Refining Abstract Interpretation-Based Static Analyses with Hints

Vincent Laviron[1] and Francesco Logozzo[2]

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

Abstract. We focus our attention on the loss of precision induced by abstract domain *operations*. We introduce a new technique, *hints*, which allows us to systematically refine the operations defined over elements of an abstract domain. We formally define hints in the abstract interpretation theory, we prove their soundness, and we characterize two families of hints: syntactic and semantic. We give some examples of hints, and we provide our experience with hints in `Clousot`, our abstract interpretation-based static analyzer for `.Net`.

1 Introduction

The three main elements of an abstract interpretation are: (i) the abstract elements (*"which properties am I interested in?"*); (ii) the abstract transfer functions (*"which is the abstract semantics of basic statements?"*); and (iii) the abstract operations (*"how do I combine the abstract elements?"*).

The loss of precision induced by the abstract elements is exemplified by Fig. 1(a). The assertion cannot be proved using only convex numerical abstract domains such as Intervals [4], Pentagons [18], Octagons [23] or even Polyhedra [8]. The reason for that is that the most precise property at the join point $\mathtt{x} == -1 \vee \mathtt{x} == 1$ cannot be exactly represented in any of those domains. For instance Intervals (Intv) approximate it with $-1 \leq \mathtt{x} \leq 1$, so that the fact that $\mathtt{x} \neq 0$ is lost. Many techniques have been proposed to overcome this problem. They essentially rely on the refinement of the *elements* of the abstract domain. Solutions include trace partitioning [16,20,9], domain completion [5], powerset construction [1,19], and abstract domain extension [25]. Abstract transfer functions may introduce an orthogonal loss of precision. For instance, in Fig. 1(b) the expression initializer for `z` is in a quadratic form. Thus no linear numerical abstract domain can precisely capture the relation between `x`, `y` and `z`. Standard domain refinements are of no help. A rough transfer function can simply abstract away `z`. A more precise one can approximate `z` with an interval. However, a compositional evaluation of the expressions which mimics the concrete one is not precise enough to discharge the assertion $-2 \leq \mathtt{z}$. Several authors suggested methods to infer optimal transfer functions in particular settings as, *e.g.*, constraint matrices [24], shape analysis [26] or constant propagation [3].

Z. Hu (Ed.): APLAS 2009, LNCS 5904, pp. 343–358, 2009.

```
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)

```
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;
}
```

(c)

Fig. 1. Examples of orthogonal losses of precision in abstract interpretations: (a) a convex domain cannot represent $\mathtt{x} \neq 0$; (b) a compositional transfer function does not infer the tightest lower bound for $\mathtt{z}$; and (c) the standard domain operations on Polyhedra are not precise enough to infer the loop invariant $\mathtt{x} \leq \mathtt{y}$

Surprisingly enough, the refinement of the *operations* over abstract elements has been widely ignored in the literature (with the exceptions of [14,1,11,12] which however focused their attention just on the widening operator).

Example. Let us consider the code snippet in Fig. 1(c). In order to prove the assertions valid, the static analysis should infer the loop invariant $\mathtt{x} \leq \mathtt{y} \leq 100 \cdot \mathtt{x}$. Different abstract domains infer different invariants. The Octagon abstract domain (Oct) is a *weakly relational* domain which captures properties in the form of $\pm\mathtt{x} \pm \mathtt{y} \leq k$. It infers the loop invariant: $0 \leq \mathtt{x} \wedge 0 \leq \mathtt{y} \wedge \mathtt{x} \leq \mathtt{y}$. The Polyhedra abstract domain (Poly) is a *fully relational* domain. It can infer and represent arbitrary linear inequalities: Abstract elements are in the form $\sum_i a_i \cdot \mathtt{x}_i \leq k$. As a consequence one expects Poly to always be more precise than Oct. However, when applied to the example, Poly infers the loop invariant: $0 \leq \mathtt{x} \wedge 0 \leq \mathtt{y} \wedge \mathtt{y} \leq 100 \cdot \mathtt{x}$: Even if Poly can *exactly* represent the constraint $\mathtt{x} \leq \mathtt{y}$, it fails inferring it! This is quite surprising. The reason for that should be found in the widening operators over the two domains. The (standard) widening over Octagons explicitly seeks an upper bound for the difference $\mathtt{x} - \mathtt{y}$ (in the example 0). The (standard) widening over Polyhedra preserves the inequalities that are stable over two loop iterations. In the example, the constraint $\mathtt{x} \leq \mathtt{y}$, even if implied by the abstract states to be widened, is never materialized. Therefore, the state after widening does not contain it either. Intuitively, in order to obtain the most precise loop invariant for DomOp, one needs to refine the widening operator for Poly to be at least as precise as the one for Oct. One way to refine the widening is by remarking that the predicate $\mathtt{x} \leq \mathtt{y}$ appears as a condition of some assertion, and then trying to explicitly materialize it. Another possible refinement is by seeking upper bounds for the expression $\mathtt{x} - \mathtt{y}$. The first is an example of *syntactic hint*. The latter is an example of *semantic hint*. Those observations can be extended and generalized to other abstract domain operators.

The case for hints. The main goal for a static analysis designer is the precision/speed trade-off. To achieve it, the common practice is to drop some of

the expressive power of the analysis while maximizing the inference power. In `Clousot`, our static analyzer for .`Net`, we needed additional flexibility. `Clousot` is mainly used to validate code contracts expressed by users in form of pre-conditions, post-conditions and object invariant. First, we observed that usual weakly-relational abstract domains are not precise enough to be used in a modular checker: for instance, it is often the case that the argument to establish an "easy" precondition (*e.g.*, $\mathtt{x} \leq \mathtt{y}$) at the call site involves a complex reasoning between several linear inequalities which require expensive abstract domains as Poly. Second, we needed `Clousot` to be adaptable, in that it can either run in an interactive environment (faster, but with more noise) or on a build machine overnight (slower, but much more precise). As a consequence, we took a different direction in the design of the abstract domains in `Clousot`: we retained the *expressive* power while we gave up some of the *inference* (*e.g.*, Pentagons [18] and Subpolyhedra [17]). Hints, introduced in this paper, are an orthogonal to the abstract domain, and they allow us to incrementaly increase the precision of the analysis by refining the transfer functions.

Contribution. We introduce hints, a new technique which allows us to systematically refine static analyses. The main ideas of hints are: (i) to have a separate module to figure out which constraints or families of constraints are of interest for the analysis; and (ii) to use such a module to refine the *operations* of the abstract domain. The main difference with related works on automatic refinement of static analyses is that hints refine the *operations* over abstract elements and *not* the elements themselves nor the transition relations. The main advantages of hints are that: (i) they enable an easy refinement of static analyses; (ii) they enable a fine-tuning of the cost/performance ratio; (iii) they make the analysis more robust with respect to implementation-related precision bugs. Hints are useful when the abstract operations are not complete w.r.t. the concrete ones, which is often the case in practice.

We formalize hints using the abstract interpretation theory, and we prove them correct w.r.t. a generic abstract interpreter. We characterize syntactic (user-defined, thresholds) and semantic hints (saturation, die-hard, computed, reductive). We show how they generalize existing techniques as, *e.g.*, widening with thresholds [2]. We applied hints to SubPolyhedra (SubPoly), a new, very efficient numerical abstract domain to propagate arbitrary linear inequalities. SubPoly has the same expressive power as Poly, but drops some of the inference to achieve scalability. Hints allow SubPoly to recover precision without giving up performances. Hints are implemented in `Clousot`, our static analyzer for .NET available at [21].

2 Abstract Interpretation Frameworks

Abstract interpretation is a general theory of approximations which formalizes the intuition that the semantics of a program is more or less precise depending on the observation level. In particular, the static analysis of a program is a semantics precise enough to capture the properties of interest and coarse enough

to be computable. The *concrete* and *abstract* semantics of a program are defined as fixpoints respectively over a concrete and an abstract domain. The concrete and the abstract domains are related by a soundness relation, which induces the soundness of the abstract semantics [6].

Static approximation: Abstract Domain. In the Galois connections approach to abstract interpretation [4], the concrete domain and the abstract domain are assumed to be two *complete* lattices, respectively $\langle \mathsf{C}, \sqsubseteq, \sqcup \rangle$ and $\langle \mathsf{A}, \bar{\sqsubseteq}, \bar{\sqcup} \rangle$. The soundness relation is expressed by a pair of monotonic functions $\langle \alpha, \gamma \rangle$, such that $\forall \mathsf{e} \in \mathsf{C}. \forall \bar{\mathsf{e}} \in \mathsf{A}.\ \alpha(\mathsf{e}) \bar{\sqsubseteq} \bar{\mathsf{e}} \iff \mathsf{e} \sqsubseteq \gamma(\bar{\mathsf{e}})$. In such a setting, the abstract join operator $\bar{\sqcup}$ is optimal in that: $\forall \mathsf{e}_1, \mathsf{e}_2 \in \mathsf{C}.\ \alpha(\sqcup(\mathsf{e}_1, \mathsf{e}_2)) = \bar{\sqcup}(\alpha(\mathsf{e}_1), \alpha(\mathsf{e}_2))$ [5]. In practice, most analyses do not require the existence of the *best* approximation for concrete elements, a sound approximation suffices. For instance, there is no best polyhedron approximating the set of concrete points $B = \{(\mathtt{x}, \mathtt{y}) \in \mathbb{R}^2 \mid \mathtt{x}^2 + \mathtt{y}^2 \leq 1\}$. However, any polyhedron including B is a sound abstraction. In the relaxed form of abstract interpretation [6], the abstract domain is not required to be complete under $\bar{\sqcup}$. It is simply a *pre-order* $\langle \mathsf{A}, \preceq, \curlyvee, \curlywedge \rangle$. The soundness relation is expressed by a monotonic concretization function $\gamma \in [\mathsf{A} \to \mathsf{C}]$, *i.e.*, no abstraction function is required. The abstract union $\curlyvee$ gathers together the information flowing from incoming edges. It is not required to be the *least* upper bound (which may not exist at all): $\forall \bar{\mathsf{e}}_0, \bar{\mathsf{e}}_1 \in \mathsf{A}.\ \bar{\mathsf{e}}_0 \preceq \bar{\mathsf{e}}_0 \curlyvee \bar{\mathsf{e}}_1 \wedge \bar{\mathsf{e}}_1 \preceq \bar{\mathsf{e}}_0 \curlyvee \bar{\mathsf{e}}_1$. It is a sound, but not *optimal*, approximation of the concrete join: $\forall \bar{\mathsf{e}}_0, \bar{\mathsf{e}}_1 \in \mathsf{A}.\ \sqcup(\gamma(\bar{\mathsf{e}}_0), \gamma(\bar{\mathsf{e}}_1)) \sqsubseteq \gamma(\curlyvee(\bar{\mathsf{e}}_0, \bar{\mathsf{e}}_1))$. The abstract intersection returns a common lower bound for the operands, which approximates the concrete meet: $\forall \bar{\mathsf{e}}_0, \bar{\mathsf{e}}_1 \in \mathsf{A}.\ \gamma(\bar{\mathsf{e}}_0) \sqcap \gamma(\bar{\mathsf{e}}_1) \sqsubseteq \gamma(\bar{\mathsf{e}}_0 \curlywedge \bar{\mathsf{e}}_1)$.

Hereafter we assume: (i) the concrete domain to be the complete lattice $\langle \mathcal{P}(\Sigma), \subseteq, \cup, \cap \rangle$ where Σ is a set of concrete program states mapping variables to values; (ii) the abstract domain to be a pre-order $\langle \mathsf{A}, \preceq, \curlyvee, \curlywedge \rangle$, therefore putting ourselves in the setting of the relaxed form of abstract interpretation.

Dynamic approximation: Widening/Narrowing. In general A is of infinite height, so that the fixpoint computation may not terminate. A widening operator $\triangledown$ should then be defined to ensure the convergence of the iteration to a *post*-fixpoint. Formally $\triangledown$ satisfies: (i) $\forall \bar{\mathsf{e}}_0, \bar{\mathsf{e}}_1 \in \mathsf{A}.\ \bar{\mathsf{e}}_0, \bar{\mathsf{e}}_1 \preceq \triangledown(\bar{\mathsf{e}}_0, \bar{\mathsf{e}}_1)$; and (ii) for each (possibly infinite) sequence of abstract elements $\bar{\mathsf{e}}_0, \bar{\mathsf{e}}_1 \ldots \bar{\mathsf{e}}_k$ the sequence defined by $\bar{\mathsf{e}}_0^{\triangledown} = \bar{\mathsf{e}}_0, \bar{\mathsf{e}}_1^{\triangledown} = \triangledown(\bar{\mathsf{e}}_0^{\triangledown}, \bar{\mathsf{e}}_1) \ldots \bar{\mathsf{e}}_k^{\triangledown} = \triangledown(\bar{\mathsf{e}}_{k-1}^{\triangledown}, \bar{\mathsf{e}}_k)$ is ultimately stationary. It is worth noting that a widening operator is not commutative. The loss of precision introduced by the widening can be partially recovered using a narrowing operator. A narrowing $\triangle$ operator satisfies: (i) $\forall \bar{\mathsf{e}}_0, \bar{\mathsf{e}}_1 \in \mathsf{A}.\ \curlywedge(\bar{\mathsf{e}}_0, \bar{\mathsf{e}}_1) \preceq \triangle (\bar{\mathsf{e}}_0, \bar{\mathsf{e}}_1) \preceq \bar{\mathsf{e}}_0, \bar{\mathsf{e}}_1$; and (ii) for each (possibly infinite) sequence of abstract elements $\bar{\mathsf{e}}_0, \bar{\mathsf{e}}_1 \ldots \bar{\mathsf{e}}_k$ the sequence defined by $\bar{\mathsf{e}}_0^{\triangle} = \bar{\mathsf{e}}_0, \bar{\mathsf{e}}_1^{\triangle} = \triangle (\bar{\mathsf{e}}_0^{\triangle}, \bar{\mathsf{e}}_1) \ldots \bar{\mathsf{e}}_k^{\triangle} = \triangle (\bar{\mathsf{e}}_{k-1}^{\triangle}, \bar{\mathsf{e}}_k)$ is ultimately stationary.

Transfer functions. It is common practice for the implementation of an abstract domain A to provide some primitive transfer functions. The assignment abstract transfer function, $\mathsf{A}.\mathsf{assign}$, 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 the 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* (A holds in all the concrete states $\gamma(\bar{\mathsf{e}})$); *false* (!A holds in all the concrete states $\gamma(\bar{\mathsf{e}})$); *bottom* (the assertion is unreached); *top* (the validity of A cannot be decided in $\gamma(\bar{\mathsf{e}})$).

3 Concrete and Abstract Semantics for a While language

We illustrate hints on a simple abstract interpreter for a while language. The concrete, reachable states semantics $[\![\cdot]\!] \in [\mathtt{Stm} \times \mathcal{P}(\Sigma) \to \mathcal{P}(\Sigma)]$ is in Fig. 3. The abstract semantics $\bar{\mathsf{s}}[\![\cdot]\!] \in [\mathtt{Stm} \times \mathsf{A} \to \mathsf{A}]$ is in Fig. 2. It is parametrized by the abstract domain $\langle \mathsf{A}, \preceq, \curlyvee, \curlywedge \rangle$ and a set of primitives assign, test, and check. The `skip` statement has no effect on the abstract state. The effects of the assignment, the assumption and the assertion are handled by the corresponding primitives of the abstract domain. Please note that for the purposes of the analysis the effects of `assume` and `assert` coincide: The assertions will be checked in a second phase, after the analysis has inferred the program invariants for all the program points. The abstract semantics of sequence is function composition. The abstract semantics of conditional: (i) pushes the guard and its negation onto the two branches; and (ii) gathers the effects using the abstract union. The abstract semantics of `while` is given in terms of fix, which computes the loop invariant as the limit of the fixpoint iterations with widening. Given a function $F \in [\mathsf{A} \to \mathsf{A}]$, fix$(F)$ is the limit of the iteration sequence: $I^0 = \bot; I^{n+1} =$ if $F(I^n) \preceq I^n$ then I^n else $I^n \triangledown F(I^n)$. The post-state for `while` is then obtained by intersecting the loop invariant with the negation of the guard. It is easy to show that for any program P, $\forall \mathsf{e} \in \mathcal{P}(\Sigma). \forall \bar{\mathsf{e}} \in \mathsf{A}.\ \mathsf{e} \subseteq \gamma(\bar{\mathsf{e}}) \implies [\![\mathtt{P}]\!](\mathsf{e}) \subseteq \gamma(\bar{\mathsf{s}}[\![\mathtt{P}]\!](\bar{\mathsf{e}}))$.

$$
\begin{aligned}
\bar{\mathsf{s}}[\![\mathtt{skip;}]\!](\bar{\mathsf{e}}) &= \bar{\mathsf{e}} & \bar{\mathsf{s}}[\![\mathtt{x = E;}]\!](\bar{\mathsf{e}}) &= \mathsf{A.assign}(\bar{\mathsf{e}}, \mathtt{x}, \mathtt{E}) \\
\bar{\mathsf{s}}[\![\mathtt{assume\ B;}]\!](\bar{\mathsf{e}}) &= \bar{\mathsf{s}}[\![\mathtt{assert\ B;}]\!](\bar{\mathsf{e}}) = \mathsf{A.test}(\mathtt{B}, \bar{\mathsf{e}}) \\
\bar{\mathsf{s}}[\![\mathtt{C_1\ C_2}]\!](\bar{\mathsf{e}}) &= \bar{\mathsf{s}}[\![\mathtt{C_2}]\!](\bar{\mathsf{s}}[\![\mathtt{C_1}]\!](\bar{\mathsf{e}})) \\
\bar{\mathsf{s}}[\![\mathtt{if(B)\ \{C_1\}else\ \{C_2\};}]\!](\bar{\mathsf{e}}) &= \bar{\mathsf{s}}[\![\mathtt{C_1}]\!](\mathsf{A.test}(\mathtt{B}, \bar{\mathsf{e}})) \curlyvee \bar{\mathsf{s}}[\![\mathtt{C_2}]\!](\mathsf{A.test}(\mathtt{!B}, \bar{\mathsf{e}})) \\
\bar{\mathsf{s}}[\![\mathtt{while(B)\ \{C\};}]\!](\bar{\mathsf{e}}) &= \text{let } \bar{\mathsf{I}} = \mathsf{fix}\lambda X.\ \bar{\mathsf{e}} \curlyvee \bar{\mathsf{s}}[\![\mathtt{C}]\!](\mathsf{A.test}(\mathtt{B}, X)) \\
&\quad \text{in } \mathsf{A.test}(\mathtt{!B}, \bar{\mathsf{I}})
\end{aligned}
$$

Fig. 2. The abstract semantics for the while-language

$$
\begin{aligned}
[\![\mathtt{skip;}]\!](\mathsf{e}) &= \mathsf{e} & [\![\mathtt{x = E;}]\!](\mathsf{e}) &= \{\sigma[\mathtt{x} \mapsto v] \mid \sigma \in \mathsf{e}, \mathbb{E}[\![\mathtt{E}]\!](\sigma) = v\} \\
[\![\mathtt{assume\ B;}]\!](\mathsf{e}) &= [\![\mathtt{assert\ B;}]\!](\mathsf{e}) = \{\sigma \in \mathsf{e} \mid \mathbb{B}[\![\mathtt{B}]\!](\sigma) = \mathit{true}\} \\
[\![\mathtt{C_1\ C_2}]\!](\mathsf{e}) &= [\![\mathtt{C_2}]\!]([\![\mathtt{C_1}]\!](\mathsf{e})) \\
[\![\mathtt{if(B)\ \{C_1\}else\ \{C_2\};}]\!](\mathsf{e}) &= [\![\mathtt{C_1}]\!](\{\sigma \in \mathsf{e} \mid \mathbb{B}[\![\mathtt{B}]\!](\sigma) = \mathit{true}\}) \cup [\![\mathtt{C_2}]\!](\{\sigma \in \mathsf{e} \mid \mathtt{!}\mathbb{B}[\![\mathtt{B}]\!](\sigma) = \mathit{true}\}) \\
[\![\mathtt{while(B)\ \{C\};}]\!](\bar{\mathsf{e}}) &= \text{let } \mathsf{I} = \bigcup_i [\![\mathtt{C}]\!]^i(\{\sigma \in \mathsf{I} \mid \mathbb{B}[\![\mathtt{B}]\!](\sigma) = \mathit{true}\}) \\
&\quad \text{in } \{\sigma \in \mathsf{I} \mid \mathtt{!}\mathbb{B}[\![\mathtt{B}]\!](\sigma) = \mathit{true}\}.
\end{aligned}
$$

Fig. 3. The reachable states semantics for the while-language

4 Hints

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.*, $\lambda(\bar{\mathsf{e}}_0, \bar{\mathsf{e}}_1, \bar{\mathsf{e}}_2).\ (\bar{\mathsf{e}}_0 \curlywedge \bar{\mathsf{e}}_1) \curlyvee (\bar{\mathsf{e}}_0 \curlywedge \bar{\mathsf{e}}_2)$).

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:*

$$\mathbb{h}_{\bar{\diamond}}(\bar{\mathsf{e}}_0 \dots \bar{\mathsf{e}}_{n-1}) \preceq \bar{\diamond}(\bar{\mathsf{e}}_0 \dots \bar{\mathsf{e}}_{n-1}) \qquad \textit{(Refinement)}$$
$$\diamond(\gamma(\bar{\mathsf{e}}_0) \dots \gamma(\bar{\mathsf{e}}_{n-1})) \sqsubseteq \gamma(\mathbb{h}_{\bar{\diamond}}(\bar{\mathsf{e}}_0 \dots \bar{\mathsf{e}}_{n-1})) \qquad \textit{(Soundness)}.$$

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. Therefore, if $\mathbb{h}^1_{\bar{\diamond}}$ and $\mathbb{h}^2_{\bar{\diamond}}$ are hints, then $\mathbb{h}^{\curlywedge}_{\bar{\diamond}}(\bar{\mathsf{e}}_0 \dots \bar{\mathsf{e}}_{n-1}) = \mathbb{h}^1_{\bar{\diamond}}(\bar{\mathsf{e}}_0 \dots \bar{\mathsf{e}}_{n-1}) \curlywedge \mathbb{h}^2_{\bar{\diamond}}(\bar{\mathsf{e}}_0 \dots \bar{\mathsf{e}}_{n-1})$ is a hint, too. Hints improve the precision of static analyses without introducing unsoundness and preserving termination:

Theorem 1 (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, $\forall \mathsf{e} \in \mathcal{P}(\Sigma). \forall \bar{\mathsf{e}} \in \mathsf{A}$.*

$$\bar{\mathsf{s}}^*[\![\mathtt{P}]\!](\bar{\mathsf{e}}) \preceq \bar{\mathsf{s}}[\![\mathtt{P}]\!](\bar{\mathsf{e}}) \qquad \textit{(Refinement)}$$
$$\mathsf{e} \subseteq \gamma(\bar{\mathsf{e}}) \implies [\![\mathtt{P}]\!](\mathsf{e}) \subseteq \gamma(\bar{\mathsf{s}}^*[\![\mathtt{P}]\!](\bar{\mathsf{e}})) \qquad \textit{(Soundness)}.$$

5 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).

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 $\mathsf{pred} \in [\mathtt{Stm} \to \mathcal{P}(\mathtt{BExp})]$ extracts from the program text the predicates appearing in conditional and loop guards. User provided hints do not affect the termination of the widening as we can only add finitely many new predicates:

Lemma 1 (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.*

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

In example of Fig. 1(b), $\mathsf{pred}(\mathtt{DomOp}) = \{\mathtt{x} \leq \mathtt{y}, 4 \leq \mathtt{x}, \mathtt{y} \leq 100 \cdot \mathtt{x}\}$. The refined domain operations keep the predicate $\mathtt{x} \leq \mathtt{y}$, which is stable among loop iterations, and hence is a loop invariant.

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") via executable annotations. 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: For instance, if in Fig. 1(c) we replace the assertion at $(*)$ with `if 10 <= x then assert 5 <= y`, then $\mathsf{pred}(\mathtt{DomOp}) = \{10 \leq \mathtt{x}, 5 \leq \mathtt{y}\}$, so that $\mathbb{h}^{\mathsf{pred}}_{\nabla_{\mathsf{Poly}}}$ cannot figure out that $\mathtt{x} \leq \mathtt{y}$, and hence the analyzer cannot prove that the assertion is valid. Semantic hints (Sect. 6.3) will fix it.

5.1 Thresholds Hints

Widening with threshold has been introduced in [2] 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 $\bar{\mathtt{e}}_0$ and $\bar{\mathtt{e}}_1$ be abstract states belonging to some numerical abstract domain. Without loss of generality we can assume that the basic facts in $\bar{\mathtt{e}}_0, \bar{\mathtt{e}}_1$ are in the form $\mathtt{p} \leq k$, where $\mathtt{p}$ is some polynomial. For instance $\mathtt{x} \in [-2, 4]$ is equivalent to $\{-\mathtt{x} \leq 2, \mathtt{x} \leq 4\}$. The standard widening preserves the linear forms with stable upper bounds: $\nabla(\bar{\mathtt{e}}_0, \bar{\mathtt{e}}_1) = \{\mathtt{p} \leq k \mid \mathtt{p} \leq k_0 \in \bar{\mathtt{e}}_0, \mathtt{p} \leq k_1 \in \bar{\mathtt{e}}_1, k = \text{if } k_1 > k_0 \text{ then } +\infty \text{ else } k_0\}$. Given a finite set of values `T`, threshold hints refine the standard widening by:

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

Lemma 2. $\mathbb{h}^{T}_{\nabla}$ *is: (i) a hint; and (ii) a widening.*

Example 1 (Widening with thresholds). Let us consider the code snippets in Fig. 4 to be analyzed with Intervals. In the both cases, the (post-)fixpoint is

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

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

Fig. 4. Two programs to be analyzed with Intervals. The iterations with widening infer the loop invariant $\mathtt{x} \in [0, +\infty]$. In the first case, the narrowing step refines the loop invariant to $\mathtt{x} \in [0, 1000]$. In the second case, the narrowing fails to refine it.

reached after the first iteration $\triangledown([0,0],[1,1]) = [0,+\infty]$. In the first case, the invariant can be improved by a narrowing step to $\triangle\ ([0,+\infty],[-\infty,1000]) = [0,1000]$ (see [4] for a definition of narrowing of Intv). In the second case, the narrowing is of no help as $\triangle\ ([0,+\infty],\curlyvee([-\infty,1000],[1002,+\infty])) = [0,+\infty]$. A widening with Thresholds $T = \{1000\}$ helps discovering the tightest loop invariant for both examples in one step as $\mathbb{h}^T_\triangledown([0,0],[1,1]) = [0,1000]$. □

Please note that user-provided hints are of no help in the previous example, as $\mathsf{pred}(\mathtt{NotEq}) = \{\mathtt{x} \neq 1000\}$ does not hold for all the operands of the widening.

The set T of thresholds is a parameter of the analyzer, which can either be provided by the user, preset to some common values (*e.g.*, $T = \{-1, 0, 1\}$), or extracted from the program text. In `Clousot`, we use a function $\mathsf{const} \in [\mathtt{Stm} \to \mathcal{P}(\mathtt{int})]$ which extracts the constants appearing in the guards. We found the hint $\mathbb{h}^{\mathsf{const}}_\triangledown$ very satisfactory: (i) it helps inferring precise *numerical* loop invariants without requiring the extra iteration steps required for applying the narrowing; and (ii) it improves the precision of the analysis of code involving disequalities, *e.g.*, Fig. 4(b). A drawback 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.

6 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).

6.1 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 [7,15]. 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 $\bar{\diamond}^\times((\bar{e}_0^0, \bar{e}_1^0)\dots(\bar{e}_0^{n-1}, \bar{e}_1^{n-1})) = (\bar{\diamond}_0(\bar{e}_0^0 \dots \bar{e}_0^{n-1}), \bar{\diamond}_1(\bar{e}_1^0 \dots \bar{e}_1^{n-1}))$. 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 2 (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 [18]. Let $\bar{z}_0 = ([\mathtt{x} \mapsto [-\infty, 0], \mathtt{y} \mapsto [1, +\infty]], [\cdot])$ and $\bar{z}_1 = ([\cdot], [\mathtt{x} \mapsto \{\mathtt{y}\}])$ be two elements of Z ($[\cdot]$ denotes the empty map). Then the Cartesian join loses all the information: $\curlyvee^\times(\bar{z}_0, \bar{z}_1) = ([\cdot], [\cdot])$. □

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

$$\mathbb{h}^\rho_{\bar{\diamond}^\times}((\bar{e}_0^0, \bar{e}_1^0)\dots(\bar{e}_0^{n-1}, \bar{e}_1^{n-1})) = \text{ let } \bar{r}^i = \rho(\bar{e}_0^i, \bar{e}_1^i) \text{ for } i \in 0\dots n-1 \text{ in } \bar{\diamond}^\times(\bar{r}^0 \dots \bar{r}^{n-1}).$$

Example 3 (Cartesian join, continued). The saturation of $\bar{z}_0$ materializes the constraint $\mathtt{x} < \mathtt{y} : \bar{r}_0 = ([\mathtt{x} \mapsto [-\infty, 0], \mathtt{y} \mapsto [1, +\infty], [\mathtt{x} \mapsto \{\mathtt{y}\}])$, and it leaves $\bar{z}_1$ 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 [23]).

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 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.

6.2 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 Ex. 2 the constraint $\mathtt{x} < \mathtt{y}$ is explicit in $\bar{z}_1$, and implicit in $\bar{z}_0$ (as $\mathtt{x} \leq 0 \wedge 1 \leq \mathtt{y} \implies \mathtt{x} < \mathtt{y}$). 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:

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

In defining the die-hard hint for $\triangledown$, 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 4. *$\mathbb{h}^{d}_{(\curlyvee,\{0,1\})}$ and $\mathbb{h}^{d}_{(\triangledown,\{0\})}$ are hints and $\mathbb{h}^{d}_{(\triangledown,\{0\})}$ is a widening.*

6.3 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 5 (Computed hints). *Let $\bar{\mathsf{e}}_0,\bar{\mathsf{e}}_1 \in \mathsf{A}$, $\varXi \in [\mathsf{A}\times\mathsf{A}\to\mathsf{A}]$ a function which returns a set of likely bounds of $\bar{\mathsf{e}}_0 \curlyvee \bar{\mathsf{e}}_1$. Then $\mathbb{h}^{\varXi}_{\curlyvee}$ below is a hint.*

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

Computed hints are useful when the abstract join $\curlyvee$ is not optimal. Otherwise, $\mathbb{h}^{\varXi}_{\curlyvee}$ 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 approximating the concrete union. As a consequence, no further information can be extracted from the operands. It is worth noting that in general $\mathbb{h}^{\varXi}_{\triangledown}$ is not a widening. However, one can extend the arguments of the previous section to define an asymmetric hint $\mathbb{h}^{\varXi}_{\triangledown}$.

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: $\forall \mathtt{E}.\ \forall \bar{\mathsf{e}} \in \mathsf{A}.\ \mathsf{A.range}(\mathtt{E},\bar{\mathsf{e}}) = [l,u] \implies \forall\sigma \in \gamma(\bar{\mathsf{e}}).\ l \le \mathbb{E}[\![\mathtt{E}]\!](\sigma) \le u$. 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 $\varXi^P$:

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

The main difference between $\mathbb{h}^{\varXi^P}_{\curlyvee}$ 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 the counter-example of Sect. 1 with the set of templates $Oct \equiv \{\mathtt{x}_0 - \mathtt{x}_1 \mid \mathtt{x}_0, \mathtt{x}_1$ are program variables$\}$. In general, template hints with Oct refine Poly so to make it as precise as Oct.

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 [8]. However the convex hull algorithm requires an expensive conversion from a tableau of linear constraints to a set of vertexes 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 abstract states on all the two-dimensional planes; and (ii) computing the planar convex hull 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. Let us consider the code in Fig. 5 from [8]. 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 [17].

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

Fig. 5. Example requiring the use of 2D-convex hull hints

6.4 Reductive Hints

Intuitively, one way to improve the precision of a unary operator is to iterate its application [13]. However, an unconditional iteration may be source of unsoundness. For instance, let $- \in [\mathsf{Intv} \to \mathsf{Intv}]$ be the operator which applies the unary minus to an interval. In general, $\forall n \in \mathbb{N}.\ \bar{\mathsf{e}} = -^{2n}(\bar{\mathsf{e}}) \neq -^{2n+1}(\bar{\mathsf{e}})$. 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 6 (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.*

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 6. Abstract compilation can be used to express ψ in terms of domain operations, their compositions and state update. Lemma 6 justifies the use of local fixpoint iterations to refine the result of the analysis. For instance, in the abstract domain $[\mathtt{Var} \to \{\mathsf{true}, \mathsf{false}, \top, \bot\}]$ the abstract compilation of the predicate $\mathtt{b1} == \mathtt{b2} \wedge \mathtt{b2} == \mathtt{b3}$ 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 pointwise extension of $\wedge$. In an initial abstract state $\mathsf{b}_0 = [\mathtt{b1}, \mathtt{b2} \mapsto \top; \mathtt{b3} \mapsto \mathsf{true}]$, $\psi(\mathsf{b}_0) = [\mathtt{b1} \mapsto \top; \mathtt{b2}, \mathtt{b3} \mapsto \mathsf{true}]$ is refined by $\psi^2(\mathsf{b}_0) = [\mathtt{b1}, \mathtt{b2}, \mathtt{b3} \mapsto \mathsf{true}] = \psi^n(\mathsf{b}_0)$, $n \geq 2$.

```
public BitArray(byte[] bytes) {
  Contract.Requires(bytes != null);

  this.m_array = new int[(bytes.Length + 3) / 4];
  this.m_length = bytes.Length * 8;
  int index = 0, j = 0;
  for (; (bytes.Length - j) >= 4; j+=4)
    this.m_array[index++] = (((bytes[j] & 0xff) | ((bytes[j + 1] & 0xff) << 8))
      | ((bytes[j + 2] & 0xff) << 0x10)) | ((bytes[j + 3] & 0xff) << 0x18);

  switch ((bytes.Length - j)) {
    case 1 : goto Label_00DB;
    case 2 : break;
    case 3 : this.m_array[index] = (bytes[j + 2] & 0xff) << 0x10; break;
    default: goto Label_00FC;
  }
  this.m_array[index] |= (bytes[j + 1] & 0xff) << 8;
Label_00DB:
    this.m_array[index] |= bytes[j] & 0xff;
Label_00FC:
    this.version = 0;
}
```

Fig. 6. Example of code from `mscorlib.dll`. Out of the 23 total array bound checks, `Clousot` with $\langle \mathsf{Pnt}, \mathbb{h}^d_{\Upsilon,\triangledown} \rangle$ validates 13, `Clousot` with $\langle \mathsf{SubPoly}, \emptyset \rangle$ validates 6 more, and `Clousot` with $\langle \mathsf{SubPoly}, \mathbb{h}^d_{\Upsilon} \rangle$ validates the remaining 4.

.

7 Experience

We implemented hints in `Clousot`, our abstract interpretation-based static analyzer for `.Net`. `Clousot` has been designed and it is used as the static checker for the CodeContracts project [21]. CodeContracts provide a language-agnostic approach to the definition of object invariants, method preconditions and postconditions. Contracts are specified by static methods of the `Contracts` class, *e.g.*, `Contracts.Requires(x! = null)`; specifies that the parameter `x` should be not null. More details on the specification language can be found in the documentation on the CodeContracts website[21]. The `Contracts` class will be shipped in the version 4.0 of the `.Net` framework [22] (at the moment of writing, in the public beta 1 phase). `Clousot` is shipped on the DevLabs [21] website, and it is available for free downloading for Academic use at `http://research.microsoft.com/en-us/projects/contracts/`.

`Clousot` analyzes each method `m` in isolation. It assumes the precondition of `m`, it progates it through the body, it computes loop invariants, and it uses the inferred invariants to validate: (i) the method postcondition; (ii) the preconditions of the methods invoked by `m`; (iii) the user provided assertions; and (iv) the absence of runtime errors (*e.g.*, null pointers, array out-of-bounds, divisions by zero, negation of MinInt ...) and of buffer overruns [10]. When a method has no annotations, `Clousot` simply assumes the worst case scenario (*e.g.*, the parameters can assume any value compatible with their type). Orthogonally, `Clousot`

can infer pre-conditions and post-conditions to help reduce the annotation burden. `Clousot` analyzes `m` incrementally. The user specifies a sequence of pairs of domains and set of hints $\langle \mathsf{A}_0, H_0 \rangle \ldots \langle \mathsf{A}_n, H_n \rangle$. `Clousot` instantiates the abstract semantics of `m` with the abstract domain A_i refined with the hints in H_i. If it cannot discharge all the proof obligations, `Clousot` tries to discharge the remaining proof obligations using the abstract domain A_{i+1} refined with the hints H_{i+1}. We designed new numerical abstract domains, ranging from imprecise yet very fast (Pnt, [18]) to very precise but more expensive (SubPoly, [17]). In the incremental setting of `Clousot`, hints allow a very fine tuning of the precision/cost ratio. For instance, the same abstract domain can be refined with several hints: the more the hints, the more precise the analysis, but also the more expensive it is.

We report the experimental results of refining the abstract operations of the two extremes of the precision spectrum of `Clousot`'s numerical abstract domains: Pnt and SubPoly. Pnt is a weakly relational domain which captures properties in the form of $\mathtt{x} \in [a, b] \wedge \mathtt{x} < \mathtt{y}$. SubPoly is a strongly relational domain which is as expressive as Poly, but drops some of the inference power to achieve scalability: Hints are cardinal to recover precision yet mantaining performace. We run the experiments on a Core2 Duo E6850@3.00 Ghz PC, with 4 GB of RAM, running Windows 7. We analyzed four of the main libraries of the current release of the `.Net` framework (v.3.5), available in every Windows distribution. The `mscorlib.dll` library is the core of the `.Net` framework: it contains the definitions for the `Object`, `Int32` ... types, but also common data structures such as `List`, `Dictionary`, and many other usefull classes (for reflection, security ...). The `System.dll` library is a higher layer on `mscorlib.dll`. `System.Web.dll` and `System.Design.dll` contain classes that simplify the access to the Web and the creation of user interfaces. In order to provide an uniform and repeatable test bench: (i) we considered shipped assemblies (hence with no annotations: The annotation processing is undergoing internally at Microsoft); (ii) we turned off the inference capabilities of `Clousot`; and (iii) we used `Clousot` *only* to check array creations and accesses (lower and upper bounds): the shipped assemblies do not contain annotations, so there are no contracts to check. The framework libraries contains tenths of thousands of array accesses, some of them are quite easy (*e.g.*, the sequential access of an array in a `for` loop) but others require inferring more complex relations between the array lengths and the indexes. For instance, Fig. 6 shows the constructor of the `BitArray` type (we picked it randomly from `Clousot`'s log). The Pnt and SubPoly abstract domains alone can be used to prove most of the array accesses correct, however, all the proof obligations can be discharged only using die-hard hints. One may object that the same result can be obtained using existing domains such as Oct or Poly. However, Oct is unable to capture the constraint $4 \cdot \mathtt{m\_array.Length} - \mathtt{bytes.Length} == 3$, which is necessary to prove that $\mathtt{index} < \mathtt{m\_array.Length}$, and Poly suffers of a huge scalability problem, which shows up even in small code snippets like the one in Fig. 6.

Figure 7 compares die-hard hints and saturation hints when used to refine the join and widening of Pnt. The figure reports the analyzed assemblies, the total number of analyzed methods, the total number of proof obligations checked (*i.e.*, array creations, lower bounds, and upper bounds), the number of proof

Assembly	Methods	P.O. Checked	Pnt Valid	Pnt Time	Pnt + $\mathbb{h}^d_{Y,\nabla}$ Valid	Pnt + $\mathbb{h}^d_{Y,\nabla}$ Time	Slow-down	Pnt + $\mathbb{h}^\rho_{\{\sqcup,\nabla\}^\times}$ Valid	Pnt + $\mathbb{h}^\rho_{\{\sqcup,\nabla\}^\times}$ Time	Slow-down
mscorlib		17 286	14 059	3:03(0)	14 293	3:10(0)	1.0x	14 220	10:33(4)	3.3x
System	15 497	12 037	9 979	2:28(0)	10 321	2:36(0)	1.0x	10 143	9:43(2)	3.7x
System.Web	23 655	14 304	12 952	2:49(0)	13 034	2:55(0)	1.0x	13 048	8:30(0)	2.9x
System.Design	12 922	10 577	9 562	2:18(0)	10 135	2:21(0)	1.0x	9 947	7:39(5)	3.2x

Fig. 7. The experimental results of refining Pnt with die-hard hints and saturation hints. Pnt with die-hard hints validates 1 231 more proof obligations. Pnt with saturation hints are 3x slower, hit 11 timeouts (2 min), and validate 425 less accesses than $\mathbb{h}^d$.

obligations validated and the analysis time for the pair-wise gathering operations and two refinements of the Pnt operations. The values in brackets denote the number of methods for which the analysis timed out. Time out was set to 2 minutes. Die-hard hints allow `Clousot` to validate 1 231 accesses more than the pair-wise joins at no extra cost. On the other hand, saturation hints induce an average 3x slow-down of the analysis, which causes the analysis to time out for 11 methods, and hence to validate 425 less accesses. We manually inspected the analysis logs. We found that $\langle \mathsf{Pnt}, \mathbb{h}^d_{Y,\nabla} \rangle$ missed only few validations w.r.t. $\langle \mathsf{Pnt}, \mathbb{h}^\rho_{\{Y,\nabla\}^\times} \rangle$. As a consequence, the use of a saturation procedure with Pnt seems to be disadvantageous: the cost is too high, and the precision can be recovered by more precise abstract domains anyway. Furthermore, we checked some of the proof obligations reported as unproven or unreachable from `Clousot`. Most of the unproven conditions are caused by the lack of contracts (mainly postconditions and object invariants). However, some of the unproven conditions turned out to be real bugs, and the unreachable ones, after fixing some bug of the analyzer, were effectively dead-code.

Figure 8 focuses on the analysis of `mscorlib` using SubPoly refined with hints. SubPoly is a very expressive abstract domain (as expressive as Poly), whose inference precision can be fine tuned thanks to hints. 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 $\mathsf{SubPoly}^*$, $\langle \mathsf{SubPoly}^*, \mathbb{h}^{\Xi^{Oct}}_Y \rangle$ and $\langle \mathsf{SubPoly}^*, \mathbb{h}^{\Xi^{2DCH}}_Y \rangle$ 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 codebase 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

SubPoly		SubPoly*		Slow	SubPoly* + $\mathbb{h}^{\Xi^{Oct}}_{\Upsilon}$		Slow	SubPoly* + $\mathbb{h}^{\Xi^{2DCH}}_{\Upsilon}$		Slow
Valid	Time	Valid	Time	down	Valid	Time	down	Valid	Time	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

Fig. 8. The experimental results analyzing `mscorlib` with SubPoly and different semantic hints. SubPoly* denotes SubPoly refined with $\mathbb{h}^{\mathsf{pred}}_{\diamond}$ and $\mathbb{h}^{d}_{\Upsilon,\triangledown}$. Computed hints significantly slow-down the analysis, but they are needed to reach a very low false alarm ratio.

as in Fig. 1(b) and Fig. 5 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.

8 Conclusions

We introduced hints, a technique to systematically refine abstract domain *operations*. Hints allow us improving the precision of abstract operation whenever those are not complete, *e.g.*, when the underlying abstract domain is not a complete lattice (the common case in practice). We formalized hints in a relaxed abstract interpretation setting, we proved their soundness, and we distinguished between syntactic and semantics hints. We showed how some existing techniques to improve the precision of static analyses, such as widening with thresholds and reductive iterations are just instances of hints. We applied hints to the numerical abstract domains defined in our abstract interpretation-based analyzer, showing how they enable a powerful tuning of the precision/cost ratio. However, hints are not restricted to numerical domains, and they can be easily generalized to other kind of domains (for instance, for heap analysis) Future work will consider combining hints with other forms of refinement, as domain refinement, counter example-based refinement, and inference of optimal transfer functions.

References

1. Bagnara, R., Hill, P.M., Zaffanella, E.: Widening operators for powerset domains. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 135–148. Springer, Heidelberg (2004)
2. 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 2003. ACM Press, New York (2003)
3. Colby, C., Lee, P.: Trace-based program analysis. In: POPL 1996. ACM Press, New York (1996)
4. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL 1977 (1977)
5. Cousot, P., Cousot, R.: Abstract interpretation and application to logic programs. Journal of Logic Programming 13(2-3), 103–179 (1992)
6. Cousot, P., Cousot, R.: Abstract interpretation frameworks. Journal of Logic and Computation 2(4) (August 1992)

7. Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: Combination of abstractions in the ASTRÉE static analyzer. In: Okada, M., Satoh, I. (eds.) ASIAN 2006. LNCS, vol. 4435, pp. 272–300. Springer, Heidelberg (2008)
8. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: POPL 1978 (1978)
9. Das, M., Lerner, S., Seigle, M.: Esp: Path-sensitive program verification in polynomial time. In: PLDI 2002, pp. 57–68 (2002)
10. Ferrara, P., Logozzo, F., Fähndrich, M.A.: Safer unsafe code in.Net. In: OOPSLA 2008 (2008)
11. Gonnord, L., Halbwachs, N.: Combining widening and acceleration in linear relation analysis. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, pp. 144–160. Springer, Heidelberg (2006)
12. Gopan, D., Reps, T.W.: Lookahead widening. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 452–466. Springer, Heidelberg (2006)
13. Granger, P.: Improving the results of static analyses programs by local decreasing iteration. In: Shyamasundar, R.K. (ed.) FSTTCS 1992. LNCS, vol. 652. Springer, Heidelberg (1992)
14. Gulavani, B.S., Chakraborty, S., Nori, A.V., Rajamani, S.K.: Automatically refining abstract interpretations. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 443–458. Springer, Heidelberg (2008)
15. Gulwani, S., McCloskey, B., Tiwari, A.: Lifting abstract interpreters to quantified logical domains. In: POPL 2008. ACM Press, New York (2008)
16. Handjieva, M., Tzolovski, S.: Refining static analyses by trace-based partitioning using control flow. In: Levi, G. (ed.) SAS 1998. LNCS, vol. 1503, pp. 200–214. Springer, Heidelberg (1998)
17. Laviron, V., Logozzo, F.: Subpolyhedra: a (more) scalable approach to infer linear inequalities. In: Jones, N.D., Müller-Olm, M. (eds.) VMCAI 2009. LNCS, vol. 5403, pp. 229–244. Springer, Heidelberg (2009)
18. Logozzo, F., Fähndrich, M.A.: Pentagons: A weakly relational abstract domain for the efficient validation of array accesses. In: SAC 2008 (2008)
19. Manevich, R., Field, J., Henzinger, T.A., Ramalingam, G., Sagiv, M.: Abstract counterexample-based refinement for powerset domains. In: Reps, T., Sagiv, M., Bauer, J. (eds.) Wilhelm Festschrift. LNCS, vol. 4444, pp. 273–292. Springer, Heidelberg (2006)
20. Mauborgne, L., Rival, X.: Trace partitioning in abstract interpretation based static analyzers. In: Sagiv, M. (ed.) ESOP 2005. LNCS, vol. 3444, pp. 5–20. Springer, Heidelberg (2005)
21. Microsoft. Codecontracts tools, `http://msdn.microsoft.com/en-us/devlabs/dd491992.aspx`
22. Microsoft. Contracts namespace, `http://msdn.microsoft.com/en-us/library/system.diagnostics.contracts(VS.100).aspx`
23. Miné, A.: The octagon abstract domain. In: WCRE 2001 (2001)
24. Monniaux, D.: Automatic modular abstractions for linear constraints. In: POPL 2009 (2009)
25. Péron, M., Halbwachs, N.: An abstract domain extending difference-bound matrices with disequality constraints. In: Cook, B., Podelski, A. (eds.) VMCAI 2007. LNCS, vol. 4349, pp. 268–282. Springer, Heidelberg (2007)
26. Yorsh, G., Reps, T.W., Sagiv, S.: Symbolically computing most-precise abstract operations for shape analysis. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 530–545. Springer, Heidelberg (2004)