j \leq \mathtt{v}_j \leq u_j \quad (j = 1, \ldots n). \end{array}$$

The Linear programming-based reduction ρ_{LP} is trivially an instance of UBLP: To infer the tightest upper bound (resp. lower bound) on a variable $\mathtt{v}_k$ in a subpolyhedron $\langle \mathsf{l};\ \mathsf{i} \rangle$ instantiate UBLP with $c = 1$ (resp. $c = -1$) subject to the linear equalities l and the numerical bounds i. UBLP can be solved in polynomial time [6]. However, polynomial time algorithms for UBLP do not perform well in practice. The Simplex method [12], exponential in the worst-case, in practice performs a lot better than other known linear programming algorithms [30]. The Simplex algorithm works by visiting the *feasible bases* (informally, the vertexes) of the polyhedron associated with the constraints. At each step, the algorithm visits the adjacent basis (vertex) that maximizes the current value of the objective by the largest amount. The iteration strategy of the Simplex guarantees the convergence to a basis which exhibits the optimal value for the objective.

The advantages of using Simplex for ρ_{LP} are that: (i) it is well-studied and optimized; (ii) it is complete in $\mathbb{R}$, *i.e.* it finds the best solution over real numbers; and (iii) it guarantees that all the information is propagated at once: $\rho_{LP} \circ \rho_{LP} = \rho_{LP}$.

The drawbacks of using Simplex are that (i) the computation over machine floating point may introduce imprecision or unsoundness in the result; and (ii) the reduction ρ_{LP} requires to solve $2 \cdot n$ UBLP problems to find the lower bound and the upper bound for each of the n variables in an abstract state. We have observed (i) in our experiences (cf. Sect. 6). There exist methods to circumvent the problem at the price of extra computational cost, *e.g.* using arbitrary precision rationals, or a combination of machine floating arithmetic and precise

Algorithm 3 The reduction algorithm ρ_{BE}, parametrized by the oracle δ

input $\langle \mathsf{l};\ \mathsf{i}\rangle \in \mathsf{SubPoly}$, $\delta \in \mathcal{P}(\{\zeta \mid \zeta \text{ is a basis change}\})$

Put l into row echelon form. Call the result l'
let $\langle \mathsf{l}^*, \mathsf{i}^*\rangle = \langle \mathsf{l}', \mathsf{i}\rangle$
for all $\zeta \in \delta$ **do**
  $\mathsf{l}^* := \zeta(\mathsf{l}^*)$
  for all $\mathtt{v}_k + a_{k+1} \cdot \mathtt{v}_{k+1} + \cdots + a_n \cdot \mathtt{v}_n = b \in \mathsf{l}^*$ **do**
    $\mathsf{i}^* := \mathsf{i}^*[\mathtt{v}_k \mapsto \mathsf{i}^*(\mathtt{v}_k) \sqcap_{\mathsf{Intv}} [\![b - a_{k+1} \cdot \mathtt{v}_{k+1} + \cdots + a_n \cdot \mathtt{v}_n]\!](\mathsf{i}^*)]$
return $\langle \mathsf{l}^*, \mathsf{i}^*\rangle$

arithmetic. Even if (i) is solved, we observed that (ii) dominates the cost of the reduction, in particular in the presence of abstract states with a large number of variables: the $2 \cdot n$ UBLP problems are *disjoints* and there is no easy way to share the sequence of bases visited by the Simplex algorithm over the different runs of the algorithm for the same abstract state.

Basis exploration-based reduction We have developed a new reduction ρ_{BE}, less subject to the drawbacks from floating point computation than ρ_{LP}, which enables a better tuning of the precision/cost ratio than the Simplex. The basic ideas are: (i) to fix *ahead* of time the bases we want to explore; and (ii) to refine at each step the variable bounds. The reduction ρ_{BE}, parametrized by a set of changes of basis δ, is formalized by Algorithm 3. First, we put the initial set of linear constraints into triangular form (row echelon form). Then, we apply the basis changes in δ and we refine all the variables *in the basis*. With respect to ρ_{LP}, ρ_{BE} is faster: (i) the number of bases to explore is statically bounded; (ii) at each step, k variables may be refined at once.

In theory, ρ_{BE} is an abstraction of ρ_{LP}, in that it may not infer the *optimal* bounds on variables (it depends on the choice of δ). In practice, we found that ρ_{LP} is much more numerically stable and it can infer better bounds than ρ_{LP}. The reason is in the handling of numerical errors in the computation. Suppose we are seeking a (lower or upper) bound for a variable using the Simplex. If we detect a numerical error (*i.e.*, a huge coefficient in the exact arithmetic computation), the only sound solution is to stop the iterations, and return the current value of the objective function as the result. On the other hand, when we detect a numerical error in ρ_{BE}, we can just skip the current basis (abstraction), and move to the next one in δ.

We are left with the problem of defining δ. We have two instantiations for it: a linear explorer and combinatorial explorer. The algorithm in Sect. 9.3.3 of [13] may also be used when the all the variables are known to be positive.

Linear Explorer (δ_L) The linear bases explorer is based on the empirical observation that in most cases having some variable $\mathtt{v}_0$ in the basis and some other variable $\mathtt{v}_1$ out of the basis is enough to infer good bounds. The explorer generates a sequence of bases δ_L with the property that for each unordered pair of distinct variables $\langle \mathtt{v}_0, \mathtt{v}_1\rangle$, it exists $\zeta \in \delta_L$ such that $\mathtt{v}_0$ is

in the basis and $\mathtt{v}_1$ is not. The sequence δ_L is defined as $\delta_L = \{\zeta_i \mid i \in [0,n],\ \mathtt{v}_i \ldots \mathtt{v}_{(i+m-1) \bmod n} \text{ are in basis for } \zeta_i\}$.

Example 5. (Reduction with the linear explorer) Let the initial state be $\mathtt{s} = \langle \mathtt{v}_0 + \mathtt{v}_2 + \mathtt{v}_3 = 1, \mathtt{v}_1 + \mathtt{v}_2 - \mathtt{v}_3 = 0;\ \mathtt{v}_0 \in [0,2], \mathtt{v}_1 \in [0,3] \rangle$, so that $\delta_L = \{\{\mathtt{v}_0, \mathtt{v}_1\}, \{\mathtt{v}_1, \mathtt{v}_2\}, \{\mathtt{v}_2, \mathtt{v}_3\}, \{\mathtt{v}_3, \mathtt{v}_0\}\}$. The reduction $\rho_{BE}(\mathtt{s})$ contains the tightest bounds for $\mathtt{v}_2, \mathtt{v}_3$: $\langle \mathtt{v}_2 + \frac{1}{2} \cdot \mathtt{v}_0 + \frac{1}{2} \cdot \mathtt{v}_1 = 0, \mathtt{v}_3 + \frac{1}{2} \cdot \mathtt{v}_0 - \frac{1}{2} \cdot \mathtt{v}_1 = 0; \mathtt{v}_0 \in [0,2], \mathtt{v}_1 \in [0,3], \mathtt{v}_2 \in [0, \frac{5}{2}], \mathtt{v}_3 \in [-\frac{1}{2}, 1] \rangle$. □

Properties of δ_L are that: (i) each variable appears exactly m times in the basis; (ii) it can be implemented efficiently as the basis change from ζ_i to $\zeta_{i+1}, i \in [0, n-1]$ requires just one variable swap; (iii) in general it is not idempotent: it may be the case that $\rho_L \circ \rho_L \neq \rho_L$; (iv) the result may depend on the initial order of variables, as shown by the next example.

Example 6 (Incompleteness of the linear explorer). Let us consider an initial state $\mathtt{s} = \langle \mathtt{v}_0 + \mathtt{v}_1 + \mathtt{v}_2 = 0, \mathtt{v}_3 + \mathtt{v}_1 = 0;\ \mathtt{v}_2 \in [0,1], \mathtt{v}_3 \in [0,1] \rangle$. The reduced state $\rho_{BE}(\mathtt{s}) = \langle \mathtt{v}_3 + \mathtt{v}_1 = 0, \mathtt{v}_2 + \mathtt{v}_0 - \mathtt{v}_1 = 0; \mathtt{v}_1 \in [-1,0], \mathtt{v}_2 \in [0,1], \mathtt{v}_3 \in [0,1] \rangle$ does not contain the bound $\mathtt{v}_0 \in [-1,1]$. □

Combinatorial Explorer (δ_C) The combinatorial explorer δ_C systematically visits all the bases. It generates all possible combinations of `m` variables trying to minimize the number of swaps at each basis change. It is very costly, but it finds the best bounds for each variable: it visits all the bases, in particular the one where the optimum is reached. The main advantage with respect to the Simplex is a better tolerance to numerical errors. However it is largely impractical because of (i) the huge cost; and (ii) the negligible gain of precision w.r.t. the use of δ_L that it showed in our benchmark examples.

5 Hints

The inference power of SubPoly can be increased using *hints*. Hints are linear functionals associated with a subpolyhedron `s`. They represent some linear inequality that *may* hold in `s`, but that it is not explicitly represented by a slack variable, or that it is not been checked to hold in `s` yet.

Hints increase the precision of joins and widenings. Let `h` be an hint, let $\mathtt{s}_0$ and $\mathtt{s}_1$ two subpolyhedra, and let $\mathtt{b} = [\![\mathtt{h}]\!](\mathtt{s}_0) \sqcup_{\mathsf{Intv}} [\![\mathtt{h}]\!](\mathtt{s}_1)$. If $\mathtt{b} \neq \top_{\mathsf{Intv}}$, then $\mathtt{h} \in \mathtt{b}$ holds in both $\mathtt{s}_0$ and $\mathtt{s}_1$, so that the constraint can be safely added to $\mathtt{s}_0 \sqcup_S \mathtt{s}_1$. That helps recovering linear inequalities that may have been dropped by the Algorithm 1. The situation for widening is similar, with the main difference that the number of hints should be bounded, to ensure convergence. Hints can be automatically generated during the analysis or they can be provided by the user in the form of annotations. In our current implementation, we have three ways to generate hints, inspired by existing solutions in the literature: program text, templates and planar convex hull. They provide very powerful hints, but some of them may be expensive.

Assembly	Methods	Bounds Checked	SubPoly with ρ_{LP} Valid	%	Time	SubPoly with ρ_{BE} Valid	%	Time	Max Vars
mscorlib.dll	18 084	17 181	14 432	84.00	73:48 (3)	14 466	84.20	23:19 (0)	373
System.dll	13 776	11 891	10 225	85.99	58:15 (2)	10 427	87.69	14:45 (0)	140
System.Web.dll	22 076	14 165	13 068	92.26	24:41 (0)	13 078	92.33	6:33 (0)	182
System.Design.dll	11 419	10 519	10 119	96.20	26:07 (0)	10 148	96.47	5:18 (0)	73
Average				89.00			89.51		

Fig. 6. The experimental results of checking array creation and accesses in representative .Net assemblies. SubPoly is instantiated with two reductions: ρ_{LP} and ρ_{BE}. Time is in minutes. The number of methods that reached the timeout (two minutes) is in parentheses. The last column reports the maximum number of variables simultaneously related by a SubPoly abstract state.

Program text hints They introduce a new hint each time a guard or assume statement (user annotation) is encountered in the analysis. This way, properties that are obvious when looking at the syntax of the program will be proved. Also, every time a slack variable β is removed, $\texttt{info}(\beta)$ is added to the hints. This is useful in the realistic case when SubPoly is used in conjunction with a heap analysis which may introduce unwanted renamings.

Template hints They consider hints of fixed shape [28]. For instance, hints in the form $\texttt{x}_0 - \texttt{x}_1$ guarantee a precision at least as good as difference bounds matrices [24], provided that the reduction is complete.

Planar convex hull hint It materializes new hints by performing the planar convex hull of the subpolyhedra to join [29]. First, it projects the interval components on every two-dimensional plane (there are a quadratic number of such planes). Then it performs the convex hull of the resulting pair of rectangles (in constant time, since the number of vertexes is at most eight). The resulting new linear constraints are a sound approximation by construction. They can be safely added to the result of the join.

6 Experience

We have implemented SubPoly on top of Clousot, our modular abstract interpretation-based static analyzer for .Net [3]. A stand-alone version of the SubPoly library is available for download [19]. Clousot directly analyzes MSIL, a bytecode target for more than seventy compilers (including C#, Managed C++, VB.NET, F#). Prior to the numerical analysis Clousot performs a heap analysis and an expression recovery analysis [21]. Clousot performs *intra*-procedural analysis and it supports assume-guarantee reasoning via Foxtrot annotations [4]. Contracts are expressed directly in the language as method calls and are persisted to MSIL using the normal compilation process of the source language. Classes and methods may be annotated with class invariants, preconditions and postconditions. Preconditions are asserted at call sites and assumed at the method entry point. Postconditions are assumed at call sites and asserted at the method

exit point. `Clousot` also checks the absence of specific errors, *e.g.* out of bounds array accesses, null dereferences, buffer overruns, and divisions by zero.

Figure 6 summarizes our experience in analyzing array creations and accesses in four libraries shipped with `.Net`. The test machine is an ordinary 2.4Ghz dual core machine, running Windows Vista. The assemblies are directly taken from the `%WINDIR%\Microsoft\Framework\v2.0.50727` directory of the PC. The analyzed assemblies do not contain contracts (We are actively working to annotate the `.Net` libraries). On average, we were able to validate almost 89.5% of the proof obligations. We manually inspected some of the warnings issued for `mscorlib.dll`. Most of them are due to lack of contracts, *e.g.* an array is accessed using a method parameter or the return value of some helper method. However, we also found real bugs (dead code and off-by-one). That is remarkable considering that `mscorlib.dll` has been tested *in extenso*. We also tried SubPoly on the examples of [11, 27, 15, 16], proving all of them.

Reduction Algorithms We run the tests using the Simplex-based and the Linear explorer-based reduction algorithms. We used the Simplex implementation shipped with the Microsoft Automatic Graph Layout tool, widely tested and optimized. The results in Fig. 6 show that ρ_{LP} is significantly slower than ρ_{BE}, and in particular the analysis of five methods was aborted as it reached the two minutes time-out. Larger time-outs did not help.

SubPoly with the reduction ρ_{LP} validates less accesses than ρ_{BE}. Two reasons for that. First, it is slower, so that the analysis of some methods is aborted and hence their proof obligations cannot be validated. Second, our implementation of the Simplex uses floating point arithmetic which induces some loss of precision. In particular we need to read back the result (a `double`) into an interval of `int`s containing it. In general this may cause a loss of precision and even worse unsoundness. We experienced both of them in our tests. For instance the 39 "missing" proof obligations in `System.Web.dll` and `System.Design.dll` (validated using ρ_{BE}, but not with ρ_{LP}) are due to floating point imprecision in the Simplex. We have considered replacing a floating point-based Simplex with one using exact rationals. However, the Simplex has the tendency to generate coefficients with large denominators. The code we analyze contains many large constants which cause the Simplex to produce enormous denominators.

SubPoly with ρ_{BE} instantiated with the linear bases explorer perform very well in practice: it is extremely fast and precise. Our implementation uses 64 bits Rationals. When an arithmetic overflow is detected, we abstract away the current computation, *e.g.*, by removing the suitable row in the matrix representation. On the negative side, the result may depend on the variables order. A "bad" variable order may cause ρ_{BE} not to infer bounds tight enough. One solution is to iterate the application of ρ_{BE} (it is not idempotent). Other solutions are: (i) to reduce the number of variables by simplifying a subpolyhedron (less bases to explore); (ii) to mark variables which can be safely kept in the basis at all times: In the best case, only one basis needs to be explored. In the general case, it still makes the reduction more precise because the bases explored are more likely to give bounds on the variables.

Max Variables It is worth noting that even if `Clousot` performs an intra-procedural analysis, the methods we analyze may be very complex, and they may require tracking linear inequalities among many abstract locations. Abstract locations are produced by the heap analysis [20], and they abstract stack locations and heap locations. Figure 6 shows that it is not uncommon to have methods which require the abstract state to track more than 100 variables. One single method of `mscorlib.dll` required to track relations among 373 distinct variables. SubPoly handles it: the analysis with ρ_{BE} took a little bit more than a minute. To the best of our knowledge those performances in presence of so many variables are largely beyond current Poly implementations. For instance, in some preliminary study we tried to instantiate `Clousot` with the Poly library included in Boogie [2]. The results were quite disappointing: under the same experimental conditions (except for a 5 minutes time out), the analysis of `System.dll` took 257 minutes, and the time out was reached more than 20 times. We did not notice any remarkable gain of precision using Poly. Furthermore, Poly is concerned by floating points soundness issues, too [5].

7 Conclusions

We introduced SubPoly, a new numerical abstract domain based on the combination of linear equalities and intervals. SubPoly can track linear inequalities involving hundreds of variables. We defined the operations of the abstract domain (order, join, meet, widening) and two reduction operators (one based on linear programming and another based on basis exploration). We found Simplex-based reduction quite unsatisfactory for program analysis purposes: because of floating point errors the result may be too imprecise or worse, unsound. We introduced then the basis exploration-based reduction, in practice more precise and faster.

SubPoly precisely propagates linear inequalities, but it may fail to infer some of them at join points. Precision can be recovered using hints either provided by the programmer in the form of program annotations; or automatically generated (at some extra cost). SubPoly worked fine on some well known examples in literature that required the use of Poly. We tried SubPoly on shipped code, and we showed that it scales to several hundreds of variables, a result far beyond the capabilities of existing Poly implementations. **Acknowledgments** Thanks to L. Nachmanson for providing us the Simplex implementation. Thanks to M. Fähndrich, J. Feret, S. Gulwani, C. Popeea and J. Smans for the useful discussions.

References

1. R. Bagnara, P.M. Hill, and E. Zaffanella. The Parma Polyhedra Library. `http://www.cs.unipr.it/ppl/`.
2. M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for Object-Oriented programs. In *FMCO'05*.
3. M. Barnett, M. Fähndrich, and F. Logozzo. Managed contract tools. `http://research.microsoft.com/downloads`.

4. M. Barnett, M. A. Fähndrich, and F. Logozzo. Foxtrot and Clousot: Language agnostic dynamic and static contract checking for `.Net`. Technical Report MSR-TR-2008-105, Microsoft Research, 2008.
5. L. Chen, A. Miné, and P. Cousot. A sound floating-point polyhedra abstract domain. In *APLAS'08*.
6. V. Chvátal. *Linear Programming*. W. H. Freeman, 1983.
7. R. Clarisó and J. Cortadella. The octahedron abstract domain. In *SAS'04*.
8. P. Cousot. The calculational design of a generic abstract interpreter. In *Calculational System Design*. NATO ASI Series F. IOS Press, Amsterdam, 1999.
9. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL'77*.
10. P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *POPL '79*.
11. P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL '78*.
12. G. B. Dantzig. Programming in linear structures. Technical report, USAF, 1948.
13. J. Feret. *Analysis of mobile systems by abstract interpretation*. PhD thesis.
14. P. Ferrara, F. Logozzo, and M. A. Fähndrich. Safer unsafe code in .Net. In *OOPSLA'08*.
15. B. S. Gulavani, S. Chakraborty, A. V. Nori, and S. K. Rajamani. Automatically refining abstract interpretations. In *TACAS'08*.
16. S. Gulwani, K. Mehra, , and T. Chilimbi. Speed: Precise and efficient static estimation of program computational complexity. In *POPL'09*.
17. M. Karr. On affine relationships among variables of a program. *Acta Informatica*, 6(2):133–151, July 1976.
18. L. Khachiyan, E. Boros, K. Borys, K. M. Elbassioni, and M. Gurvich. Generating all vertices of a polyhedron is hard. In *SODA'06*.
19. V. Laviron and F. Logozzo. The Subpoly Library. `http://research.microsoft.com/downloads`.
20. F. Logozzo. Cibai: An abstract interpretation-based static analyzer for modular analysis and verification of Java classes. In *VMCAI'07*, 2007.
21. F. Logozzo and M. A. Fähndrich. On the relative completeness of bytecode analysis versus source code analysis. In *CC'08*.
22. F. Logozzo and M. A. Fähndrich. Pentagons: A weakly relational abstract domain for the efficient validation of array accesses. In *SAC'08*.
23. B. Meyer. *Object-Oriented Software Construction (2nd Edition)*. Professional Technical Reference. Prentice Hall, 1997.
24. A. Miné. The octagon abstract domain. In *WCRE 2001*.
25. M. Müller-Olm and H. Seidl. Precise interprocedural analysis through linear algebra. In *POPL'04*.
26. E. Rodríguez-Carbonell and D. Kapur. Automatic generation of polynomial invariants of bounded degree using abstract interpretation. *Sci. Comput. Program.*, 64(1), 2007.
27. S. Sankaranarayanan, F. Ivancic, and A. Gupta. Program analysis using symbolic ranges. In *SAS'07*.
28. S. Sankaranarayanan, H. B. Sipma, and Z. Manna. Scalable analysis of linear systems using mathematical programming. In *VMCAI'05*.
29. A. Simon, A. King, and J. Howe. Two variables per linear inequality as an abstract domain. In *LOPSTR'02*.
30. D. A. Spielman and S.-H. Teng. Smoothed analysis of algorithms: Why the simplex algorithm usually takes polynomial time. *J. ACM.*, 51(3), 2004.