Pentagons: A Weakly Relational Abstract Domain for the Efficient Validation of Array Accesses

Francesco Logozzo

Microsoft Research, Redmond, WA, USA

Manuel Fähndrich

Microsoft Research, Redmond, WA, USA

Abstract

We introduce Pentagons (Pntg), a weakly relational numerical abstract domain useful for the validation of array accesses in byte-code and intermediate languages (IL). This abstract domain captures properties of the form of $\mathtt{x} \in [a, b] \wedge \mathtt{x} < \mathtt{y}$. It is more precise than the well known Interval domain, but it is less precise than the Octagon domain.

The goal of Pntg is to be a lightweight numerical domain useful for adaptive static analysis, where Pntg is used to quickly prove the safety of most array accesses, restricting the use of more precise (but also more expensive) domains to only a small fraction of the code.

We implemented the Pntg abstract domain in `Clousot`, a generic abstract interpreter for .NET assemblies. Using it, we were able to validate 83% of array accesses in the core runtime library `mscorlib.dll` in a little bit more than 3 minutes.

Key words: Abstract Domains, Abstract Interpretation, Bounds checking, Numerical Domains, Static Analysis, .NET Framework

1 Introduction

The goal of an abstract interpretation-based static analysis is to statically infer properties of the execution of a program that can be used to ascertain the

Email addresses: `logozzo@microsoft.com` (Francesco Logozzo), `maf@microsoft.com` (Manuel Fähndrich).

absence of certain runtime failures. Traditionally, such tools focus on proving the absence of out of bound memory accesses, divisions by zero, overflows, or null dereferences.

The heart of an abstract interpreter is the abstract domain, which captures the properties of interest for the analysis. In particular, several *numerical* abstract domains have been developed, *e.g.*, [14,27,35], that are useful to check properties such as out of bounds and division by zero, but also aliasing [36], parametric predicate abstraction [12] and resource usage [30].

In this paper we present Pentagons, Pntg, a new numerical abstract domain designed and implemented as part of `Clousot`, a generic static analyzer based on abstract interpretation of MSIL [19]. We intend `Clousot` to be used by developers during coding and testing phases. It should therefore be scalable, yet sufficiently precise. To achieve this aim, `Clousot` is designed to adaptively choose the necessary precision of the abstract domain, as opposed to fixing it *before* the analysis (*e.g.*, [24]). Thus, `Clousot` must be able to discharge most of the "easy checks" very quickly, hence focusing the analysis only on those pieces of code that require a more precise abstract domain or fixpoint strategy.

`Clousot` uses the abstract domain of Pntg to quickly analyze .NET assemblies and discharge most of the proof obligations from the successive phases of the analysis. As an example let us consider the code in Figure 1, taken from the basic component library of .NET. `Clousot`, instantiated with the abstract domain Pntg, automatically discovers the following invariant at program point $(*)$:

$$0 \leq \texttt{num} < \texttt{array.Length} \wedge 0 \leq \texttt{num2} < \texttt{array.Length}$$

This is sufficient to prove that $0 \leq \texttt{index} < \texttt{array.Length}$, *i.e.*, the array is never accessed outside of its bounds.

The elements of Pntg are of the form $\texttt{x} \in [a, b] \wedge \texttt{x} < \texttt{y}$, where `x` and `y` are program variables and a, b are rationals. Such elements allow expressing (most) bounds of program variables, and in particular those of array indices: intervals $[a, b]$ take care of the numerical part (*e.g.*, to check array underflows $0 \leq a$), and inequalities $\texttt{x} < \texttt{y}$ handle the symbolic reasoning (*e.g.*, to check array overflows $\texttt{x} < \texttt{arr.Length}$).

Pntg is therefore an abstract domain more precise than the Intervals, Intv [13], as it adds symbolic reasoning, but it is less precise than Octagons, Oct [27], as it cannot for instance capture equalities such as $\texttt{x} + \texttt{y} == 22$. We found that Pntg is precise enough to validate 83% of the array bound accesses (lower and upper) in `mscorlib.dll`, the main library in the .NET platform, in less than 4 minutes. Similar results are obtained for the other assemblies of the .NET framework. Thus, Pntg fits well with the programming style adopted in this library. Nevertheless, it is not the ultimate abstract domain for bounds

```
int BinarySearch(ulong[] array, ulong value)
{
  int num = 0;
  int num2 = array.Length - 1;
  while (num <= num2)
  { (*)
    int index = (num + num2) >> 1;
    ulong num4 = array[index];
    if (value == num4)
      return index;
    if (num4 < value)
      num = index + 1;
    else
      num2 = index - 1;
  }
  return ~num;
}
```

Fig. 1. Example from `mscorlib.dll`. `Pntg` infers the loop invariant $0 \leq$ `num` $\wedge$ `num2` $<$ `array.Length` which is enough to prove that $0 \leq$ `index` $<$ `array.Length` holds at array access.

analysis. For instance, when used on part of `Clousot`'s implementation, it validates only 72% of the accesses.

2 Basics of Abstract interpretation

Abstract interpretation is a theory of approximations, [13]. It captures the intuition that semantics are more or less precise depending on the observation level. The observation level is formalized by the notion of an abstract domain. An abstract domain $\bar{\mathsf{D}}$ is a complete lattice $\langle \mathsf{E}, \sqsubseteq, \bot, \top, \sqcup, \sqcap \rangle$, where E is the set of abstract elements, ordered according to relation $\sqsubseteq$. The smallest abstract element is $\bot$, the largest is $\top$. The join $\sqcup$, and the meet $\sqcap$, are also defined. When the abstract domain $\bar{\mathsf{D}}$ does not respect the ascending chain condition, then a widening operator $\triangledown$ must be used to enforce the termination of the analysis. With a slight abuse of notation, sometimes we will confuse an abstract domain $\bar{\mathsf{D}}$ with the set of its elements E.

A domain $\bar{\mathsf{D}}$ is the abstraction of a concrete domain $\mathsf{C} = \langle \mathsf{C}, \preceq, \bot, \top, \curlyvee, \curlywedge \rangle$, if it exists a pair of monotonic functions (Galois connection) $\langle \alpha, \gamma \rangle$ such that: (i) $\alpha \in [\mathsf{C} \to \bar{\mathsf{D}}]$ (abstraction); (ii) $\gamma \in [\bar{\mathsf{D}} \to \mathsf{C}]$ (concretization); and (iii) $\forall \mathsf{c} \in \mathsf{C}. \forall \mathsf{d} \in \bar{\mathsf{D}}.\ \alpha(\mathsf{c}) \sqsubseteq \mathsf{d} \Leftrightarrow \mathsf{c} \preceq \gamma(\mathsf{d})$ (soundness). In a Galois connection, one adjoint determines the other [13], and in particular the concretization function uniquely defines the abstraction: $\alpha = \lambda \mathsf{c}.\ \sqcap \{\mathsf{d} \mid \mathsf{c} \preceq \gamma(\mathsf{d})\}$, so that often we

will omit of the two.

The *best* abstract transfer function for a concrete transfer function $\mathsf{t} \in [\mathsf{C} \to \mathsf{C}]$, is $\mathsf{t}^{\mathsf{a}}_{*} = \alpha \circ \mathsf{t} \circ \gamma$ [13]. In general, the best abstract transfer function is not computable, and a sound approximation t^{a}, such that $\forall \bar{\mathsf{d}} \in \bar{\mathsf{D}}.\ \mathsf{t}^{\mathsf{a}}_{*}(\bar{\mathsf{d}}) \sqsubseteq \mathsf{t}^{\mathsf{a}}(\bar{\mathsf{d}})$, is often good enough.

Given a concrete domain C and two domains $\bar{\mathsf{D}}_1$ and $\bar{\mathsf{D}}_2$, related to C respectively by two Galois connections $\langle \alpha_1, \gamma_1 \rangle$ and $\langle \alpha_2, \gamma_2 \rangle$, then the relation $\equiv$ defined as $\langle d_1, d_2 \rangle \equiv \langle d'_1, d'_2 \rangle$ iff $\gamma_1(d_1) \curlywedge \gamma_2(d_2) = \gamma_1(d'_1) \curlywedge \gamma_2(d'_2)$ is an equivalence relation. The quotient domain $(\bar{\mathsf{D}}_1 \times \bar{\mathsf{D}}_2)_{/\equiv}$ is the *reduced* Cartesian product of $\bar{\mathsf{D}}_1$ and $\bar{\mathsf{D}}_2$. We let $\bar{\mathsf{D}}_1 \otimes \bar{\mathsf{D}}_2$ denote the reduced product of two abstract domains.

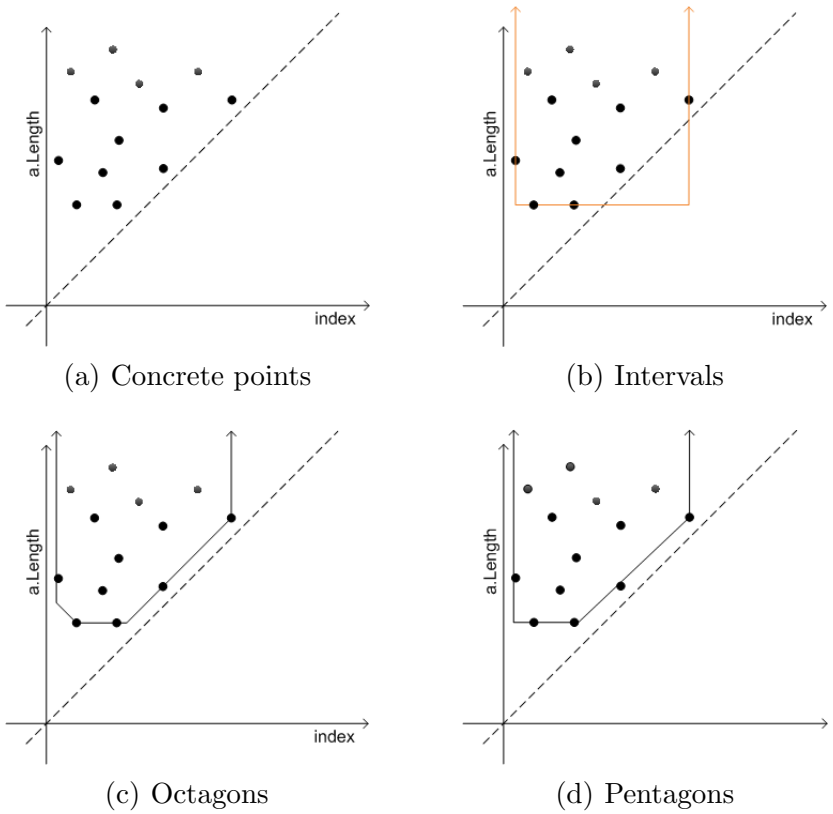
3 Numerical Abstract Domains

A *numerical* abstract domain $\bar{\mathsf{N}}$ is an abstract domain which approximates sets of numerical values, *e.g.*, one such concretization is $\gamma \in [\bar{\mathsf{N}} \to \mathcal{P}(\Sigma)]$, where $\Sigma = [\texttt{Vars} \to \mathbb{Z}]$ is an environment, mapping variables to integers.

When designing numerical abstract domains, one wants to fine tune the cost-precision ratio. Consider the points in Figure 2(a). They represent the concrete values that two variables, `index` and `a.Length`, can take at a given program point for *all* possible executions. As there may be many such values or an unbounded number of them, computing this set precisely is either too expensive or infeasible. Abstract domains over-approximate such sets and thereby make them tractable.

3.1 Intervals

A first abstraction of the points in Fig 2(a) can be made by retaining only the minimum and maximum values of variables `index` and `a.Length`. This is called interval abstraction. Graphically, it boils down to enveloping the concrete values with a rectangle, as depicted in Figure 2(b). The abstract domain of intervals is very cheap, as it requires storing only two integers for each variable, and all the operations can be performed in linear time (w.r.t. the number of variables). However, it is also quite imprecise, in particular because it cannot capture relations between variables. For instance, in Figure 2(b) the fact that $\texttt{index} < \texttt{a.Length}$ is lost.



(a) Concrete points (b) Intervals

(c) Octagons (d) Pentagons

Fig. 2. The concrete points, and some approximations depending on the numerical abstract domain

3.2 Octagons

A more precise abstraction is obtained by using the abstract domain of Octagons, Oct. Oct keeps relations of the form $\pm\mathsf{x} \pm \mathsf{y} \leq k$. When applied to our concrete points, the octagon enveloping them is shown in Figure 2(c). Oct can capture relations between two variables—desirable when analyzing array bounds—but its complexity is $\mathcal{O}(n^2)$ in space and $\mathcal{O}(n^3)$ in time. The cubic complexity is a consequence of the closure algorithm used by all the domain operations. Bagnara *et al.* gave a precise bound for it in [3]. The standard closure operator on Oct performs $20n^3 + 24n^2$ coefficient operations, that can be reduced to $16n^3 + 4n^2 + 4n$ with a smarter algorithm.

While having polynomial complexity, Oct unfortunately does not scale if many variables are kept in the same octagon. For this reason the technique of buckets has been independently introduced in [6] and [37]. The intuition behind it is to create many octagons, each relating few variables, *e.g.*, no more than 4. The problem with this technique is how to choose the bucketing of variables. Existing heuristics use the structure of the source program.

$$
\begin{aligned}
\text{Order:} \quad & [a_1, b_1] \sqsubseteq_i [a_2, b_2] \iff a_1 \geq a_2 \wedge b_1 \leq b_2 \\
\text{Bottom:} \quad & [a, b] = \bot_i \iff a > b \\
\text{Top:} \quad & [a, b] = \top_i \iff a = -\infty \wedge b = +\infty \\
\text{Join:} \quad & [a_1, b_1] \sqcup_i [a_2, b_2] = [\min(a_1, a_2), \max(b_1, b_2)] \\
\text{Meet:} \quad & [a_1, b_1] \sqcap_i [a_2, b_2] = [\max(a_1, a_2), \min(b_1, b_2)] \\
\text{Widening:} \quad & [a_1, b_1] \triangledown_i [a_2, b_2] = [a_1 \leq a_2 ? a_2 : -\infty, b_1 \geq b_2 ? b_2 : +\infty]
\end{aligned}
$$

Fig. 3. Lattice operations over single intervals

3.3 Pentagons

The approximation of the concrete points with Pntg is given in Figure 2(d). Elements of Pntg have the form of $\mathtt{x} \in [a, b] \wedge \mathtt{x} < \mathtt{y}$, where x and y are variables and a and b belong to some underlying numerical set as $\mathbb{Z}$ or $\mathbb{Q}$, extended with $-\infty$ and $+\infty$. A pentagon keeps lower and upper bounds for each variable, so it is as precise as intervals, but it also keeps strict inequalities among variables so that it enables a (limited) form of symbolic reasoning. It is worth noting that the region of the plane that is delimited by a (two dimensional) pentagon may not be closed. In fact, if the underlying numerical values are in $\mathbb{Q}$, then $\mathtt{x} < \mathtt{y}$ denotes an open surface of $\mathbb{Q}^2$, whereas if they are in $\mathbb{Z}$, then $\mathtt{x} < \mathtt{y}$ is equivalent to $\mathtt{x} \leq \mathtt{y} - 1$, which is a closed region of $\mathbb{Z}^2$.

We found pentagons quite efficient in practice. The complexity is $\mathcal{O}(n^2)$, both in time and space. Furthermore, in our implementation we perform the expensive operation (the closure) either lazily or in an incomplete (but sound) way, so that the domain shows an almost linear behavior in practice.

4 Interval Environments

The elements of the abstract domain of intervals, Intv, are $\{[i, s] \mid i, s \in \mathbb{Z} \cup \{-\infty, +\infty\}\}$. The formal definition of the lattice operations on intervals is recalled in Figure 3. The order is the interval inclusion, the bottom element is the empty interval (*i.e.*, an interval where $s < i$), the largest element is the line $[-\infty, +\infty]$, the join and the meet are respectively the convex hull and the intersection of intervals. The widening preserves the bounds which are stable.

The concretization function, $\gamma_{\mathsf{Intv}} \in [\mathsf{Intv} \to \mathcal{P}(\mathbb{Z})]$ is defined as $\gamma_{\mathsf{Intv}}([i, s]) = \{z \in \mathbb{Z} \mid i \leq z \leq s\}$.

The abstract domain of interval environments, Boxes, is the functional lifting of Intv, *i.e.*, $\mathsf{Boxes} = [\mathtt{Vars} \to \mathsf{Intv}]$. The lattice operations are hence the

$$
\begin{array}{rl}
\text{Order:} & b_1 \sqsubseteq_b b_2 \iff \forall \mathtt{x} \in b_1.b_1(\mathtt{x}) \sqsubseteq_i b_2(\mathtt{x}) \\
\text{Bottom:} & b = \bot_b \iff \exists \mathtt{x} \in b.b(\mathtt{x}) = \bot_i \\
\text{Top:} & b = \top_b \iff \forall \mathtt{x} \in b.b(\mathtt{x}) = \top_i \\
\text{Join:} & b_1 \sqcup_b b_2 = \lambda \mathtt{x}.b_1(\mathtt{x}) \sqcup_i b_2(\mathtt{x}) \\
\text{Meet:} & b_1 \sqcap_b b_2 = \lambda \mathtt{x}.b_1(\mathtt{x}) \sqcap_i b_2(\mathtt{x}) \\
\text{Widening:} & b_1 \triangledown_b b_2 = \lambda \mathtt{x}.b_1(\mathtt{x}) \triangledown_i b_2(\mathtt{x})
\end{array}
$$

Fig. 4. Lattice operations of interval environments

$$
\begin{array}{rl}
\text{Order:} & s_1 \sqsubseteq_s s_2 \iff \forall \mathtt{x} \in s_2.s_1(\mathtt{x}) \supseteq s_2(\mathtt{x}) \\
\text{Bottom:} & s = \bot_s \iff \exists \mathtt{x}, \mathtt{y} \in s.\mathtt{y} \in s(\mathtt{x}) \wedge \mathtt{x} \in s(\mathtt{y}) \\
\text{Top:} & s = \top_s \iff \forall \mathtt{x} \in s.s(\mathtt{x}) = \emptyset \\
\text{Join:} & s_1 \sqcup_s s_2 = \lambda \mathtt{x}.s_1(\mathtt{x}) \cap s_2(\mathtt{x}) \\
\text{Meet:} & s_1 \sqcap_s s_2 = \lambda \mathtt{x}.s_1(\mathtt{x}) \cup s_2(\mathtt{x}) \\
\text{Widening:} & s_1 \triangledown_s s_2 = \lambda \mathtt{x}.s_1(\mathtt{x}) \subseteq s_2(\mathtt{x}) ? s_2(\mathtt{x}) : \emptyset
\end{array}
$$

Fig. 5. Lattice operations of strict upper bounds

functional extension of those in Figure 3, as shown by Figure 4.

The concretization of a box, $\gamma_{\mathsf{Boxes}} \in [\mathsf{Boxes} \to \mathcal{P}(\Sigma)]$ is defined as $\gamma_{\mathsf{Boxes}}(f) = \{\sigma \in \Sigma \mid \forall \mathtt{x}.\mathtt{x} \in f \implies \sigma(\mathtt{x}) \in \gamma_{\mathsf{Intv}}(f(\mathtt{x}))\}$.

The assignments and the guards in the interval environment are defined as usual in interval arithmetic [11].

5 Strict upper bounds

The abstract domain of strict upper bounds Sub is a special case of the zone abstract domains [28,17], which keeps symbolic information in the form of $\mathtt{x} < \mathtt{y}$. We represent elements of Sub with maps $\mathtt{x} \mapsto \{\mathtt{y}_1, \ldots \mathtt{y}_n\}$ with the meaning that $\mathtt{x}$ is strictly smaller than each of the $\mathtt{y}_i$. Maps enable very efficient implementations. The formal definition of the lattice operations for Sub is in Figure 5.

Roughly, the fewer constraints the less information is present. As a consequence, the order is given by the (pointwise) superset inclusion, the bottom environment is one which contains a contradiction $\mathtt{x} < \mathtt{y} \wedge \mathtt{y} < \mathtt{x}$ and the lack of information, *i.e.*, the top element is represented by the empty set. The join is (pointwise) set intersection: at a join point we want to keep those relations

that hold on both (incoming) branches. The meet is (pointwise) set union: relations that hold on either the left *or* the right branch. Finally, widening is defined in the usual way: we keep those constraints that are stable in successive iterations.

The concretization function, $\gamma_{\mathsf{Sub}} \in [\mathsf{Sub} \to \mathcal{P}(\Sigma)]$ is defined as $\gamma_{\mathsf{Sub}}(s) = \cap_{\mathtt{x} \in s}\{\sigma \in \Sigma \mid \mathtt{y} \in s(\mathtt{x}) \Longrightarrow \sigma(\mathtt{x}) < \sigma(\mathtt{y})\}$.

We deliberately skipped the discussion of the closure operation until now. One may expect to endow the Sub abstract domain with a saturation rule for transitivity such as

$$\frac{\mathtt{y} \in s(\mathtt{x}) \quad \mathtt{z} \in s(\mathtt{y})}{\mathtt{z} \in s(\mathtt{x})}$$

and apply it to the abstract values prior to applying the join in Figure 5, thereby inferring and retaining the maximum possible constraints. However it turns out that the systematic application of the saturation rule requires $\mathcal{O}(n^3)$ operations, which voids the efficiency advantage of Pntg. In `Clousot`, we chose to not perform the closure, and instead improved the precision of individual transfer functions. They infer new relations $\mathtt{x} < \mathtt{y}$ and use a limited transitivity driven by the program under analysis. So, for instance:

$$\begin{aligned}
[\![\mathtt{x} := \mathtt{y} - 1]\!](s) &= s[\mathtt{x} \mapsto \{\mathtt{y}\} \cup s(\mathtt{y})], \text{if } \mathtt{x} \text{ does not appear in } s \\
[\![\mathtt{x} == \mathtt{y}]\!](s) &= s[\mathtt{x}, \mathtt{y} \mapsto s(\mathtt{x}) \cup s(\mathtt{y})] \\
[\![\mathtt{x} < \mathtt{y}]\!](s) &= s[\mathtt{x} \mapsto s(\mathtt{x}) \cup s(\mathtt{y}) \cup \{\mathtt{y}\}] \\
[\![\mathtt{x} \leq \mathtt{y}]\!](s) &= s[\mathtt{x} \mapsto s(\mathtt{x}) \cup s(\mathtt{y})]
\end{aligned}$$

because we know that (i) if we subtract a positive constant from a variable we obtain a result strictly smaller [1], that (ii) when we compare two variables for equality they must satisfy the same constraints, and that (iii) for each $\mathtt{z}$ such that $\mathtt{y} < \mathtt{z}$, if $\mathtt{x} < \mathtt{y}$ or $\mathtt{x} \leq \mathtt{y}$ then $\mathtt{x} < \mathtt{z}$.

6 Pentagons

A first approach to combine the numerical properties captured by Intv, and the symbolic ones captured by Sub is to consider the Cartesian product $\mathsf{Intv} \times \mathsf{Sub}$. Such an approach is equivalent to running the two analyses in parallel, without any exchange of information between the two domains. A better solution is to perform the *reduced* Cartesian product $\mathsf{Intv} \otimes \mathsf{Sub}$. Roughly, the reduced Cartesian product of a product lattice smashes together the pairs that have

[1] In this paper we ignore overflows. However our abstract semantics of arithmetic expressions in `Clousot` takes care of them.

the same concrete meaning. The Pntg abstract domain is an abstraction of the reduced product and is more precise than the Cartesian product.

The lattice operations are defined in Figure 6. The functions sup and inf are defined as $\inf([a,b]) = a$ and $\sup([a,b]) = b$.

The order on Pntg is a refined version of the pairwise order: a pentagon $\langle b_1, s_1\rangle$ is smaller than a pentagon $\langle b_2, s_2\rangle$ iff the interval environment b_1 is included in b_2 and for all the symbolic constraints $\mathtt{x} < \mathtt{y}$ in s_2, either $\mathtt{x} < \mathtt{y}$ is an explicit constraint in s_1 or it is implied by the interval environment b_1, *i.e.*, the numerical upper bound for $\mathtt{x}$ is strictly smaller than the numerical lower bound for $\mathtt{y}$.

A pentagon is bottom if either its numerical component *or* the symbolic component are. A pentagon is top if both the numerical component *and* the symbolic component are.

For the numerical part, the join operator pushes the join to the underlying Intv abstract domain, and for the symbolic part, it keeps the constraints which are either explicit in the two operators *or* which are explicit in one operator, and implied by the numerical domain in the other component. We will further discuss the join, cardinal for the scalability and the precision of the analysis below.

The meet and the widening operators simply delegate the meet and the widening to the underlying abstract domains. Note that we do not perform any closure before widening in order to avoid well known convergence problems arising from the combination of widening and closure operations [27].

6.1 Cost and Precision of the Join

One may ask why we defined the join over Pntg as in Figure 6. In particular, a more natural definition may be to first close the two operands, by deriving all the symbolic and numerical constraints, and then perform the join. This is for instance how the standard join of Oct works. More formally one may want to have a closure for a pentagon $\langle b, s\rangle$ defined by:

$$b^* = \bigsqcap_{\mathtt{x}<\mathtt{y}\in s} [\![\mathtt{x} < \mathtt{y}]\!](b)$$
$$s^* = \lambda \mathtt{x}.s(\mathtt{x}) \cup \{\mathtt{y} \in b \mid \mathtt{x} \neq \mathtt{y} \Longrightarrow \sup(b^*(\mathtt{x})) < \inf(b^*(\mathtt{y}))\}$$

The closure first refines the interval environment by assuming all the strict inequalities of the Sub domain. Then, it closes the element of the Sub domain by adding all the strict inequalities implied by the numerical part of the abstract domain.

$$
\begin{array}{rl}
\text{Order:} & \langle b_1, s_1\rangle \sqsubseteq_p \langle b_2, s_2\rangle \iff b_1 \sqsubseteq_b b_2 \\
& \qquad \wedge(\forall \mathtt{x} \in s_2 \forall \mathtt{y} \in s_2(\mathtt{x}).\mathtt{y} \in s_1(\mathtt{x}) \vee \ \sup(b_1(\mathtt{x})) < \inf(b_1(\mathtt{y})))\\
\text{Bottom:} & \langle b, s\rangle = \bot_p \Rightarrow b = \bot_b \vee s = \bot_s \\
\text{Top:} & \langle b, s\rangle = \top_p \iff b = \top_b \wedge s = \top_s \\
\text{Join:} & \langle b_1, s_1\rangle \sqcup_p \langle b_2, s_2\rangle = \\
& \quad \text{let } b^{\sqcup} = b_1 \sqcup_b b_2 \\
& \quad \text{let } s^{\sqcup} = \lambda \mathtt{x}.s'(\mathtt{x}) \cup s''(\mathtt{x}) \cup s'''(\mathtt{x}) \\
& \qquad \text{where } s' = \lambda \mathtt{x}.s_1(\mathtt{x}) \cap s_2(\mathtt{x}) \\
& \qquad \text{and } s'' = \lambda \mathtt{x}.\{\mathtt{y} \in s_1(\mathtt{x}) \mid \sup(b_2(\mathtt{x})) < \inf(b_2(\mathtt{y}))\} \\
& \qquad \text{and } s''' = \lambda \mathtt{x}.\{\mathtt{y} \in s_2(\mathtt{x}) \mid \sup(b_1(\mathtt{x})) < \inf(b_1(\mathtt{y}))\} \\
& \quad \text{in } \langle b^{\sqcup}, s^{\sqcup}\rangle \\
\text{Meet:} & \langle b_1, s_1\rangle \sqcap_p \langle b_2, s_2\rangle = \langle b_1 \sqcap_b b_2, s_1 \sqcap_s s_2\rangle \\
\text{Widening:} & \langle b_1, s_1\rangle \triangledown_p \langle b_2, s_2\rangle = \langle b_1 \triangledown_b b_2, s_1 \triangledown_s s_2\rangle
\end{array}
$$

Fig. 6. The lattice operations over Pentagons

As a consequence, the closure-based join $\sqcup_p^*$ can be defined as

$$\langle b_1, s_1\rangle \sqcup_p^* \langle b_2, s_2\rangle = \langle b_1^* \sqcup_b b_2^*, s_1^* \sqcup_s s_2^*\rangle.$$

The complexity of $\sqcup_p^*$ is $\mathcal{O}(n^2)$, as for getting s^* we need to consider all the pairs of intervals in b^*.

Performing a quadratic operation at each join point imposes a serious slowdown of the analysis. We experienced the quadratic blowup in our tests (Section 8).

As a consequence we defined a safe approximation of the join as in Figure 6. The idea behind $\sqcup_p$ is to avoid materializing new symbolic constraints, but just to keep those which are present in one of the two operators, and implied by the numerical part of the other operand. If needed, some implied relations may be recovered later (hence lazily), after the join point. The next example illustrates this on an assertion following a join point.

Example. Let us consider the code in Figure 7(a), to be analyzed with some initial pentagon $\langle b, s\rangle$ which does not mention $\mathtt{x}$ and $\mathtt{y}$. Using $\sqcup_p^*$, one gets the post-state

$$p_1 = \langle b[\mathtt{x} \mapsto [-2, 0], \mathtt{y} \mapsto [1, 3]], s[\mathtt{x} \mapsto \{\mathtt{y}\}]\rangle.$$

With $\sqcup_p$ the result is

$$p_2 = \langle b[\mathtt{x} \mapsto [-2, 0], \mathtt{y} \mapsto [1, 3]], s]\rangle.$$

```
if (...)
  x = 0; y = 3;
else
  x = −2; y = 1;
```
(a) Non-strict abstraction

```
if (...)
  x = 0; y = 3;
else
  x = −2; y = 0;
```
(b) Strict abstraction

Fig. 7. Difference in precision between $\sqcup_p^*$ and $\sqcup_p$

Suppose that we would like to discharge `assert x < y` following the conditional. The first pentagon, p_1 already contains the constraint `x < y`, thus proving the assertion is as complex as a simple table lookup. On the other hand, the symbolic part of p_2 does not contain the explicit constraint `x < y`, but it is implied by the numerical part. Proving the assertion with p_2 requires two table lookups and an integer comparison. □

One may argue that $\sqcup_p$ is just a lazy version of $\sqcup_p^*$. However it turns out that the abstraction is strict, in that there are cases where $\sqcup_p$ introduces a loss of information that cannot be recovered later, as shown by the next example.

Example. Let us consider the code in Figure 7(b), to be analyzed with some initial pentagon $\langle b, s\rangle$, which does not mention `x` and `y`. Using the closure-based join, $\sqcup_p^*$ one obtains the pentagon

$$p_3 = \langle b[\mathtt{x} \mapsto [-2, 0], \mathtt{y} \mapsto [0, 3]], s[\mathtt{x} \mapsto \{\mathtt{y}\}]\rangle.$$

which implies that `x` and `y` cannot be equal to 0 at the same time. On the other hand, $\sqcup_p$ returns

$$p_4 = \langle b[\mathtt{x} \mapsto [-2, 0], \mathtt{y} \mapsto [0, 3]], s]\rangle.$$

which does not exclude the case when $\mathtt{x} = \mathtt{y} = 0$. As a consequence, `assert x+ y` $\neq 0$ cannot be proved using p_4, whereas it can be with p_3. □

Even if the previous example shows that there may be some loss of precision induced by using $\sqcup_p$, we found it negligible in practice (see Sect. 8). We also tried a hybrid solution, where we fixed some $\overline{n}$. If the cardinality of the abstract elements to join was $n < \overline{n}$, then the we used $\sqcup_p^*$, otherwise we used $\sqcup_p$. However, we did not find any values for $\overline{n}$ with a better cost-precision trade-off.

6.2 Transfer Functions

Analysis precision also heavily depends on the precision of the transfer functions. Using Pntg we can refine the transfer functions for some MSIL instruc-

```
assume x >= 0 & y >= 0;
if (x > y)
  r := sub x y;
  assert r >= 0;
```
(a) Underflow checking

```
assume len >= 0;
r := rem x len;
assert r < len;
```
(b) Overflow checking

Fig. 8. Common code patterns in `mscorlib.dll`. The instructions `sub` and `rem` denote respectively subtraction and remainder. Proving the two assertions requires a combination of numerical and symbolic information.

tions which have a non-trivial behavior depending on the operators. We illustrate the situation with two representative MSIL instructions: subtraction and remainder. The precise handling of subtraction is cardinal to prove the absence of array underflows, whereas the precise handling of remainder is cardinal to prove the absence of array overflows.

6.2.1 Subtraction

The concrete semantics for `sub x y` subtracts `x` from `y` and pushes the result onto the evaluation stack, [16].

The transfer function for `sub` in Intv first evaluates `x` and `y` to intervals, then it performs interval subtraction. For instance, in an interval environment b such that $b = [\mathtt{x} \mapsto [-1, 2], \mathtt{y} \mapsto [0, 4]]$ then $[\![\mathtt{r} := \mathtt{sub\ x\ y}]\!](b) = b[\mathtt{r} \mapsto [-5, 2]]$.

On the other hand, when the bounds are not finite, the interval transfer function for `sub` may be quite imprecise, as shown by the next example.

Example. Let us consider the code snippet in Figure 8(a), to be analyzed with Intv. The abstract state after the `assume` statement is $b_1 = [\mathtt{x} \mapsto [0, +\infty], \mathtt{y} \mapsto [0, +\infty]]$. The guard refines b_1 to $b_2 = [\mathtt{x} \mapsto [1, +\infty], \mathtt{y} \mapsto [0, +\infty]]$, as it can never be the case that `x == y == 0`. The assignment does not derive any interesting bound for `r`, as $[1, +\infty] - [0, +\infty] = \top_i$, so that the assertion cannot be proved. □

The code snippet if Figure 8(a) abstracts away a common pattern we have found in .NET assemblies. A precise handling of such situations is cardinal to prove the absence of array access underflows. In Pentagons, we refine the numerical information captured by Intv with the symbolic information captured by Sub. Assuming `r` to be a fresh variable, the transfer function for `sub` in

Pntg is defined as:

$$[\![\mathtt{r} := \mathtt{sub\ x\ y}]\!](\langle b, s\rangle) = \langle b[\mathtt{r} \mapsto (b(\mathtt{x}) - b(\mathtt{y})) \sqcap_i (\mathtt{x} \in s(\mathtt{y})?[1, +\infty] : \top_i)], \\ s[\mathtt{r} \mapsto b(\inf(\mathtt{y})) > 0?\{\mathtt{x}\} \cup s(\mathtt{x}) : \emptyset]$$

because i) if we know that $\mathtt{y} < \mathtt{x}$, then $\mathtt{x} - \mathtt{y}$ should be at least 1, and ii) if we subtract a strictly positive quantity from $\mathtt{x}$, then $\mathtt{r} < \mathtt{x}$, and if $\mathtt{t}$ is a strict upper bound for $\mathtt{x}$, then it is for $\mathtt{r}$, too.

Example. Let us consider the code in Figure 8(a) to be analyzed with Pntg. The abstract state after the `assume` statement is $p_1 = \langle[\mathtt{x} \mapsto [0, +\infty], \mathtt{y} \mapsto [0, +\infty]], \emptyset\rangle$. The guard is precisely captured by the symbolic component of the abstract domain, so p_1 is refined to $p_2 = \langle[\mathtt{x} \mapsto [1, +\infty], \mathtt{y} \mapsto [0, +\infty]], \mathtt{y} \mapsto \{\mathtt{x}\}\rangle$. The transfer function for `sub` uses such information to derive a tighter bound for `r`. The abstract state after the assignment is $p_3 = \langle[\mathtt{r} \mapsto [0, +\infty], \mathtt{x} \mapsto [1, +\infty], \mathtt{y} \mapsto [0, +\infty]], \mathtt{y} \mapsto \{\mathtt{x}\}\rangle$, which is enough to prove the assertion. □

6.2.2 Remainder

Intuitively, `rem u d` computes the remainder of the division `u/d`. The precise handling of the remainder is important as many expressions used to access arrays in `mscorlib.dll` include the remainder operation. According to the definition of `rem` in Part. III, Sect. 3.55 of [16], the sign of the result is the sign of `u` and $0 \leq |\mathtt{rem\ u\ d}| < |\mathtt{d}|$ holds. Therefore in order to derive the constraint $\mathtt{rem\ u\ d} < \mathtt{d}$ one must know that $\mathtt{d} \geq 0$.

The transfer function for `rem` in Intv can infer useful upper bounds whenever `d` is finite, but it infers unhelpful bounds when `d` is infinite.

The transfer function for `rem` in Sub cannot infer lower bounds, and worse, no upper bounds, for it cannot determine the sign of `d`.

The transfer function for `rem` in Pntg has the necessary information. It uses Intv to determine if `d` is non-negative in the pre-state, then constrains the result using Sub, modeling the assignment more precisely.

$$[\![\mathtt{r} := \mathtt{rem\ u\ d}]\!](\langle b, s\rangle) = \langle [\![\mathtt{r} := \mathtt{rem\ u\ d}]\!](b), s[\mathtt{x} \mapsto (\inf(b(\mathtt{d})) \geq 0)?\{\mathtt{d}\} : \emptyset\rangle.$$

Example Let us consider the code in Figure 8(b). Intervals alone cannot prove the assertion: The upper bound for `len` is $+\infty$, so that any interesting relation between `r` and `len` can be inferred. Strict upper bounds do not capture the numerical assumption $\mathtt{len} \geq 0$, so they cannot deduce that $\mathtt{r} < \mathtt{len}$. The transfer function for `rem` for Pentagons is precise enough to deduce from the sign `len` to deduce that $\mathtt{r} < \mathtt{len}$ and hence to prove the assertion. □

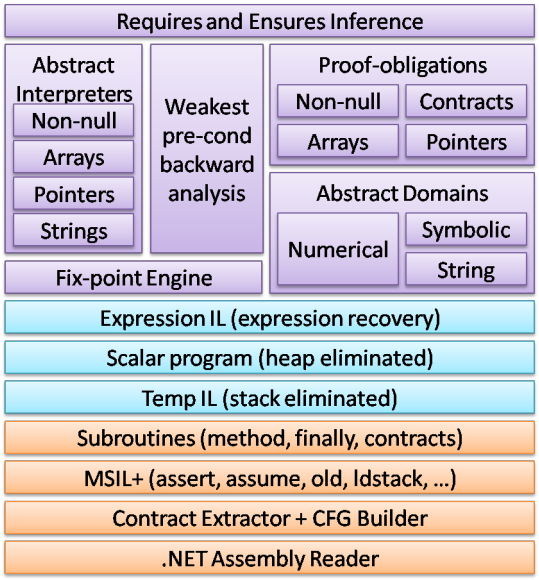


Fig. 9. The `Clousot` architecture.

7 Clousot

We have implemented the abstract domain `Pntg` in our analyzer for .NET assemblies, `Clousot`. `Clousot` is part of the Code Contracts tools [5]. Code Contracts provide a language-agnostic way to express coding assumptions in .NET programs. The contracts take the form of preconditions, postconditions, and object invariants. `Clousot` checks each method in isolation, by assuming the precondition and asserting the postcondition.

The detailed structure of the analyzer is given in Figure 9. `Clousot` directly analyzes assemblies as produced by different .NET compilers as `csc` (for C#), `vbc` (for VB), `fsc` (for F#), etc. At the bottom of the stack are the code providers, which provide a low-level and stack-based, view of the code. `Clousot` has a pluggable architecture that allows for different code providers and contract providers. For instance, one code provider can read assemblies from disk, another could be a compiler, providing the code being compiled directly.

The basic code fragment used is a *subroutine*. We use subroutines to represent several different aspects of code, such as fault/finally handlers, pre- and post-conditions, and the normal method bodies. These snippets are composed to form a complete control flow graph. This approach makes it easy, for example, to form inheritance chains for object invariants and postconditions, as well as sharing precondition subroutines from all call-sites.

The higher abstraction levels take care of presenting a uniform view of the code for specific analyses. They take care of (i) injecting contracts as `assume`/

```
if (*)
  x := −1;
else
  x := 1;
assert x < 0||x > 0;
```
(a) Disjunctive assertion

```
Caller()
  return FreshArray(5);

FreshArray(int l)
  return new int[l − 2];
```
(b) Overflow checking

Fig. 10. Assertion checking and precondition propagation in `Clousot`. In the first case, a local backward analysis is used to discharge the assertion in the two incoming paths. In the second case, the implicit proof obligation $\texttt{l} \geq 2$ of `NewArray` is propagated to the caller.

`assert` instructions; (ii) removing the evaluation stack; (iii) abstracting the heap; and, (iv) reconstructing the expressions that were lost by compilation [25].

The value analyses at the top of the stack *view* the code as a normalized scalar program (similar to SSA form), where all the heap accesses have been resolved and the contracts have been turned into a series of `assume` and `assert` statements. In particular, when analyzing a public method `m`, of a class `C`, the precondition of `m` and the object invariant of `C` are turned into assumptions at the entry point, and the postcondition and the object invariant of `C` are injected as assertions at the exit point of `m`. When analyzing a method which invokes `m`, the precondition of `m` is asserted at the program point just before the call, and the postcondition is assumed just after the call.

`Clousot` has two main phases: Analysis and checking. The analysis phase is a forward abstract interpretation instantiated with a user-specified abstract domain. The entry (abstract) state is propagated through the method body, and loops are handled with usual fixpoint computation. In theory, `Clousot` infers an invariant for each program point. In practice, it stores only the invariants at the loop headers to save memory.

In the checking phase, `Clousot` collects the proof obligations and tries to validate them. There are two kinds of proof obligations: (i) implicit, as for instance array bounds checking and non-null dereferences; and (ii) explicits as preconditions, postconditions, and user-provided assertions. If `Clousot` cannot discharge a proof obligation, it iteratively refines the inferred invariants re-analyzing the method with more precise abstract domains, as *e.g.* SubPolyhedra [23]. If the iterative refinement fails, then the analyzer performs a local backward analysis trying to validate the assertion for all the incoming paths. This is useful to prove disjunctive assertions. For instance, in Figure 10(a), `Pntg` (as well as other non-disjunctive abstract domains) cannot validate the assertion. However, it can be propagated backwards to the two branches of

Assembly	Bounds checked	Intv Valid	Intv %	Intv Time	Intv × Sub Valid	Intv × Sub %	Intv × Sub Time
mscorlib.dll	17 181	12 538	72.98	3:38	14 263	83.02	3:03
System.dll	11 891	9 574	80.51	3:01	10 319	86.78	2:28
System.Web.dll	14 165	12 350	87.19	3:39	13 030	91.99	2:49
System.Design.dll	10 519	9 322	88.62	2:56	10 132	96.32	2:18
Average			81.45			88.82	

Fig. 11. The experimental results of the analyzer instantiated with Intervals alone, and with Intervals combined with Strict inequalities.

the conditional, and then discharged in each of them. If the backward analysis fails, `Clousot` tries to push the proof obligation to the callers of the method. For instance, in Figure 10(b), the array creation induces an implicit proof obligation $\texttt{l} \geq \texttt{2}$ which cannot be discharged locally. `Clousot` propagates it to the callers and checks it at the call sites (in the example $\texttt{5} \leq \texttt{2}$ trivially holds). If none of the checking above succeed, then `Clousot` reports a warning to the user.

8 Experimental Results

We present the experimental results of instantiating `Clousot` with Pntg. For arrays, `Clousot` tries to validate that (i) the expression for a `newarr` instruction is non-negative, and (ii) the index for the `ldelem`, `stelem`, and `ldelema` instructions is greater than or equal to zero and strictly smaller than the length of the array.

Figure 11, 12 and 13 summarize the results of running the analysis on a subset of the .NET framework assemblies. The analyzed assemblies are taken from the directory %`WINDIR`%\`Microsoft`\ `Framework`\`v2.0.50727` of our laptop without modification or preprocessing. The annotation of the framework libraries is ongoing [5]. To provide an uniform test bench, we turned off the inter-method capabilities of `Clousot`. All experiments were conducted on a Centrino 2 duo processor at 2.16 GHz, with 4 GB of RAM, running Windows Vista.

We ran the analysis with five different domains: Intv alone and the Cartesian product Intv×Sub (Figure 11); Pntg without and with constraint closure (Figure 12); and Oct (Figure 13). We set a time out of 2 minutes per method. In order to provide an uniform test bench, we switched off the backward analysis and the precondition inference.

	Pntg $\sqcup_p$			Pntg $\sqcup_p^*$			
Assembly	Valid	%	Time	Valid	%	Time	Timeout
`mscorlib.dll`	14 293	83.19	3:10	14 220	82.77	10:33	1
`System.dll`	10 321	86.80	2:36	10 143	85.30	9:43	1
`System.Web.dll`	13 034	92.02	2:55	13 048	92.11	8:30	0
`System.Design.dll`	10 135	96.35	2:21	9 947	94.56	7:39	1
Average		88.89			88.10		3

Fig. 12. The experimental results of the analyzer instantiated with Pentagons and two different joins.

The results show that with Pntg, `Clousot` is able to validate on average 88.9% of all array accesses in a little bit more than 3 minutes for the analyzed .NET assemblies. As for the memory footprint, the analyzer never exceeded 300 Mbytes of RAM.

8.1 Pentagons and non relational domains

Figure 11 shows that combining Intv with symbolic upper bounds validates on average almost 10% more array accesses than Intv alone for a modest extra cost. Pntg without closure validate 78 extra accesses. Pntg with closure produces almost no extra precision but the analysis time sensibly increases. The analysis of three methods was aborted because it reached the 2 minutes timeout. We manually inspected those methods. They turned out to be long methods (more than 1 300 instructions) with complex control flow graphs using many distinct integer constants. Constants are captured by the numerical component of Pntg. At join points, the closure step of $\sqcup_p^*$ materialized a quadratic number of new symbolic constraints which caused the slow down.

8.2 Pentagons and Octagons

Figure 13 presents the running times of Clousot instantiated with Octagons. Our implementation of Oct uses sharing and sparse arrays to optimize performances. Octagons caused an explosion of the analysis time: 35 methods timed out. A larger timeout did not helped. We inspected some of those methods. Once again, the problem is related to the propagation of numerical constants. For instance, if $\mathtt{b} = 1$ and $\mathtt{y} = 2$, then the closure operation on Oct deduces the constraints $\mathtt{b} - \mathtt{y} \leq -1$, $\mathtt{b} + \mathtt{y} \leq 3$, $-\mathtt{b} - \mathtt{y} \leq -3$ and $-\mathtt{b} + \mathtt{y} \leq 1$. However, if `b` and `y` correspond in the source code respectively to a `bool` variable and `int`

variable (Boolean are compiled to integers) then those constraints are meaningless. It turns out that octagon constraints may relate too many *logically* unrelated variables. This behavior has already been observed by Miné [29] and Venet [37], who proposed a solution based on the use of buckets. The main idea behind buckets is to decompose an octagon of n variables into a set of k smaller octagons (buckets) each one with at most n/k variables (typically four). Buckets may share some variables (pivots).

In our setting, *i.e.* the analysis of MSIL, it is not clear how to partition variables into buckets or how to select the pivots. Syntactic scope-based approaches do not work: in MSIL nested scopes inside methods are flatten. Semantic based approaches as some form of backward analysis or type inference may be as expensive as the analysis with Pntg itself.

Pentagons are not a replacement for Oct. Octagons can validate array accesses which are out of the scope of Pntg, for instance because they involve the relation between two variables and a numerical offset. We manually inspected the analysis logs for `mscorlib.dll`. Octagons can validate 177 more array accesses than Pentagons. To increase precision, `Clousot` allows a combination of Pntg and Oct where constants are represented *only* in Pntg and Oct *only* tracks symbolic relations involving *exactly* two variables, that is constraints in the form $\mathtt{x} - \mathtt{y} \leq k$ or $\mathtt{x} + \mathtt{y} \leq k$.

In general we found Oct not to be a good compromise for `Clousot`. On one hand, the cost to validate implicit proof obligation as array accesses with Oct is too elevated to be used in a build environment. On the other hand, Oct are not expressive enough to support assume/guarantee reasoning. In our experience, while most (numerical) preconditions have the shape of Octagonal constraints (*e.g.*$\mathtt{x} - \mathtt{y} < k$), proving that they are established at the call site it requires a relatively complex reasoning that often involves more than just variables (*e.g.*. Figure 1 of [23]). Pntg provide a better compromise than Oct in the managed contracts setting. They allow to quickly discharge "easy" proof obligations, and to leave the more complex ones to more expressive yet expensive abstract domains as SubPolyhedra.

9 Related work

Early works on numerical abstract domains focused on program optimization. Kildall described constant propagation in [21], the first example of a numerical abstract domain. Karr refined constant propagation using linear *equalities* [20]. Cousot and Cousot [13] introduced Intv as an example of abstract interpretation applied to bounds checking elimination.

	Oct	
Assembly	Time	Timeout
`mscorlib.dll`	1:38:43	20
`System.dll`	1:09:00	13
`System.Web.dll`	21:49	1
`System.Design.dll`	17:44	1
		35

Fig. 13. The execution times of the analyzer instantiated with Octagons

Cousot and Halbwachs noticed that numerical abstract domain can be used for program verification [14]. They used Polyhedra, a numerical abstract domain to infer linear *inequalities*, *i.e.* constraints in the form $a_1 \cdot \mathtt{x}_1 + a_2 \cdot \mathtt{x}_2 \ldots a_n \mathtt{x}_n \cdot \leq k$. Polyhedra are very powerful: they can be used for bounds checking, integer overflow detection, timing analysis, alias analysis, etc. On the other hand, they encounter serious scalability issues [18]. Later research focused on optimizing Polyhedra.

The model checking and the constraint programming communities developed Difference bounds matrices (DBM) to handle constraints in the form $\mathtt{x} - \mathtt{y} \leq k$ or $\mathtt{x} \leq k$. DBM are used to model-check timed automata [2,9] or answer queries [32]. Miné extended DBM to fully-featured abstract domain (Oct, [26,27]) able to infer constraints in the form $\pm\mathtt{x} \pm \mathtt{y} \leq k$ in polynomial time. A nice property of Oct is that the underlying implementation can be easily parallelized, *e.g.* to exploit the power of modern graphics cards [4]. Octagons have been generalized by Octahedra which capture linear constraints with *unary* coefficients: $\pm\mathtt{x}_1 \pm \mathtt{x}_2 \cdots \pm \mathtt{x}_n \leq k$. Octahedra have an exponential worst case complexity which is reached in practice, [8].

Sankaranarayanan *et al.* proved that a polynomial time algorithm for linear inequalities inference can be obtained either (i) by fixing the number of linear equations *before* the analysis [34], or (ii) by fixing a partial order between the variables in the program [33]. In our setting, neither of the two hypotheses is realistic.

Laviron and Logozzo introduced SubPolyhedra [23], which retain the same expressive power of Polyhedra, but they give up some of the inference power in order to achieve scalability. SubPolyhedra precision can be finely tuned using hints.

Popeea *et al.* [31] presented an interesting analysis to infer sufficient preconditions to eliminate bounds check inside method bodies. They underlying domain to their technique is Polyhedra. It would be interesting to see if faster

results can be obtained using Pntg.

Xi and Pfenning [38] presented a type checker to eliminate array bound checking in ML programs. Courbot *et al.* [10] advocated the use of formal methods to optimize Java programs. Unlike those two approaches, Pntg can synthesize loop invariants, hence presenting an higher level of automation.

Our analysis competes in performance with analyses developed to be used by the JIT, but it seems more precise: we got an average precision close to 89% versus the 45% reported in [7].

Dor *et al.* [15] use Poly and Allamigeon *et al.* [1] use Intv and a whole program analysis to check overruns in string manipulation. For the modular nature of the code that we analyze we cannot perform a whole program analysis, and the use of Intv without any symbolic reasoning on upper bounds produces an analysis which is too imprecise (cf. Figure 11). Larochelle *et al.* [22] present an approach for buffer overrun checking similar to ours, using contracts and static checking. However, their static analysis is limited by the syntax-oriented handling of loop invariants.

10 Conclusions

We presented a new numerical abstract domain, Pntg. We described its lattice operations, discussed its complexity and presented an optimized algorithm for the join operator which runs in (almost) linear time (instead of quadratic).

This abstract domain sits, as precision and cost are concerned, in between the abstract domains of intervals and octagons.

We used Pntg to validate on average over 89% of array accesses in four major .NET assemblies in a couple of minutes. The remaining unproven accesses are discharge by using more precise, yet expensive domains on demand.

Acknowledgments. We would like to thank the Anindya Banerjee, Corneliu Popeea, Pietro Ferrara and Vincent Laviron.

References

[1] X. Allamigeon, W. Godard, and C. Hymans. Static analysis of string manipulations in critical embedded C programs. In *SAS'06*. Springer-Verlag, August 2006.

[2] R. Alur and D. L. Dill. Automata for modeling real-time systems. In *ICALP'90*, July 1990.

[3] R. Bagnara, P. M. Hill, E.Mazzi, and E. Zaffanella. Widening operators for weakly-relational numeric abstractions. In *SAS'05*. Springer-Verlag, September 2005.

[4] F. Banterle and R. Giacobazzi. A fast implementation of the octagon abstract domain on graphics hardware. In *SAS'07*, August 2007.

[5] M. Barnett, M. Fähndrich, and F. Logozzo. Codecontracts for .net. `http://msdn.microsoft.com/en-us/devlabs/dd491992.aspx`.

[6] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *PLDI'03*. ACM Press, June 2003.

[7] R. Bodík, R. Gupta, and V. Sarkar. ABCD: Eliminating array bounds checks on demand. In *PLDI'00*. ACM Press, 2000.

[8] R. Clarisó and J. Cortadella. The octahedron abstract domain. *Sci. Comput. Program.*, 64(1), 2007.

[9] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, 1999.

[10] A. Courbot, M. Pavlova, G. Grimaud, and J.-J. Vandewalle. A low-footprint java-to-native compilation scheme using formal methods. In *CARDIS'06*, LNCS. Springer-Verlag, April 2006.

[11] P. Cousot. The calculational design of a generic abstract interpreter. In *Calculational System Design*. NATO ASI Series F. IOS Press, Amsterdam, 1999.

[12] P. Cousot. Verification by abstract interpretation. In *Verification: Theory and Practice*. Springer-Verlag, 2003.

[13] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL'77*. ACM Press, January 1977.

[14] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL '78*. ACM Press, January 1978.

[15] N. Dor, M. Rodeh, and M. Sagiv. CSSV: towards a realistic tool for statically detecting all buffer overflows in c. In *PLDI'03*. ACM Press, 2003.

[16] ECMA. Standard ECMA-335, Common Language Infrastructure (CLI). `http://www.ecma-international.org/publications/standards/Ecma-335.htm`, Ecma International, 2006.

[17] S. Gaubert, E. Goubault, A. Taly, and S. Zennou. Static analysis by policy iteration on relational domains. In *ESOP'07*, April 2007.

[18] N. Halbwachs, D. Merchat, and L. Gonnord. Some ways to reduce the space dimension in polyhedra computations. *Formal Methods in System Design*, 29(1):79–95, 2006.

[19] ECMA Int. Standard ECMA-355, Common Language Infrastructure, June 2006.

[20] M. Karr. On affine relationships among variables of a program. *Acta Informatica*, 6(2):133–151, July 1976.

[21] G. A. Kildall. A unified approach to global program optimization. In *POPL '73*. ACM Press, October 1973.

[22] D. Larochelle and D. Evans. Statically detecting likely buffer overflow vulnerabilities. In *2001 USENIX Security Symposium*, August 2001.

[23] V. Laviron and F. Logozzo. Subpolyhedra: a (more) scalable approach to infer linear inequalities. In *VMCAI'09*, January 2009.

[24] F. Logozzo. Cibai: An abstract interpretation-based static analyzer for modular analysis and verification of Java classes. In *VMCAI'07*. Springer-Verlag, January 2007.

[25] F. Logozzo and M. A. Fähndrich. On the relative completeness of bytecode analysis versus source code analysis. In *CC'08*, LNCS. Springer-Verlag, March 2008.

[26] A. Miné. A new numerical abstract domain based on difference-bounds matrices. In *PADO'01*. Springer-Verlag, May 2001.

[27] A. Miné. The octagon abstract domain. In *WCRE 2001*. IEEE Computer Society, October 2001.

[28] A. Miné. A few graph-based relational numerical abstract domains. In *SAS'02*, September 2002.

[29] A. Miné. *Weakly Relational Numerical Abstract Domains*. PhD thesis, École Polythechnique, 2004.

[30] J. Navas, Ed. Mera, P. López-García, and M. V. Hermenegildo. User-definable resource bounds analysis for logic programs. In *ICLP'07*. Springer-Verlag, September 2007.

[31] C. Popeea, D. N. Xu, and W.-N. Chin. A practical and precise inference and specializer for array bound checks elimination. In *PEPM'08*, 2008.

[32] P. Z. Revesz. The constraint database approach to software verification. In *VMCAI'07*, January 2007.

[33] S. Sankaranarayanan, F. Ivancic, and A. Gupta. Program analysis using symbolic ranges. In *SAS'07*, August 2007.

[34] S. Sankaranarayanan, H. B. Sipma, and Z. Manna. Scalable analysis of linear systems using mathematical programming. In *VMCAI'05*, pages 25–41, January 2005.

[35] A. Simon, A. King, and J. M. Howe. Two variables per linear inequality as an abstract domain. In *LOPSTR'02*. Springer-Verlag, 2002.

[36] A. Venet. Nonuniform alias analysis of recursive data structures and arrays. In *SAS'02*. Springer-Verlag, September 2002.

[37] A. Venet and G. P. Brat. Precise and efficient static array bound checking for large embedded c programs. In *PLDI'04*. ACM Press, July 2004.

[38] H. Xi and F. Pfenning. Eliminating array bound checking through dependent types. In *PLDI'98*. ACM Press, 1998.