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

Francesco Logozzo & Manuel Fähndrich
Microsoft Research
{ logozzo, maf }@microsoft.com

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 less than 8 minutes.

Keywords

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 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.*, [6, 9, 11], that are useful to check properties such as out of bounds and division by zero, but also aliasing [12], parametric predicate abstraction [3] and resource usage [10].

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. 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.*, [8]). 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 Fig. 1, taken from the basic component library of .NET. `Clousot`, instantiated with the abstract domain Pntg, automatically discovers the following invariant at program point $(*)$:

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

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

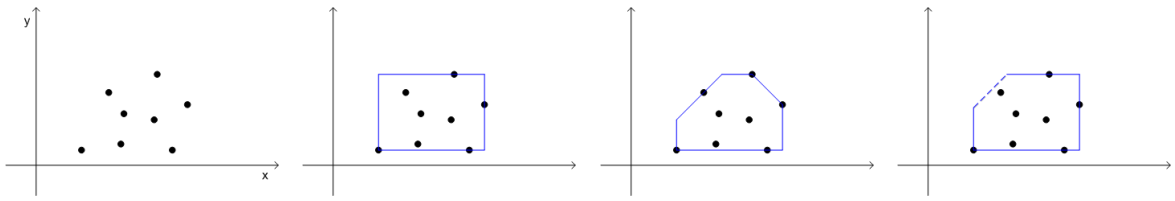
The elements of Pntg are of the form $\mathtt{x} \in [a, b] \wedge \mathtt{x} < \mathtt{y}$, where $\mathtt{x}$ and $\mathtt{y}$ are program variables and a, b are rationals. Such elements allow expressing (most) bounds of program variables, and in particular those of array indices: intervals $[a, b]$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

Pntg is therefore an abstract domain more precise than the widespread Intervals, Intv [4], as it adds symbolic reasoning, but it is less precise than Octagons, Oct [9], as it cannot for instance capture equalities such as $\mathtt{x} + \mathtt{y} == 22$. 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 8 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 analysis. In fact, when used on part of `Clousot`'s implementation, it validates only 65.6% of the accesses.

2. NUMERICAL ABSTRACT DOMAINS

Abstract interpretation is a theory of approximations, [4]. 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 $\langle \mathsf{E}, \sqsubseteq, \bot, \top, \sqcup, \sqcap \rangle$,

SAC'08 March 16-20, 2008, Fortaleza, Ceará, Brazil



(a) Concrete points (b) Intervals (c) Octagons (d) Pentagons

Figure 2: The concrete points, and some approximations depending on the numerical abstract domain

```
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;
}
```

Figure 1: Example from `mscorlib.dll`. Pntg infers $0 \leq ((\texttt{num} + \texttt{num2}) >> 1) < \texttt{array.Length}$ at array access

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.

An abstract domain $\bar{\mathsf{D}}$, is related to a concrete domain C by a monotonic concretization function, $\gamma \in [\bar{\mathsf{D}} \to \mathsf{C}]$. 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 Fig. 2(a). They represent the concrete values that two variables, `x` and `y`, 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.

Intervals

A first abstraction of the points in Fig 2(a) can be made by retaining only the minimum and maximum values of variables `x` and `y`. This is called interval abstraction. Graphically, it boils down to enveloping the concrete values with a rectangle, as depicted in Fig. 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 Fig. 2(b) the fact that $\texttt{y} < \texttt{x}$ is lost.

Octagons

A more precise abstraction is obtained by using the abstract domain of Octagons, Oct. Oct keeps relations of the form $\pm\texttt{x} \pm \texttt{y} \leq k$. When applied to our concrete points, the octagon enveloping them is shown in Fig. 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 [1]. 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 [2] and [13]. 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.

Pentagons

The approximation of the concrete points with Pntg is given in Fig. 2(d). Elements of Pntg have the form of $\texttt{x} \in [a, b] \wedge \texttt{x} < \texttt{y}$, where `x` and `y` are variables and a and b belong to some underlying numerical set as $\mathbb{Z}$ or $\mathbb{Q}$. 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. If fact, if the underlying numerical values are in $\mathbb{Q}$, then $\texttt{x} < \texttt{y}$ denotes an open surface of $\mathbb{Q}^2$, whereas if they are in $\mathbb{Z}$, then $\texttt{x} < \texttt{y}$ is equivalent to $\texttt{x} \leq \texttt{y} - 1$, 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.

Order:	$[a_1, b_1] \sqsubseteq_i [a_2, b_2] \iff a_1 \geq a_2 \wedge b_1 \leq b_2$
Bottom:	$[a, b] = \bot_i \iff a > b$
Top:	$[a, b] = \top_i \iff a = -\infty \wedge b = +\infty$
Join:	$[a_1, b_1] \sqcup_i [a_2, b_2] = [\min(a_1, a_2), \max(b_1, b_2)]$
Meet:	$[a_1, b_1] \sqcap_i [a_2, b_2] = [\max(a_1, a_2), \min(b_1, b_2)]$
Widening:	$[a_1, b_1] \nabla_i [a_2, b_2] = [a_1 > a_2 ? a_2 : -\infty, b_1 < b_2 ? b_2 : +\infty]$

Table 1: Lattice operations over single intervals

3. 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 Tab. 1. 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 union 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 functional extension of those in Tab. 1, as shown by Tab. 3.

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 assignment and the guards in the interval environment are defined as usual in interval arithmetic.

4. STRICT UPPER BOUNDS

The abstract domain of strict upper bounds Sub is a special case of the zone abstract domains, 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 $\mathtt{y}_i$. The formal definition of the lattice operations for Sub is in Tab. 2.

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

Table 2: Lattice operations of strict upper bounds

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 (if the number of variables is fixed, the join suffices).

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

Order:	$b_1 \sqsubseteq_b b_2 \iff \forall \mathtt{x} \in b_1.b_1(\mathtt{x}) \sqsubseteq_i b_2(\mathtt{x})$
Bottom:	$b = \bot_b \iff \exists \mathtt{x} \in b.b(\mathtt{x}) = \bot_i$
Top:	$b = \top_b \iff \forall \mathtt{x} \in b.b(\mathtt{x}) = \top_i$
Join:	$b_1 \sqcup_b b_2 = \lambda \mathtt{x}.b_1(\mathtt{x}) \sqcup_i b_2(\mathtt{x})$
Meet:	$b_1 \sqcap_b b_2 = \lambda \mathtt{x}.b_1(\mathtt{x}) \sqcap_i b_2(\mathtt{x})$
Widening:	$b_1 \nabla_b b_2 = \lambda \mathtt{x}.b_1(\mathtt{x}) \nabla_i b_2(\mathtt{x})$

Table 3: Lattice operations of interval environments

The Need for Closure

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 Tab. 2, 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:

$$\begin{array}{rcl} [\![\mathtt{x} := \mathtt{y} - \mathtt{1}]\!](s) & = & s[\mathtt{x} \mapsto \{\mathtt{y}\}] \\ [\![\mathtt{x} == \mathtt{y}]\!](s) & = & s[\mathtt{x}, \mathtt{y} \mapsto s(\mathtt{x}) \cup s(\mathtt{y})] \\ [\![\mathtt{x} < \mathtt{y}]\!](s) & = & s[\mathtt{x} \mapsto s(\mathtt{x}) \cup s(\mathtt{y}) \cup \{\mathtt{y}\}] \\ [\![\mathtt{x} \leq \mathtt{y}]\!](s) & = & s[\mathtt{x} \mapsto s(\mathtt{x}) \cup s(\mathtt{y})] \end{array}$$

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 $\mathtt{x} < \mathtt{z}$.

5. 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 $\mathsf{Intv} \otimes \mathsf{Sub}$, [5]. The elements of the reduced cartesian product satisfy the following relation

$$\forall \langle b, s \rangle \in \mathsf{Intv} \otimes \mathsf{Sub}.\gamma_{\mathsf{Intv} \otimes \mathsf{Sub}}(\langle b, s \rangle) \subseteq \gamma_{\mathsf{Intv}}(b) \cap \gamma_{\mathsf{Sub}}(s)$$

i.e., the analysis on the reduced lattice is more precise than the pairwise composition of the two analyses. The Pntg abstract domain is an abstraction of the reduced product and is more precise than the cartesian product:

$$\mathsf{Intv} \otimes \mathsf{Sub} \prec \mathsf{Pntg} \prec \mathsf{Intv} \times \mathsf{Sub}.$$

The lattice operations are defined in Tab. 4. The functions sup and inf are defined as $\inf([a, b]) = a$ and $\sup([a, b]) = b$.

The order on pentagons 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

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

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

Table 4: The lattice operations over pentagons

constraint in s_1 or it is implied by the interval environment b_1, *i.e.*, the numerical upper bound for `x` is strictly smaller than the numerical lower bound for `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 widenings and closure operations [9].

Cost and Precision of the Join

One may ask why we defined the join over Pntg as in Tab. 4. 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.

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 $\Omega(n^2)$, 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. In our experience, when used for `mscorlib.dll`, we got that the running time of the analyzer went up to more than 45 minutes.

As a consequence we defined a safe approximation of the join as in Tab. 4. 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

```
if (...)                    if (...)
  x = 0; y = 3;               x = 0; y = 3;
else                        else
  x = −2; y = 1;              x = −2; y = 0;
```

(a) Non-strict abstraction (b) Strict abstraction

Figure 3: Difference in precision between $\sqcup_p^*$ and $\sqcup_p$

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 Fig. 3(a), to be analyzed with some initial pentagon $\langle b, s\rangle$ which does not mention `x` and `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

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

Suppose that we'd 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 Fig. 3(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 `x = y = 0`. As a consequence, `assert x + y ≠ 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. 6). 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.

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 instructions which have a non-trivial behavior depending on the operators.

Let us illustrate this situation using the `rem` instruction of MSIL. Intuitively, `rem u d` computes the reminder of the division `u/d`. The precise handling of the remainder is important as many expressions used to access arrays in `mscorlib.dll`

Assembly	Bounds checked	Intv Valid	Intv %	Intv Time	Intv × Sub Valid	Intv × Sub %	Intv × Sub Time	Pntg $\sqcup_p$ Valid	Pntg $\sqcup_p$ %	Pntg $\sqcup_p$ Time	Pntg $\sqcup_p^*$ Valid	Pntg $\sqcup_p^*$ %	Pntg $\sqcup_p^*$ Time
mscorlib.dll	17 052	12 416	72.79	5:08	14 059	82.42	7:03	14 162	83.02	7:25	14 162	83.02	61:39
System.dll	11 609	9 298	80.09	3:38	9 979	85.95	4:56	9 993	86.07	5:10	-	-	-
System.Web.dll	14 202	12 313	86.69	3:54	12 952	91.19	5:39	12 964	91.28	5:49	12 964	91.28	18:55
System.Design.dll	10 072	8 854	87.90	3:52	9 562	94.93	5:01	9 586	95.17	5:17	9 610	95.41	43:18
Average			80.99			87.92			88.22			-	

Table 5: The experimental results of the analyzer with different abstract domains

include the remainder operation. According to the definition of `rem` in Part. III, Sect. 3.55 of [7], the sign of the result is the sign of `u` and $0 \leq |\texttt{rem u d}| < |\texttt{d}|$ holds. Therefore in order to derive the constraint $\texttt{rem u d} < \texttt{d}$ one must know that $\texttt{d} \geq 0$.

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.

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

6. EXPERIMENTS

We have implemented the abstract domain Pntg in our analyzer for .NET assemblies, `Clousot`. Prior to the array bound analysis, `Clousot` performs heap abstraction and expression analysis. 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. All experiments were conducted on a Centrino 2 duo processor at 2.16 GHz, with 4 GB of RAM, running Windows Vista.

Tab. 5 summarizes 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.

We ran the analysis with four different domains, shown in the different columns: intervals alone, the cartesian product Intv×Sub, Pntg without constraint closure, and finally Pntg with constraint closure.

The table shows that combining Intv with symbolic upper bounds validates on average almost 7% more array accesses than Intv alone for only a modest extra cost. Pntg without closure validate an extra 0.3% of accesses at little extra cost, whereas Pntg with closure produces almost no extra precision but the analysis time blows up. The time for the missing run was in excess of 90 minutes. Overall, the results show that with Pntg, `Clousot` is able to validate on average 88.2% of all array accesses in under 7 minutes for the analyzed .NET assemblies.

As for the memory footprint, the analyzer never exceeded 300 Mbytes of RAM.

7. 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 88% of array accesses in four major .NET assemblies in a couple of minutes. We plan to discharge the remaining unproven accesses by using more precise, yet expensive domains on demand.

Acknowledgments. We would like to thank the Anindya Banerjee, Pietro Ferrara and the anonymous referees.

8. REFERENCES

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

[2] 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.

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

[4] 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, Jan. 1977.

[5] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *POPL '79*, pages 269–282. ACM Press, Jan. 1979.

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

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

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

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

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

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

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

[13] 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.