Int J Softw Tools Technol Transfer (2011) 13:585–601
DOI 10.1007/s10009-011-0199-5

VMCAI 2009

SubPolyhedra: a family of numerical abstract domains for the (more) scalable inference of linear inequalities

Vincent Laviron · Francesco Logozzo

Published online: 24 May 2011

Abstract We introduce SubPolyhedra (SubPoly), a new family of numerical abstract domains to infer and propagate linear inequalities. The key insight is that the reduced product of linear equalities and intervals produces powerful yet scalable analyses. Abstract domains in SubPoly are as expressive as Polyhedra, but they drop some of the deductive power to achieve scalability. The cost/precision ratio of abstract domains in the SubPoly family can be fine-tuned according to the precision one wants to retain at join points, and the algorithm used to infer the tighter bounds on intervals. 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 the state-of-the-art implementations of Polyhedra.

Keywords Abstract interpretation · Abstract domains · Loop invariants · Numerical abstract domains · Static analysis

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, etc.) and programmer annotations in the form of preconditions, postconditions, object invariants and assertions ("contracts" [28]). 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 [13] introduced the Polyhedra numerical abstract domain (Poly). Poly infers 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 moving in two orthogonal directions: either only considering 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 $\pm \mathtt{x} \pm \mathtt{y} \leq c$) [29], TVPI ($a \cdot \mathtt{x} + b \cdot \mathtt{y} \leq c$) [36], Pentagons ($\mathtt{x} \leq \mathtt{y} \wedge a \leq \mathtt{x} \leq b$) [27], Stripes ($\mathtt{x} + a \cdot (\mathtt{y} + \mathtt{z}) > b$) [15] 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) [19,34] and methods to generate polynomial invariants, e.g. [22,31,32].

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:

V. Laviron
École Normale Supérieure, 45, rue d'Ulm, Paris, France
e-mail: Vincent.Laviron@ens.fr

F. Logozzo (✉)
Microsoft Research, Redmond, WA, USA
e-mail: logozzo@microsoft.com

```
class StringBuilder {
  int m_ChunkLength; char[] m_ChunkChars;
  // ...
  public void Append(int wb, int count) {
    Contract.Requires(wb >= 2 * count);
    if (count + m_ChunkLength > m_ChunkChars.Length)
(*)   CopyChars(wb, m_ChunkChars.Length - m_ChunkLength);
    // ... }

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

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

(i) *propagating* the given constraints:

$$\texttt{wb} \geq 2 \cdot \texttt{count}$$
$$\texttt{count} + \texttt{m\_ChunkLength} > \texttt{m\_ChunkChars.Length}$$

(ii) *deducing* the precondition for `CopyChars`:

$$\texttt{wb} \geq 2 \cdot (\texttt{m\_ChunkChars.Length} - \texttt{m\_ChunkLength})$$

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 does not propagate the second constraint (it involves three variables); Pentagons and Octagons cannot represent any of the constraints; Stripes can propagate both constraints, but because of the incomplete closure it cannot deduce the precondition. Bounded domains does the job, provided we fix *before* the analysis the template for 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 elevated: the analysis will be limited to few tens of variables.

1.1 SubPolyhedra

We propose a new family of numerical abstract domains, 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 [20], 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. The reason for that is that the pairwise join on LinEq and Intv is in general less precise than the join on Poly. To mitigate this loss of precision, we introduce a technique called hints [23], which enable recovering some of the precision. This technique is not limited to SubPoly, and indeed we show that several existing refinement techniques can be seen as a particular case of hints.

The cardinal operation for SubPoly is the join, which computes a compact yet precise upper approximation of two incoming abstract states. The join of SubPoly is parameterized by the: (i) the reduction algorithm, which propagates the information between LinEq and Intv; and (ii) the hints, which recover information lost at join points. Every instantiation of the (reduction, hints) produces a new abstract domain in the SubPoly family, allowing the fine tuning of the cost/precision ratio. The most imprecise yet fast abstract domain in the SubPoly family is the one in which the reduction is the simple identity (no interval is refined) and the no hints are used. The most precise yet expensive abstract domain is one where the reduction is a complete linear programming algorithm and the hints are the usual Poly join.

1.2 Reduction

Let us consider the example in Fig. 2, taken from [33]. 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 simplification of SubPoly propagates the interval, refining the

```
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 [33]. SubPoly infers the loop invariant $\texttt{x} - \texttt{i} = \texttt{y} - \texttt{j} \wedge \texttt{x} \geq 0$, propagates it and prove the assertion

```
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 [17]. SubPoly infers the loop invariant $x \leq y \leq 100 \cdot x \wedge z = 10 \cdot w$, propagates it out of the loop, and proves that the assertion is unreached

linear constraint to $y = j - i$. This is enough to prove the assertion (in conjunction with the if-statement guard). It is worth noting that unlike [33] SubPoly does not require any hypothesis on the order of variables to prove the assertion.

1.3 Join and hints

Let us consider the code in Fig. 3, taken from [17]. The loop invariant required to prove that the assertion is unreached (and hence that the program is correct) is $x \leq y \leq 100 \cdot x \wedge z = 10 \cdot w$. Without hints, SubPoly can only infer $z = 10 \cdot w$. *Template* hints, inspired by [34], are used to recover linear inequalities that are dropped by the imprecision of the join: In the example, the template is $x - y \leq b$, and the analysis automatically figures out that $b = 0$. *Planar Convex hull* hints, inspired by [36], 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 $y \leq 100 \cdot x$. It is worth noting that SubPoly does not need any of the techniques of [17] to infer the loop invariant.

2 Abstract interpretation

2.1 Abstract domains

We assume the concrete domain to be the complete Boolean lattice of environments, $\mathsf{C} = \langle \mathcal{P}(\Sigma), \subseteq, \emptyset, \Sigma, \cup, \cap \rangle$, where $\Sigma = [\mathtt{Vars} \to \mathbb{Z}]$. An abstract domain A is a tuple $\langle \bar{\mathsf{D}}, \gamma, \bar{\sqsubseteq}, \bar{\bot}, \bar{\top}, \bar{\sqcup}, \bar{\sqcap}, \triangledown, \rho, \sigma \rangle$. The set of abstract elements $\bar{\mathsf{D}}$ is related to the concrete domain by a *monotonic* concretization function $\gamma \in [\bar{\mathsf{D}} \to \bar{\mathsf{C}}]$. With an abuse of notation, we will not distinguish between an abstract domain and the set of its elements. The approximation order is $\bar{\sqsubseteq}$ is a sound approximation of the concrete order:

$$\forall \bar{\mathsf{d}}_0, \bar{\mathsf{d}}_1 \in \bar{\mathsf{D}}.\ \bar{\mathsf{d}}_0 \bar{\sqsubseteq} \bar{\mathsf{d}}_1 \Longrightarrow \gamma(\bar{\mathsf{d}}_0) \subseteq \gamma(\bar{\mathsf{d}}_1)$$

The smallest element is $\bar{\bot}$, the largest element is $\bar{\top}$. The join operator $\bar{\sqcup}$ satisfies:

$$\forall \bar{\mathsf{d}}_0, \bar{\mathsf{d}}_1 \in \bar{\mathsf{D}}.\ \bar{\mathsf{d}}_0 \bar{\sqsubseteq} \bar{\mathsf{d}}_0 \bar{\sqcup} \bar{\mathsf{d}}_1 \wedge \bar{\mathsf{d}}_1 \bar{\sqsubseteq} \bar{\mathsf{d}}_0 \bar{\sqcup} \bar{\mathsf{d}}_1$$

The meet operator $\bar{\sqcap}$ satisfies:

$$\forall \bar{\mathsf{d}}_0, \bar{\mathsf{d}}_1 \in \bar{\mathsf{D}}.\ \bar{\mathsf{d}}_0 \bar{\sqcap} \bar{\mathsf{d}}_1 \bar{\sqsubseteq} \bar{\mathsf{d}}_0 \wedge \bar{\mathsf{d}}_0 \bar{\sqcap} \bar{\mathsf{d}}_1 \bar{\sqsubseteq} \bar{\mathsf{d}}_1$$

The widening $\triangledown$ ensures the convergence of the fixpoint iterations, i.e. it satisfies:

(i) $\forall \bar{\mathsf{d}}_0, \bar{\mathsf{d}}_1 \in \bar{\mathsf{D}}.\ \bar{\mathsf{d}}_0 \bar{\sqsubseteq} \bar{\mathsf{d}}_0 \triangledown \bar{\mathsf{d}}_1 \wedge \bar{\mathsf{d}}_1 \bar{\sqsubseteq} \bar{\mathsf{d}}_0 \triangledown \bar{\mathsf{d}}_1$

(ii) for each sequence of abstract elements $\bar{\mathsf{d}}_0, \bar{\mathsf{d}}_1, \ldots, \bar{\mathsf{d}}_k$ the sequence defined by:
$\bar{\mathsf{d}}_0^{\triangledown} = \bar{\mathsf{d}}_0, \bar{\mathsf{d}}_1^{\triangledown} = \bar{\mathsf{d}}_0^{\triangledown} \triangledown \bar{\mathsf{d}}_1, \ldots, \bar{\mathsf{d}}_k^{\triangledown} = \bar{\mathsf{d}}_{k-1}^{\triangledown} \triangledown \bar{\mathsf{d}}_k$
is ultimately stationary.

In general, we do not require abstract elements to be in some canonical or closed form, i.e. there may exist $\bar{\mathsf{d}}_0, \bar{\mathsf{d}}_1 \in \mathsf{D}$, such that $\bar{\mathsf{d}}_0 \neq \bar{\mathsf{d}}_1$, but $\gamma(\bar{\mathsf{d}}_0) = \gamma(\bar{\mathsf{d}}_1)$. The *reduction* operator $\rho \in [\bar{\mathsf{D}} \to \bar{\mathsf{D}}]$ puts an abstract element into a (pseudo-) canonical form without adding or losing any information: $\forall \bar{\mathsf{d}}.\ \gamma(\rho(\bar{\mathsf{d}})) = \gamma(\bar{\mathsf{d}}) \wedge \rho(\bar{\mathsf{d}}) \bar{\sqsubseteq} \bar{\mathsf{d}}$. We do not require ρ to be idempotent. The *simplification* operator $\sigma \in [\bar{\mathsf{D}} \to \bar{\mathsf{D}}]$ removes redundancies in an abstract state. It may introduce some loss of precision: $\forall \bar{\mathsf{d}}.\ \gamma(\bar{\mathsf{d}}) \subseteq \gamma(\sigma(\bar{\mathsf{d}}))$.

In most of the literature, reduction and simplification are not given the status of lattice operation. However, several domains use internally some specific operations that give a more adapted representation of the abstract state, for a given operation. For instance, there is an operation on Octagons that is called *closure*, and which has the properties of a reduction operator. We believe that this is general enough to warrant adding two operators to the standard abstract domain definition. Of course, the identity is always a reduction operator and a simplification operator, so it can be defined even for domains which have no corresponding specific operation. Those operators are particularly important when the abstract elements considered are representations of mathematical objects, such that some objects have multiple equivalent representations.

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 or functional extensions.

2.2 Transfer functions

It is common practice for the implementation of an abstract domain A to provide three abstract transfer functions: one for the assignment, one for the handling of tests, and one to perform abstract checking. The abstract transfer function for assignment, $\mathsf{A}.\mathsf{assign}$, is an over-approximation of the states reached after the concrete assignment ($\mathbb{E}[\![\mathtt{E}]\!](\sigma)$ denotes the evaluation of the expression $\mathtt{E}$ in the state σ):

$$\forall \mathtt{x}, \mathtt{E}.\forall \bar{\mathsf{e}} \in \mathsf{A}.$$
$$\{\sigma[\mathtt{x} \mapsto v] \mid \sigma \in \gamma(\bar{\mathsf{e}}), \mathbb{E}[\![\mathtt{E}]\!](\sigma) = v\}$$
$$\subseteq \gamma(\mathsf{A.assign}(\bar{\mathsf{e}}, \mathtt{x}, \mathtt{E}))$$

The test abstract transfer function, A.test, filters the input states ($\mathbb{B}[\![\mathtt{B}]\!](\sigma)$ denotes the evaluation of a Boolean expression B in the state σ):

$$\forall \mathtt{B}.\forall \bar{\mathsf{e}} \in \mathsf{A}.\ \{\sigma \in \gamma(\bar{\mathsf{e}}) \mid \mathbb{B}[\![\mathtt{B}]\!](\sigma) = \mathit{true}\} \subseteq \gamma(\mathsf{A.test}(\bar{\mathsf{e}}, \mathtt{B})).$$

The abstract checking A.check verifies if an assertion A holds in an abstract state $\bar{\mathsf{e}}$. It has four possible outcomes: *true*, meaning that A holds in all the concrete states $\gamma(\bar{\mathsf{e}})$; *false*, meaning that !A holds in all the concrete states $\gamma(\bar{\mathsf{e}})$; *bottom*, meaning that the assertion is unreached; *top*, meaning that the validity of A cannot be decided in $\gamma(\bar{\mathsf{e}})$. Formally, A.check satisfies $\forall \mathtt{A}.\ \forall \bar{\mathsf{e}} \in \mathsf{A}$:

$$\mathsf{A.check}(\mathtt{A}, \bar{\mathsf{e}}) = v \Rightarrow \forall \sigma \in \gamma(\bar{\mathsf{e}}).$$
$$\mathbb{B}[\![\mathtt{A}]\!](\sigma) = v, v \in \{\mathit{true}, \mathit{false}\}$$
$$\mathsf{A.check}(\mathtt{A}, \bar{\mathsf{e}}) = \mathit{bot} \Rightarrow \gamma(\bar{\mathsf{e}}) = \emptyset$$
$$\mathsf{A.check}(\mathtt{A}, \bar{\mathsf{e}}) = \mathit{top} \Rightarrow \exists \sigma_0, \sigma_1 \in \gamma(\bar{\mathsf{e}}).$$
$$\mathbb{B}[\![\mathtt{A}]\!](\sigma_0) \neq \mathbb{B}[\![\mathtt{A}]\!](\sigma_1)$$

2.3 Intervals

The abstract domain of interval environments is $\langle \mathsf{Intv}, \gamma_{\mathsf{Intv}}, \dot{\sqsubseteq}_{\mathsf{Intv}}, \dot{\bot}_{\mathsf{Intv}}, \dot{\top}_{\mathsf{Intv}}, \dot{\sqcup}_{\mathsf{Intv}}, \dot{\sqcap}_{\mathsf{Intv}}, \dot{\nabla}_{\mathsf{Intv}} \rangle$. The abstract elements are maps from program variables to open intervals. The concretization of an interval environment $\bar{\mathsf{i}}$ is

$$\gamma_{\mathsf{Intv}}(\bar{\mathsf{i}}) = \{s \in \Sigma \mid \forall \mathtt{x} \in \mathrm{dom}(\bar{\mathsf{i}}).$$
$$\bar{\mathsf{i}}(\mathtt{x}) = [a, b] \wedge a \leq s(\mathtt{x}) \leq b\}.$$

The lattice operations are the functional extension of those in Fig. 4. The reduction and the simplification for intervals are the identity function. All the domain operations can be implemented in linear time.

2.4 Linear equalities

The abstract domain of linear *equalities* is $\langle \mathsf{LinEq}, \gamma_{\mathsf{LinEq}}, \bar{\sqsubseteq}_{\mathsf{LinEq}}, \bar{\bot}_{\mathsf{LinEq}}, \bar{\top}_{\mathsf{LinEq}}, \bar{\sqcup}_{\mathsf{LinEq}}, \bar{\sqcap}_{\mathsf{LinEq}} \rangle$. The elements are sets of linear equalities, their meaning is given by the set of concrete states which satisfy the constraints, i.e.

$$\gamma_{\mathsf{LinEq}} = \lambda \bar{\mathsf{I}}.\ \left\{ s \in \Sigma \mid \forall \left(\sum a_i \cdot \mathtt{x}_i = b \right) \in \bar{\mathsf{I}}.\ \sum a_i \cdot s(\mathtt{x}_i) = b \right\}.$$

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 satisfies the ascending chain condition, so that the join suffices to ensure analysis termination. The reduction and the simplification are just Gaussian elimination. The complexity of the domain operations is subsumed by the complexity of Gaussian elimination, which is cubic.

2.5 Polyhedra

The abstract domain of linear *inequalities* is $\langle \mathsf{Poly}, \gamma_{\mathsf{Poly}}, \bar{\sqsubseteq}_{\mathsf{Poly}}, \bar{\bot}_{\mathsf{Poly}}, \bar{\top}_{\mathsf{Poly}}, \bar{\sqcup}_{\mathsf{Poly}}, \bar{\sqcap}_{\mathsf{Poly}}, \bar{\nabla}_{\mathsf{Poly}} \rangle$. The elements are sets of linear inequalities, the concretization is the set of concrete states which satisfy the constraints, i.e.

$$\gamma_{\mathsf{Poly}} = \lambda \bar{\mathsf{p}}.\ \left\{ s \in \Sigma \mid \forall \left(\sum a_i \cdot \mathtt{x}_i \leq b \right) \in \bar{\mathsf{p}}.\ \sum a_i \cdot s(\mathtt{x}_i) \leq b \right\}.$$

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 and the simplification, respectively, infer the set of generators and remove 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. $\bar{\sqcap}_{\mathsf{Poly}}$), some require the geometrical representation (e.g. $\bar{\sqcup}_{\mathsf{Poly}}$), and some others require both (e.g. $\bar{\sqsubseteq}_{\mathsf{Poly}}$). The conversion between the two representations is exponential in the number of variables, and it cannot be done better [21].

Order:	$[a_1, b_1] \dot{\sqsubseteq}_{\mathsf{Intv}} [a_2, b_2] \iff a_1 \geq a_2 \wedge b_1 \leq b_2$
Bottom:	$[a, b] = \dot{\bot}_{\mathsf{Intv}} \iff a > b$
Top:	$[a, b] = \dot{\top}_{\mathsf{Intv}} \iff a = -\infty \wedge b = +\infty$
Join:	$[a_1, b_1] \dot{\sqcup}_{\mathsf{Intv}} [a_2, b_2] = [\min(a_1, a_2), \max(b_1, b_2)]$
Meet:	$[a_1, b_1] \dot{\sqcap}_{\mathsf{Intv}} [a_2, b_2] = [\max(a_1, a_2), \min(b_1, b_2)]$
Widening:	$[a_1, b_1] \dot{\nabla}_{\mathsf{Intv}} [a_2, b_2] =$
	$[\mathtt{if}\ a_1 > a_2\ \mathtt{then}\ a_2\ \mathtt{else}\ -\infty,$
	$\mathtt{if}\ b_1 < b_2\ \mathtt{then}\ b_2\ \mathtt{else}\ +\infty]$

Fig. 4 Lattice operations over single intervals

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.

3.1 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 an 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.

3.2 Domain structure

The elements of SubPoly belong to the reduced product LinEq $\times$ 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 $\mathtt{info}(\beta) = \sum a_i \cdot \mathtt{x}_i$).

3.3 Concretization

An element of SubPoly can be interpreted as a polyhedron by projecting out the slack variables: $\gamma_S^{\mathsf{Poly}} \in [\mathsf{SubPoly} \to \mathsf{Poly}]$ is

$$\gamma_S^{\mathsf{Poly}} = \lambda\langle \bar{\mathtt{l}};\ \bar{\mathtt{i}}\rangle.\ \pi_{\mathtt{Var_S}}(\bar{\mathtt{l}} \cup \{a \leq \mathtt{v} \leq b \mid \bar{\mathtt{i}}(\mathtt{v}) = [a, b]\}),$$

where π denotes the projection of variables in Poly. The concretization $\gamma_S \in [\mathsf{SubPoly} \to \mathcal{P}(\Sigma)]$ is then $\gamma_S = \gamma_{\mathsf{Poly}} \circ \gamma_S^{\mathsf{Poly}}$.

3.4 Approximation order

The order on SubPoly may be defined in terms of order over Poly. Given two SubPolyhedra $\bar{\mathtt{s}}_0, \bar{\mathtt{s}}_1$, the most precise order relation $\bar{\sqsubseteq}_S^*$ is

$$\bar{\mathtt{s}}_0 \bar{\sqsubseteq}_S^* \bar{\mathtt{s}}_1 \iff \gamma_S^{\mathsf{Poly}}(\bar{\mathtt{s}}_0) \bar{\sqsubseteq}_{\mathsf{Poly}} \gamma_S^{\mathsf{Poly}}(\bar{\mathtt{s}}_1).$$

However, $\bar{\sqsubseteq}_S^*$ may be too expensive to compute: it involves mapping SubPolyhedra in the dual representation of Poly. 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:

$$\begin{aligned}
&\langle \bar{\mathtt{l}}_0;\ \bar{\mathtt{i}}_0\rangle \bar{\sqsubseteq}_S \langle \bar{\mathtt{l}}_1;\ \bar{\mathtt{i}}_1\rangle \iff \\
&\quad \exists \theta.\ \mathtt{Var_S}(\langle \bar{\mathtt{l}}_0;\ \bar{\mathtt{i}}_0\rangle) \xrightarrow{\text{inj}} \mathtt{Var_S}(\langle \bar{\mathtt{l}}_1;\ \bar{\mathtt{i}}_1\rangle). \\
&\quad \forall \beta \in \mathtt{Var_S}(\langle \bar{\mathtt{l}}_0;\ \bar{\mathtt{i}}_0\rangle).\ \mathtt{info}(\beta) = \mathtt{info}(\theta(\beta)) \\
&\qquad \wedge \theta(\langle \bar{\mathtt{l}}_0;\ \bar{\mathtt{i}}_0\rangle) \dot{\sqsubseteq} \langle \bar{\mathtt{l}}_1;\ \bar{\mathtt{i}}_1\rangle.
\end{aligned}$$

In general, $\bar{\sqsubseteq}_S \subsetneq \bar{\sqsubseteq}_S^*$. In practice, $\bar{\sqsubseteq}_S$ 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 $\bar{\sqsubseteq}_S$ satisfactory in our experience.

One other important consequence of using a weak approximation order is that we are not always able to tell whether two abstract elements are actually equivalent representations of the same geometric shape. This is why, unlike some other domains like Poly, in SubPoly the elements do not correspond to a geometrical shape, even up to equivalence; there are some elements that correspond to the same polyhedron, but are not comparable with our weak ordering.

3.5 Bottom

An element of SubPoly is equivalent to bottom if after a reduction one of the two components is bottom:

$$\bar{\mathtt{s}} = \bot_S \iff \rho(\bar{\mathtt{s}}) = \langle \bar{\mathtt{l}}, \bar{\mathtt{i}}\rangle \wedge (\bar{\mathtt{i}} = \dot{\bot}_{\mathsf{Intv}} \vee \bar{\mathtt{l}} = \bot_{\mathsf{LinEq}}).$$

3.6 Top

An element of SubPoly is top if after the simplification both components are top:

$$\bar{\mathtt{s}} = \top_S \iff \sigma(\bar{\mathtt{s}}) = \langle \bar{\mathtt{l}}, \bar{\mathtt{i}}\rangle \wedge \bar{\mathtt{i}} = \dot{\top}_{\mathsf{Intv}} \wedge \bar{\mathtt{l}} = \top_{\mathsf{LinEq}}.$$

3.7 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 an element of SubPoly after that the reduction has inferred the tightest bounds: $[\![\sum a_i \cdot \mathtt{v}_i]\!]\ \langle \bar{\mathtt{l}};\ \bar{\mathtt{i}}\rangle = \textbf{let}\ \langle \bar{\mathtt{l}}^*;\ \bar{\mathtt{i}}^*\rangle = \rho(\langle \bar{\mathtt{l}};\ \bar{\mathtt{i}}\rangle)\ \textbf{in}\ \sum a_i \cdot \bar{\mathtt{i}}^*(\mathtt{v}_i)$.

3.8 Join

As with the order, one could define a most precise join operation by concretizing on Poly, doing the convex hull, then abstracting again to SubPoly. However, this is a very expensive operation, and the aim of SubPoly is to give faster, but potentially less precise, operations. So we define instead a specific join algorithm $\sqcup_S$ 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

Algorithm 1 The join $\sqcup_S$ on SubPolyhedra

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

let $\langle \bar{l}'_i;\ \bar{i}'_i\rangle = \langle \bar{l}_i;\ \bar{i}_i\rangle$
{Step 1. Propagate the information of the slack variables}
for all $\beta \in \mathtt{Var_S}(\bar{l}_i) \setminus \mathtt{Var_S}(\bar{l}_{\underline{i}})$ **do**
  $\langle \bar{l}'_{\underline{i}};\ \bar{i}'_{\underline{i}}\rangle := \langle \bar{l}'_{\underline{i}} \sqcap_{\mathsf{LinEq}} \{\beta = \mathtt{info}(\beta)\};\ \bar{i}'_{\underline{i}}\rangle$
{Step 2. Perform the point-wise join on the saturated operands}
let $\langle \bar{l}_\sqcup;\ \bar{i}_\sqcup\rangle = \rho(\langle \bar{l}'_0;\ \bar{i}'_0\rangle) \dot{\sqcup} \rho(\langle \bar{l}'_1;\ \bar{i}'_1\rangle)$
{Step 3. Hints: Recover the lost information }
let D_i be the linear equalities dropped from $\bar{l}'_i$ at the previous step
for all $\kappa \in D_i$ **do**
  if κ contains no slack variable **then**
    let $\bar{i}_{s_\kappa} = [\![\mathtt{s}_\kappa]\!]\ \langle \bar{l}'_{\underline{i}};\ \bar{i}'_{\underline{i}}\rangle$
    if $\bar{i}_{s_\kappa} \neq \top_{\mathsf{Intv}}$ **then**
      let β be a fresh slack variable
      $\langle \bar{l}_\sqcup;\ \bar{i}_\sqcup\rangle := \langle \bar{l}_\sqcup \sqcap_{\mathsf{LinEq}} \{\beta = \kappa\};\ \bar{i}_\sqcup \dot{\sqcap}_{\mathsf{Intv}} \{\beta = \bar{i}_{s_\kappa} \sqcup_{\mathsf{Intv}} [0,0]\}\rangle$
  else if κ contains exactly one slack variable β **then**
    let $\bar{i}_{s_\kappa} = [\![\mathtt{s}_\kappa]\!]\ \langle \bar{l}'_{\underline{i}};\ \bar{i}'_{\underline{i}}\rangle$
    if $\bar{i}_{s_\kappa} \neq \top_{\mathsf{Intv}}$ **then**
      $\langle \bar{l}_\sqcup;\ \bar{i}_\sqcup\rangle := \langle \bar{l}_\sqcup \sqcap_{\mathsf{LinEq}} \{\kappa\};\ \bar{i}_\sqcup \dot{\sqcap}_{\mathsf{Intv}} \{\beta = \bar{i}_{s_\kappa} \sqcup_{\mathsf{Intv}} \bar{i}_i(\beta)\}\rangle$
  else {κ contains strictly more than one slack variable}
    continue
return $\langle \bar{l}_\sqcup;\ \bar{i}_\sqcup\rangle$

operands of the join, but that were not preserved by the previous step. The join is defined by the Algorithm 1 (We let $\underline{0} = 1, \underline{1} = 0$). We illustrate it with examples.

Example 1 (Steps 1 & 2) Let us consider the code in Fig. 5a. After the assumption, the abstract states on the left branch and the right branch are, respectively, $\bar{\mathtt{s}}_0 = \langle \mathtt{x} - \mathtt{y} = \beta_0;\ \beta_0 \in [-\infty, 0]\rangle$ and $\bar{\mathtt{s}}_1 = \langle \mathtt{x} - \mathtt{y} = \beta_1;\ \beta_1 \in [-\infty, 5]\rangle$. 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: $\bar{\mathtt{s}}'_0 = \langle \mathtt{x} - \mathtt{y} = \beta_0 = \beta_1;\ \beta_0 \in [-\infty, 0]\rangle$ and $\bar{\mathtt{s}}'_1 = \langle \mathtt{x} - \mathtt{y} = \beta_1 = \beta_0;\ \beta_1 \in [-\infty, 5]\rangle$. Step 2 requires the reduction of the operands. The interval for β_1 (resp., β_0) in $\bar{\mathtt{s}}'_0$ (resp., $\bar{\mathtt{s}}'_1$) is refined: $\rho(\bar{\mathtt{s}}'_0) = \langle \mathtt{x} - \mathtt{y} = \beta_0 = \beta_1;\ \beta_0 \in [-\infty, 0], \beta_1 \in [-\infty, 0]\rangle$ and $\rho(\bar{\mathtt{s}}'_1) = \langle \mathtt{x} - \mathtt{y} = \beta_1 = \beta_0;\ \beta_0 \in [-\infty, 5], \beta_1 \in [-\infty, 5]\rangle$. The pairwise join gets the expected invariant: $\bar{\mathtt{s}}_\sqcup = \rho(\bar{\mathtt{s}}'_0) \dot{\sqcup} \rho(\bar{\mathtt{s}}'_1) = \langle \mathtt{x} - \mathtt{y} = \beta_0 = \beta_1;\ \beta_0 \in [-\infty, 5], \beta_1 \in [-\infty, 5]\rangle$. □

Example 2 (Non-trivial information for slack variables) Let us consider the code snippet in Fig. 5b. 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 $\mathtt{y} \leq \mathtt{z}$. □

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

```
if(...)
 { assume x == y; assume y <= z; }
else
 { assume x <= y; assume y == z; }
```
(b)

Fig. 5 Examples illustrating the need for Step 1 in the join algorithm

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

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

Fig. 6 Examples illustrating the need for the Step 3 in the join and the widening

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 LinEq × 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. 6a. The analysis of the two branches of the conditional produces the abstract states: $\bar{\mathtt{s}}_0 = \langle \mathtt{x} - 3 \cdot \mathtt{y} = 0;\ \dot{\top}_{\mathsf{Intv}}\rangle$ and $\bar{\mathtt{s}}_1 = \langle \mathtt{x} = 0, \mathtt{y} = 1;\ \mathtt{x} \in [0,0], \mathtt{y} \in [1,1]\rangle$. 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 the corresponding linear form can be bounded by the "other" branch. The linear form in D_0 is also bounded in the right branch: $[\![\mathtt{x} - 3 \cdot \mathtt{y}]\!](\bar{\mathtt{s}}_1) = [-3, -3]$ $(\neq \dot{\top}_{\mathsf{Intv}})$. Therefore, it is meaningful to add a slack variable β corresponding to this linear form to the result. The linear forms of D_1 cannot be bounded on the left branch, so they are discarded. The abstract state after the join is then $\bar{\mathtt{s}}_\sqcup = \langle \mathtt{x} - \mathtt{y} = \beta;\ \beta \in [-3, 0]\rangle$. □

Algorithm 2 The widening ∇_S on SubPolyhedra

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

let $\langle \bar{l}'_i ; \bar{i}'_i \rangle = \langle \bar{l}_i ; \bar{i}_i \rangle$
{Step 1. Propagate the information of the slack variables}
for all $\beta \in \mathtt{Var_S}(\bar{l}_0) \setminus \mathtt{Var_S}(\bar{l}_1)$ **do**
  $\langle \bar{l}'_0 ; \bar{i}'_0 \rangle := \langle \bar{l}'_0 \sqcap_{\mathsf{LinEq}} \{\beta = \mathtt{info}(\beta)\} ; \bar{i}'_0 \rangle$
{Step 2. Perform the point-wise widening}
let $\langle \bar{l}_\nabla ; \bar{i}_\nabla \rangle = \langle \bar{l}'_0 ; \bar{i}'_0 \rangle \dot{\nabla} \rho(\langle \bar{l}'_1 ; \bar{i}'_1 \rangle)$
{Step 3. Recover the lost information }
let D_0 be the linear equalities dropped from $\bar{l}'_0$ at the previous step
for all $\kappa \in D_0$ **do**
  if κ contains no slack variables **then**
    let $\bar{i}_{s_\kappa} = [\![\mathtt{s}_\kappa]\!] \langle \bar{l}'_1 ; \bar{i}'_1 \rangle$
    if $\bar{i}_{s_\kappa} \neq \top_{\mathsf{Intv}}$ **then**
      let β be a fresh slack variable
      $\langle \bar{l}_\nabla ; \bar{i}_\nabla \rangle := \langle \bar{l}_\nabla \sqcap_{\mathsf{LinEq}} \{\beta = \kappa\} ; \bar{i}_\nabla \dot{\sqcap}_{\mathsf{Intv}} \{\beta = [0, 0] \nabla \bar{i}_{s_\kappa}\} \rangle$
  else if κ contains exactly one slack variable β **then**
    let $\bar{i}_{s_\kappa} = [\![\mathtt{s}_\kappa]\!] \langle \bar{l}'_1 ; \bar{i}'_1 \rangle$
    if $\bar{i}_{s_\kappa} \neq \top_{\mathsf{Intv}}$ **then**
      $\langle \bar{l}_\nabla ; \bar{i}_\nabla \rangle := \langle \bar{l}_\nabla \sqcap_{\mathsf{LinEq}} \{\kappa\} ; \bar{i}_\nabla \dot{\sqcap}_{\mathsf{Intv}} \{\beta = \bar{i}_0(\mathtt{v}) \nabla \bar{i}_{s_\kappa}\} \rangle$
  else {κ contains strictly more than one slack variable}
    continue
return $\langle \bar{l}_\nabla ; \bar{i}_\nabla \rangle$

3.9 Meet

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

3.10 Widening

The algorithm for the widening 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 [30,35].

Example 4 (Refinement step for the widening) Let us consider the code snippet in Fig. 6b. The entry state to the loop is $\bar{\mathtt{s}}_0 = \langle \mathtt{i} - \mathtt{k} = 0 ; \dot{\top}_{\mathsf{Intv}} \rangle$. The state after one iteration is $\bar{\mathtt{s}}_1 = \langle \mathtt{i} - \mathtt{k} = 1 ; \dot{\top}_{\mathsf{Intv}} \rangle$. We apply the widening operator. Step 1 does not refine the states as there are no slack variables. The pairwise widening of Step 2 lose 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 $[\![\mathtt{i} - \mathtt{k}]\!] (\bar{\mathtt{s}}_1) = [1, 1]$ so that $\bar{\mathtt{s}}_\nabla = \langle \mathtt{i} - \mathtt{k} = \beta ; \ \beta \in [0, +\infty] \rangle$. □

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

Proof sketch Algorithm 2 ensures that the number of linear equalities at any step is at most the number of equalities in the first step. So there exists a point from which no more slack variables will be added. Existing slack variables may be renamed to fresh ones to avoid conflicts. In the definition of $\bar{\sqsubseteq}_S$ the renaming θ takes care of those. Up to the renaming, the widening is the pairwise widening, which is convergent and whose stability can be checked by the pairwise partial order. □

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` [2].

4.1 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):

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

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 an element of $\mathsf{SubPoly}\ \langle \bar{l} ; \bar{i} \rangle$ instantiate UBLP with $c = 1$ (resp., $c = -1$) subject to the linear equalities $\bar{l}$ and the numerical bounds $\bar{i}$.

UBLP can be solved in polynomial time [6]. However, polynomial time algorithms for UBLP do not perform well in practice. The Simplex method [14], exponential in the worst-case, in practice performs a lot better than other known linear programming algorithms [37]. The Simplex algorithm works by visiting the *feasible bases* (informally, the vertices) 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: $\rho_{\mathrm{LP}} \circ \rho_{\mathrm{LP}} = \rho_{\mathrm{LP}}$.

The drawbacks of using Simplex are that (i) the computation over machine floating point may introduce

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

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

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

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. 7). There exists 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 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.

4.2 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_{BE} 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 loss of precision in floating point computations or 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.

4.2.1 Linear explorer (δ_L)

The linear bases explorer is based on the empirical observation that in most cases, to infer the tightest bounds for some variable v_0, you need to have it in the basis while some other variable v_1 is out of the basis. Following this, the linear explorer generates a sequence of bases δ_{L} with the property that for each unordered pair of distinct variables $\langle \mathrm{v}_0, \mathrm{v}_1 \rangle$, there exists $\zeta \in \delta_{\mathrm{L}}$ such that v_0 is in the basis and v_1 is not. The sequence δ_{L} is defined as $\delta_{\mathrm{L}} = \{\zeta_i \mid i \in [0, n],\ \mathrm{v}_i, \ldots, \mathrm{v}_{(i+m-1) \bmod n} \text{ are in basis for } \zeta_i\}$.

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

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 ζ_{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 $\rho_{\mathrm{L}} \circ \rho_{\mathrm{L}} \neq \rho_{\mathrm{L}}$; (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 $\bar{\mathsf{s}} = \langle \mathrm{v}_0 + \mathrm{v}_1 + \mathrm{v}_2 = 0, \mathrm{v}_3 + \mathrm{v}_1 = 0;\ \mathrm{v}_2 \in [0, 1], \mathrm{v}_3 \in [0, 1] \rangle$. The reduced state $\rho_{\mathrm{BE}}(\bar{\mathsf{s}}) = \langle \mathrm{v}_3 + \mathrm{v}_1 = 0, \mathrm{v}_2 + \mathrm{v}_0 - \mathrm{v}_1 = 0; \mathrm{v}_1 \in [-1, 0], \mathrm{v}_2 \in [0, 1], \mathrm{v}_3 \in [0, 1] \rangle$ does not contain the bound $\mathrm{v}_0 \in [-1, 1]$. □

4.2.2 Combinatorial explorer (δ_C)

The combinatorial explorer δ_{C} systematically visits all the bases. It generates all possible combinations of $\mathtt{m}$ 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 relative loss of precision of the join operator incited us to search some ways to recover precision on imprecise operators. We discovered several of those, and they share some properties that led us to define a new concept called hints, which generalizes all of them as well as several known refining techniques.

Hints are precision improving operators which can be used to systematically refine and improve the precision of domain operations in abstract interpretation. Domain operations are either *basic* domain operations (e.g. $\curlyvee$ or $\curlywedge$) or their compositions (e.g. $\lambda(\bar{e}_0, \bar{e}_1, \bar{e}_2).\ (\bar{e}_0 \curlywedge \bar{e}_1) \curlyvee (\bar{e}_0 \curlywedge \bar{e}_2)$).

Definition 1 (Hint, $\mathbb{h}$) Let $\diamond \in [\mathsf{C}^n \to \mathsf{C}]$ be a concrete domain operation defined over a concrete domain $\langle \mathsf{C}, \sqsubseteq, \sqcup, \sqcap \rangle$. Let $\bar{\diamond} \in [\mathsf{A}^n \to \mathsf{A}]$ be the abstract counterpart for $\diamond$ defined over the abstract domain $\langle \mathsf{A}, \preceq, \curlyvee, \curlywedge \rangle$. A hint $\mathbb{h}_{\bar{\diamond}} \in [\mathsf{A}^n \to \mathsf{A}]$ is such that:

$\mathbb{h}_{\bar{\diamond}}(\bar{e}_0, \ldots, \bar{e}_{n-1}) \preceq \bar{\diamond}(\bar{e}_0, \ldots, \bar{e}_{n-1})$(Refinement)

$\diamond(\gamma(\bar{e}_0), \ldots, \gamma(\bar{e}_{n-1})) \sqsubseteq \gamma(\mathbb{h}_{\bar{\diamond}}(\bar{e}_0, \ldots, \bar{e}_{n-1}))$(Soundness).

The first condition states that $\mathbb{h}_{\bar{\diamond}}$ is a more precise operations than $\bar{\diamond}$. The second condition requires $\mathbb{h}_{\bar{\diamond}}$ to be a sound approximation of $\diamond$. An important property of hints is that they can be designed separately and the combined to obtain a more precise hint.

Lemma 1 (Hints combination) *If $\mathbb{h}^1_{\bar{\diamond}}$ and $\mathbb{h}^2_{\bar{\diamond}}$ are hints, then*

$$\mathbb{h}^{\curlywedge}_{\bar{\diamond}}(\bar{e}_0, \ldots, \bar{e}_{n-1}) = \mathbb{h}^1_{\bar{\diamond}}(\bar{e}_0, \ldots, \bar{e}_{n-1}) \curlywedge \mathbb{h}^2_{\bar{\diamond}}(\bar{e}_0, \ldots, \bar{e}_{n-1})$$

is a hint.

Proof sketch (Refinement) follows from the definition of $\curlywedge$. *(Soundness)* is because $\diamond(\gamma(\bar{e}_0), \ldots, \gamma(\bar{e}_{n-1})) \sqsubseteq \gamma(\mathbb{h}^1_{\bar{\diamond}}(\bar{e}_0, \ldots, \bar{e}_{n-1})) \sqcap \gamma(\mathbb{h}^2_{\bar{\diamond}}(\bar{e}_0, \ldots, \bar{e}_{n-1})) \sqsubseteq \gamma(\mathbb{h}^1_{\bar{\diamond}}(\bar{e}_0, \ldots, \bar{e}_{n-1}) \curlywedge \mathbb{h}^2_{\bar{\diamond}}(\bar{e}_0, \ldots, \bar{e}_{n-1}))$. □

The next theorem states that hints improve the precision of static analyses without introducing unsoundness and preserving termination:

Theorem 2 (Refinement of the abstract semantics) *Let $\mathbb{h}_{\triangledown}$ and $\mathbb{h}_{\curlyvee}$ be two hints refining, respectively, the widening and the abstract union, and let $\mathbb{h}_{\triangledown}$ be a widening operator. Let $\bar{\mathsf{s}}^*[\![\cdot]\!]$ be the abstract semantics obtained from $\bar{\mathsf{s}}[\![\cdot]\!]$ by replacing $\triangledown$ with $\mathbb{h}_{\triangledown}$ and $\curlyvee$ with $\mathbb{h}_{\curlyvee}$. Let $\mathtt{P}$ be a program. Then, $\forall \mathsf{e} \in \mathcal{P}(\Sigma).\forall \bar{\mathsf{e}} \in \mathsf{A}$.*

$\bar{\mathsf{s}}^*[\![\mathtt{P}]\!](\bar{\mathsf{e}}) \preceq \bar{\mathsf{s}}[\![\mathtt{P}]\!](\bar{\mathsf{e}})$ *(Refinement)*

$\mathsf{e} \subseteq \gamma(\bar{\mathsf{e}}) \Longrightarrow [\![\mathtt{P}]\!](\mathsf{e}) \subseteq \gamma(\bar{\mathsf{s}}^*[\![\mathtt{P}]\!](\bar{\mathsf{e}}))$ *(Soundness).*

Proof sketch The cases to consider are those for the conditional and the while loop. The conditional can be proven by structural induction. The while loop by instantiating the abstract fixpoint transfer theorem of [11]. □

5.1 Syntactic hints

Syntactic hints use some part of the program text to refine the operations of the abstract domain. They exploit user annotations to preserve as much information as possible in gathering operations (user-provided hints), and systematically improve the widening heuristics to find tighter loop invariants (thresholds hints).

5.1.1 User-provided hints

They are the easiest, and probably cheapest form of hints. First, we collect all the predicates appearing as assertions or as guards. Then, the gathering operations are refined by explicitly checking for each collected predicate `B`, if it holds for *all* the operands. If this is the case, `B` is added to the result. The predicate seeker $\mathsf{pred} \in [\mathtt{Stm} \to \mathcal{P}(\mathtt{BExp})]$ is defined in Fig. 7. User provided hints do not affect the termination of the widening as we can only add finitely many new predicates.

Lemma 2 (User-provided hints) *Let $\diamond \in \{\curlyvee, \triangledown\}$, and let $\mathtt{P}$ be a program. Then: (i) $\mathbb{h}^{\mathsf{pred}}_{\diamond}$ defined below is a hint; and (ii) $\mathbb{h}^{\mathsf{pred}}_{\triangledown}$ is a widening operator.*

$$\begin{aligned}\mathbb{h}^{\mathsf{pred}}_{\diamond}(\bar{\mathsf{e}}_0, \bar{\mathsf{e}}_1) = \text{let } S \\ = \{\mathtt{B} \in \mathsf{pred}(\mathtt{P}) \mid \mathsf{A.check}(\mathtt{B}, \bar{\mathsf{e}}_0) = \mathit{true} \\ \wedge\ \mathsf{A.check}(\mathtt{B}, \bar{\mathsf{e}}_1) = \mathit{true}\} \\ \text{in } \mathsf{A.test}(\textstyle\bigwedge_{\mathtt{B} \in S} \mathtt{B}, \diamond(\bar{\mathsf{e}}_0, \bar{\mathsf{e}}_1)).\end{aligned}$$

Proof sketch Note that (1) implies that $\mathsf{A.test}(\mathtt{b1} \wedge \mathtt{b2}, \bar{\mathsf{e}}) \preceq \mathsf{A.test}(\mathtt{b1}, \bar{\mathsf{e}})$, which is enough to prove *(Refinement)*. The soundness condition (1) of check guarantees that no inconsistent predicate is added to the result, implying *(Soundness)*. □

Example 7 (Refined SubPoly operations) In example of Fig. 8, $\mathsf{pred}(\mathtt{DomOp}) = \{\mathtt{x} \leq \mathtt{y}, 4 \leq \mathtt{x}, \mathtt{y} \leq 100 \cdot \mathtt{x}\}$. The refined domain operations keep the predicate $\mathtt{x} \leq \mathtt{y}$, which is stable among loop iterations, and hence is a loop invariant. □

$$\begin{aligned}\mathsf{pred}(\mathtt{skip;}) &= \emptyset \\ \mathsf{pred}(\mathtt{x = E;}) &= \emptyset \\ \mathsf{pred}(\mathtt{assert\ B;}) &= \mathsf{atomize}(\mathtt{B}) \\ \mathsf{pred}(\mathtt{assume\ B;}) &= \mathsf{atomize}(\mathtt{B}) \\ \mathsf{pred}(\mathtt{C\ C'}) &= \mathsf{pred}(\mathtt{C}) \cup \mathsf{pred}(\mathtt{C'}) \\ \mathsf{pred}(\mathtt{if(B)\ \{C\}else\ \{C'\}}) &= \mathsf{atomize}(\mathtt{B}) \cup \mathsf{pred}(\mathtt{C}) \cup \mathsf{pred}(\mathtt{C'}) \\ \mathsf{pred}(\mathtt{while(B)\ \{C\}}) &= \mathsf{atomize}(\mathtt{B}) \cup \mathsf{pred}(\mathtt{C}) \\ \\ \mathsf{atomize}(\mathtt{B}_1 \wedge \mathtt{B}_2) &= \mathsf{atomize}(\mathtt{B}_1) \cup \mathsf{atomize}(\mathtt{B}_2) \\ \mathsf{atomize}(\mathtt{B}_1 \vee \mathtt{B}_2) &= \mathsf{atomize}(\mathtt{B}_1) \cup \mathsf{atomize}(\mathtt{B}_2) \\ \mathsf{atomize}(\mathtt{B}) &= \{\mathtt{B}\} \quad \text{otherwise.}\end{aligned}$$

Fig. 7 The functions pred and atomize collect the atomic predicates in statements and Boolean expressions

We found user-provided hints very useful in Clousot, our abstract interpretation based static analyzer for .Net. Clousot analyzes methods in isolation, and supports assume/guarantee reasoning ("contracts" [28]) via executable annotations [2]. Precision in propagating and checking program annotations is crucial to provide a satisfactory user experience. User-provided hints help to reach this goal as the analyzer makes sure that at each joint point no user annotation is lost, if it is implied by the incoming abstract states. They make the analyzer more robust w.r.t. incompleteness of Υ or a buggy implementation which may cause Υ to return a more abstract element than the one predicted by the theory. The downside is that user-provided hints are syntactically based:

Example 8 (Fragility of user-provided hints) Let us consider again the code in Fig. 8. If we replace the assertion at $(*)$ with `if 10 <= x then assert 5 <= y`, then $\mathsf{pred}(\texttt{DomOp}) = \{10 \leq \texttt{x}, 5 \leq \texttt{y}\}$, so that $\mathbb{h}^{\mathsf{pred}}_{\nabla_{\mathsf{Poly}}}$ cannot figure out that $\texttt{x} \leq \texttt{y}$, and hence the analyzer cannot prove that the assertion is valid. Semantic hints (Sect. 5.2.3) will fix it. □

5.1.2 Thresholds hints

Widening with threshold has been introduced in [4] to improve the precision of standard widenings over non-relational or weakly relational domains. Roughly, the idea of a widening with thresholds is to stage the extrapolation process, so that before projecting a bound to the infinity, values from a set T are considered as candidate bounds. The set T can be either provided by the user or it can be extracted from the program text. The widening with thresholds is just another form of hint. Let $\bar{\texttt{e}}_0$ and $\bar{\texttt{e}}_1$ be abstract states belonging to some numerical abstract domain. Without loss of generality we can assume that the basic facts in $\bar{\texttt{e}}_0, \bar{\texttt{e}}_1$ are in the form $\texttt{p} \leq k$, where p is some polynomial. For instance $\texttt{x} \in [-2, 4]$ is equivalent to $\{-\texttt{x} \leq 2, \texttt{x} \leq 4\}$. The standard widening preserves the linear forms with stable upper bounds: $\nabla(\bar{\texttt{e}}_0, \bar{\texttt{e}}_1) = \{\texttt{p} \leq k \mid \texttt{p} \leq k_0 \in \bar{\texttt{e}}_0, \texttt{p} \leq k_1 \in \bar{\texttt{e}}_1, k = \text{if } k_1 > k_0 \text{ then } +\infty \text{ else } k_0\}$. Given a finite set of values T, threshold hints refine the standard widening by:

$$\mathbb{h}^T_\nabla(\bar{\texttt{e}}_0, \bar{\texttt{e}}_1) = \{\texttt{p} \leq k \mid \texttt{p} \leq k_0 \in \bar{\texttt{e}}_0, \texttt{p} \leq k_1 \in \bar{\texttt{e}}_1, k = \text{if } k_1 > k_0 \text{ then } \min\{t \in T \cup \{+\infty\} \mid k_1 \leq t\} \text{ else } k_0\}.$$

```
void DomOp()
{ int x = 0, y = 0;
  while (...)
  { if (...) { x++; y += 100; }
    else if (...)
     if (x >= 4) { x++; y++; }
   }
    (*) assert x <= y;
    assert y <= 100 * x;
}
```

Fig. 8 Example requiring user-provided hints

```
void LessThan() {
  int x = 0;
  while (x < 1000) {
    x++;
  }
}
```

(a) Narrowing

```
void NotEq() {
  int x = 0;
  while (x != 1000) {
    x++;
  }
}
```

(b) Thresholds

Fig. 9 Two programs to be analyzed with intervals. The iterations with widening infer the loop invariant $\texttt{x} \in [0, +\infty]$. In the first case, the narrowing step refines the loop invariant to $\texttt{x} \in [0, 1000]$. In the second case, the narrowing fails to refine it

Lemma 3 *$\mathbb{h}^T_\nabla$ is: (i) a hint; and (ii) a widening.*

Proof sketch *Refinement* and *Soundness* are a direct consequence of definition of $\mathbb{h}^T_\nabla$. Termination follows from the fact that T is finite. □

Example 9 (Widening with thresholds) Let us consider the code snippets in Fig. 9 to be analyzed with Intervals. In the both cases, the (post-)fixpoint is reached after the first iteration $\nabla([0, 0], [1, 1]) = [0, +\infty]$. In the first case, the invariant can be improved by a narrowing step to $\triangle([0, +\infty], [-\infty, 1{,}000]) = [0, 1{,}000]$ (see [9] for a definition of narrowing of Intv). In the second case, the narrowing is of no help as $\triangle([0, +\infty], \Upsilon([-\infty, 1{,}000], [1, 002, +\infty])) = [0, +\infty]$. A widening with Thresholds $T = \{1{,}000\}$ helps discovering the tightest loop invariant for both examples in one step as $\mathbb{h}^T_\nabla([0, 0], [1, 1]) = [0, 1{,}000]$. □

Please note that user-provided hints are of no help in the previous example, as $\mathsf{pred}(\texttt{NotEq}) = \{\texttt{x} \neq 1{,}000\}$ does not hold for all the operands of the widening.

We are left with problem of generating the set T of thresholds. A common practice in static analyzers is to have $T = \{-1, 0, 1\}$. A better solution is to have the user provide T, left as parameter of the analyzer. This is the approach of [4]. In Clousot we chose a slightly different solution, which consists in populating T with the constants appearing in the program text. Constants are fetched from the source using a function $\mathsf{const} \in [\texttt{Stm} \to \mathcal{P}(\texttt{int})]$ defined as one may expect. We found $\mathbb{h}^{\mathsf{const}}_\nabla$ very satisfactory. The hint $\mathbb{h}^{\mathsf{const}}_\nabla$: (i) helps inferring precise *numerical* loop invariants without requiring the extra iteration steps required for applying the narrowing; and (ii) improves the precision of the

analysis of code involving disequalities, e.g. Fig. 9b. A drawback of threshold hints is that the set T may grow too large, slowing down the convergence of the fixpoint iterations. In `Clousot`, we infer thresholds on a per-method basis, which helps maintaining the cardinality of T quite small.

5.2 Semantic hints

Semantic hints provide a more refined yet more expensive form of operator refinement. For instance, they exploit information in the abstract states to materialize constraints that were implied by the operands (saturation hints, die-hard hints and template hints) or they iterate the application of operators to get a more precise abstract state (reductive hints).

5.2.1 Saturation hints

A common way to design abstract interpreters is to build the abstract domain as a composition of basic abstract domains, which interact through a well-defined interface [5,12,18]. Formally, given two abstract domains $\mathsf{A}_0, \mathsf{A}_1$, the Cartesian product $\mathsf{A}^\times = \mathsf{A}_0 \times \mathsf{A}_1$ is still an abstract domain, whose operations are defined as the point-wise extension of those over A_0 and A_1. Let $\bar{\diamond}_i \in [\mathsf{A}_i^n \to \mathsf{A}_i]$, $i \in \{0, 1\}$, then

$$\begin{aligned}&\bar{\diamond}^\times((\bar{\mathsf{e}}_0^0, \bar{\mathsf{e}}_1^0), \ldots, (\bar{\mathsf{e}}_0^{n-1}, \bar{\mathsf{e}}_1^{n-1}))\\ &\quad = (\bar{\diamond}_0(\bar{\mathsf{e}}_0^0, \ldots, \bar{\mathsf{e}}_0^{n-1}), \bar{\diamond}_1(\bar{\mathsf{e}}_1^0, \ldots, \bar{\mathsf{e}}_1^{n-1}))\end{aligned}$$

The Cartesian product enables the modular design (and refinement) of static analyses. However, a naive design which does not consider the flow of information between the abstract elements may lead to imprecise analyses, as illustrated by the following example.

Example 10 (Cartesian join) Let us consider the abstract domain $\mathsf{Z} = \mathsf{Intv} \times \mathsf{LT}$, where $\mathsf{LT} = [\mathtt{Var} \to \mathcal{P}(\mathtt{Var})]$ is an abstract domain capturing the "*less than*" relation between *variables*. For instance, $\mathtt{x} < \mathtt{y} \wedge \mathtt{x} < \mathtt{z}$ is represented in LT by $[\mathtt{x} \mapsto \{\mathtt{y}, \mathtt{z}\}]$. The domain operations are defined as one may expect [27]. Let $\bar{\mathsf{z}}_0 = ([\mathtt{x} \mapsto [-\infty, 0], \mathtt{y} \mapsto [1, +\infty]], [\cdot])$ and $\bar{\mathsf{z}}_1 = ([\cdot], [\mathtt{x} \mapsto \{\mathtt{y}\}])$ be two elements of Z ($[\cdot]$ denotes the empty map). Then the Cartesian join loses all the information: $\curlyvee^\times(\bar{\mathsf{z}}_0, \bar{\mathsf{z}}_1) = ([\cdot], [\cdot])$. □

A common solution is: (i) saturate the operands; and (ii) apply the operation pairwise. The saturation materializes all the constraints implicitly expressed by the product abstract state. Let $\rho \in [\mathsf{A}^\times \to \mathsf{A}^\times]$ be a saturation (*a.k.a.* closure) procedure. Then the next lemma provides a systematic way to refine an operator $\bar{\diamond}^\times$.

Lemma 4 *The operator* $\mathbb{h}^\rho_{\bar{\diamond}^\times}$ *below is a hint.*

$$\begin{aligned}&\mathbb{h}^\rho_{\bar{\diamond}^\times}((\bar{\mathsf{e}}_0^0, \bar{\mathsf{e}}_1^0), \ldots, (\bar{\mathsf{e}}_0^{n-1}, \bar{\mathsf{e}}_1^{n-1}))\\ &\quad = \text{let } \bar{\mathsf{r}}^i = \rho(\bar{\mathsf{e}}_0^i, \bar{\mathsf{e}}_1^i) \text{ for } i \in 0, \ldots, n-1 \text{ in } \bar{\diamond}^\times(\bar{\mathsf{r}}^0, \ldots, \bar{\mathsf{r}}^{n-1}).\end{aligned}$$

Example 11 (Cartesian join, continued) The saturation of $\bar{\mathsf{z}}_0$ materializes the constraint $\mathtt{x} < \mathtt{y} : \bar{\mathsf{r}}_0 = ([\mathtt{x} \mapsto [-\infty, 0], \mathtt{y} \mapsto [1, +\infty], [\mathtt{x} \mapsto \{\mathtt{y}\}])$, and it leaves $\bar{\mathsf{z}}_1$ unchanged. The constraint $\mathtt{x} < \mathtt{y}$ is now present in both the operands, and it is retained by the pairwise join. □

It is worth noting that in general $\mathbb{h}^\rho_\nabla$ does not guarantee the convergence of the iterations, as the saturation procedure may re-introduce constraints which were abstracted away from the widening (e.g. Fig. 10 of [29]).

Saturation hints can provide very precise operations for Cartesian abstract interpretations: They allow the analysis to get additional precision by combining the information present in different abstract domains. The quality of the result depends on the quality of the saturation procedure. The main drawbacks of saturation hints are that: (i) the iteration convergence is not ensured, so that extra care should be put in the design of the widening; (ii) the systematic application of saturation may cause a dramatic slow-down of the analysis. In our experience with the combination of domains implemented in `Clousot`, we found that the slow-down introduced by saturation hints was too high to be practical. Die-hard hints, introduced in the next section, are a better solution to achieve precision without giving up scalability.

5.2.2 Die-hard hints

These hints are based on the observation that often the constraints that one wants to keep at a gathering point often appears explicitly in one of the operands. For instance in Example 10 the constraint $\mathtt{x} < \mathtt{y}$ is explicit in $\bar{\mathsf{z}}_1$, and implicit in $\bar{\mathsf{z}}_0$ (as $\mathtt{x} \le 0 \wedge 1 \le \mathtt{y} \implies \mathtt{x} < \mathtt{y}$). Therefore, $\mathtt{x} < \mathtt{y}$ holds for all the operands of the join so it is sound to add it to its result. Die-hard hints generalize and formalize it. They work in three steps: (i) apply the gathering operation, call the result $\bar{\mathsf{r}}$; (ii) collect the constraints C that are explicit in one of the operands, but are neither present nor implied by $\bar{\mathsf{r}}$; and (iii) add to $\bar{\mathsf{r}}$ all the constraints in C which are implied by *all* the operands. Formally:

$$\begin{aligned}\mathbb{h}^d_{(\bar{\diamond}, I)}(\bar{\mathsf{e}}_0, \bar{\mathsf{e}}_1) = \ &\text{let } \bar{\mathsf{r}} = \bar{\diamond}(\bar{\mathsf{e}}_0, \bar{\mathsf{e}}_1),\\ &C = \cup_{i \in I}\{\kappa \in \bar{\mathsf{e}}_i \mid \mathsf{A}.\mathsf{check}(\kappa, \bar{\mathsf{r}}) = \mathit{top}\}\\ &\text{let } S = \{\kappa \in C \mid \mathsf{A}.\mathsf{check}(\kappa, \bar{\mathsf{e}}_0) =\\ &\qquad \mathsf{A}.\mathsf{check}(\kappa, \bar{\mathsf{e}}_1) = \mathit{true}\}\\ &\text{in } \mathsf{A}.\mathsf{test}\left(\wedge_{\kappa \in S}\kappa, \bar{\mathsf{r}}\right).\end{aligned}$$

In defining the die-hard hint for ∇, one should pay attention to avoid loops which re-introduce a constraint that as been dropped by the widening. One way to do it is to have an

asymmetric hint, which restricts C only to the first operand (e.g. the candidate invariant):

Lemma 5 $\mathbb{h}^d_{(\Upsilon,\{0,1\})}$ *and* $\mathbb{h}^d_{(\nabla,\{0\})}$ *are hints and* $\mathbb{h}^d_{(\nabla,\{0\})}$ *is a widening.*

5.2.3 Computed hints

Hints can be inferred from the abstract states themselves. By looking at some properties of the elements involved in the operation, one can try to guess useful hints.

Lemma 6 (Computed hints) *Let* $\bar{\mathsf{e}}_0, \bar{\mathsf{e}}_1 \in \mathsf{A}$, $\Xi \in [\mathsf{A} \times \mathsf{A} \to \mathsf{A}]$ *a function which returns a set of likely bounds of* $\bar{\mathsf{e}}_0 \curlyvee \bar{\mathsf{e}}_1$. *Then* $\mathbb{h}^{\Xi}_{\curlyvee}$ *below is a hint.*

$$\mathbb{h}^{\Xi}_{\curlyvee}(\bar{\mathsf{e}}_0, \bar{\mathsf{e}}_1) = \text{let } S = \{\mathtt{B} \in \Xi(\bar{\mathsf{e}}_0, \bar{\mathsf{e}}_1) \mid \mathsf{A.check}(\mathtt{B}, \bar{\mathsf{e}}_0) = \mathit{true} \wedge \mathsf{A.check}(\mathtt{B}, \bar{\mathsf{e}}_1) = \mathit{true}\} \text{ in } \mathsf{A.test}(\textstyle\bigwedge_{\mathtt{B}\in S} \mathtt{B}, \bar{\mathsf{e}}_0 \curlyvee \bar{\mathsf{e}}_1).$$

Computed hints are useful when the abstract join $\curlyvee$ is not optimal. Otherwise, $\mathbb{h}^{\Xi}_{\curlyvee}$ is no more precise than $\bar{\sqcup}$. For instance, in a Galois connections-based abstract interpretation, $\bar{\sqcup}$ is optimal, in that it returns the most precise abstract element overapproximating the concrete union. As a consequence, no further information can be extracted from the operands. It is worth noting that in general $\mathbb{h}^{\Xi}_{\nabla}$ is not a widening. However, one can extend the arguments of the previous section to define an asymmetric hint $\mathbb{h}^{\Xi}_{\nabla}$.

The next two kinds of hints (template hints and 2D-convex hull hints) are examples of computed hints.

5.2.4 Template hints

Let $\mathsf{A.range} \in [\mathtt{Exp} \times \mathsf{A} \to \mathsf{Intv}]$ be a function that returns the range for an expression in some abstract state, e.g. it satisfies: $\forall \mathtt{E}.\ \forall \bar{\mathsf{e}} \in \mathsf{A}.\ \mathsf{A.range}(\mathtt{E}, \bar{\mathsf{e}}) = [l, u] \implies \forall \sigma \in \gamma(\bar{\mathsf{e}}).\ l \leq \mathbb{E}[\![\mathtt{E}]\!](\sigma) \leq u$. If $\mathsf{A.range}(\mathtt{E}, \bar{\mathsf{e}}_i) = [l_i, u_i]$ for $i \in \{0, 1\}$, then $\gamma(\sqcup_{\mathsf{Intv}}([l_0, u_0], [l_1, u_1]))$ is an upper bound for $\mathtt{E}$ in $\cup(\gamma(\bar{\mathsf{e}}_0), \gamma(\bar{\mathsf{e}}_1))$. As a consequence given a set P of polynomial forms, one can design the guessing function Ξ^P:

$$\Xi^P(\bar{\mathsf{e}}_0, \bar{\mathsf{e}}_1) = \{l \leq \mathtt{p} \leq u \mid \mathtt{p} \in P \wedge [l, u] = \sqcup_{\mathsf{Intv}}(\mathsf{A.range}(\mathtt{p}, \bar{\mathsf{e}}_0), \mathsf{A.range}(\mathtt{p}, \bar{\mathsf{e}}_1))\}.$$

The main difference between $\mathbb{h}^{\Xi^P}_{\curlyvee}$ and syntactic hints is that the bounds for the polynomials in P are *semantic*, as they are inferred from the abstract states and not from the program text. For instance, computed hints infer the right invariant in Example 8 using the set of templates $\mathit{Oct} \equiv \{\mathtt{x}_0 - \mathtt{x}_1 \mid \mathtt{x}_0, \mathtt{x}_1$ areprogram variables$\}$. In general, template hints with *Oct* refine SubPoly so to make it as precise as Oct.

```
void Foo() {
  int i = 2, j = 0;
  while (...) {
    if (...) { i = i + 4; }
    else     { i = i + 2; j++; } }
  assert  2 <= i - 2 * j; }
```

Fig. 10 Example requiring the use of 2D-convex hull hints to infer the right invariant, expressed by the assertion

5.2.5 2D-convex hull hints

New linear inequalities can be discovered at join points using the convex hull algorithm. For instance, the standard join on Poly is defined in that way [13]. However the convex hull algorithm requires an expensive conversion from a tableau of linear constraints to a set of vertices and generators, which causes the analysis time to blow up. A possible solution is to consider a planar convex hull, which computes possible linear relations between *pairs* of variables by: (i) projecting the Intvpart of the abstract states on all the two-dimensional planes; and (ii) computing the planar convex hull (of two rectangles, a particularly simple case) on those planes. Planar convex hull, combined with a smart representation of the abstract elements allows us to automatically discover complex invariants without giving up performances.

Example 12 (2D-convex hull) Let us consider the code in Fig. 10 taken from [13]. At a price of exponential complexity, Poly can infer the correct loop invariant, and prove the assertion correct. SubPoly refined with 2D-convex hull hints can prove the assertion, yet keeping a worst-case polynomial complexity [24]. □

5.2.6 Reductive hints

Intuitively, one way to improve the precision of a unary operator is to iterate its application [16]. However, an unconditional iteration may be source of unsoundness, as shown by the following example.

Example 13 (Unsoundness of unconditional iterations) Let $- \in [\mathsf{Intv} \to \mathsf{Intv}]$ be the operator which applies the unary minus to an interval. In general, $\forall n \in \mathbb{N}.\ \bar{\mathsf{e}} = -^{2n}(\bar{\mathsf{e}}) \neq -^{2n+1}(\bar{\mathsf{e}})$ so that the iterations are unstable. □

We say that a function f is *reductive* if $\forall x.\, f(x) \sqsubseteq x$; and *closing* if it is reductive and $\forall x.\, f(f(x)) = f(x)$.

Lemma 7 (Reductive hints) *Let* $\diamond \in [\mathsf{C} \to \mathsf{C}]$ *be a unary operator and* $\bar{\diamond} \in [\mathsf{A} \to \mathsf{A}]$ *its abstract counterpart. Let* $\diamond$ *be closing,* $\bar{\diamond}$ *be reductive, and* $n \geq 0$. *Then* $\mathbb{h}_{\bar{\diamond}}(\bar{\mathsf{e}}) = \bar{\diamond}^n(\bar{\mathsf{e}})$ *is a hint.*

Proof sketch (Refinement) follows from the definition. To prove *(Soundness)*, it is enough to prove that $\diamond(\gamma(\bar{\mathsf{e}})) \subseteq$

$\gamma(\bar{\diamond}^2(\bar{\mathsf{e}}))$. It holds as $\diamond(\gamma(\bar{\mathsf{e}})) = \diamond^2(\gamma(\bar{\mathsf{e}})) \subseteq \diamond(\gamma(\bar{\diamond}(\bar{\mathsf{e}}))) \subseteq \gamma(\bar{\diamond}^2(\bar{\mathsf{e}}))$. □

The main application of reductive hints is to improve the precision in handling the guards in non-relational abstract domains. Given a Boolean guard `B` and an abstract domain A, $\psi \equiv \lambda\bar{\mathsf{e}}.\ \mathsf{A.test}(\mathtt{B}, \bar{\mathsf{e}})$ is an abstract operator which satisfies the hypotheses of Lemma 7. Abstract compilation can be used to express ψ in terms of domain operations, their compositions and state update. Lemma 7 justifies the use of local fixpoint iterations to refine the result of the analysis.

Example 14 Let us consider the following Boolean expression:

$$\mathtt{b1} == \mathtt{b2} \wedge \mathtt{b2} == \mathtt{b3}$$

Its abstract compilation in an abstract domain $[\mathtt{Var} \to \{\mathsf{true}, \mathsf{false}, \top, \bot\}]$ is :

$$\psi \equiv \lambda\mathsf{b}.(\mathsf{b}[\mathtt{b1}, \mathtt{b2} \mapsto \mathsf{b}(\mathtt{b1}) \wedge \mathsf{b}(\mathtt{b2})]) \\ \dot{\wedge}(\mathsf{b}[\mathtt{b2}, \mathtt{b3} \mapsto \mathsf{b}(\mathtt{b2}) \wedge \mathsf{b}(\mathtt{b3})])$$

where $\dot{\wedge}$ denotes the point-wise extension of $\wedge$. In an initial abstract state $\mathsf{b}_0 = [\mathtt{b1}, \mathtt{b2} \mapsto \top; \mathtt{b3} \mapsto \mathsf{true}]$, $\psi(\mathsf{b}_0) = [\mathtt{b1} \mapsto \top; \mathtt{b2}, \mathtt{b3} \mapsto \mathsf{true}]$, and $\psi^2(\mathsf{b}_0) = [\mathtt{b1}, \mathtt{b2}, \mathtt{b3} \mapsto \mathsf{true}] = \psi^n(\mathsf{b}_0), n \geq 2$. □

6 Refinements for SubPolyhedra

6.1 Precision improvement: hints

Many of the hints presented above can be used to improve the precision of SubPoly. User-provided hints provide a simple but efficient way to deal with programs that are not too complicated, and the intervals part of SubPoly can of course use the threshold hints.

Saturation hints are impractical for SubPoly, but die-hard hints are very useful; indeed, step 3 of the join algorithm (1) can be seen as a particular case of die-hard hints, with the difference that the constraints marked as deleted in the join of the linear equalities domain might not have been actually present in the initial states, but have been introduced in place of an equivalent one by Karr's algorithm.

Both kinds of computed hints are useful, but in a different way : they can be used for tricky programs that require some complex reasoning, but in most cases the algorithm already returns a good enough result, and use of those hints only slows down the analysis. As an example, template hints can be used to manually set the invariants to infer on a complicated program. They can also be used to guarantee some minimal precision (e.g., at least the precision of octagons) if one can afford the extra time it costs. 2D-convex hull hints are useful in the case of pointer access validation, to infer bounds with non-unary coefficients between pairs of variables.

Reductive hints can also be used on cases involving reduction, because our reduction is not idempotent; however, reduction is rather expensive and there is no guarantee of an actual gain of precision in this case, so we do not use it in practice. Instead, when precision matters, we use reduction with the combinatorial explorer, which does have guarantees on the precision of the result.

6.2 Speed improvement: simplification

The simplification operator σ removes redundant information from an abstract element. It is required neither for soundness nor completeness nor to improve the precision of the analysis (unlike ρ), but it is cardinal to the implementation of scalable analyses. The simplification σ of an element of SubPoly $\langle \bar{\mathsf{l}};\ \bar{\mathsf{i}} \rangle$ reduces the number of variables in $\bar{\mathsf{l}}$, which is the more expensive domain, without losing any precision. It consists in the application of the following three rules:

(Const) If an equality $\mathtt{v} = b$ is detected, $\mathtt{v}$ is projected from $\bar{\mathsf{l}}$ and added to $\bar{\mathsf{i}}$;
(Slack) If a slack variable β does not appear in $\langle \bar{\mathsf{l}};\ \bar{\mathsf{i}} \rangle$, then it should be removed;
(Dep) If $\langle \bar{\mathsf{l}};\ \bar{\mathsf{i}} \rangle$ implies $\beta_0 + a \cdot \beta_1 = b$, then one between β_0 and β_1 can be removed.

The rationale behind (Const) is that constants are very expensive when represented in LinEq but very cheap if represented with Intv; (Slack) performs a kind of garbage collection, by removing slack variables β which are in the domain of $\langle \bar{\mathsf{l}};\ \bar{\mathsf{i}} \rangle$, but such that β does not appear in any of the constraints of $\bar{\mathsf{l}}$ and $\bar{\mathsf{i}}(\beta) = \top_{\mathsf{Intv}}$; (Dep) is justified by the fact that after refining the intervals for both variables, removing one of the slack variables does not change the concretization of the abstract element. (Const) is useful when we introduce a new slack variable; (Slack) helps reducing the number of slack variables after joins; and (Dep) is applied as a pre-step of the reduction, to reduce the number of variables and hence make it faster.

7 Experience

We have implemented SubPoly on the top of `Clousot`, our modular abstract interpretation-based static analyzer for `.Net`. `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 [26]. `Clousot` performs *intra*-procedural analysis and it supports assume-guarantee reasoning via `Foxtrot` annotations [2,15]. Contracts are expressed directly in the language as method calls and are persisted to

Table 1 The experimental results of checking array creation and accesses

Assembly	Bounds		Simplex ρ_{LP}			Linear explorer ρ_{BE}			Max
	Methods	Checked	Valid	%	Time	Valid	%	Time	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		

SubPoly is instantiated with two reductions ρ_{LP} and ρ_{BE}. Time is expressed in minutes, the time-out per method is set to 2 min (in parentheses). The last column reports the maximum number of variables related by an element of SubPoly

Table 2 The experimental results analyzing `mscorlib` with SubPoly and different semantic hints and no-reduction

SubPoly		SubPoly*			SubPoly* + $\mathbb{lh}_{Y}^{\Xi^{Oct}}$			SubPoly* + $\mathbb{lh}_{Y}^{\Xi^{2DCH}}$		
Valid	Time	Valid	Time	Slow down	Valid	Time	Slow down	Valid	Time	Slow down
14,230	4:29 (0)	14,432	20:22 (0)	4.5x	13,948	81:24 (20)	18.2x	14,396	36:33 (7)	8.1x

SubPoly* denotes SubPoly refined with $\mathbb{lh}_{\diamond}^{\text{pred}}$ and $\mathbb{lh}_{Y,\nabla}^{d}$. Computed hints significantly slow-down the analysis, but they are needed to reach a very low false alarm ratio

MSIL using the normal compilation process of the source language (cf. Appendix A). Classes and methods are 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.

Table 1 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 standard .NET directory of our PC. The shipped versions of the 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 [13,17,33], proving all of them.

7.1 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 Table 1 show that ρ_{LP} is significantly slower than ρ_{BE}, and in particular the analysis of five methods was aborted as it reached the 2-min 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 some 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 `float`) into an interval of `ints` 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. However, the result may depend on the variables order. A "bad" variable order may cause ρ_{BE} not to infer bounds tight enough. Possible solutions are: (i) to reduce the number of variables using σ (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.

7.2 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 [25], and they abstract stack locations and heap locations. Table 1 shows that it is not uncommon to have methods which requires 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 the presence of so many variables are largely beyond state-of-the-art Poly implementations.

7.3 Hints

Table 2 focuses on the analysis of `mscorlib` using SubPoly refined with hints and no-reduction. The first column in the table shows the results of the analysis with no hints. This is roughly equivalent to precisely propagating arbitrary linear equalities and intervals, with limited inference and no propagation of information between linear equalities and intervals. User-provided hints and die-hard hints add more inference power, at the price of a still acceptable slow-down. Computed hints (with Octagons and 2D-convex hull) further slow-down of the analysis, causing the analysis of various methods to time out. We manually inspected the analysis logs to investigate the differences. Ignoring the methods that timed-out, with respect to SubPoly*, $\langle \text{SubPoly}^*, \mathbb{h}^{\Xi^{Oct}}_{\Upsilon} \rangle$ and $\langle \text{SubPoly}^*, \mathbb{h}^{\Xi^{2DCH}}_{\Upsilon} \rangle$ report respectively 125 and 124 less false positives. Out of those, only 13 overlap.

One may wonder if computed hints are needed at all. We observed that, when considering annotated code (unfortunately, just a small fraction of the overall code base at the moment of writing), one needs to refine the operations of the abstract domains with hints in order to get a very low (and hence acceptable) false alarms ratio (around 0.5%). In fact, even if (relatively) rare, assertions as in Figs. 10 and 11b are present in real code. Thanks to the incremental structure of `Clousot`, we do not need to run SubPoly with *all* the hints on *all* the analyzed methods, but we can focus the highest precision only on the few methods which require it.

```
void AbsEl(int x)
{ if(...)  x  =-1;
  else     x  = 1;

  assert   x != 0;
}
```

(a)

```
void Transfer(int x, y)
{ assume 2 <= x <= 3;
  assume -1 <= y <= 1;

  int z = (x + y) * y;

  assert -2 <= z;
}
```

(b)

Fig. 11 Examples of orthogonal losses of precision in abstract interpretations: **a** a convex domain cannot represent $\mathtt{x} \neq 0$ and **b** a compositional transfer function does not infer the tightest lower bound for `z`

8 Conclusion

We introduced SubPoly, a new numerical abstract domain based on the combination of linear equalities and intervals. SubPoly can track linear inequalities involving hundred of variables. We defined the operations of the abstract domain (order, join, meet, widening); the simplification operator (to speed up the analysis); 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 existing Poly implementations.

Acknowledgments Thanks to Lev Nachmanson for providing us the Simplex implementation. Thanks to Manuel Fähndrich, Jérôme Feret, Corneliu Popeea for the useful discussions.

Appendix A: Foxtrot

`Foxtrot` is a language independent solution for contract specifications in `.Net`. It does not require any source language support or compiler modification. Preconditions and postconditions are expressed by invocations of static methods (`Contract.Requires` and `Contract.Ensures`) at the start of methods. Class invariants are contained in a method with an opportune name (`ObjectInvariant`) or tagged by a special attribute (`[ObjectInvariant]`). Dummy static methods are used to express meta-variables

such as e.g. `Contract.Old(x)` for the value in the pre-state of `x` or `Contract.WritableBytes(p)` for the length of the memory region associated with `p`. These contracts are persisted to MSIL using the standard source language compiler.

Contracts in the `Foxtrot` notation (using static method calls) can express arbitrary boolean expressions as preconditions and postconditions. We expect the expressions to be side effect free (and only call side-effect free methods). We use a separate purity checker to optionally enforce this [3].

A binary rewriter tool enables dynamic checking. It extracts the specifications and instruments the binary with the appropriate runtime checks at the applicable program points, taking contract inheritance into account. Most `Foxtrot` contracts can be enforced at runtime.

For static checking, `Foxtrot` contracts are presented to `Clousot` as simple `assert` or `assume` statements. E.g., a precondition of a method appears as an assumption at the method entry, whereas it appears as an assertion at every call-site.

Appendix B: Simplex algorithm

We recall some basic facts about the Simplex algorithm, and in particular the notion of basis. The Simplex algorithm finds the best solution to the problem:

$$\begin{array}{l}\text{maximize } c^{\mathrm{T}}\mathrm{v}\\ \text{subject to } A\,\mathrm{v} = b\end{array}$$

There may be also interval constraints ($l_i \leq \mathrm{v}_i \leq u_i$), but they are not important for the notion of basis. The problem above can be rewritten in matricial form as

$$(A|b)\begin{pmatrix}\mathrm{v}\\ -1\end{pmatrix} = 0.$$

We let $S = (A|b)$. There are infinitely many matrices S with the same space of solutions as $S\,\mathrm{v} = 0$, so we can make a few assumptions on S. First, we can use Gaussian elimination get an upper triangular matrix (row echelon form). Gaussian elimination updates the matrix by adding to a row a linear combination of the other rows of the matrix, which does not change the space of solutions; after several of such updates, the result is triangular. We can then remove all zero rows and divide each row by its leading coefficient (which is the left-most non-zero coefficient). These operations do not change the space of solutions. As Gaussian elimination guarantees that the leading coefficient of each row is strictly right of the leading coefficients of the rows above it, there is at most one leading coefficient in each column. The variables whose columns contain a leading coefficient are called *basic variables*, the ones whose columns do not contain a leading coefficient are called *non-basic variables*. The set of basic variables is the *basis*. It is also convenient to have the columns corresponding to basic variables containing only zeros except for a single one (the leading coefficient). This can be achieved from the previous matrix by a way similar to Gaussian elimination.

The Simplex algorithm starts with a matrix in this form, and at each iteration changes the basis. *Changing the basis* consists in choosing a basic variable, v_b, with the associated row r (the row whose leading coefficient is in the column for v_b), then choosing a non-basic variable v_n whose coefficient c in the row r is non-zero, then divide the row r by c, and use row operations to make all the other coefficients in the column for v_n zeros. The basis is now the previous basis plus v_n minus v_b (and so v_b is now non-basic and v_n is now basic). Note that the matrix may not be triangular anymore; this is not required for the simplex algorithm. The simplex algorithm uses the cost function c and the bounds l_i and u_i on variables to change the basis. Furthermore the simplex chooses the variables to ensures that c will not be zero; if this is not the case, a zero coefficient means that a particular exchange is not possible.

References

1. Bagnara, R., Hill, P.M., Zaffanella, E.: The Parma Polyhedra Library. http://www.cs.unipr.it/ppl/ (2011)
2. Barnett, M., Fähndrich, M.A., Logozzo, F.: Foxtrot and Clousot: Language Agnostic Dynamic and Static Contract Checking for `.Net`. Technical Report MSR-TR-2008-105. Microsoft Research (2008)
3. Barnett, M., Fändrich, M., Garbervetsky, D., Logozzo, F.: Annotations for (more) precise points-to analysis. In: IWACO 2007 (July 2007)
4. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: A static analyzer for large safety-critical software. In: PLDI'03. ACM Press, New York (2003)
5. Chang, B.-Y.E., Leino, K.R.M.: Abstract interpretation with alien expressions and heap structures. In: VMCAI'05. Springer, Berlin (2005)
6. Chvátal, V.: Linear Programming. W.H. Freeman, New York (1983)
7. Clarisó, R., Cortadella, J.: The octahedron abstract domain. In: SAS'04 (2004)
8. Cousot, P.: The calculational design of a generic abstract interpreter. In: Calculational System Design. NATO ASI Series F. IOS Press, Amsterdam (1999)
9. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL'77 (1977)
10. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: POPL'79 (1979)
11. Cousot, P., Cousot, R.: Abstract interpretation and application to logic programs. J. Logic Program. **13**(2–3), 103–179 (1992)
12. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: Combination of abstractions in the ASTRÉE static analyzer. In: ASIAN'06. LNCS, vol. 4435, pp. 272–300. Springer, Berlin (2006)

13. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: POPL'78 (1978)
14. Dantzig, G.B.: Programming in Linear Structures. Technical Report. USAF (1948)
15. Ferrara, P., Logozzo, F., Fähndrich, M.A.: Safer unsafe code in .Net. In: OOPSLA'08 (2008)
16. Granger, P.: Improving the results of static analyses programs by local decreasing iteration. In: FSTTCS'92. Springer, Berlin (1992)
17. Gulavani, B.S., Chakraborty, S., Nori, A.V., Rajamani, S.K.: Automatically refining abstract interpretations. In: TACAS'08 (2008)
18. Gulwani, S., McCloskey, B., Tiwari, A.: Lifting abstract interpreters to quantified logical domains. In: POPL'08. ACM Press, New York (2008)
19. Gulwani, S., Srivastava, S., Venkatesan, R.: Program analysis as constraint solving. In: PLDI'08 (2008)
20. Karr, M.: On affine relationships among variables of a program. Acta Inform. **6**(2), 133–151 (1976)
21. Khachiyan, L., Boros, E., Borys, K., Elbassioni, K.M., Gurvich, M.: Generating all vertices of a polyhedron is hard. In: SODA'06 (2006)
22. Kovács, L.: Reasoning algebraically about p-solvable loops. In: TACAS'08. Springer, Berlin (2008)
23. Laviron, V., Logozzo, F.: Refining abstract interpretation-based static analyses with hints. In: APLAS'09 (2009)
24. Laviron, V., Logozzo, F.: Subpolyhedra: a (more) scalable approach to infer linear inequalities. In: VMCAI'09 (2009)
25. Logozzo, F.: Cibai: an abstract interpretation-based static analyzer for modular analysis and verification of Java classes. In: VMCAI'07 (2007)
26. Logozzo, F., Fähndrich, M.A.: On the relative completeness of bytecode analysis versus source code analysis. In: CC'08 (2008)
27. Logozzo, F., Fähndrich, M.A.: Pentagons: A weakly relational abstract domain for the efficient validation of array accesses. In: SAC'08 (2008)
28. Meyer, B.: Object-Oriented Software Construction, 2nd ed., Professional Technical Reference. Prentice Hall, Upper Saddle River (1997)
29. Miné, A.: The octagon abstract domain. In: WCRE 2001 (2001)
30. Miné, A.: Weakly Relational Numerical Abstract Domains. PhD Thesis. École Polythechnique (2004)
31. Müller-Olm, M., Seidl, H.: Precise interprocedural analysis through linear algebra. In: POPL'04 (2004)
32. Rodríguez-Carbonell, E., Kapur, D.: Automatic generation of polynomial invariants of bounded degree using abstract interpretation. Sci. Comput. Program. **64**(1), 54–75 (2007)
33. Sankaranarayanan, S., Ivancic, F., Gupta A.: Program analysis using symbolic ranges. In: SAS'07 (2007)
34. Sankaranarayanan, S., Sipma, H.B., Manna, Z.: Scalable analysis of linear systems using mathematical programming. In: VMCAI'05 (2005)
35. Simon, A.: Value-Range Analysis of C Programs. Springer, New York (2008)
36. Simon, A., King, A., Howe, J.: Two variables per linear inequality as an abstract domain. In: LOPSTR'02 (2002)
37. Spielman, D.A., Teng, S.-H.: Smoothed analysis of algorithms: Why the simplex algorithm usually takes polynomial time. J. ACM **51**(3), 385–463 (2004)