

A Parametric Segmentation Functor for Fully Automatic and Scalable Array Content Analysis

Patrick Cousot
ENS & CIMS–NYU
cousot@ens.fr, pcousot@cs.nyu.edu

Radhia Cousot
CNRS, ENS, & MSR Redmond
radhia.cousot@ens.fr

Francesco Logozzo
MSR Redmond
logozzo@microsoft.com

Abstract

We introduce FunArray, a parametric segmentation abstract domain functor for the fully automatic and scalable analysis of array content properties. The functor enables a natural, painless and efficient lifting of existing abstract domains for scalar variables to the analysis of uniform compound data-structures such as arrays and collections (as well as matrices when instantiating the functor on itself). The analysis automatically and semantically divides arrays into consecutive non-overlapping possibly empty segments. Segments are delimited by sets of bound expressions and abstracted uniformly. All bound expressions appearing in a set are equal in the concrete. The FunArray can be naturally combined via reduced product with any existing analysis for scalar variables. The bound expressions, the segment abstractions and the reduction operator are the three parameters of the analysis. Once the functor has been instantiated with fixed parameters, the analysis is fully automatic.

We first prototyped FunArray in Arrayal to adjust and experiment with the abstractions and the algorithms to obtain the appropriate precision/ratio cost. Then we implemented it into Clousot, an abstract interpretation-based static contract checker for .NET. We empirically validated the precision and the performance of the analysis by running it on the main libraries of .NET and on its own code. We were able to infer thousands of invariants and verify the implementation with a modest overhead (circa 1%). To the best of our knowledge this is the first analysis of this kind applied to such a large code base, and proven to scale.

Categories and Subject Descriptors D.2.4 [Software Engineering]: Program Verification; D.3.1 [Programming Languages]: Formal Definitions and Theory; F.3.1 [Logics and Meanings of Programs]: Specifying and Verifying and Reasoning about Programs—Mechanical verification, assertions, invariants; F.3.2 [Logics and Meanings of Programs]: Semantics of Programming Languages

General Terms Algorithms, Design, Languages, Performance, Reliability, Security, Verification

Keywords Static analysis, Abstract interpretation, Array content analysis

1. Introduction

Our goal is to augment static analyzers for very large programs with a new fully automatic static analysis determining properties of array elements with good precision but at low cost so as to scale up. The approach is in the context of abstract interpretation [6]. The first objective of the array content analysis is to reduce the false alarms due to accessing array elements which analysis is often imprecise, in particular because their proper initialization is unknown. The second objective is to allow for automatically proving

user provided pre/post conditions and assertions of moderate complexity on arrays (such as the non relational property “all elements are initialized” but not the relational one “the array is sorted” as in [5]). To cope with verification, we want to be able to adjust the cost/precision ratio towards more or less precision, one extreme being the classical analysis by array smashing, the other being an element by element analysis of maximal precision and cost.

2. Motivating Example

Let us consider the example in Fig. 1, extracted from the public constructor of the Random class of the .NET framework. The constructor initializes all the elements of the private array SeedArray to be ≥ -1 . The initialization process is quite complex, relying on some number theory properties which are out-of-the scope of the paper. The precondition requires the parameter Seed not to be the smallest 32-bits integer, to prevent Math.Abs from throwing an OverflowException. Next, an array of 56 elements is allocated and assigned to SeedArray. The last array element is set to the value of Seed, whereas all the others are zero (because of .NET semantics). The first loop (Loop 1), sets all the elements of indexes 1...54 to be ≥ -1 according to the non-contiguous indexing sequence: 21, 42, 8, ..., leaving the first and the last elements unchanged. Therefore the assertion at the end of Loop 1 holds. The next loop (Loop 2) shakes the values in the array, updating the last element of the array but not the first. To prove the second assertion one should prove that (i) the last element of SeedArray is definitely updated in the inner loop to a ≥ 1 value; and that (ii) the inner loop is executed at least once.

Array expansion The first and most precise approach for proving the two assertions: (i) expands the 56 cells of the array to 56 local variables; (ii) fully unrolls the loops. The example will then become intractable, even with up-to-date hardware and tools. We totally unrolled the first loop, we sliced the second loop according to some “interesting” variables (manually determined), and we tried to prove the second postcondition using Boogie [2] and the state-of-the-art SMT solver Z3 [8]. We let the verification process run for a whole week-end without getting an answer. The theorem prover was overcome by the large number of case splits it had to perform (because of conditionals in loop bodies and the lack of primitive support for the remainder operation which had to be axiomatized).

Array smashing At the opposite side of the precision spectrum there is the smashing of all the array elements into one summary location. It is immediate that this is not going to work. For instance in Loop 1, the value of SeedArray[55] is smashed with the others, concluding that any value can be written anywhere in the array.

Predicate abstraction The method of Qadeer and Flanagan [12] uses some easy syntactic heuristics to derive the predicates used for the abstraction, which unfortunately do not work here. For instance, one needs to know that $1 \leq \text{index} < 55$ to determine that the last element of SeedArray is never overwritten in Loop 1, or that $\text{num3} \geq -1$. Both properties cannot be inferred with syntactic heuristics.

```

public Random(int Seed) {
    Contract.Requires(Seed != Int32.MinValue);
    int num2 = 161803398 - Math.Abs(Seed);
    this.SeedArray = new int[56];
    this.SeedArray[55] = num2;

    int num3 = 1;
    // Loop 1
    for (int i = 1; i < 55; i++) {
        int index = (21 * i) % 55;
        this.SeedArray[index] = num3; // (*)
        num3 = num2 - num3;
        if (num3 < 0) num3 += 2147483647;
        num2 = this.SeedArray[index];
    }
    Contract.Assert(Contract.Forall( // (**)
        0, this.SeedArray.Length - 1, i => a[i] >= -1));
    // Loop 2
    for (int j = 1; j < 5; j++) {
        // Loop 3
        for (int k = 1; k < 56; k++) {
            this.SeedArray[k] -= this.SeedArray[1 + (k + 30) % 55];
            if (this.SeedArray[k] < 0)
                this.SeedArray[k] += 2147483647;
        }
    }
    Contract.Assert(Contract.Forall(0, // (***)
        this.SeedArray.Length, i => a[i] >= -1));
}

```

Figure 1. A motivating example taken from the core library of .NET. `Contract`.`{Requires, Assert, ForAll}` is the CodeContracts API (adopted in .NET from v4.0) to express pre-conditions, assertions and bounded universal quantifications [3].

Array Partitioning The array partitioning approach of Gopan, Reps and Sagiv [14] (later improved by Peron and Halbwegs [16]) separates the task of array partitioning from that of establishing array properties. Given a partition of the array into slices, the analysis populates the slices with some abstract value. The partitioning is done either syntactically or by some pre-analysis. The syntactic approach (used in the examples of [14, 16, 24]) simply does not work here (e.g. it cannot determine which array element is written at `(*)`), and in general it is unfeasible in the generic setting of the bytecode analysis, where high-level syntactic structures are compiled away. As a consequence, at the early stages of this work, we tried to implement the second approach in Clousot. The idea was to first perform a preliminary analysis of indices to provide a restricted domain for each loop, and then to perform the array analysis (generalizing [16, Sect. 15]). Performance turned out to be extremely bad. The first analysis generated *too many* partition slices (also noticed by Dillig *et al* [10, Sect. 4]). The second analysis needed to replay the index analysis (e.g. to distinguish the first iteration from all the others) and the partition analysis (e.g. to track how abstract values flowed between partitions). The analysis of the example induced a $28\times$ slowdown with respect to a run of Clousot without the array analysis. We have therefore developed a new approach (subject of this paper) in which: (i) the scalar analysis and the array analysis are performed at the same time (which is also more precise [7]); (ii) the array segmentation is automatically and semantically discovered by the analysis; and (iii) in the segmentation we admit possibly empty segments. In particular possibly empty segments are a winning choice because they allow to compactly represent array partitions avoiding the exponential multiplication of slices of the aforementioned works (Sect. 4.4).

Under-approximations and Templates The technique of Gulwani, McCloskey and Tiwari [15] is extremely powerful yet expensive. It requires: (i) the user to provide templates for the ar-

ray invariants; and (ii) the abstract domain to perform under-approximations for the index variable. It can infer all the invariants of our example, provided some refinement in the handling of transition functions for quantified facts and in the under-approximation algorithm. Their technique uses uninterpreted functions and a guess & prove cycle to determine precedents for guards. Unfortunately, the abstract domain of uninterpreted functions exposes a double-exponential complexity [15], which seriously affects the analysis cost. According to [15, Sect.5.2], *at best* the quantified domain induces a 70% slowdown of their analyzer, and *at most* a 1800% slowdown (w.r.t. a normal run) on small examples. As a comparison, the functor abstract domain presented in this paper induces a mere 1% slowdown with respect to a normal Clousot run on huge, production quality libraries (cf. Sect. 12), yet presenting a high precision.

Deductive methods Program verifiers *à la* ESC/Java 2 [4] or Spec# [1] require the user to provide loop invariants. In our running example, we needed to provide a few extra-annotations (9 to be exact) to help both tools prove the assertions in the code. First we have to add the invariant on the content of `SeedArray` to every loop in the code. Then, we added the loop invariant $\text{num3} \geq -1 \wedge i \geq 1$ to Loop 1, $j \geq 1$ to Loop 2 and $k \geq 1$ to Loop 3. In general, such program verifiers are very powerful but the extra-annotations impose a burden that very few professional programmers are willing to pay. Furthermore, deductive verification-based tools can *check* the correctness of a program fragment (e.g. a method), but they cannot *infer* facts to be used on larger pieces of code (e.g. class invariants to verify whole classes [22]).

Theorem prover-based The method of Kovacs and Voronkov [19] uses a saturation theorem prover to generate loop invariants. The idea is to encode the changes to an array at the i -th iteration as a quantified fact and then to systematically apply resolution to derive a closed form (one not mentioning the loop iteration i). A problem with such a technique is termination, for instance to determine when the “right” loop invariant has been produced by a saturation step. This may require a human help (stopping the saturation process when a postcondition does not work: for instance if we remove the first assertion in Fig. 1, then the process may go on forever). Furthermore, their method is based on the use of monotonic changes to the array (which is not the case for Loop 1) and it requires a pre-analysis of indexes (causing an extra slow-down).

The techniques of Jhala and McMillan [17, 25] and of Seghir, Podelski and Wies [30] make use of the loop postconditions to be proven in order to infer the quantified loop invariants. Suppose we remove `(**)` and `(***)` from the example. Then their techniques (unlike ours) cannot infer the postcondition that all the elements of `SeedArray` are initialized to a value ≥ -1 at the end of the `Random` constructor. In practice, such a postcondition is needed, for instance to prove that it is an object invariant for the class `Random` [21] and hence to prove the safety of the public methods. Furthermore, the techniques above do not always guarantee termination.

Our Approach Our analysis infers *all* the invariants for Fig. 1 without user interaction: no templates, no annotations nor partitions are required, no hypotheses are done on the structure of the source program. The invariants are inferred even if the assertions are removed from the code. The code is analyzed in (a little bit less than) 60 milliseconds (50 milliseconds for reading the bytecode, performing a stack analysis, heap analysis, non-null, and numerical analysis alone).

The analysis is an instance of FunArray, which we introduce in this paper. FunArray is a functor abstract domain which lifts existing analyses for scalar values to uniform compound data structures as arrays or collections. In this paper we will concentrate on arrays, but it is immediate to see how the results generalize to collections

as found in mainstream object-oriented languages such as C# or Java as well as matrices when instantiating the functor on itself.

The FunArray analysis automatically divides the array into possibly empty segments delimited by a set of segment bounds. The content of each segment is uniformly abstracted. The FunArray analysis can be combined via a reduced product with an abstraction for scalar variables. Therefore the FunArray has three parameters: (i) the expressions used to describe the segment bounds; (ii) the abstract domain to abstract the segment values; and (iii) the operator used to perform the reduction between the FunArray itself and the abstract domain abstracting the scalar variables.

When the three parameters above are chosen to be: (i) simple expressions in the form k or $x + k$; (ii) and (iii) intervals [6] then for Loop 1 our analysis infers that all the values of the arrays with indexes in the range $1 \dots 54$ are greater or equal to -1 , and that the last element of the array is not overwritten. The FunArray uses the information to infer the segmentation below, which is enough to prove the assertion (**) (values in brackets are bounds, intervals denote the abstraction for the array elements in the bounds, see Sect. 4.3).

```
{0} [-1,+∞] {55} [+∞,-∞] {56}. (1)
```

For Loop 3, the analysis discovers that *all* the array elements, including the last one, have been overwritten with a value ≥ -1 :

```
{0} [-1,+∞] {56}. (2)
```

The loop invariant for Loop 2 is then the union of the two invariants above, that is (1) as the first segmentation subsumes the second one. If directly propagated after the loop, this invariant is too weak to prove the assertion (***). The imprecision is originated by the fact that we are not considering that the body of Loop 2 is executed at least once, so that `SeedArray[55]` is overwritten at least once (because of (1)). Standard static analysis techniques such as loop unrolling or backwards goal propagation can be used to recover the needed precision, and hence refine the abstract post-state of Loop 2 to (2). This highlights another advantage of our analysis, which benefits *for free* of precision refinement techniques applied to the analyzer.

3. Our Contribution

The main advantages of our analysis can be summarized as:

1. The array segmentation is automatically and semantically inferred during the analysis. The segments are consecutive, without holes (a hole being just another segment). The segments derive from the way array elements are modified and accessed. Segments are delimited by bounds, in increasing order, denoted by sets of simple symbolic expressions with equal but unknown values;
2. The combinatorial explosion in the handling of disjunctions is avoided by considering symbolic segment bounds as well as possibly empty segments;
3. The relations between array indexes and array elements can be inferred by abstracting pairs made of the array index and the value of the corresponding array element (vs. abstracting array element values only);
4. The precision/cost ratio in the abstraction of the array content can be finely tuned using a functor abstract domain: the array content analysis is parameterized by the abstract domain representing symbolic segment bound expressions, the domain abstracting the pairs (index, value) in segments, and the abstract domain assigning values to segment bound expressions.
5. By instantiating the array segmentation abstract domains functor with different abstract domains, different static analyzers can be automatically generated with different cost/precision ratios

```
void Init(int[] A) {
/* 0: */   int i = 0;
/* 1: */   while /* 2: */ (i < A.Length) {
/* 3: */     A[i] = 0;
/* 4: */     i = i + 1;
/* 5: */   }
/* 6: */ }
```

Figure 2. The fully initialized example. We want to prove that $\forall i \in [0, A.Length) : A[i] = 0$ at program point 6.

allowing the cost versus precision tradeoff of the analysis to be tuned depending on the target application at no re-programming cost of the static analyzer.

We have first implemented our technique in a research prototype `ArrayA1`, to quickly experiment with the algorithms and adjust the abstractions. Then we fully implemented it in `Clousot`, an industrial-quality static contract checker for .NET based on abstract interpretation. The functor abstract domain enabled a natural lifting of the abstract domains, already present in `Clousot`, to array contents. To the users, this is exposed as a simple checkbox in the development environment. We validated the precision of the analysis by using it to check its own implementation. The analysis is extremely fast: we estimated the cost of the array analysis on `Clousot` to be less than 1% of the total running time when running it on production code (Sect. 12). To the best of knowledge, this is the first analysis of this kind applied to such a large scale.

4. Array Initialization

We explain the details of our technique on the initialization example of Fig. 2, slightly more general than Loop 3. We illustrate how we avoid the combinatorial explosion on the partial initialization example of Fig. 3.

4.1 Manual proof

A manual proof of the exit specification would involve a loop invariant at program point 2 stating that if $A.Length = 0$ then $i = 0$ and the array A is empty or else $A.Length \geq 1$ in which case either $i = 0$ and the array A is not initialized or else $i > 0$ so that $A[0] = A[1] = \dots = A[i - 1] = 0$. Formally the invariant $(A.Length = 0 \wedge i = 0) \vee (A.Length \geq 1 \wedge 0 \leq i \leq A.Length \wedge \forall j \in [0, i - 1] : A[j] = 0)$ holds at point 2.

This invariant shows that array content analyses must be able to: (i) express disjunctions of array descriptions; (ii) express properties of array segments (that is sequences of values of consecutive array elements); and (iii) relate the symbolic limits $0, i - 1, A.Length - 1$ of these segments to the scalar program variables.

4.2 Automatic proof and the meaning of the abstract invariant predicates

In our array segmentation analysis instantiated *e.g.* with constant propagation [18], we automatically get the abstract invariant predicates

```
p1 = [ A: {0 i} T {A.Length}? ]
p2 = [ A: {0} 0 {i}? T {A.Length}? ]
p6 = [ A: {0} 0 {A.Length i}? ]
```

where p_i is the abstract invariant predicate at program point $i = 1, \dots, 6$. In this example the properties of scalar variables need not be used. The abstract values for constant propagation can be \perp (*i.e.*, bottom \perp , meaning unreachable), an integer constant (meaning equal to that constant), or \top (*i.e.*, top \top , meaning unknown).

In the array environments such as $[A: \{0 i\} T \{A.Length\} ?]$ in p_1 , each array of the program (such as A) has its content described by a segmentation (such as $\{0 i\} T \{A.Length\} ?$). From the symbolic segment bounds such as $\{0 i\}$ and $\{A.Length\} ?$ we

know that $i = 0$ (since all expressions in a bound are equal) and that $0 = i \leq A.Length$ (since the segment bounds are in increasing order, strictly increasing in absence of $?$). The segments are not empty, except if the upper bound of the segment is marked with $?$. The segments are consecutive without holes, since a hole can always be represented by a segment (possibly empty if the hole may or may not be absent). Each segment uniformly describes the array elements within that segment bounds, lower bound included, upper bound excluded. In $\{0 \ i\} \ T \ \{A.Length\}?$, the array element abstract value is T , meaning in the constant propagation analysis, that the array values are unknown. So the invariant p_1 states that $i = 0 \leq A.Length \wedge \forall j \in [0, A.Length) : A[j] \in \mathbb{Z}$. In particular, when $i = A.Length = 0$, the interval $[0, A.Length)$ is empty, so the quantified expression holds vacuously.

The invariant p_2 states that $0 \leq i \leq A.Length$ (in absence of question marks $?$ these inequalities would be strict), that $A[0] = A[1] = \dots = A[i-1] = 0$ when $i > 0$ and that the values $A[i], A[i+1], \dots, A[A.Length-1]$ are unknown when $A.Length > i$. So the array is divided into consecutive non-overlapping segments, which may be empty and are delimited by symbolic expressions in increasing order. The abstraction of the array elements within one segment is uniform but different segments can have different abstract properties.

In order to avoid combinatorial explosion, disjunctions appear in restricted form only either as possible segment emptiness, or symbolic bounds which may have different values, or in the segment content analysis (see Sect. 11.1). For example, the post-condition p_6 expresses that either the array is empty (*i.e.* $A.Length = i = 0$) or else $A.Length = i > 0$ and all array elements are initialized to 0.

Please note that the case $A.Length < 0$ is excluded. This comes from the initial condition stating that $A.Length \geq 0$ since most programming languages like C, C# and Java do not allow arrays of negative size. We handle all such runtime errors including division by zero, index out of bounds, ... by stopping execution. This is a sound treatment of their undefined semantics in absence of runtime errors but may otherwise miss some other possible erroneous executions (following from the fact that execution goes on in practice with an undefined semantics).

4.3 Detailed unreeling of the initialization example analysis

We now consider the details of the analysis of the code of Fig. 2 with constant propagation. The initial condition $A.Length \geq 0$ is recorded in the segmentation of array A .

$$p_0 = [A: \{0\} \ T \ \{A.Length\}?$$

The assignment $i = 0$; sets the value of the scalar variable i to 0. The equality $i = 0$ is valid after the assignment and so is recorded in the lower bound of the array segment. Initially $p_2 = p_3 = \dots = p_5 = \perp$ denotes unreachability of the loop so that the abstract loop invariant is initially $p_2 = p_1 \sqcup p_5 = p_1$ (here \sqcup is the join in the constant abstract domain: $x \sqcup \perp = \perp \sqcup x = x, x \sqcup \top = \top \sqcup x = \top, i \sqcup i = i$, and $i \sqcup j = \top$ when $i \neq j$).

$$p_2 = p_1 = p_0[i=0] = [A: \{0 \ i\} \ T \ \{A.Length\}?$$

The loop is entered when $i < A.Length$ so that the array, hence its only segment, cannot be empty so $?$ is dropped:

$$p_3 = p_2[i < n] = [A: \{0 \ i\} \ T \ \{A.Length\}$$

The analysis of the array assignment $A[i] = 0$; splits the array segment around the index i and assigns to the array element the value of expression 0 in the constant domain that is 0:

$$p_4 = p_3[A[i]=0] = [A: \{0 \ i\} \ 0 \ \{1 \ i+1\} \ T \ \{A.Length\}?$$

Please note that the segment $i \dots i+1$ is definitely not empty while the segment $i+1 \dots A.Length$ may be empty. The scalar variable

assignment $i = i + 1$; is invertible since the old value of i is the new value of variable i decremented by 1. So the segment bounds involving variable i have to be modified accordingly:

$$p_5 = p_4[i=i+1] = [A: \{0 \ i-1\} \ 0 \ \{1 \ i\} \ T \ \{A.Length\}?$$

The next approximation of the loop invariant is $p_2 = p_1 \sqcup p_5$. This join first involves the unification of the segment $\{0 \ i\} \ T \ \{A.Length\}$ of p_1 and that $\{0 \ i-1\} \ 0 \ \{1 \ i\} \ T \ \{A.Length\}?$ of p_5 . Keeping only the expressions appearing in both segmentations, we get $\{0 \ i\} \ T \ \{A.Length\}$ and $\{0\} \ 0 \ \{i\} \ T \ \{A.Length\}?$. Splitting the bound $\{0 \ i\}$ we get $\{0\} \ \perp \ \{i\} \ T \ \{A.Length\}$ so that the union with $\{0\} \ 0 \ \{i\} \ T \ \{A.Length\}?$ can now be performed segmentwise in the constant domain $\{0\} \ \perp \ \sqcup \ 0 \ \{i\} \ T \ \{A.Length\} \ (\perp \ \top = \top)$ since the segments may be empty in at least one of the cases (that is $\perp \ \top = \perp$ for non-empty segments and otherwise $\perp \ \top = \top$ for possibly empty ones). We get

$$p_2 = p_1 \cup p_5 = [A: \{0\} \ 0 \ \{i\} \ T \ \{A.Length\}?$$

The next iteration is similar

$$\begin{aligned} p_3 &= p_2[i < A.Length] = [A: \{0\} \ 0 \ \{i\} \ T \ \{A.Length\} \] \\ p_4 &= p_3[A[i]=0] = [A: \{0\} \ 0 \ \{i\} \ 0 \ \{i+1\} \ T \ \{A.Length\} \] \\ p_5 &= p_4[i=i+1] = [A: \{0\} \ 0 \ \{i-1\} \ 0 \ \{i\} \ T \ \{A.Length\} \] \\ p_2 &= p_1 \cup p_5 = [A: \{0\} \ 0 \ \{i\} \ T \ \{A.Length\} \] \end{aligned}$$

so that we have reached a fixpoint. It remains to compute

$$p_6 = p_2[i \geq A.Length] = [A: \{0\} \ 0 \ \{A.Length, i\} \ T \ \{A.Length\}?$$

where $A.Length = i$ since the segmentation of p_2 provides the information that $0 \leq i \leq A.Length$.

The array content analysis always terminates since the only two reasons for non-termination are impossible:

1. The array might have infinitely many symbolic segments as in $\{0\} \dots \{n-3\} \dots \{n-2\} \dots \{n-1\} \dots \{n\}$ which is prevented by segmentation unification and widening;
2. A segment might take successive strictly increasing abstract values which is prevented by the use of a widening/narrowing convergence acceleration for segment content analysis [6]. No widening was necessary for constant propagation which satisfies the ascending chain condition ($\perp \sqsubseteq i \sqsubseteq \top, i \in \mathbb{Z}$).

4.4 Partial Array Initialization

Full array initialization is a very well studied example, and array-partitioning techniques perform reasonably well on it [14, 16]. However, partial array initialization (Fig. 3) illustrates the multiplication of partitions which makes those techniques not-scalable. At the end of the loop, our analysis (instantiated with constant propagation) infers the following segmentation for C :

$$\{0\} \ 1 \ \{j\} \ T \ \{i, A.Length, C.Length\}?$$

which compactly captures the fact that C may be empty (when $0 = j = i$), may be not initialized (when $j = 0$), may be partially initialized (when $0 < j < i$), may be fully initialized (when $0 < j = i$). Compare it with partition-based approaches where the abstract state at the end of the loop contains four disjuncts: one representing the concrete state when none of the C elements is initialized ($j = 0$), two representing the partial initialization of C distinguishing when $j+1 < C.Length$ or $j < C.Length$, and one representing the total initialization ($j == C.Length$) ([14, 7.2]). We tried this example using our early implementation of [16] and we got a $2 \times$ slow-down with respect to a normal run of Clousot (it is worth noting that the experimental results reported in [14] and those in [15] are even worse than our first implementation). For this example, Clousot lifted with the functor abstract domain was so fast that we were unable to measure its impact on the performances: the additional cost is in the order of magnitude the noise of the virtual machine (JIT, garbage collector ...) *i.e.* few milliseconds.

```

void InitPartial(int[] A, int[] C) {
  Contract.Requires(A.Length == C.Length);
  int i = 0, j = 0;
  while (i < A.Length) {
    if (p(A[i])) // For some predicate p
      C[j++] = 1;
    i++;
  } }

```

Figure 3. Partial array initialization. Partition-based techniques use four partitions encoding the fact that at loop exit C may be empty, partially filled, almost-totally filled or totally filled. Our analysis: (i) compactly represents the same information with only one segmentation; and (ii) infers the segmentation automatically.

5. Abstract Domains and Functors

An *abstract domain* \mathbf{D} includes a set $\overline{\mathcal{D}}$ of abstract properties as well as abstract functions and operations $\mathbf{D}.op$ for the partial order structure of abstract properties (\sqsubseteq), the join (\sqcup), the meet (\sqcap), convergence acceleration operators: widening (∇) and narrowing (Δ), the abstract property transformers involved in the definition of the semantics of the programming language: the abstract evaluation of program arithmetic and Boolean expressions, the assignment to scalar variables ... [6]. A monotonic concretization function γ provides the meaning of abstract properties in terms of concrete properties.

An *abstract domain functor* \mathbf{D} is a function from the parameter abstract domains $\mathbf{D}_1, \dots, \mathbf{D}_n$ to a new abstract domain $\mathbf{D}(\mathbf{D}_1, \dots, \mathbf{D}_n)$. The term “functor” is mutated from OCaml terminology. The formal parameters $\mathbf{D}_1, \dots, \mathbf{D}_n$ of the abstract domain functor \mathbf{D} can be instantiated to various actual abstract domains without needing to rewrite the code of the static analyzer. So various abstractions can be experimented at no programming cost. The abstract domain functor $\mathbf{D}(\mathbf{D}_1, \dots, \mathbf{D}_n)$ composes abstract properties $\overline{\mathcal{D}}_1, \dots, \overline{\mathcal{D}}_n$ of the parameter abstract domains $\mathbf{D}_1, \dots, \mathbf{D}_n$ to build a new class of abstract properties $\overline{\mathcal{D}}$ (e.g. abstract environments mapping program numerical variables to intervals) and operations (e.g. assignment of an interval to a variable). For short, we can omit the parameters writing \mathbf{D} or op when the parameters $\mathbf{D}_1, \dots, \mathbf{D}_n$ are clear from the context.

6. Concrete Semantics

We describe the elements of the semantics of programming languages to which our array content analysis does apply, that is scalar variables, simple expressions, and unidimensional arrays and corresponding assignments.

6.1 Scalar Variables Semantics The operational semantics of scalar variables with basic types (`bool`, `char`, `int`, `float`, etc) is assumed to be concrete variable environments $\rho \in \mathcal{R}_v$ mapping variables names $i \in \mathbb{X}$ to their values $\rho(i) \in \mathcal{V}$ so that $\mathcal{R}_v \triangleq \mathbb{X} \mapsto \mathcal{V}$.

6.2 Simple Expressions Semantics The program simple expressions $e \in \mathbb{E}$ containing only constant, scalar variables, and mathematical unary and binary operators have a semantics $\llbracket e \rrbracket \rho$ in the concrete variable environment ρ so that $\llbracket e \rrbracket \in \mathcal{R}_v \mapsto \mathcal{V}$. For simplicity, the values in our examples are chosen to be integers (so $\mathcal{V} = \mathbb{Z}$). The semantics of scalar variable assignment is as usual $\llbracket i := e \rrbracket \rho \triangleq \rho[i := \llbracket e \rrbracket \rho]$ where $\rho[i := v](i) = v$ and $\rho[i := v](j) = \rho(j)$ when $j \neq i$.

6.3 Unidimensional Arrays Semantics The operational semantics of array variables (such as $A \in \mathbb{A}$) are concrete array environments $\theta \in \mathcal{R}_a$ mapping array names $A \in \mathbb{A}$ to their values $\theta(A) \in \mathcal{A} \triangleq \mathcal{R}_v \times \mathbb{E} \times \mathbb{E} \times (\mathbb{Z} \mapsto (\mathbb{Z} \times \mathcal{V}))$ so that $\mathcal{R}_a \triangleq \mathbb{A} \mapsto \mathcal{A}$.

In order to be able to relate array element values to their indexes, we assume that the concrete value of an array A is a quadruple $a = (\rho, A.low, A.high, A) \in \mathcal{A}$, where:

- $\rho \in \mathcal{R}_v$ is a scalar variable environment (Sect. 6.1);
- $A.low \in \mathbb{E}$ is an expression (0 in our examples) which value $\llbracket A.low \rrbracket \rho$ evaluated in the variable environment ρ yields the integer lower bound of the array;
- $A.high \in \mathbb{E}$ ($A.Length$ in our examples) is an expression which value $\llbracket A.high \rrbracket \rho$ evaluated in the variable environment ρ yields the integer upper bound of the array;
- A maps an index $i \in [\llbracket A.low \rrbracket \rho, \llbracket A.high \rrbracket \rho]$ to a pair $A(i) = (i, v)$ of the index i and the corresponding array element value v .

The instrumented semantics of arrays makes explicit the fact that arrays relate indexes to indexed element values by considering array elements to be a pair of an index and an array element value. This instrumented semantics is in contrast with the classical semantics $a \in [\ell, h] \mapsto \mathcal{V}$ of arrays mapping indexes in $[\ell, h]$ to array element values in \mathcal{V} . The explicit inclusion of the array bounds is useful to handle arrays of parametric length such as JavaScript arrays or collections in managed languages. Nevertheless, the examples here consider arrays of fixed length, maybe unknown, with $A.low = 0$. The inclusion of the concrete variable environment is also necessary to explain segments (which are sub-arrays whose bounds may symbolically coincide at different program points although they may take different concrete values over time, so that the length of the segment can vary during execution as shown e.g. in Sect. 4.3 by p1 and p5).

The semantics of an array element access $A[e]$ is classical. The expression e is evaluated to an index i . The array variable A is evaluated to its array value $a = (\rho, A.low, A.high, A)$ where ρ is the concrete variable environment. It is a “buffer overrun” runtime error if $i < \llbracket A.low \rrbracket \rho$ or $\llbracket A.high \rrbracket \rho \leq i$, in which case the value of $A[i]$ is undefined so that program execution is assumed to stop. Otherwise the index is in-bounds so $A(i) = (i, v)$ is well-defined and v is the value, in the classical sense, of the array element $A[e]$. Obviously storing (i, v) instead of v is useless but for the fact that the instrumented semantics can be used to make the array content analysis more precise.

Example 1 Let us consider the initialization example of Fig. 2 with the *additional* assumption that $A.Length > 1$. At program point 6 the final values of the scalar variables are given by ρ_6 such that $\rho_6(i) = \rho_6(A.Length) = n$ where $n > 1$ is the unknown array length. The final value of A is $a_6 = (\rho_6, 0, A.Length, A_6)$ with $A_6(i) = (i, i)$ for all $i \in [0, n)$. Because $\rho_6, 0$, and $A.Length$ are easily understood from the context, we write $A[i] = (i, i)$ by abuse of notation where the value i of i is assumed to be in-bounds. \square

In the analysis of the example of Fig. 2, the pair $A[i] = (i, v)$ was first abstracted to v , which is the case for all non-relational abstract domains such as constant propagation which cannot establish a relation between the index i and the array element value v .

Array properties are sets of concrete array values and so belong to $\wp(\mathcal{A})$.

We have no hypotheses on expressions but $\mathbb{Z} \subseteq \mathbb{E}$ and $\mathbb{X} \subseteq \mathbb{E}$ so that the expressions used in segment bounds can at least be integer constants or scalar variables, which is necessary in most programming languages to express bounds.

7. The Variable and Expression Abstract Domains

7.1 Scalar variable abstraction

We let \mathbf{X} be an abstract domain encoding program variables including a special variable v_0 which value is assumed to be always zero

so $\bar{\mathcal{X}} = \mathbb{X} \cup \{v_0\}$ where $v_0 \notin \mathbb{X}$. Operations include the equality comparison of variables.

Properties and property transformers of concrete variable environments in $\wp(\mathcal{R}_v)$ are abstracted by the variable environment abstract domain $\mathbf{R}(\mathbf{X})$ which depends on the variable abstract domain \mathbf{X} (so that \mathbf{R} is an abstract domain functor). The abstract properties $\bar{\rho} \in \bar{\mathcal{R}}$ are called abstract variable environments. The concretization $\gamma_v(\bar{\rho})$ denotes the set of concrete variable environments having this abstract property. It follows that $\gamma_v \in \bar{\mathcal{R}} \mapsto \wp(\mathcal{R}_v)$.

The static analysis of scalar variables may or may not be relational. For non-relational abstractions, $\wp(\mathcal{R}_v)$ is first abstracted to $\mathbb{X} \mapsto \wp(\mathcal{V})$ and $\bar{\mathcal{R}} \triangleq \mathbb{X} \mapsto \bar{\mathcal{V}}$ where the abstract domain \mathbf{V} abstracts properties of values in \mathcal{V} with concretization $\gamma_v \in \bar{\mathcal{V}} \mapsto \wp(\mathcal{V})$.

7.2 Expressions in simple normal form

The symbolic expressions appearing in segment bounds belong to the expression abstract domain $\mathbf{E}(\mathbf{X})$. The abstract properties $\bar{\mathcal{E}}$ consist in a set of symbolic expressions depending on the variables in $\bar{\mathcal{X}}$ restricted to a canonical normal form plus the bottom expression \perp corresponding to unreachability and the top expression \top abstracting all symbolic expressions which cannot be put in the considered normal form. The array bound expressions are assumed to be converted in canonical normal form (e.g. via auxiliary variables, so that $\dots A[B[i]] \dots$ becomes $\dots \{\text{int } x; x := B[i]; A[x]\} \dots$). Different canonical forms for expressions correspond to different expression abstract domains $\mathbf{E}(\mathbf{X})$.

In our examples, and in the Clousot implementation, the abstract expressions $\bar{\mathcal{E}}$ are restricted to the normal form $v + k$ where $v \in \bar{\mathcal{X}}$ is an integer variable plus an integer constant $k \in \mathbb{Z}$ ($v_0 + k$ represents the integer constant k). An alternative example of convenient normal form would be linear expressions $a.v + b$ where v is a variable and $a, b \in \mathbb{Z}$ ($a = 0$ for constants).

7.3 Concretization

Given an abstract domain for scalar variables with concretization $\gamma_v \in \bar{\mathcal{R}} \mapsto \wp(\mathbb{X} \mapsto \mathbb{Z})$, the concretization $\gamma_e(\mathbf{e})\bar{\rho}$ of an expression $\mathbf{e} \in \bar{\mathcal{E}}$ depends on the abstract value $\bar{\rho} \in \bar{\mathcal{R}}$ of the scalar variables in $\bar{\mathcal{X}}$ and is the set of possible concrete values of the expression. So $\gamma_e \in \bar{\mathcal{E}} \mapsto \bar{\mathcal{R}} \mapsto \wp(\mathcal{V})$ such that $\gamma_e(\perp)\bar{\rho} \triangleq \emptyset$, $\gamma_e(\top)\bar{\rho} \triangleq \mathcal{V}$, $\gamma_e(v_0 + i)\bar{\rho} \triangleq \{i\}$, and otherwise $\gamma_e(v + i)\bar{\rho} \triangleq \{\rho(v) + i \mid \rho \in \gamma_v(\bar{\rho})\}$.

7.4 Abstract operations on expressions in simple normal form

Simple operations are defined on symbolic expressions in normal form such as the check that an expression depends or not on a given variable, or the substitution of an expression for a variable in an expression followed by its reduction in normal form, returning \top if impossible.

Given two expressions in normal form, we must be able to answer the question of their equality and inequality, which in the abstract is always true, false or unknown. These abstract equality and inequality tests of expressions may be more or less sophisticated. We consider below three cases of increasing complexity, which one is chosen can be a parameter of the analysis.

Syntactic comparisons In their simplest form the comparisons can be purely syntactic. For example $v + i = v' + j$ is true if and only if $v = v'$ and $i = j$, false if $v = v'$ and $i \neq j$ and unknown otherwise. Similarly $v + i < v' + j$ is true if and only if $v = v'$ and $i < j$, false if $v = v'$ and $i \geq j$ and unknown otherwise. The comparison of i and $v + j$ where $v \neq v_0$ always has an unknown result. This is very simple, rapid, but rather imprecise.

Variable comparisons An immediate refinement consists in using the abstract information $\bar{\rho} \in \bar{\mathcal{R}}$ available on scalar variables. This is always possible since the corresponding abstract domains, whether relational or not, do have primitives to handle program conditional expressions.

For example assume that $\mathbf{R}(\mathbf{X})$ is an interval analysis, $\bar{\rho}(v) = [a, b]$, and $\bar{\rho}(v') = [a', b']$. The comparison $v + i < v' + j$ is true when $v = v'$ and $i < j$ or $v \neq v'$ but (using the abstract variable environment) $b + i < a' + j$, false when $v = v'$ and $i \geq j$ or $v \neq v'$ but (using the abstract variable environment) $b' + j \leq a + i$ and unknown otherwise. Relational domains such as DBM [9] and Octagons [27] can directly answer such questions. In that case the expression abstract domain \mathbf{E} is an abstract domain functor $\mathbf{E}(\mathbf{X}, \mathbf{R}(\mathbf{X}))$ depending on the variable abstract domain \mathbf{X} and the variable environment abstract domain $\mathbf{R}(\mathbf{X})$.

Please note that comparison of expressions $\mathbf{e}, \mathbf{e}' \in \bar{\mathcal{E}}$ must be done for all possible variable abstract domains \mathbf{R} which requires all of them to share a common abstract interface for expression comparison. A reasonable choice is to translate the comparison of normal expressions in $\bar{\mathcal{E}}$ to that of program expressions which anyway have to be evaluated in the abstract using \mathbf{R} .

Segmentation-based comparisons The information in the array segmentation can be used to symbolically compare expressions. In fact a segmentation $\{e_1^1 \dots e_{m_1}^1\} \dots \{e_1^2 \dots e_{m_2}^2\} [?] \dots \{e_1^n \dots e_{m_n}^n\} [?^n]$ maintains the information that $e_1^1 = \dots = e_{m_1}^1 \leq e_1^2 = \dots = e_{m_2}^2 \leq \dots \leq e_1^n = \dots = e_{m_n}^n$ (where the i -th inequality is strict when $[?^{i+1}]$ is empty and not strict when $[?^{i+1}]$ is $?$). In its simplest form, two expressions are known to be equal if they appear in the same segment bound, unequal if they appear in different segment bounds of the same array (strictly when separated by at least one $_$), and otherwise their comparison is unknown.

More sophisticated algorithms can be used depending on the allowed syntactic form of normal expressions.

For example, in the case of expressions of the restricted form $v + i$, $i \in \mathbb{Z}$ where constant expressions are represented by the distinguished variable v_0 which value is assumed to always be zero, we can use Pratt's algorithm [28] to compare their symbolic values. A graph matrix is constructed with an edge (v, v') labelled $i - j$ whenever $v + i \leq v' + j$ (respectively $i - j + 1$ when $v + i < v' + j$) is derived from a segmentation of some array. Equalities are represented by two inverse inequalities. Arcs between incomparable variables are marked $+\infty$ (including when $i - j$ or $i - j + 1$ overflows so that the relation is abstracted away). The Roy-Warshall-Floyd all-pairs shortest paths/transitive closure algorithm [29] is used to derive all possible comparisons derived by repeated application of the transitivity of comparisons. A cycle (v, v) for a variable v in the transitive closure matrix means impossibility, that is unreachability in the concrete. A constraint $v + k \leq v'$ holds when the label of arc (v, v') is less than or equal to k in the transitive closure matrix.

Another similar example is Shostak algorithm [31] for comparison of linear expressions of the form $a.v + b.v' \leq c$ where $a, b, c \in \mathbb{Z}$.

8. Segment Bounds Abstract Domain Functor

The segment bound abstract domain functor \mathbf{B} takes any of the expression abstract domains \mathbf{E} discussed in Sect. 7.2 and produces an instantiated segment bound abstract domain $\mathbf{B}(\mathbf{E})$ whose abstract properties are sets of expressions $\bar{\mathcal{B}} \triangleq \wp(\bar{\mathcal{E}} \setminus \{\perp, \top\})$. The empty set \emptyset denotes unreachability while non-empty sets $\{e_1 \dots e_m\}$ of expressions $e_1, \dots, e_m \in \bar{\mathcal{E}}$ are all equivalent symbolic denotations of some concrete value (generally unknown in the abstract except when one of the e_i is a constant).

8.1 Concretization

The concretization $\gamma_b \in \overline{\mathcal{B}} \mapsto \wp(\mathcal{R}_v)$ of segment bounds is the set of scalar variables concrete environments ρ making the concrete values of all expressions in the set to be equal $\llbracket e_1 \rrbracket \rho = \dots = \llbracket e_m \rrbracket \rho$. So $\gamma_b(\emptyset) = \emptyset$ and $\gamma_b(S) = \{\rho \mid \forall e, e' \in S : \llbracket e \rrbracket \rho = \llbracket e' \rrbracket \rho\}$ where $\llbracket e \rrbracket \rho$ is the concrete value of expression e in the concrete environment ρ (for example $\llbracket v_0 + c \rrbracket \rho \triangleq c$ and otherwise $\llbracket v + c \rrbracket \rho = \rho(v) + c$).

When normal expressions and segment bounds are simplified and compared in the context of variable abstract environments $\overline{\rho} \in \overline{\mathcal{R}}$ (Sect. 7.4), the concretization can be chosen as $\gamma_b \in \overline{\mathcal{B}} \mapsto \overline{\mathcal{R}} \mapsto \wp(\mathcal{R}_v)$ such that $\gamma_b(S)\overline{\rho} = \{\rho \in \gamma_v(\overline{\rho}) \mid \forall e, e' \in S : \llbracket e \rrbracket \rho = \llbracket e' \rrbracket \rho\}$.

8.2 Abstract operations on segment bounds

The segment bound abstract domain operations include basic set operations (such as the empty and singleton constructors, test for emptiness, inclusion, strict inclusion, and equality, union, intersection) as well as a widening (when the normal form of expressions does not enforce the finiteness of the number of expressions which can all have the same concrete value). A simple widening limits the number of expressions that can appear to a maximum given as a parameter of the analysis.

In order to handle non-invertible assignments to scalar variables, the segment bounds abstract domain $\mathbf{B}(\mathbf{E})$ has an operation that eliminates from a set of expressions all the expressions that contain a given variable (using the check provided by the expression domain parameter \mathbf{E} in Sect. 7.4).

Similarly, to handle invertible assignments to scalar variables, an operation is available to substitute an expression for a variable in all expressions of a set, a resulting expression being eliminated from the set when the expression domain parameter cannot put it in normal form. After a side-effect free assignment $i=e$; or an equality test $i==e$, we have $i = e$ so $e'(e)$ can be added to a segment bound containing the expression $e'(i)$ provided the expression domain parameter \mathbf{E} can put $e'(e)$ in normal form.

Based on the comparison of expressions in sections Sect. 7.4 the segment bounds abstract domain functor can compare sets of equal expressions. For example $s < s'$ is true if there exists an expression e in s and expression e' in s' such that the expression domain parameter \mathbf{E} can determine that $e < e'$ is true. $s < s'$ is false if the expression domain parameter \mathbf{E} can determine that $e < e'$ for all $e \in s$ and $e' \in s'$. Otherwise the comparison $s < s'$ has an unknown result.

9. Array Element Abstract Domain

The array element abstract domain \mathbf{A} abstracts properties of pairs (index, value of indexed array element). The concretization is $\gamma_a \in \overline{\mathcal{A}} \mapsto \wp(\mathbb{Z} \times \mathcal{V})$.

Properties in $\wp(\mathbb{Z} \times \mathcal{V})$ can or cannot be first abstracted to $\wp(\mathcal{V})$ when we do not want to relate array element values to their index. In the first case we have a relational analysis (e.g. in Sect. 11.1), in the second a non-relational (e.g. in Sect. 4.3).

10. Conversion between the Variable and Array Element Abstract Domains

In general the variable and array elements abstractions do differ so that a conversion from one to the other is needed.

A variable to array element abstract property conversion is involved in an assignment $A[i] = e$; (handled as $A[i] = (i, e)$), while an array element to variable property conversion is required in an assignment $x := A[i]$;, and no conversion is required in $A[i] := A[j]$; or $i = j$;. This is taken care of by a conversion abstract domain providing the two conversion functions. There-

fore, the analysis must be parameterized by a conversion abstract domain functor $\mathbf{C}(\mathbf{A}, \mathbf{R})$ which contains two conversion functions from variable abstract properties in $\overline{\mathcal{R}}$ to abstract array elements properties in $\overline{\mathcal{A}}$ and inversely. This domain can also abstract (i, e) into e to get array content analyses not relating indexes to indexed array elements.

11. FunArray: The Array Segmentation Abstract Domain Functor

The array segmentation abstract domain $\mathbf{S}(\mathbf{B}(\mathbf{E}), \mathbf{A}, \mathbf{R})$ abstracts a set of possible array contents by consecutive, non-overlapping segments covering all array elements. The precision/cost ratio of the array segmentation analysis can be adjusted to a specific application domain by changing the abstraction \mathbf{R} of scalar variable environments, the normal form \mathbf{E} of symbolic expressions, hence that of the segment bounds $\mathbf{B}(\mathbf{E})$, the abstraction \mathbf{A} of the abstract array elements, as well as the various parameters of these abstract domains (such as the degree of refinement of expression comparison in \mathbf{E} in Sect. 7.4, hence of segment bounds comparison of $\mathbf{B}(\mathbf{E})$ in Sect. 8.2).

11.1 Examples of array segmentation functor instantiations

To illustrate the possibility of relating the value of array elements to their index, let us consider the static analysis of

```

int n = 10, i = 0;
int[] A = new int[n];

/* 1: */ while /* 2: */ (i < n) {
/* 3: */   A[i] = 0;
/* 4: */   i = i + 1;
/* 5: */   A[i] = -16;
/* 6: */   i = i + 1;
/* 7: */ }
/* 8: */

```

(3)

typical of data transfer protocols where even and odd numbered packets contain data of different types e.g. [13, Sec. 6.6.3], [20]. We will combine parity (where \perp (i.e. \perp) is unreachable, o is odd, e is even, T (i.e. \top) is unknown) and intervals.

Example 2 The first abstraction is the reduced product [7] of parity and intervals where pairs of a parity and an interval denote the conjunction of both properties (with a reduction e.g. of bounds by parity (such as $(e, [0, 9]) \rightarrow (e, [0, 8])$) and parity for constant intervals (such as $(T, [1, 1]) \rightarrow (o, [1, 1])$). In the following analysis of (3) this abstraction is used both for variables and array elements (hence ignoring their relationship to indexes since $\llbracket (i, e) \rrbracket \overline{\rho} = (\text{parity}(\llbracket e \rrbracket \overline{\rho}), \text{interval}(\llbracket e \rrbracket \overline{\rho}))$).

```

p1 = [ A: <{0 i}> (T, [-oo,+oo]) {n 10}>> ]
     [ i: (e, [0,0]) n: (e, [10,10]) ]
p2 = [ A: <{0}> (e, [-16,0]) {i}? (T, [-oo,+oo]) {n 10}?>> ]
     [ i: (e, [0,10]) n: (e, [10,10]) ]
p8 = [ A: <{0}> (e, [-16,0]) {n 10 i}>> ]
     [ i: (e, [10,10]) n: (e, [10,10]) ]

```

The analysis of i starts with the initial value $(e, [0, 0])$ and is $(e, [0, 2])$ after one iteration which is widened to $(e, [0, +oo])$ hence stable. The narrowing phase starts with the test $i < n$ where n in $[10, 10]$ so i is in $(e, [0, 9])$ hence $(e, [0, 8])$ by reduction through evenness. After one more iteration we get back $(e, [0, 10])$ to narrow $(e, [0, +oo])$ which is $(e, [0, 10])$ and is a fixpoint. \square

Example 3 The second abstraction is the reduced cardinal power [7] of intervals by parity whose abstract properties have the form $(o \rightarrow i_o, e \rightarrow i_e)$ meaning that the interval is i_o (resp. i_e) when the parity is o (resp. e). In the following non-relational analysis of (3), we use the reduced product of parity and intervals for simple variables and the power of parity by interval for array elements

(hence ignoring their relationship to indexes since $\llbracket(i, e)\rrbracket\bar{\rho}$ maps *parity*($\llbracket e \rrbracket\bar{\rho}$) to *interval*($\llbracket e \rrbracket\bar{\rho}$)). For example $(o \rightarrow _|_, e \rightarrow [-16, 0])$ means that the indexed array elements must be even with value included between -16 and 0 .

```
p1 = [ A : <{0 i} (o -> [-oo,+oo], e -> [-oo,+oo]) {n,10}> ]
      [ i : (e, [0,0]) n : (e, [10,10]) ]
p2 = [ A : <{0} (o -> \_|\_, e -> [-16,0]) {i}? ,
      (o -> [-oo,+oo], e -> [-oo,+oo]) {n,10}?> ]
      [ i : (e, [0,10]) n : (e, [10,10]) ]
p8 = [ A : <{0} (o -> \_|\_, e -> [-16,0]) {n 10 i}> ]
      [ i : (e, [10,10]) n : (e, [10,10]) ]
```

Observe that the abstraction is more powerful but the result is exactly the same as in the above analysis using the reduced product since $(o \rightarrow _|_, e \rightarrow [-16, 0])$ is exactly $(e, [-16, 0])$ on array elements. \square

Example 4 The third abstraction also uses the reduced cardinal power of intervals by parity, but this time in a relational way for arrays thus relating the parity of an index to the interval of possible variation of the corresponding element (so $\llbracket(i, e)\rrbracket\bar{\rho}$ is a map of *parity*($\llbracket i \rrbracket\bar{\rho}$) to *interval*($\llbracket e \rrbracket\bar{\rho}$)). We get

```
p1 = [ A : <{0 i} (o -> [-oo,+oo], e -> [-oo,+oo]) {n 10}> ]
      [ i : (e, [0,0]) n : (e, [10,10]) ]
p2 = [ A : <{0} (o -> [-16,-16], e -> [0,0]) {i}?
      (o -> [-oo,+oo], e -> [-oo,+oo]) {n 10}?> ]
      [ i : (e, [0,10]) n : (e, [10,10]) ]
p8 = [ A : <{0} (o -> [-16,-16], e -> [0,0]) {n 10 i}> ]
      [ i : (e, [10,10]) n : (e, [10,10]) ]
```

so that the array elements with odd index are shown to be equal to -16 while those of even index are zero. \square

11.2 Abstract Predicates

The array segmentation abstract predicates belong to $\bar{\mathcal{S}} \triangleq \{(\bar{\mathcal{B}} \times \bar{\mathcal{A}}) \times (\bar{\mathcal{B}} \times \bar{\mathcal{A}} \times \{ _, ? \})^k \times (\bar{\mathcal{B}} \times \{ _, ? \}) \mid k \geq 0\} \cup \{\perp\}$ and have the form

$$\{e_1^1 \dots e_{m_1}^1\} P_1 \{e_1^2 \dots e_{m_2}^2\} [?] P_2 \dots P_{n-1} \{e_1^n \dots e_{m_n}^n\} [?]^n$$

where

- the segment bounds $\{e_1^i \dots e_{m_i}^i\} \in \bar{\mathcal{B}}, i \in [1, n], n > 1$, are finite non-empty sets of symbolic expressions in normal form $e_j^i \in \bar{\mathcal{E}}$ as respectively considered in Sect. 8 and Sect. 7.2;
- the $P_i \in \bar{\mathcal{A}}$ are abstract predicates chosen in an abstract domain \mathbf{A} denoting possible values of pairs (index, indexed array element) in a segment; and
- the optional question mark $[?]^i$ follows the upper bound of a segment. Its presence $?$ means that the segment might be empty. Its absence $_$ means that the segment cannot be empty. Because this information is attached to the segment upper bound (which is also the lower bound of the next segment), the lower bound $\{e_1^1 \dots e_{m_1}^1\}$ of the first segment never has a question mark. $(\{ _, ? \}, \preceq, \Upsilon, \lambda)$ is a complete lattice with $_ < ?$.

The symbolic expressions $e_i^k \in \bar{\mathcal{E}}$ in a given segment bound depend on scalar variables but not on array elements hence $\mathbf{A}[\mathbf{A}[i]]$ should be handled as $\mathbf{x}=\mathbf{A}[i]$; $\mathbf{A}[\mathbf{x}]$ so that the auxiliary variable \mathbf{x} can appear in a segment bound for array \mathbf{A} . The consecutive segment bounds are in strictly increasing order in the concrete except when followed by a question mark meaning that the preceding block may be empty. There is no hole between segments (since this hole can always be viewed as another segment whose properties are unknown). The first block limit always contains an expression in normal form denoting the array lower bound while the last block always contains an expression in normal form denoting the array upper bound. Within one block the abstraction is uniform (but can be relational, since the array semantics of Sec. 6.3 can relate the array value $\mathbf{A}[i]$ to the index i). A possible refinement would be

to introduce relationships between segment emptiness marks (so as to express that in $\{0\} 0\{i\} ? \top \{n\} ?$ both segments cannot be simultaneously empty), which we do not do for the sake of efficiency.

11.3 Concretization

Given the concretizations $\gamma_v \in \bar{\mathcal{R}} \mapsto \wp(\mathcal{R}_v)$ for the variable abstract domain and $\gamma_a \in \bar{\mathcal{A}} \mapsto \wp(\mathbb{Z} \times \mathcal{V})$ for the array elements abstract domain, the concretization γ_s of an abstract array segmentation is an array property so $\gamma_s \in \bar{\mathcal{S}} \mapsto \wp(\mathcal{A})$.

The concretization of a segment $B P B' [?]$ is the set of arrays whose elements in the segment $[B, B']$ satisfy the abstract property P ($_ < _$ stands for $<$ while $< ?$ stands for \leq):

$$\begin{aligned} \gamma'_s(B P B' [?])\bar{\rho} &\triangleq \\ \{(\rho, \ell, h, A) \mid \rho \in \gamma_v(\bar{\rho}) \wedge \forall \mathbf{e}_1, \mathbf{e}_2 \in B : \forall \mathbf{e}'_1, \mathbf{e}'_2 \in B' : \\ &\llbracket \ell \rrbracket \rho \leq \llbracket \mathbf{e}_1 \rrbracket \rho = \llbracket \mathbf{e}_2 \rrbracket \rho < [?] \llbracket \mathbf{e}'_1 \rrbracket \rho = \llbracket \mathbf{e}'_2 \rrbracket \rho \leq \llbracket h \rrbracket \rho \wedge \\ &\forall i \in \llbracket [\mathbf{e}_1] \rrbracket \rho, \llbracket [\mathbf{e}'_1] \rrbracket \rho \} : A(i) \in \gamma_a(P)\} \end{aligned}$$

The concretization of an array segmentation $B_1 P_1 B_2 [?] P_2 \dots P_{n-1} B_n [?]^n$ is the set of arrays whose elements in all segments $[B_i, B_{i+1}], i = 1, \dots, n-1$ satisfy abstract property P_i and whose lower and upper bounds are respectively given by B_1 and B_n .

$$\begin{aligned} \gamma_s(B_1 P_1 B_2 [?] P_2 \dots P_{n-1} B_n [?]^n)\bar{\rho} &\triangleq \\ \{(\rho, \ell, h, A) \in \bigcap_{i=1}^{n-1} \gamma'_s(B_i P_i B_{i+1} [?]^{i+1})\bar{\rho} \mid \\ &\forall \mathbf{e}_1 \in B_1 : \llbracket [\mathbf{e}_1] \rrbracket \rho = \llbracket \ell \rrbracket \rho \wedge \forall \mathbf{e}_n \in B_n : \llbracket [\mathbf{e}_n] \rrbracket \rho = \llbracket h \rrbracket \rho\} \end{aligned}$$

and $\gamma_s(\perp) = \emptyset$.

11.4 Abstract operations

Abstract value of an indexed array element Assume that we have to evaluate $\llbracket \mathbf{A}[\mathbf{e}] \rrbracket\bar{\rho}$ for the array \mathbf{A} abstracted by the segmentation $B_1 P_1 B_2 [?] P_2 \dots P_{n-1} B_n [?]^n$. The expression $B_1 \leq \mathbf{e} \leq B_n$ is evaluated in the abstract and a warning is emitted if the result is unreachable (dead code), false (definite error) or unknown (potential error). Let B_ℓ be the largest segment bound such that $B_\ell \leq \mathbf{e}$ is true (B_1 otherwise) and B_h be the smallest segment bound such that $\mathbf{e} < B_h$ is true (B_n otherwise, assuming that execution goes on only in absence of buffer overrun). The value of $\llbracket \mathbf{A}[\mathbf{e}] \rrbracket\bar{\rho}$ is then $\bigsqcup_{k=\ell}^{h-1} P_k$ where \bigsqcup is the join in the domain \mathbf{A} abstracting (index, value of indexed array element) pairs. A call to a conversion function of \mathbf{C} is necessary if this abstract value in \mathbf{A} must be converted to a variable abstract value in \mathbf{R} .

Assignment to an array element In an array element assignment $\mathbf{A}[\mathbf{e}] = \mathbf{e}'$ with abstract variable environment $\bar{\rho}$ where the array \mathbf{A} is abstracted by the segmentation $B_1 P_1 B_2 [?] P_2 \dots P_{n-1} B_n [?]^n$, we first determine the range of segments such that $B_\ell \leq \mathbf{e} < B_h$ is definitely true. The segmentation of \mathbf{A} can be thought of as being abstracted to $B_1 P_1 \dots B_\ell [?] (\bigsqcup_{k=\ell}^{h-1} P_k) B_h [?] P_h \dots P_{n-1} B_n [?]^n$ where $[?]$ is $?$ if all the $[?]^{i+1}, \dots, [?]^h$ are $?$ (so that the block $B_\ell \dots B_h$ can then be empty) and $_$ otherwise. Of course it may happen that $h = \ell + 1$ in which case only one segment is concerned or $\ell = 1$ and $h = n$ in which case all segments are smashed. In all cases, the assignment is definitely in the segment $B_\ell \dots B_h$ (may be at its borders). This segment is split. Let $P \in \bar{\mathcal{A}}$ be the abstraction of the value of the pair $(\mathbf{e}, \mathbf{e}')$ in \mathbf{A} . After the array element assignment, the array segmentation of \mathbf{A} becomes $B_1 P_1 \dots B_\ell [?] (\bigsqcup_{k=\ell}^{h-1} P_k) \{\mathbf{e}\} [?] P \{\mathbf{e} + 1\} (\bigsqcup_{k=\ell}^{h-1} P_k) B_h [?] P_h \dots P_{n-1} B_n [?]^n$. $[?]_i$ is $?$ unless the segment bounds comparison discussed in Sect. 8.2 can determine that $B_\ell < \{\mathbf{e}\}$ is always true. Similarly, $[?]_r$ is $_$ when $\{\mathbf{e} + 1\} < B_h$ for sure and $?$ otherwise.

There are special cases. When the index expression e or $e+1$ or both cannot be put in the normal form of \mathbf{E} , we have to merge the corresponding segments, to get $B_1P_1 \dots B_\ell[?] (P \sqcup \bigsqcup_{k=\ell}^{h-1} P_k) B_h[?] P_h \dots P_{n-1} B_n[?]$ in the worst case. This is the case of the assignment (*) in Fig. 1. If the segment bounds comparison can determine that $B_\ell = \{e\}$ we get $B_1P_1 \dots (B_\ell \cup \{e\})[?] P(\{e+1\}) (\bigsqcup_{k=\ell}^{h-1} P_k) B_h[?] P_h \dots P_{n-1} B_n[?]$. Similarly if $\{e+1\} = B_h$ for sure, we get $B_1P_1 \dots B_\ell[?] (\bigsqcup_{k=\ell}^{h-1} P_k) \{e\}[?] P(\{e+1\}) \cup B_h) P_h \dots P_{n-1} B_n[?]$.

Test of an array element A test $c(A[e])$ of an array element $A[e]$ can be done by getting the abstract value of (i, v) of $(e, A[e])$ in \mathbf{A} , restricting the abstract value (i, v) by restricting i to the array bounds (execution is assumed to stop in case of buffer overrun) and v to the test $c(v)$, and assigning the restricted value back to the array element $A[e]$. For a simpler uniform treatment of tests involving both scalar variables and array elements, (i, v) in \mathbf{A} can be converted to the variable abstract domain \mathbf{R} and back after handling the test using \mathbf{C} .

Invertible assignment to a scalar variable In an invertible assignment to a scalar variable $x = f(x, \vec{y})$ where $\vec{y} = y_1, \dots, y_m$, we have $x_{\text{new}} = f(x_{\text{old}}, \vec{y})$, where f is the value of f , x_{old} denotes the value of the variable x before assignment, x_{new} denotes the value of the variable x after assignment, f has no side effect so the values \vec{y} of the variables \vec{y} are not changed, and $x_{\text{old}} = f^{-1}(x_{\text{new}}, \vec{y})$. For such an invertible assignment, all occurrences of the variable x in the expressions in the segment bounds must be replaced by the expression $f^{-1}(x, \vec{y})$ and the resulting expressions simplified into canonical normal form, if any, and dropped otherwise. In case a segment bound becomes empty because normalization is impossible, the two adjacent segments must be joined. For the example in Fig. 2, we have:

```
/* [ A : <{0} 0 {i}? 0 {i+1} T {n}? > ] [...] */
i = i + 1;
/* [ A : <{0} 0 {i-1}? 0 {i} T {n}? > ] [...] */
```

Non-invertible assignment to a scalar variable In a non invertible assignment $x = f(x, \vec{y})$, no inverse $x_{\text{old}} = f^{-1}(x_{\text{new}}, \vec{y})$ is available, for example in $x = f(\vec{y})$. For such a non invertible assignment $x = e$, all expressions in the segment bounds containing occurrences of the variable x must be eliminated from these segment bounds. In case a segment bound becomes empty, the two adjacent segments must be joined. Then the variable x and expression e are added to the segment bound containing an expression e' such that $\llbracket e' = e \rrbracket \bar{p}$ is definitely true in the abstract. In the simple purely syntactic case of Sect. 7.4, $\llbracket e' = e \rrbracket \bar{p} = \text{true}$ is under-approximated by $e' = e$ so that x is added to all segment bounds containing e . For the example of Fig. 2 we have:

```
/* [ A : <{0} T {n}? > ] [ i : T n : T ] */
i = 0;
/* [ A : <{0} i T {n}? > ] [ i : 0 n : T ] */
```

In case of an assignment $x = f(\vec{y})$, $x \notin \vec{y}$ where $y_i = f_i^{-1}(x_{\text{new}}, y_1, \dots, y_{i-1}, y_{i+1}, \dots, y_m)$ is available, the expression $e(f_i^{-1}(x_{\text{new}}, y_1, \dots, y_{i-1}, y_{i+1}, \dots, y_m))$ can be added to all segment bounds containing an expression $e(y_i)$ whenever simplification in canonical normal form is possible. For example:

```
/* [ A : {0} 0 {i+5, j+7} T {10, n} ] [...] */
i = j - 7;
/* [ A : {0} 0 {i+14, j+7} T {10, n} ] [...] */
```

Comparison of scalar variable expressions The equality comparison $e = e'$ (where e and e' have equivalent normal forms) together with the segment bounds comparison discussed in Sect. 8.2 that may be able to determine that $e = B_i^1$ is always true will

add e and e' to B_i . If moreover, $e' = B_j$ for sure and $i < j$ then the segmentation $B_1P_1B_2[?]P_2 \dots P_{n-1}B_n[?]$ will be reduced to $B_1P_1B_2[?]P_2 \dots P_{i-1}(B_i \cup B_{i+1} \cup \dots \cup B_j \cup \{e, e'\})[?]P_j \dots P_{n-1}B_n[?]$ or to unreachability when one of the $[?^{i+1}]$, \dots , $[?^j]$ is \sqcup (since then $e < e'$). The comparison $e \leq e'$ has the same effect when $j < i$ or when $e < e'$ and $i = j$.

Otherwise disequality $e <> e'$ and strict inequality $e < e'$ with $e = B_i$ and $e' = B_{i+1}$ can be used to remove a doubt ? on the possible emptiness of a segment $B_iP_iB_{i+1}$? which becomes $B_iP_iB_{i+1}$.

Segmentation unification Given two segmentations with equal extremal segment bounds (in general for the same array), the objective of segmentation unification is to modify the two segmentations so that they coincide. There is a lattice of solutions. The most precise solution is the coarsest common refinement of both segmentations. The most imprecise solution just joins all segments and keeps the extremal bounds only. The most refined unification is obtained by splitting segments, the resulting segment have all the segments whose bounds do appear in one or the other initial segmentation. The least refined unification is obtained by merging consecutive segments, indeed all of them, just keeping the common extremal bounds.

Because one segment can be empty in one segmentation (like $\{0 \ i\}$) and non-empty in the other one (like $\{0\}P\{1, i\}$) segmentation must include the splitting of empty segments (like $\{0 \ i\} \rightarrow \{0\}P'\{i\}$). Such an empty segment splitting is used in the comparison/join/meet/widening/narrowing of segments (which are not all commutative) so that the abstract value P' of the created empty segment must be chosen as the left/right neutral element of the considered operation (e.g. P' is \perp for join, \top for meet, \perp on the left and \top on the right of the partial order \sqsubseteq).

The segmentations involved in a unification are usually related to different program contexts. Assume we want to unify $\{0 \ i-1\}P_1\{i\}$ and $\{0 \ i-2\}P_2\{i\}$ (obviously in two different contexts). The coarsest common refinement is $\{i-2\} \perp \{0\} \perp \{i-1\} ? P_1 \{i\}$ for $\{0, i-1\} P_1 \{i\}$ and $\{i-2\} \perp \{0\} ? P_2 \{i-1\} P_2 \{i\}$ for $\{0, i-2\} P_2 \{i\}$ (which would yield the join $\{i-2\} \perp \{0\} ? P_2 \{i-1\} ? P_1 \cup P_2 \{i\}$). However, it might be the case that $i < 2$ from the abstract variable environment, in which case the expression $i-2$ in the lower bound of the first refined segmentation is undefined. This shows that the well-definedness of the coarsest common refinement, if any, depends upon the abstract variable environment. Because we want the array segmentation analysis to have the possibility of being completely independent of the variable analysis (see Sect. 11.4), our segmentation unification does not provide the coarsest or even minimal common refinements but one which is always well-defined in absence of knowledge of the contexts of the segmentations and does terminate.

Segmentation unification algorithm First the compatibility of the two segmentations is checked to verify that they do have common lower and upper bounds (this should always be the case since they correspond to the same array whose bounds should be the limit of both segmentations). The unification proceeds recursively from left to right and maintains the invariant that the left part is already unified (\sqcup_l (resp. \sqcup_r) is the left (resp. right) neutral element).

1. $B[?_1] P_1 B'_1[?_1] \dots$ and $B[?_2] P_2 B'_2[?_2] \dots$ have same lower bounds and so keep the first segments as they are and go on with $B'_1[?_1] \dots$ and $B'_2[?_2] \dots$.
2. In case $(B \cup B_1)[?_1] P_1 B'_1[?_1] \dots$ and $B[?_2] P_2 B'_2[?_2] \dots$ with $B_1 \neq \emptyset$ and $B \cap B_1 = \emptyset$, let \bar{B}_1 be the set of expressions in B_1 appearing in the second segmentation blocks B'_2, \dots .
 - 2.1 If \bar{B}_1 is empty then go on with $B[?_1] P_1 B'_1[?_1] \dots$ and $B[?_2] P_2 B'_2[?_2] \dots$ following case 1.

¹ Recall that $e = B$ is $\exists e' \in B : e = e'$.

2.2 Otherwise go on with $B[?_1] \sqcup_l B_1? P_1 B'_1[?'_1] \dots$ and $B[?_2] P_2 B'_2[?'_2] \dots$ as in case 1.

3. The symmetrical case is similar.

4. In case $(B \cup B_1)[?_1] P_1 B'_1[?'_1] \dots$ and $(B \cup B_2)[?_2] P_2 B'_2[?'_2] \dots$ with $B_1, B_2 \neq \emptyset$ and $B \cap B_1 = B \cap B_2 = \emptyset$, let \bar{B}_1 (resp. \bar{B}_2) be the set of expressions in B_1 (resp. B_2) appearing in the second (resp. first) segmentation blocks $B'_2, \dots (B'_1, \dots)$.

4.1 If \bar{B}_1 and \bar{B}_2 are both empty, go on with $B[?_1] P_1 B'_1[?'_1] \dots$ and $B[?_2] P_2 B'_2[?'_2] \dots$ as in case 1.

4.2 Else if \bar{B}_1 is empty (so that \bar{B}_2 is not empty) then go on with $B[?_1] P_1 B'_1[?'_1] \dots$ and $B[?_2] \sqcup_r \bar{B}_2? P_2 B'_2[?'_2] \dots$ (where \sqcup_r is the right neutral element).

4.3 The symmetrical case is similar.

4.4 Finally if \bar{B}_1 and \bar{B}_2 are both non-empty then go on with $B[?_1] \sqcup_l \bar{B}_1? P_1 B'_1[?'_1] \dots$ and $B[?_2] \sqcup_r \bar{B}_2? P_2 B'_2[?'_2] \dots$ as in case 1.

5. In case $B_1[?_1] P_1 B'_1[?'_1] \dots$ and $B_2[?_2] P_2 B'_2[?'_2] \dots$ with $B_1 \cap B_2 = \emptyset$, we cannot be on the first left segment block so we have on the left $B_0[?_0] P_0 B_1[?_1] P_1 B'_1[?'_1] \dots$ and $B'_0[?'_0] P'_0 B_2[?_2] P_2 B'_2[?'_2] \dots$ and go on by merging these consecutive blocks $B_0[?_0] P_0 \sqcup P_1 B'_1[?'_1] \wedge ?'_1 \dots$ and $B'_0[?'_0] P'_0 \sqcup P_2 B'_2[?'_2] \wedge ?'_2 \dots$

6. Finally, at the end either we are left with the right limits that have both been checked to be equal or else we have $B_1[?_1] P_1 B'_1[?'_1]$ and $B_2[?_2]$ with $B'_1 = B_2$. Because we have maintained the invariant that B_1 is always equal to B_2 in the concrete (so necessarily $[?'_1] = ?$ since then $B_1 = B_2 = B'_1$), and so we end up with $(B_1 \cup B'_1 \cup B_2)[?_1]$ and $(B_1 \cup B'_1 \cup B_2)[?_2]$ \square

In the analysis of the example of Fig. 2, we have to unify $\langle \{0 \ i\} T \{n\} \rangle$ and $\langle \{0 \ i-1\} 0 \ \{1 \ i\} T \{n\} \rangle$ which becomes $\langle \{0\} \perp \{i\} T \{n\} \rangle$ and $\langle \{0\} 0 \ \{1 \ i\} T \{n\} \rangle$ by 4.3 and we go on with $\{i\} T \{n\}$ and $\langle \{1 \ i\} T \{n\} \rangle$ which, by the symmetric in 3 of 2.1 becomes $\{i\} T \{n\}$ and $\langle \{i\} T \{n\} \rangle$ so we go on with $\langle \{n\} \rangle$ and $\langle \{n\} \rangle$ which terminates the recursion by 6, thus returning $\langle \{0\}, \perp, \{i\} T \{n\} \rangle$ and $\langle \{0\} 0 \ \{i\} T \{n\} \rangle$. Their array segmentation join is then $\langle \{0\} 0 \ \{i\} T \{n\} \rangle$ (taking the disjunction \vee of potential emptiness).

The algorithm never adds any new expression to the segment bounds nor increments the total number of segment bounds in splits and so does terminate.

Partial order/join/meet/widening/narrowing For an array segmentation join $\mathbf{S}.\sqcup$, a (\perp, \perp) -segmentation unification is performed and then the array element abstract domain join $\mathbf{A}.\sqcup$ is applied segmentwise. For the meet $\mathbf{S}.\sqcap$, a (\top, \top) -segmentation unification is performed and then a segmentwise meet $\mathbf{A}.\sqcap$. For the widening $\mathbf{S}.\nabla$, a (\perp, \perp) -segmentation unification is performed and then a segmentwise widening $\mathbf{A}.\nabla$. Moreover, the widening merges consecutive segments with same abstract value. Widening could also be used to limit the size of segment bound sets and/or the number of segments given as parameters of the analysis. For the narrowing $\mathbf{S}.\Delta$, a (\top, \top) -segmentation unification is performed and then a segmentwise narrowing $\mathbf{A}.\Delta$. For the partial order $\mathbf{S}.\sqsubseteq$, a (\perp, \top) -segmentation unification is performed before returning the conjunction of the segmentwise comparisons $\mathbf{A}.\sqsubseteq$. The potential emptiness indications must also be taken into account, that is $\sqsubseteq = \sqsubseteq \prec ? = ?$.

Array segmentation reduction A program analysis is the product of a segmentation analysis for arrays and the analysis of scalar variables. The two analyses can be completely independent which is an important feature for the array segmentation analysis to be easily inserted in any analyzer without having to make any hypoth-

```

InitBackwards(int[] A) {
    int i = A.Length;
    /* 1: */ while /* 2: */ (0 < i) {
    /* 3: */     i = i - 1;
    /* 4: */     A[i] = 0;
    /* 5: */ }
    /* 6: */ }

```

Figure 4. Example of a backwards initialization. Array segmentation reduction is needed to prove the postcondition $\forall j \in [0, A.Length). A[j] = 0$.

esis on the static analyzer. The consequence is that the result may not be as precise as possible. Let us illustrate this phenomenon on the following program.

Using the independent product of interval abstractions for array elements and scalar variables, the post condition derived by the static analyzer with Sect. 7.4 at program point 6 is

```

[ A: <\{0\} [-oo,+oo] \{i\}? [0,0] \{A.Length\}?? ]
[ i: [0,0] A.Length: [2,+oo] ]

```

It states that it is possible that $i = 0$ but the array segmentation analysis cannot prove that this is indeed always the case. It is in general always more precise to consider the reduced product of the array and variable analyses [7]. This consists in iterating reduction operators that propagate information for one abstract domain to the other. For example it may be useful to propagate the relational information of array segmentation (equality of expressions in a segment bounds and segment bounds in increasing order (strictly increasing in absence of ?)), unless a more precise relational domain is already used for scalar variables. In the other direction, the information provided by the scalar variable analysis can be propagated to segmentations. A possibly empty segment $\dots B[?] P B'?$ can be reduced to a non-empty one $\dots B[?] P B'?$ if the scalar variables environment $\bar{\rho}$ implies $\exists e \in B : \exists e' \in B' : \llbracket e \rrbracket \bar{\rho} < \llbracket e' \rrbracket \bar{\rho}$ is always true in the abstract (the abstract test returning either \perp , true, false, or unknown). Similarly, a possibly empty segment $\dots B[?] P B'?$ may be definitely empty and reduced to the bound $\dots (B \cup B')[?] \dots$ when $\exists e \in B : \exists e' \in B' : \llbracket e \rrbracket \bar{\rho} = \llbracket e' \rrbracket \bar{\rho} = true$.

In the reduction example of Fig. 4, the fact that $i \in [0, 0]$ implies that the segment $\{0\} [-oo, +oo] \{i\}$ is empty, in which case the reduction automatically yields

```

[ A: <\{0 \ i\} [0,0] \{A.Length\} > ]
[ i: [0,0] A.Length: [2,+oo] ]

```

which is exactly the expected result at program point 6.

12. Experience

12.1 CodeContracts and Clousot

CodeContracts allow the language-agnostic specification of contracts (preconditions, postconditions and object-invariants [3, 26]). The CodeContracts API is included in .NET starting from v4.0. Clousot is an abstract interpretation-based static analyzer developed at MSR Redmond used to statically check: (i) contracts; and (ii) the absence of common runtime errors such as non-null dereferences or buffer overruns [11]. Clousot is used both inside and outside Microsoft on large production projects, and counts more than 20000 external downloads at the moment of writing. When a method is analyzed, its preconditions is turned into an assumption and its postcondition into an assertion. For each method call appearing in the method body, its precondition is turned into an assertion and the postcondition into an assumption. Object-invariants are assumed at the entry of public methods and asserted at the exit point (a detailed description of the object-invariants treatment is out-of-the scope of this paper). Further assertions are generated

from the body text: *e.g.* when an array is accessed the indexing expression is better being in bounds. Clousot analyzes the bytecode, which presents several advantages (independence from the compiler, the language, the language version ...) but also some drawbacks (lack of program structure ...) [23]. After reading the bytecode, extracting the contracts, creating the control flow graph, and simplifying the program, a heap analysis is run so to resolve aliasing, and the program is turned into a scalar form. On the top of the program scalar form several forward value analyses are run, and their results are used to discharge the assertions. If an assertion cannot be discharged, then the analysis is refined by using a more precise abstract domain or a goal-directed backward propagation. If refinement does not work, then a warning is reported to the user. Warnings are issued because of a lack of knowledge (*e.g.* missing postcondition, precondition too weak ...), incompleteness of the analysis (inevitable in all the static analyses), or because of too complex assertions (*e.g.* quadratic inequalities).

Before this work, quantified assertions over arrays (*e.g.* all the elements are non-null) were not understood by Clousot which reported warnings for assertions as *e.g.* the ones in Fig. 1. Furthermore, the analysis was also very imprecise in handling array loads (and iterations over collections), so that each time a value was loaded from an array, nothing could be stated on that value, hence the worst case was assumed, degrading the analysis precision. Fixing those issues were the #1 request from Clousot’s users.

12.2 Implementation of FunArray in Clousot

We fully implemented FunArray in Clousot. To the user (typically a programmer with no background in formal methods) FunArray is exposed as a simple check-box in Visual Studio. When the check-box is enabled, the FunArray is transparently instantiated with the abstract domains in Clousot. The array analysis is orthogonal to the other components of Clousot, so that it can benefit of precision improvements for free. For instance, if a more precise scalar variable abstract environment is used, then the FunArray analysis is likely to get more precise.

In the current implementation: (i) the abstract domain used for the segment elements is the disjoint union of intervals with non-nullness; (ii) the expressions used in the segment limits are the simple expressions of Sect. 7.2 augmented with explicit casts to model the fact that `a.Length` and `(int) a.Length` both denote the length of an array `a` in the bytecode; and (iii) reduction is achieved via the generic interfaces exposed by Clousot’s abstract domains.

We validated the performances of the analysis by running on large, production quality libraries. We validated its precision by running it on its own implementation (Clousot is written in C#).

12.3 Analysis of large libraries

We report the experience of running Clousot on the main libraries of the .NET framework. The `mscorlib.dll` and `System.dll` libraries provide core functionalities such as basic types, optimized data structures, cryptographic primitives, date manipulation, interfaces with the operating system, etc. The other libraries focus on database interfacing (`System.data.dll`), bitmap manipulations (`System.Drawings.dll`), WEB contents (`System.Web.dll`), XML parsing and creation (`System.Xml.dll`). Libraries have been authored by several different programmers over the years, hence present all kinds of different programming styles and optimizations. We randomly inspected a large set of methods containing loops with arrays. We found few cases of code as Fig. 2, whereas Fig. 3 is a lot more common pattern. Other common idioms include the initialization using multiple loops, conditional initialization of an array prefix (or postfix) followed (or not) by the initialization of the remaining array segment (or a sub-segment).

From our manual inspection we deduced that simple partitions based on the assumption that arrays are uniformly traversed from the first to the last element (*e.g.* [24]) simply do not apply to existing .NET code. Other syntactic-based partitioning heuristics do not apply as well (roughly because at bytecode level the structure of loops has been compiled away).

At the beginning of this project, our first attempt was to implement the technique of [16] on the top of a semantic analysis to determine the array partitions. Performances turned out to be extremely bad: up to $100\times$ slower w.r.t. a normal run of Clousot. The main reasons for the bad performance were: (i) the large number of generated slices (because of lack of possibly empty segments, unlike us); (ii) the need to re-run the analysis once the partition is discovered (*e.g.* to distinguish the first iteration from all the others); (iii) the cost of partition changes (detailed in [16, Sect. 5]). More generally, the problem of [14, 16] is that they abstract too much the concrete environment, *e.g.* by separating the array partitioning from the discovery of array partitions (we do it at the same time instead) and *e.g.* by forgetting the relative positioning of array slices (we have consecutive segments). The information lost because of the rough abstraction should be recovered during the array analysis.

Next, we developed FunArray in which we made sure that: (i) the array analysis is run *at the same time* as the scalar variable analysis; and (ii) explicit partition enumeration is avoided by means of possibly empty segments, *i.e.* at each program point there is at most *one* approximation for a given array. We first sketched the analysis in a research prototype (ArrayaL), to experiment and adjust the algorithms and then we validated it by integrating it in Clousot.

Our analysis turned out to be extremely fast. We report the experimental results in Tab. 1. For each library, we report the number of functions, the analysis time *without* the array analysis, the analysis time *with* the array analysis, the slowdown, and the number of inferred non-trivial array invariants in function postconditions. The workbench is a 2.4GHz Core 2 duo laptop running Windows 7 and .NET v3.5. The first observation is that the FunArray introduces a negligible analysis slowdown (less than 1%) whereas it discovers a thousands of non-trivial array invariants. Interestingly enough, we did not encounter any corner case causing the analysis time to blow up (unlike previous published similar techniques). This fact makes us comfortable to state that the analysis scales up well. We were also positively impressed by the fact that the analysis was able to handle complex initialization patterns such as the one in Fig. 1, which were not considered at all during its designing, meaning that the analysis is robust enough to handle *unexpected* code (a problem that unfortunately afflicts several static analyses usually developed for few coding patterns).

Lib	# func.	time	time w.		Δ	# inv
			arr.			
<code>mscorlib.dll</code>	21 475	4:06	4:15	0:09	2 430	
<code>System.dll</code>	15 489	3:40	3:46	0:06	1 385	
<code>System.data.dll</code>	12 408	4:49	4:55	0:06	1 325	
<code>System.Drawings.dll</code>	3 123	0:28	0:29	0:01	289	
<code>System.Web.dll</code>	23 647	4:56	5:02	0:06	840	
<code>System.Xml.dll</code>	10 510	3:59	4:16	0:17	807	

Table 1. The execution time with and without the array analysis, the slow-down and the number of non-trivial array invariants. Time is in minutes. The incidence of array analysis is a mere 1%.

12.4 Analysis of annotated code

To validate the precision of the analysis in the context of contract checking we ran it against its own implementation in Clousot. We implemented FunArray with a pair of *mutable* sequences, one

for the bounds and the other for the array elements. Each sequence should only contain non-null elements. Sequences are implemented as partially filled arrays (to optimize cache hits). The class `NonNullSeq` abstracts sequences, allowing for in-place insertion, manipulation, update and removal of elements (13 methods in total). The object invariant of `NonNullSeq` states that all the elements in the partially filled array are non-null. The analysis of `NonNullSeq` without the functor analysis takes 5.36 seconds, reporting 11 warnings (out of 210 proof obligations). The analysis of `NonNullSeq` with the functor analysis takes 3.85 seconds, with 0 warnings! Therefore the more precise analysis is also faster. The reason why is that the array analysis induces a negligible slowdown and discover more facts on the program, which can be directly used to discharge the proof obligations, without moving to more refined analyses. Once `NonNullSeq` has been verified, then we considered the `FunArray` implementation (78 methods), where we were able to prove 61 further proof obligations, out of 1800 total (`FunArray` analysis cost was negligible). The remaining 8 warnings are issued by a possible violation of the precondition of the segment unification algorithm (out-of-reach of `C1ousot`, and maybe of existing SMT solvers). Overall, our experience matched the feedback from our users (we do not have access to their code though): `FunArray` reduced the number of false positives at a negligible cost.

13. Conclusions

Beyond segmentation, the smashing within segments, and the treatment of disjunctions (through possibly empty segments, symbolic bounds with different instances, and relation of array values to indexes), the main idea is to derive the segment bounds through array accesses in array element tests and assignments (so that we do not rely on the end-user or other analyses/tools to infer the segment bounds) as well as the segment unification for partial order, joins, meets, widenings, and narrowings to avoid explosion. Although expressiveness of array segmentation is limited, our analysis is self-contained without hidden hypotheses. It has proved to be simple enough to scale up in production-quality static analysis tools (whereas a previous attempt based on [14, 16] did not). We used it to validate its own implementation, effectively reducing to zero the false alarms.

The approach is applicable to matrices of higher dimensions by recursively instantiating the functor on an array instantiation. The work can be extended to relational properties among segments by using an auxiliary scalar variable to denote the value of any element of a segment and relating the values of these auxiliary scalar variables for different segments by a relational abstraction in the scalar environment. This would handle the partitioning in QuickSort. Intra-segment relational properties can also be considered by using several auxiliary scalar variables x_i, x_j, \dots to denote the values of elements indexed $i \leq j \leq \dots$ within the segment and relating them in the scalar environment. This would handle sorting algorithms [5]. Of course inter-segments and intra-segment relational properties can be combined. The extra cost makes those relational analyses probably inappropriate in a general-purpose and scalable static verifier such as `C1ousot`.

References

[1] M. Barnett, K. Leino, and W. Schulte. The Spec# programming system: An overview. *CASSIS'04*, LNCS 3362, 49–69. Springer, 2005.

[2] M. Barnett, B.-Y. Chang, R. DeLine, B. Jacobs, and K. Leino. Boogie: A modular reusable verifier for object-oriented programs. *FMCO'05*, LNCS 4111, 364–387. Springer, 2006.

[3] M. Barnett, M. Fähndrich, and F. Logozzo. Embedded contract languages. *SAC'10*. ACM Press, 2010.

[4] P. Chalin, J. Kiniry, G. Leavens, and E. Poll. Beyond assertions:

Advanced specification and verification with JML and ESC/Java2. *FMCO'05*, LNCS 4111, 77–101. Springer, 2006.

[5] P. Cousot. Verification by abstract interpretation. *Verification – Theory & Practice*, LNCS 2772, 243–268. Springer, 2003.

[6] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. *4th POPL*, 238–252. ACM Press, 1977.

[7] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. *6th POPL*, 269–282. ACM Press, 1979.

[8] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. *TACAS'08*, LNCS 4963, 337–340. Springer, 2008.

[9] D. Dill. Timing assumptions and verification of finite-state concurrent systems. *Automatic Verification Methods for Finite State Systems*, LNCS 407, 197–212. Springer, 1989.

[10] I. Dillig, T. Dillig, and A. Aiken. Fluid updates: Beyond strong vs. weak updates. *ESOP'10*, LNCS 6012, 246–266. Springer, 2010.

[11] P. Ferrara, F. Logozzo, and M. Fähndrich. Safer unsafe code in .NET. *OOPSLA'08*. ACM Press, 2008.

[12] C. Flanagan and S. Qadeer. Predicate abstraction for software verification. *29th POPL*, 191–202. ACM Press, 2002.

[13] Garmin Int. Garmin device interface specification. Technical report, Garmin Int., Inc., Olathe, 2006. www.garmin.com/support/pdf/iop_spec.pdf.

[14] D. Gopan, T. Reps, and S. Sagiv. A framework for numeric analysis of array operations. *32nd POPL*, 338–350. ACM Press, 2005.

[15] S. Gulwani, B. McCloskey, and A. Tiwari. Lifting abstract interpreters to quantified logical domains. *35th POPL*, 235–246. ACM Press, 2008.

[16] N. Halbwachs and M. Péron. Discovering properties about arrays in simple programs. *PLDI'2008*, 339–348. ACM Press, 2008.

[17] R. Jhala and K. McMillan. Array abstractions from proofs. *CAV'07*, LNCS 4590, 193–206. Springer, 2007.

[18] G. Kildall. A unified approach to global program optimization. *1st POPL*, 194–206. ACM Press, 1973.

[19] L. Kovács and A. Voronkov. Finding loop invariants for programs over arrays using a theorem prover. *FASE'2009*, LNCS 5503, 470–485. Springer, 2009.

[20] S.-H. Lee and D.-H. Cho. Packet-scheduling algorithm based on priority of separate buffers for unicast and multicast services. *Electronics Letters*, 39(2):259–260, 2003.

[21] F. Logozzo. Class-level modular analysis for object oriented languages. *SAS'03*, LNCS 2694, 37–54. Springer, 2003.

[22] F. Logozzo. *Modular static analysis of object-oriented languages*. Thèse de doctorat en informatique, École polytechnique, 2004.

[23] F. Logozzo and M. Fähndrich. On the relative completeness of bytecode analysis versus source code analysis. *CC'08*, LNCS 4959, 197–212. Springer, 2008.

[24] M. Marron, D. Stefanovic, M. Hermenegildo, and D. Kapur. Heap analysis in the presence of collection libraries. *PASTE'07*, 31–36. ACM Press, 2007.

[25] K. L. McMillan. Quantified invariant generation using an interpolating saturation prover. *TACAS'08*, LNCS 4963, 197–212. Springer, 2008.

[26] B. Meyer. *Eiffel: The Language*. Prentice Hall, 1991.

[27] A. Miné. The octagon abstract domain. *Higher-Order and Symbolic Computation*, 19:31–100, 2006.

[28] V. Pratt. Two easy theories whose combination is hard. Technical report, MIT, 1977. boole.stanford.edu/pub/sefnp.pdf.

[29] B. Roy. Transitivité et connexité. *Comptes-Rendus de l'Académie des Sciences de Paris, Sér. A-B*, 249:216–218, 1959.

[30] M. Seghir, A. Podelski, and T. Wies. Abstraction refinement for quantified array assertions. *SAS'09*, LNCS 5673, 3–18. Springer, 2009.

[31] R. Shostak. Deciding linear inequalities by computing loop residues. *JACM*, 28(4):769–779, 1981.