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

Patrick Cousot
École normale supérieure &
New York University
Courant Institute of Mathematical Sciences
cousot@ens.fr, pcousot@cs.nyu.edu

Radhia Cousot
Centre National de la Recherche Scientifique
École normale supérieure &
Microsoft Research, Redmond
radhia.cousot@ens.fr

Francesco Logozzo
Microsoft Research, 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. 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 symbolic expressions appearing in a bound set are equal in the concrete. The `FunArray` can be naturally combined via reduced product with any existing analysis for scalar variables. The analysis is presented as a general framework parameterized by the choices of bound expressions, segment abstractions and the reduction operator. 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 non-trivial 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—formal methods, validation, assertion checkers; D.3.1 [*Programming Languages*]: Formal Definitions and Theory—semantics; 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—Program analysis.

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

Keywords Abstract interpretation, Array abstraction, Array content analysis, Array property inference, Invariant synthesis, Static analysis, Program verification.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

POPL'11, January 26–28, 2011, Austin, Texas, USA.
Copyright © 2011 ACM 978-1-4503-0490-0/11/01. . . $10.00

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 [7]. 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 [6]). 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 $1 \ldots 54$ to be ≥ -1 according to the non-contiguous indexing sequence: $21, 42, 8, \ldots$, 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 [10]. We let the verification process run for a whole week-end without getting an answer. The theorem prover

```
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 preconditions, assertions and bounded universal quantifications [3].

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 [15] 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 $1 \leq$ `index` < 55 to determine that the last element of `SeedArray` is never overwritten in `Loop 1`, or that `num3` ≥ -1. Both properties cannot be inferred with syntactic heuristics.

Array Partitioning The array partitioning approach of Gopan, Reps and Sagiv [17] (later improved by Péron and Halbwachs [19]) 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 [17, 19, 29]) 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 pre-analysis approach in `Clousot` [14]. 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 [19, Sect. 15]). Performance turned out to be extremely bad. The first pre-analysis generated *too many* partition slices (also noticed by Dillig *et al* [12, 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 [8]); (ii) the array segmentation is automatically and semantically discovered by the analysis; and (iii) the segmentation admits possibly empty segments. In particular, possibly empty segments are a winning choice because they enable a compact representation for array partitions avoiding the exponential multiplication of slices of the aforementioned works (Sect. **4.4**). Yang *et al* remarked similar advantages when using possibly empty list segments for shape analysis [37].

Under-approximations and Templates The technique of Gulwani, McCloskey and Tiwari [18] is extremely powerful yet expensive. It requires: (i) the user to provide templates for the array 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 [18], which seriously affects the analysis cost. According to [18, 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 [5] 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 `num3` $\geq -1 \wedge$ `i` ≥ 1 to `Loop 1`, `j` ≥ 1 to `Loop 2` and `k` ≥ 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 for. 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 [26]).

Theorem prover-based The method of Kovács and Voronkov [23] 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 [20, 30] and of Seghir, Podelski and Wies [35] 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` [26] and hence to prove the safety of the public methods. Furthermore, the techniques above do not always guarantee termination.

The fluid updates technique of Dillig, Dillig and Aiken [12] is very expressive and it can be extended to precisely track complex containers properties [13]. It is exposed to a potential exponential explosion too. Theoretically, their technique is the lifting to the reduced cardinal power [8] of a points-to analysis. Practically, every time an array is accessed or created, the points-to edges are modified, new constraints are added and calls to an SMT solver are issued to prove the (un-)feasibility of the edge(s) and simplify the constraints. This may negatively influence the performance of the analysis and also affect the precision (whenever the expressions go out of the language treated by the SMT solver, as for instance the reminder in Fig. **1**).

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 a sequence of possibly empty segments delimited by a set of segment bounds. The content of each segment is uniformly abstracted. The array analysis can be combined via a reduced product with an abstraction for scalar variables. Therefore the `FunArray` has three main parameters: (i) the expressions used to describe the segment bounds; (ii) the abstract domain used to abstract the segment values; and (iii) the abstract domain used to abstract scalar variables. When the three parameters above are chosen to be: (i) simple expressions in the form k or $\mathtt{x} + k$ where `x` is a variable and k is an integer [33]; (ii) and (iii) intervals [7] then for `Loop 1` our analysis infers that all the values of the arrays with indexes in the range $1 \ldots 54$ 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,+oo] {55} [+oo,-oo] {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,+oo] {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 [14, Sect. 6] 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. By semantically, we mean that the subdivision of the array is done during the analysis using *semantic* information, unlike the aforementioned approaches which derive the partition by looking at the syntactic structure of the program. 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 abstract domain abstracting the pairs (index, value) in segments, the abstract domain assigning values to segment bound expressions, and the reduction between those domains.
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 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 `Arrayal`, 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** and on the array rearrangement example of Fig. **4**.

4.1 Manual proof

A manual proof of the exit specification would involve a loop invariant at program point `2` stating that if $\mathtt{A.Length} = 0$ then $\mathtt{i} = 0$ and the array `A` is empty or else $\mathtt{A.Length} \geqslant 1$ in which case either $\mathtt{i} = 0$ and the array `A` is not initialized or else $\mathtt{i} > 0$

```
          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 $\forall i \in [0, \texttt{A.Length}) : \texttt{A}[i] = 0$ at program point 6.

so that $\texttt{A[0]} = \texttt{A[1]} = \ldots = \texttt{A[i} - 1\texttt{]} = 0$. Formally the invariant $(\texttt{A.Length} = 0 \wedge \texttt{i} = 0) \vee (\texttt{A.Length} \geqslant 1 \wedge 0 \leqslant \texttt{i} \leqslant \texttt{A.Length} \wedge \forall j \in [0, \texttt{i}) : \texttt{A}[j] = 0)$ holds at point 2 [(1)].

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 $0, \texttt{i}-1, \texttt{A.Length}-1$ 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 [22], 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 pi is the abstract invariant predicate at program point $i = 1, \ldots, 6$. In this example the properties of scalar variables need not be used. The abstract values for constant propagation can be $\_|\_$ (*i.e.*, bottom $\bot$, meaning unreachable), an integer constant (meaning equal to that constant), or T (*i.e.*, top $\top$, meaning unknown).

In the array environments such as `A: {0 i} T {A.Length}?` in p1, 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 $\texttt{i} = 0$ (since all expressions in a bound are equal) and that $0 = \texttt{i} \leqslant \texttt{A.Length}$ (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 $\top$ 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 ($\top$). So the invariant p1 states that $\texttt{i} = 0 \leqslant \texttt{A.Length} \wedge \forall j \in [0, \texttt{A.Length}) : \texttt{A}[j] \in \mathbb{Z}$. In particular, when $\texttt{i} = \texttt{A.Length} = 0$, the interval $[0, \texttt{A.Length})$ is empty, so the quantified expression holds vacuously.

The invariant p2 states that $0 \leqslant \texttt{i} \leqslant \texttt{A.Length}$ (in absence of question marks ? these inequalities would be strict), that $\texttt{A[0]} = \texttt{A[1]} = \ldots = \texttt{A[i} - 1\texttt{]} = 0$ when $\texttt{i} > 0$ and that the values A[i], A[i + 1], ..., A[A.Length − 1] are unknown when $\texttt{A.Length} > \texttt{i}$. 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.

(1) This invariant can also be written $0 \leqslant \texttt{i} \leqslant \texttt{A.Length} \wedge \forall j \in [0, \texttt{i}) : \texttt{A}[j] = 0$ with the convention that $[0, -1] = \emptyset$ is the empty set in which case $\texttt{A}[j]$ is not evaluated, which is made explicit by a disjunction of cases (marked ? in segmentation bounds).

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 p6 expresses that either the array is empty (*i.e.* $\texttt{A.Length} = \texttt{i} = 0$) or else $\texttt{A.Length} = \texttt{i} > 0$ and all array elements are initialized to 0.

Please note that the case $\texttt{A.Length} < 0$ is excluded. This comes from the initial condition stating that $\texttt{A.Length} \geqslant 0$ 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 $\texttt{A.Length} \geqslant 0$ is recorded in the segmentation of array A.

```
p0 = 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 $\texttt{p2} = \texttt{p3} = \ldots = \texttt{p5} = \bot$ denotes unreachability of the loop so that the abstract loop invariant is initially $\texttt{p2} = \texttt{p1} \sqcup \texttt{p5} = \texttt{p1}$ (using the join $\sqcup$ in the constant abstract domain for segments: $x \sqcup \bot = \bot \sqcup x = x$, $x \sqcup \top = \top \sqcup x = \top$, $i \sqcup i = i$, and $i \sqcup j = \top$ when $i \neq j$).

```
p2 = p1 = p0[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:

```
p3 = p2[i<A.Length] = 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:

```
p4 = p3[A[i]=0] = A: {0 i} 0 {1 i+1} T {A.Length}?
```

Please note that the segment i ... i+1 is definitely not empty while the segment i+1 ... 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:

```
p5 = p4[i=i+1] = A: {0 i-1} 0 {1 i} T {A.Length}?
```

The next approximation of the loop invariant is $\texttt{p2} = \texttt{p1} \sqcup \texttt{p5}$. This join first involves the unification of the segment $\{0\,\texttt{i}\}\top\{\texttt{A.Length}\}$ of p1 and that $\{0\ \texttt{i} - 1\}0\{1\ \texttt{i}\}\top\{\texttt{A.Length}\}?$ of p5. Keeping only the expressions appearing in both segmentations, we get $\{0\ \texttt{i}\}\top\{\texttt{A.Length}\}$ and $\{0\}0\{\texttt{i}\}\top\{\texttt{A.Length}\}?$. Splitting the bound $\{0\ \texttt{i}\}$ we get $\{0\}\bot\{\texttt{i}\}?\top\{\texttt{A.Length}\}$ so that the union with $\{0\}0\{\texttt{i}\}\top\{\texttt{A.Length}\}?$ can now be performed segmentwise in the constant domain $\{0\}\bot \sqcup 0\{\texttt{i}\}(? \curlyvee \lrcorner)\top \sqcup \top\{\texttt{A.Length}\}(\lrcorner \curlyvee ?) = \{0\}0\{\texttt{i}\}?\top\{\texttt{A.Length}\}?$ since the segments may be empty in at least one of the cases (that is $\lrcorner \curlyvee \lrcorner = \lrcorner$ for non-empty segments and otherwise $\lrcorner \curlyvee ? = ? \curlyvee \lrcorner = ? \curlyvee ? = ?$ for possibly empty ones). We get

```
p2 = p1 U p5 = A: {0} 0 {i}? T {A.Length}?
```

The next iteration is similar:

```
p3 = p2[i<A.Length] =  A: {0} 0 {i}? T {A.Length}
p4 = p3[A[i]=0] =  A: {0} 0 {i}? 0 {i+1} T {A.Length}?
p5 = p4[i=i+1] =  A: {0} 0 {i-1}? 0 {i} T {A.Length}?
p2 = p1 U p5 =  A: {0} 0 {i}? T {A.Length}?
```

```
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.

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

```
p6 = p2[i>=A.Length] = A: {0} 0 {A.Length,i}?
```

where `A.Length` $=$ `i` since the segmentation of `p2` provides the information that $0 \leqslant$ `i` $\leqslant$ `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} ... ... {n-3} ... {n-2} ... {n-1} ... {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 [7]. No widening was necessary for constant propagation which satisfies the ascending chain condition ($\bot \sqsubset i \sqsubset \top, i \in \mathbb{Z}$).

4.4 Partial Array Initialization

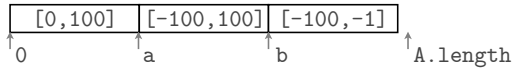
Full array initialization is a very well studied example, and array-partitioning techniques perform reasonably well on it [17, 19]. However, partial array initialization (Fig. **3**) illustrates the multiplication of partitions which makes those techniques not-scalable. At the end of the loop, our analysis (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`) ([17, 7.2]). We tried this example using our early implementation of [19] 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 [17] and those in [18] 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.

4.5 Array in-situ rearrangement example

The in-situ array rearrangement algorithm of Fig. **4** [4, 23] maintains an invariant




```
          void Rearrangement(int[] A) {
           Contract.Requires(A.length > 1);
           Contract.Requires(Contract.Forall(0,A.length,
             i => (-100 <= A[i] && A[i] <= 100)));
           int a = 0, b = A.length;
/* 1: */   while /* 2: */ (a < b) {
/* 3: */     if A[a] >= 0 then {
/* 4: */       a = a + 1;
/* 5: */     } else {
/* 6: */       b = b - 1;
/* 7: */       int x = A[a]; A[a] = A[b]; A[b] = x;
/* 8: */     } }
/* 9: */   }
```

Figure 4. The array in-situ rearrangement example.

where positive numbers are on the left of `a`, the negative numbers are on the right, from `b` included, and in the middle, between `a` and `b` $-$ 1 the numbers remain to be handled. If `A[a]` is positive, the limit `a` is moved to the right. Otherwise, `A[a]` is exchanged with `A[b-1]` and `b` is moved to the left. The algorithm terminates when the central zone is empty. This invariant which is automatically inferred by the automatic array segmentation analysis illustrates the interest of using possibly empty segments:

```
p1 = (A: {0 a} [-100,100] {b A.length}
      a:[0,0] b:[2,+oo] A.length:[2,+oo])
p2 = (A: {0}[0,100]{a}?[-100,100]{b}?[-100,-1]{A.length}?
      a:[0,+oo] b:[0,+oo] A.length:[2,+oo])
p9 = (A: {0} [0,100] {b a}? [-100,-1] {A.length}?
      a:[0,+oo] b:[0,+oo] A.length:[2,+oo])
```

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 ...[7]. 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 $\mathbf{D}_1, \ldots, \mathbf{D}_n$ to a new abstract domain $\mathbf{D}(\mathbf{D}_1, \ldots, \mathbf{D}_n)$. The term "functor" is mutated from OCaml terminology. The formal parameters $\mathbf{D}_1, \ldots, \mathbf{D}_n$ 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 $\mathbf{D}(\mathbf{D}_1, \ldots, \mathbf{D}_n)$ composes abstract properties $\overline{\mathcal{D}}_1, \ldots, \overline{\mathcal{D}}_n$ of the parameter abstract domains $\mathbf{D}_1, \ldots, \mathbf{D}_n$ 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 $\mathbf{D}_1, \ldots, \mathbf{D}_n$ 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

variable names $\mathtt{i} \in \mathbb{X}$ to their values $\rho(\mathtt{i}) \in \mathcal{V}$ so that $\mathcal{R}_v \triangleq \mathbb{X} \mapsto \mathcal{V}$. In the following we let $\mathtt{A.Length} \in \mathbb{X}$ be the name denoting the length of the array A.

6.2 Simple Expressions Semantics The program simple expressions $\mathtt{e} \in \mathbb{E}$ containing only constant, scalar variables, and mathematical unary and binary operators have a semantics $[\![\mathtt{e}]\!]\rho$ in the concrete variable environment ρ so that $[\![\mathtt{e}]\!] \in \mathcal{R}_v \mapsto \mathcal{V}$. 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 $[\![\mathtt{i := e}]\!]\rho \triangleq \rho[\mathtt{i} := [\![\mathtt{e}]\!]\rho]$ where $\rho[\mathtt{i} := v](\mathtt{i}) = v$ and $\rho[\mathtt{i} := v](\mathtt{j}) = \rho(\mathtt{j})$ when $\mathtt{j} \neq \mathtt{i}$.

6.3 Unidimensional Arrays Semantics The operational semantics of array variables (such as $\mathtt{A} \in \mathbb{A}$) are concrete array environments $\theta \in \mathcal{R}_a$ mapping array names $\mathtt{A} \in \mathbb{A}$ to their values $\theta(\mathtt{A}) \in \mathcal{A} \triangleq \mathcal{R}_v \times \mathbb{E} \times \mathbb{E} \times (\mathbb{Z} \mapsto (\mathbb{Z} \times \mathcal{V}))$ 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 $a = (\rho, \mathtt{A.low}, \mathtt{A.high}, A) \in \mathcal{A}$, where:

- $\rho \in \mathcal{R}_v$ is a scalar variable environment (Sect. **6.1**);
- $\mathtt{A.low} \in \mathbb{E}$ is an expression (0 in our examples) which value $[\![\mathtt{A.low}]\!]\rho$ evaluated in the variable environment ρ yields the integer lower bound of the array;
- $\mathtt{A.high} \in \mathbb{E}$ (A.Length in our examples) is an expression which value $[\![\mathtt{A.high}]\!]\rho$ evaluated in the variable environment ρ yields the integer upper bound of the array;
- A maps an index $i \in [[\![\mathtt{A.low}]\!]\rho, [\![\mathtt{A.high}]\!]\rho)$ 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 $\mathtt{A.low} = 0$. 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 $a = (\rho, \mathtt{A.low}, \mathtt{A.high}, A)$ where ρ is the concrete variable environment. It is a "buffer overrun" runtime error if $i < [\![\mathtt{A.low}]\!]\rho$ or $[\![\mathtt{A.high}]\!]\rho \leqslant i$, 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 $\mathtt{A.Length} > 1$. At program point 6 the final values of the scalar variables are given by ρ_6 such that $\rho_6(\mathtt{i}) = \rho_6(\mathtt{A.Length}) = n$ where $n > 1$ is the unknown array length. The final value of A is $a_6 = (\rho_6, 0, \mathtt{A.Length}, A_6)$ with $A_6(i) = (i, 0)$ for all $i \in [0, n)$. Because ρ_6, 0, and A.Length are easily understood from the context, we write $\mathtt{A[i]} = (i, 0)$ by abuse of notation where the value i of i is assumed to be in-bounds. □

In the analysis of the example of Fig. **2**, the pair $\mathtt{A}[i] = (i, v)$ 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 $\wp(\mathcal{A})$ (relations between array values are thus abstracted away).

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 $\mathtt{v}_0$ which value is assumed to be always zero so $\overline{\mathcal{X}} = \mathbb{X} \cup \{\mathtt{v}_0\}$ where $\mathtt{v}_0 \notin \mathbb{X}$. 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 $\overline{\rho} \in \overline{\mathcal{R}}$ are called abstract variable environments. The concretization $\gamma_v(\overline{\rho})$ denotes the set of concrete variable environments having this abstract property. It follows that $\gamma_v \in \overline{\mathcal{R}} \mapsto \wp(\mathcal{R}_v)$.

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 $\overline{\mathcal{R}} \triangleq \mathbb{X} \mapsto \overline{\mathcal{V}}$ where the abstract domain $\mathbf{V}$ abstracts properties of values in $\mathcal{V}$ with concretization $\gamma_v \in \overline{\mathcal{V}} \mapsto \wp(\mathcal{V})$.

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 $\overline{\mathcal{E}}$ consist in a set of symbolic expressions depending on the variables in $\overline{\mathcal{X}}$ restricted to a canonical normal form plus the bottom expression $\bot$ 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 `...A[B[i]]...` becomes `...{int x; x := B[i]; A[x]}...`). 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 $\overline{\mathcal{E}}$ are restricted to the normal form $\mathtt{v} + k$ where $\mathtt{v} \in \overline{\mathcal{X}}$ is an integer variable plus an integer constant $k \in \mathbb{Z}$ ($\mathtt{v}_0 + k$ represents the integer constant k). An alternative example of convenient normal form would be linear expressions $a.\mathtt{v} + b$ 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 $\gamma_v \in \overline{\mathcal{R}} \mapsto \wp(\mathbb{X} \mapsto \mathbb{Z})$, the concretization $\gamma_e(\mathtt{e})\overline{\rho}$ of an expression $\mathtt{e} \in \overline{\mathcal{E}}$ depends on the abstract value $\overline{\rho} \in \overline{\mathcal{R}}$ of the scalar variables in $\overline{\mathcal{X}}$ and is the set of possible concrete values of the expression. So $\gamma_e \in \overline{\mathcal{E}} \mapsto \overline{\mathcal{R}} \mapsto \wp(\mathcal{V})$ such that $\gamma_e(\bot)\overline{\rho} \triangleq \emptyset$, $\gamma_e(\top)\overline{\rho} \triangleq \mathcal{V}$, $\gamma_e(\mathtt{v}_0 + i)\overline{\rho} \triangleq \{i\}$, and otherwise $\gamma_e(\mathtt{v} + i)\overline{\rho} \triangleq \{\rho(\mathtt{v}) + i \mid \rho \in \gamma_v(\overline{\rho})\}$.

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 $\mathtt{v} + i = \mathtt{v'} + j$ is true if and only if $\mathtt{v} = \mathtt{v'}$ and $i = j$, false if $\mathtt{v} = \mathtt{v'}$ and $i \neq j$ and unknown otherwise. Similarly $\mathtt{v} + i < \mathtt{v'} + j$ is true if and only if $\mathtt{v} = \mathtt{v'}$ and $i < j$, false if $\mathtt{v} = \mathtt{v'}$ and $i \geqslant j$ and unknown otherwise. The comparison of i and $\mathtt{v}+j$ where $\mathtt{v} \neq \mathtt{v}_0$ always has an unknown result. This is very simple, rapid, but rather imprecise.

Variable comparisons An immediate refinement consists in using the abstract information $\overline{\rho} \in \overline{\mathcal{R}}$ 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, $\overline{\rho}(\mathtt{v}) = [a, b]$, and $\overline{\rho}(\mathtt{v'}) = [a', b']$. The comparison $\mathtt{v} + i < \mathtt{v'} + j$ is true when $\mathtt{v} = \mathtt{v'}$ and $i < j$ or $\mathtt{v} \neq \mathtt{v'}$ but (using the abstract variable environment) $b + i < a' + j$, false when $\mathtt{v} = \mathtt{v'}$ and $i \geqslant j$ or $\mathtt{v} \neq \mathtt{v'}$ but (using the abstract variable environment) $b' + j \leqslant a + j$ and unknown otherwise. Relational domains such as DBM [11] and Octagons [32] 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 $\mathtt{e}, \mathtt{e}' \in \overline{\mathcal{E}}$ 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 $\overline{\mathcal{E}}$ 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 $\{e^1_1 \ \ldots \ e^1_{m^1}\} \ \ldots \ \{e^2_1 \ \ldots \ e^2_{m^2}\}[?^2] \ \ldots \ \{e^n_1 \ \ldots \ e^n_{m^n}\}[?^n]$ maintains the information that $e^1_1 = \ldots = e^1_{m^1} \leqslant e^2_1 = \ldots = e^2_{m^2} \leqslant \ldots \leqslant e^n_1 = \ldots = e^n_{m^n}$ (where the i-th inequality is strict when $[?^{i+1}]$ is empty $\lrcorner$ 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 $\lrcorner$), 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 $\mathtt{v} + i, i \in \mathbb{Z}$ where constant expressions are represented by the distinguished variable $\mathtt{v}_0$ which value is assumed to always be zero, we can use Pratt's algorithm [33] to compare their symbolic values. A graph matrix is constructed with an edge $(\mathtt{v}, \mathtt{v'})$ labelled $i - j$ whenever $\mathtt{v}+i \leqslant \mathtt{v'}+j$ (respectively $i-j+1$ when $\mathtt{v}+i < \mathtt{v'}+j$) 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 [34] is used to derive all possible comparisons derived by repeated application of the transitivity of comparisons. A cycle $(\mathtt{v}, \mathtt{v})$ for a variable $\mathtt{v}$ in the transitive closure matrix means impossibility, that is unreachability in the concrete. A constraint $\mathtt{v}+k \leqslant \mathtt{v'}$ holds when the label of arc $(\mathtt{v}, \mathtt{v'})$ is less than or equal to k in the transitive closure matrix.

Another similar example is Shostak algorithm [36] for comparison of linear expressions of the form $a.\mathtt{v} + b.\mathtt{v'} \leqslant c$ 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 $\overline{\mathcal{B}} \triangleq \wp(\overline{\mathcal{E}} \setminus \{\bot, \top\})$. The empty set $\emptyset$ denotes unreachability while non-empty sets $\{e_1 \ \ldots \ e_m\}$ of expressions $e_1, \ldots, e_m \in \overline{\mathcal{E}}$ 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 $[\![e_1]\!]\rho = \ \ldots \ = [\![e_m]\!]\rho$. So $\gamma_b(\emptyset) = \emptyset$ and $\gamma_b(S) = \{\rho \mid \forall \mathtt{e}, \mathtt{e'} \in S : [\![\mathtt{e}]\!]\rho = [\![\mathtt{e'}]\!]\rho\}$ where $[\![\mathtt{e}]\!]\rho$ is the concrete value of expression $\mathtt{e}$ in the concrete environment ρ (for example $[\![\mathtt{v}_0 + c]\!]\rho \triangleq c$ and otherwise $[\![\mathtt{v} + c]\!]\rho = \rho(\mathtt{v}) + c$).

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 $\gamma_b(S)\overline{\rho} = \{\rho \in \gamma_v(\overline{\rho}) \mid \forall \mathtt{e}, \mathtt{e'} \in S : [\![\mathtt{e}]\!]\rho = [\![\mathtt{e'}]\!]\rho\}$.

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 there exists $e \in s$ and $e' \in s'$ such that $e \geq e'$. 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 $\wp(\mathbb{Z} \times \mathcal{V})$ may not or may 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. Therefore, 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 (Sect. **7.1**), the normal form $\mathbf{E}$ of symbolic expressions (Sect. **7.2**), hence that of the segment bounds $\mathbf{B}(\mathbf{E})$ (Sect. **8**), the abstraction $\mathbf{A}$ of the abstract array elements (Sect. **9**), as well as the various parameters of these abstract domains (such are the degree of refinement of expression comparison in $\mathbf{E}$ in Sect. **7.4**, hence of segment bounds comparison of $\mathbf{B}(\mathbf{E})$ (Sect. **8.2**) and the conversions (Sect. **10**).

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.* [16, Sec. 6.6.3], [25]. We will combine parity (where `_|_` (*i.e.* $\bot$) 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 [8] 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 $[\![(i,e)]\!]\overline{\rho} = (parity([\![e]\!]\overline{\rho}), interval([\![e]\!]\overline{\rho}))$).

```
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. □

Example 3 The second abstraction is the reduced cardinal power [8] of intervals by parity whose abstract properties have the form `(o -> `i_o`,e -> `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 $[\![(i,e)]\!]\overline{\rho}$ maps $parity([\![e]\!]\overline{\rho})$ to $interval([\![e]\!]\overline{\rho})$). For example `(o -> _|_,e -> [-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 i 10},
      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 in Ex. **1** using the reduced product since `(o -> _|_,e -> [-16,0])` is exactly `(e,[-16,0])` on array elements. □

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 $[\![(i,e)]\!]\overline{\rho}$ is a map of $parity([\![i]\!]\overline{\rho})$ to $interval([\![e]\!]\overline{\rho})$). 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. □

11.2 Abstract Predicates

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

$$\{e_1^1 \ldots e_{m^1}^1\}\, P_1\, \{e_1^2 \ldots e_{m^2}^2\}[?^2]\, P_2\, \ldots\, P_{n-1}\, \{e_1^n \ldots e_{m^n}^n\}[?^n]$$

where

- the segment bounds $\{e_1^i \ldots e_{m^i}^i\} \in \overline{\mathcal{B}}, i \in [1,n], n > 1$, are finite non-empty sets of symbolic expressions in normal form $e_j^i \in \overline{\mathcal{E}}$ as respectively considered in Sect. **8** and Sect. **7.2**;
- the $P_i \in \overline{\mathcal{A}}$ are abstract predicates chosen in an abstract domain $\mathbf{A}$ denoting possible values of pairs (index, indexed array element) in a segment of Sect. **9**; and

- the optional question mark $[?^i]$ follows the upper bound of a segment. Its presence `?` means that the segment might be empty. Its absence $\lrcorner$ 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 $\{e^1_1 \ \ldots \ e^1_{m^1}\}$ of the first segment never has a question mark. $(\{\lrcorner, ?\}, \preccurlyeq, \lrcorner, ?, \curlyvee, \curlywedge)$ is a complete lattice with $\lrcorner \prec ?$.

The symbolic expressions $e^k_i \in \overline{\mathcal{E}}$ in a given segment bound depend on scalar variables but not on array elements hence `A[A[i]]` should be handled as `x=A[i]; A[x]` so that the auxiliary variable `x` can appear in a segment bound for array `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 Sect. **6.3** can relate the array value `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}? T {n}?` both segments cannot be simultaneously empty), which we do not do for the sake of efficiency.

11.3 Concretization

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

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 ($<_\lrcorner$ stands for $<$ while $<?$ stands for $\leqslant$):

$$\begin{array}{l}\gamma'_s(B\ P\ B'\ [?])\overline{\rho} \quad \triangleq \\ \quad \{(\rho, \ell, h, A) \mid \rho \in \gamma_v(\overline{\rho}) \wedge \forall \mathtt{e}_1, \mathtt{e}_2 \in B : \forall \mathtt{e}'_1, \mathtt{e}'_2 \in B' : \\ \qquad [\![\ell]\!]\rho \leq [\![\mathtt{e}_1]\!]\rho = [\![\mathtt{e}_2]\!]\rho <[?]\ [\![\mathtt{e}'_1]\!]\rho = [\![\mathtt{e}'_2]\!]\rho \leq [\![h]\!]\rho \wedge \\ \qquad\quad \forall i \in \left[\ [\![\mathtt{e}_1]\!]\rho, [\![\mathtt{e}'_1]\!]\rho\right) : A(i) \in \gamma_a(P)\}\end{array}$$

The concretization of an array segmentation $B_1 P_1 B_2 [?^2] P_2 \ \ldots\ P_{n-1} B_n [?^n]$ is the set of arrays whose elements in all segments $[B_i, B_{i+1}), i = 1, \ldots, n-1$ satisfy abstract property P_i and whose lower and upper bounds are respectively given by B_1 and B_n.

$$\begin{array}{l}\gamma_s(B_1 P_1 B_2 [?^2] P_2 \ldots P_{n-1} B_n [?^n])\overline{\rho} \quad \triangleq \\ \quad \{(\rho, \ell, h, A) \in \bigcap_{i=1}^{n-1} \gamma'_s(B_i\ P_i\ B_{i+1} [?^{i+1}])\overline{\rho} \mid \\ \qquad \forall \mathtt{e}_1 \in B_1 : [\![\mathtt{e}_1]\!]\rho = [\![\ell]\!]\rho \wedge \forall \mathtt{e}_n \in B_n : [\![\mathtt{e}_n]\!]\rho = [\![h]\!]\rho\}\end{array}$$

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

11.4 Segmentation unification

Given two segmentations with *compatible* extremal segment bounds (in general for the same array), the objective of segmentation unification is to modify the two segmentations so that they coincide. By compatible we mean that the first (the last) segment bounds should have a non-empty intersection. In practice this is always the case as the first segment bound always contains `0` and the last segment bound always contains the symbolic name for the array length (*e.g.*, `A.Length`).

On the* best *unification The problem of segmentation unification admits a partially ordered set of solutions, in general not forming a lattice.

The minimal elements, hence the least precise unifications are those where all the segments are joined, and only the extremes are preserved.

Example 5 Let `a` and `b` be distinct variables. When unifying `{0}...{a b}` with `{0}...{a b c}`, both segmentations `{0}...{a}` and `{0}...{b}` are minimal solutions, but not comparable. □

The maximal elements, hence the most precise unifications are the coarsest common refinements of both segmentations.

Example 6 When unifying `{0}...{a}...{b}...{c}` with `{0}...{b}...{a}...{c}` both segmentations `{0}...{a}...{c}` and `{0}...{b}...{c}` are maximal , but not comparable solutions. □

In general, *a* solution is such that the bounds: (i) do appear in one or the other initial segmentation; and (ii) preserve the original orderings.

One segment can be empty in one segmentation (like `{0 i}`) and non-empty in the other one (like $\{0\}P\{1,i\}$). Therefore 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 $\bot$ for join, $\top$ for meet, $\bot$ 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:

Example 7 Assume we want to unify `{0 i-1}P1{i}` and `{0 i-2}P2{i}` (obviously in two different contexts). The coarsest common refinement is `{i-2}`$\bot$`{0}`$\bot$`{i-1}?P1{i}` for `{0, i-1}P1{i}` and `{i-2}`$\bot$`{0}?P2{i-1}P2{i}` for `{0,i-2}P2{i}` (which would yield the join `{i-2}`$\bot$`{0}?P2{i-1}?P1` $\cup$ `P2{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. □

Therefore, the well-definedness of the coarsest common refinement, if any, depends upon the abstract variable environment, too.

To sum up, we want the array segmentation analysis: (i) to have the possibility of being completely independent of the variable analysis (see Sect. **11.7**); (ii) to have a deterministic behavior in presence of several maximal common refinements. Therefore we present a segmentation unification which does not provide any guarantee on the maximality of the result, but instead one which: (i) is always well-defined in absence of knowledge of the contexts of the segmentations; (ii) does terminate; (iii) is deterministic.

The segmentation unification algorithm The first step of the algorithm is checking the compatibility of the two input segmentations to verify that they do have common lower and upper bounds.

Then, the unification proceeds recursively from left to right and maintains the invariant that the left part is already unified. We let $\bot_l$ (resp. $\bot_r$) denote the left (resp. right) neutral element.

1. $B[?_1]\ P_1\ B'_1[?'_1] \ldots$ and $B[?_2]\ P_2\ B'_2[?'_2] \ldots$ have same lower bounds and so keep the first segments as they are and go on with $B'_1[?'_1] \ldots$ and $B'_2[?'_2] \ldots$.
2. In case $(B \cup B_1)[?_1]\ P_1\ B'_1[?'_1] \ldots$ and $B[?_2]\ P_2\ B'_2[?'_2] \ldots$ with $B_1 \neq \emptyset$ and $B \cap B_1 = \emptyset$, let $\overline{B}_1$ be the set of expressions in B_1 appearing in the second segmentation blocks $B'_2, \ldots$.
   - 2.1 If $\overline{B}_1$ is empty then go on with $B[?_1]\ P_1\ B'_1[?'_1] \ldots$ and $B[?_2]\ P_2\ B'_2[?'_2] \ldots$ following case 1.
   - 2.2 Otherwise go on with $B[?_1]\ \bot_l\ B_1?\ P_1\ B'_1[?'_1] \ldots$ and $B[?_2]\ P_2\ B'_2[?'_2] \ldots$ as in case 1.
3. The symmetrical case is similar.
4. In case $(B \cup B_1)[?_1] P_1 B'_1[?'_1] \ldots$ and $(B \cup B_2)[?_2] P_2 B'_2[?'_2] \ldots$ with $B_1, B_2 \neq \emptyset$ and $B \cap B_1 = B \cap B_2 = \emptyset$, let $\overline{B}_1$ (resp.

$\overline{B}_2$) be the set of expressions in B_1 (resp. B_2) appearing in the second (resp. first) segmentation blocks $B'_2, \ldots$ ($B'_1, \ldots$).

4.1 If $\overline{B}_1$ and $\overline{B}_2$ are both empty, go on with $B[?_1]P_1B'_1[?'_1]\ldots$ and $B[?_2]\ P_2\ B'_2[?'_2]\ldots$ as in case 1.

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

4.3 The symmetrical case is similar.

4.4 Finally if $\overline{B}_1$ and $\overline{B}_2$ are both non-empty then go on with $B[?_1]\bot_l\overline{B}_1?\,P_1\,B'_1[?'_1]\ldots$ and $B[?_2]\bot_r\overline{B}_2?\,P_2\,B'_2[?'_2]\ldots$ as in case 1.

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

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 $(B_1 \cup B'_1 \cup B_2)[?_2]$ □

Example 8 In the analysis of the example of Fig. **2**, we have to unify `{0 i} T {n}` and `{0 i-1} 0 {1 i} T {n}?` which becomes `{0}` $\bot$ `{i}? T {n}` and `{0} 0 {1 i} T {n}?` by 4.3 and we go on with `{i}? T {n}` and `{1 i} T {n}?` which, by the symmetric in 3 of 2.1 becomes `{i}? T {n}` and `{i} T {n}?` so we go on with `{n}` and `{n}?` which terminates the recursion by 6, thus returning `{0}`$\bot$`{i}? T {n}` and `{0} 0 {i} T {n}?`. Their array segmentation join is then `{0} 0 {i}? T {n}?` (taking the disjunction $\curlyvee$ of potential segment 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. The algorithm has a look-ahead of 1 (cf. case 5). It can be easily refined to provide a larger look-ahead at a price of an increased complexity. However, an advantage of the algorithm is that its behavior and output are deterministic. For instance, in the Ex. **6** our algorithm returns the non-maximal segmentation `{0}...{c}`. A 2-look-ahead algorithm should make the choice between the two maximal solutions.

In general, the algorithm can be easily adapted and refined to take into account specific knowledge when comparing segment bounds (for instance the total order induced by constants Sect. **12.2**).

11.5 Partial order/join/meet/widening/narrowing

For an array segmentation join $\mathbf{S}.\sqcup$, a $(\bot,\bot)$-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 $(\bot,\bot)$-segmentation unification is performed and then a segmentwise widening $\mathbf{A}.\nabla$. Moreover, the widening merges consecutive segments with same abstract value. Widenings 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 $(\bot,\top)$-segmentation unification is performed before returning the conjunction of the segmentwise comparisons $\mathbf{A}.\ \sqsubseteq$. The potential segment emptiness indications must also be taken into account, that is $\lrcorner = \lrcorner \prec ? = ?$.

11.6 Abstract Transfer Functions

Abstract value of an indexed array element Assume that we have to evaluate $[\![\texttt{A[e]}]\!]\overline{\rho}$ for the array `A` abstracted by the segmentation $B_1P_1B_2[?^2]P_2\ldots P_{n-1}B_n[?^n]$. The expression $B_1 \leqslant \texttt{e} \leqslant B_n$ 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 $B_\ell \leqslant \texttt{e}$ is true (B_1 otherwise) and B_h be the smallest segment bound such that $\texttt{e} < B_h$ is true (B_n otherwise, assuming that execution goes on only in absence of buffer overrun). The value of $[\![\texttt{A[e]}]\!]\overline{\rho}$ is then $\bigsqcup_{k=\ell}^{h-1} P_k$ where $\sqcup$ 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 `A[e] = e'` with abstract variable environment $\overline{\rho}$ where the array `A` is abstracted by the segmentation $B_1P_1B_2[?^2]P_2\ldots P_{n-1}B_n[?^n]$, we first determine the range of segments such that $B_\ell \leqslant \texttt{e} < B_h$ is definitely true. The segmentation of `A` can be thought of as being abstracted to $B_1P_1\ldots B_\ell[?^\ell](\bigsqcup_{k=\ell}^{h-1} P_k)B_h[?']P_h\ldots P_{n-1}B_n[?^n]$ where $[?']$ is ? if all the $[?^{\ell+1}],\ldots,[?^h]$ are ? (so that the block $B_\ell\ldots B_h$ can then be empty) and $\lrcorner$ 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 $B_\ell\ldots B_h$ (may be at its borders). This segment is split. Let $P \in \overline{\mathcal{A}}$ be the abstraction of the value of the pair `(e,e')` in $\mathbf{A}$. After the array element assignment, the array segmentation of `A` becomes $B_1P_1\ldots B_\ell[?^\ell](\bigsqcup_{k=\ell}^{h-1} P_k)\{\texttt{e}\}[?_l]P\{\texttt{e}+1\}(\bigsqcup_{k=\ell}^{h-1} P_k)B_h[?_r]P_h\ldots P_{n-1}B_n[?^n]$. $[?_l]$ is ? unless the segment bounds comparison discussed in Sect. **8.2** can determine that $B_\ell < \{\texttt{e}\}$ is always true. Similarly, $[?_r]$ is $\lrcorner$ when $\{\texttt{e}+1\} < B_h$ 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 $B_1P_1\ldots B_\ell[?^\ell](P\sqcup\bigsqcup_{k=\ell}^{h-1} P_k)$ $B_h[?']P_h\ldots P_{n-1}B_n[?^n]$ in the worst case. This is the case of the assignment `(*)` in Fig. **1**. If the segment bounds comparison can determine that $B_\ell = \{\texttt{e}\}$ we get $B_1P_1\ldots(B_\ell\cup\{\texttt{e}\})[?_l]P\{\texttt{e}+1\}(\bigsqcup_{k=\ell}^{h-1} P_k)B_h[?_r]P_h\ldots P_{n-1}B_n[?^n]$. Similarly if $\{\texttt{e}+1\} = B_h$ for sure, we get $B_1P_1\ldots B_\ell[?^\ell](\bigsqcup_{k=\ell}^{h-1} P_k)\{\texttt{e}\}[?_l]P(\{\texttt{e}+1\}\cup B_h)P_h\ldots P_{n-1}B_n[?^n]$.

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 $\mathbf{C}$.

Assumption for the content of an array For a statement in the form of `assume` $\forall i \in [a,b).\ c($`A[e]`$)$ (assuming $a \leq b$ are within the array bounds): (i) we infer an abstract segment value as in the previous case (let us call it `P`); (ii) we abstract the bounds a, b in the quantification to their best possible approximation `a, b` in the bounds abstract domain; (iii) we materialize the most generic abstract segmentation

```
A: {0} T {a}? P {b}? T {A.Length}?
```

and we use the scalar abstract domain to get rid of some of the uncertainties. Finally the so-obtained abstract predicate is intersected

(using the meet operation) with the abstract value in the pre-state of A. For example, using the abstract domain of intervals:

```
/* (A: {0} T {A.Length}, k:[10,10], A.Length:[10, +oo]) */
assume forall i: [0, k) => A[i] >= 0;
/* (A: {0} [0, +oo] {k} T {A.Length}?, ...) */
```

In fact, the materialized abstract segment is

```
A: {0} T {0}? [0, +oo] {k}? T {A.Length}?,
```

which can be reduced using the abstract domain of scalars to:

```
A: {0} [0, +oo] {k} T {A.Length}?
```

The constant zero is trivially equal to itself and k is positive but it may be equal to the length of the array A. Finally the value in the pre-state of A is refined via the meet operation.

Invertible assignment to a scalar variable In an invertible assignment to a scalar variable $\mathtt{x} = \mathtt{f}(\mathtt{x}, \vec{\mathtt{y}})$ where $\vec{\mathtt{y}} = \mathtt{y}_1, \ldots, \mathtt{y}_m$, 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 $\vec{\mathtt{y}}$ 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 $f^{-1}(\mathtt{x}, \vec{\mathtt{y}})$ 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 $\mathtt{x} = f(\mathtt{x}, \vec{\mathtt{y}})$, no inverse $x_{\text{old}} = f^{-1}(x_{\text{new}}, \vec{y})$ is available, for example in $\mathtt{x} = f(\vec{\mathtt{y}})$. For such a non invertible assignment $\mathtt{x} = \mathtt{e}$, 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 $\mathtt{e}'$ such that $[\![\mathtt{e}' == \mathtt{e}]\!]\overline{\rho}$ is definitely true in the abstract. In the simple purely syntactic case of Sect. **7.4**, $[\![\mathtt{e}' == \mathtt{e}]\!]\overline{\rho} = \mathit{true}$ is under-approximated by $\mathtt{e}' = \mathtt{e}$ 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 $\mathtt{x} = f(\vec{\mathtt{y}})$, $\mathtt{x} \notin \vec{\mathtt{y}}$ where $y_i = f_i^{-1}(x_{\text{new}}, y_1, \ldots, y_{i-1}, y_{i+1}, \ldots, y_m)$ is available, the expression $\mathtt{e}(f_i^{-1}(x_{\text{new}}, y_1, \ldots, y_{i-1}, y_{i+1}, \ldots, y_m))$ can be added to all segment bounds containing an expression $\mathtt{e}(\mathtt{y}_i)$ 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 $\mathtt{e} = B_i$[(2)] is always true will add e and e' to B_i. Moreover, if $\mathtt{e}' = B_j$ for sure and $i < j$ then the segmentation $B_1 P_1 B_2 [?^2] P_2 \ldots P_{n-1} B_n [?^n]$ will be reduced to $B_1 P_1 B_2 [?^2] P_2 \ldots P_{i-1} (B_i \cup B_{i+1} \cup \ldots \cup B_j \cup \{\mathtt{e}, \mathtt{e}'\}) [?^i] P_j \ldots P_{n-1} B_n [?^n]$ or to unreachability when one of

(2) Recall that $\mathtt{e} = B$ is $\exists \mathtt{e}' \in B : \mathtt{e} = \mathtt{e}'$.

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

Figure 5. Example of a backwards initialization. Array segmentation reduction is needed to prove the postcondition $\forall j \in [0, \mathtt{A.Length}).\ \mathtt{A}[j] = 0$.

the $[?^{i+1}], \ldots, [?^j]$ is $\sqcup$ (since then e < e'). The comparison e ⩽ 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 $\mathtt{e} = B_i$ and $\mathtt{e}' = B_{i+1}$ can be used to remove a doubt ? on the possible emptiness of a segment $B_i P_i B_{i+1}?$ which becomes $B_i P_i B_{i+1}$.

11.7 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 hypothesis on the static analyzer. The consequence is that the result may not be as precise as possible. Let us illustrate this phenomenon on the program of Fig. **5**.

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 [8]. 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 $\ldots B[?]\ P\ B'? \ldots$ can be reduced to a non-empty one $\ldots B[?]\ P\ B' \ldots$ if the scalar variables environment $\overline{\rho}$ implies $\exists \mathtt{e} \in B : \exists \mathtt{e}' \in B' : [\![\mathtt{e}]\!]\overline{\rho} < [\![\mathtt{e}']\!]\overline{\rho}$ is always *true* in the abstract (the abstract test returning either $\bot$, *true*, *false*, or *unknown*). Similarly, a possibly empty segment $\ldots B[?] P B'? \ldots$ may be definitely empty and reduced to the bound $\ldots (B \cup B')[?] \ldots$ when $\exists \mathtt{e} \in B : \exists \mathtt{e}' \in B' : [\![\mathtt{e}]\!]\overline{\rho} = [\![\mathtt{e}']\!]\overline{\rho} = \mathit{true}$.

In the reduction example of Fig. **5**, the fact that $\mathtt{i} \in [0, 0]$ 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. Implementation

12.1 CodeContracts and Clousot

CodeContracts allow the language-agnostic specification of contracts (preconditions, postconditions and object-invariants [3, 31]).

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. `Clousot` is used both inside and outside Microsoft on large production projects. 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 ...) [27]. 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. The heap analysis makes some assumptions on parameter aliasing, we refer the interested reader to [14]. On the top of the program scalar form several forward value analyses are run, and their results are used to discharge the assertions. Assertions are discharged using simple built-in decision procedures. 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 a main 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.

Functor instantiation 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) the scalar numerical abstract domain is the one provided by `Clousot` in the particular run. Several numerical abstract domains are available in `Clousot`, among them: Pentagons [28] combined with Linear equalities [21], Subpolyhedra [24].

Clousot numerical domains are composed according to a tree topology [9]. Reduction is achieved via pushing or pulling of information. An abstract domain can *push* information to abstract domains of lower rank, and *pull* information from all other domains. The segmentation abstract domain is at the root of the tree. It pushes: bottom (contradiction), expression equality (when the two expressions appears in the same bound), array values (*e.g.*, in `x = a[i]`, if it determines that `a[i]!= null`, then it pushes the information `x != null`). It pulls: comparisons (`exp1 < exp2`, `exp1!= exp2`, `exp1 == exp2` ...), integer constants (all the variables which are known to be definitely constants), intervals (e.g. in `x = a[exp]`, which is the range for `exp`?), abstract values (*e.g.* in `a[i] = z`, which is the abstract value for `z`?).

The array analysis is currently implemented as a value analysis run *on the top of* the heap analysis, when in particular array aliasing has been resolved. As a consequence, the abstract transfer functions implemented in the analyzer are very similar to those described in Sect. **11**, with some adjustements to meet the peculiarietes of the `.NET` semantics and `Clousot` infrastructure.

Array creation When an array is created with the instruction `newarr A exp` then the segmentation

```
{0} d { A.Length, exp }?
```

is materialized, where `d` denotes the *abstraction* in the segment value abstract domain of the default value for the type of the array elements (*e.g.*, 0 is the default value for `int` and `null` is the default value for reference types). The so-materialized segmentation is then refined pulling some information from the scalar variables abstract domain. In the particular case, `Clousot` first asks the numerical abstract domain if $\texttt{exp} > 0$. If the answer is *true* then the uncertainity `?` is dropped. If the answer is *false*, then it means that $\texttt{exp} == 0$ (as if $\texttt{exp} < 0$, then a buffer overrun occurs, causing the concrete execution to stop, and having some other `Clousot` analysis reporting the bug). Therefore the *empty* segmentation is returned. Otherwise, the original segmentation could not be refined, and it is returned as it is.

Assertions `Clousot` implements a very simple and specialized decision procedure to check whether an quantified assertion over arrays holds at a given program point. Given the statement $\texttt{assert}\ \forall i \in [\texttt{e1}, \texttt{e2}).\ c(\texttt{A}[i])$, `Clousot` uses the numerical abstract domain to provide a lower bound l for `e1` and an upper bound u for `e2` to be used to determine an upper-approximation v for the values of $\texttt{A}[i]$ in the range $[l, u)$. The expression $c(v)$, which does not contain any reference to array values, is then constructed and its truth is decided by the internal decision procedures commonly used in `Clousot`.

Example 9 Let us consider the example in Fig. **2**, and let us suppose that at the end of the method there was the statement $\texttt{assert}\ \forall i \in [0, \texttt{A.Length}).\ \texttt{A}[i] == 0$. `Clousot` infers the limits to be 0 and `A.Length`, and it uses the abstract state `p6` to determine that in such a range $v = \texttt{A}[i] = 0$. The consequent proof obligation $0 == 0$ is then trivially discharged. □

Function calls Function calls are not-inlined as `Clousot` relies on their contract instead. So they are turned into two instructions, an `assert` for the precondition, and an `assume` for the postcondition.

Segmentation Unification In the implementation we slightly refined the algorithm of Sect. **11.4** to cope with some *noise* produced by the bytecode representation and heap analysis. In general, in a segmentation done at bytecode level one will find many more extra-variables than those that one may expect, in particular when reasoning at the equivalent source code.

The first modification is the introduction of a *purification* step, which pre-processes the segmentations to remove those segment bounds (and hence segments) containing variables appearing only in one of the two segmentations. Of course those would have disappeared anyway with the original algorithm, but the *purification* step

has the advantage of making the algorithm faster and more precise, as shown by the next example.

Example 10 Suppose we need to unify the segmentation `{0} ... {a} ...{b}...{A.Length}` with `{0}... {b}... {A.Length}`. The purification step removes `{a}` from the first segmentation, enabling the segmentation unification algorithm to produce a more precise result. Without purification, we would have needed a deeper look-ahead for the algorithm to prevent the abstraction of the bound `{b}`. □

The second modification is a refinement of the step 5 to take into account constants: If B_1 and B_2 contain a constant then we can use this information to materialize new bounds, as illustrated by the next example.

Example 11 Suppose we need to unify `{0}...{25}...{26} ...{A.Length}` with `{0}... {20}... {30}... {A.Length}`. The result of the unification is `{0}...{20}...{25} ...{26} ...{30} ...{A.Length}`. □

13. Experimental Evaluation

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#).

13.1 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. For instance, this is how we picked the example in Fig. **1**. 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.* [29]) 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). We conclude that an array analysis technique in order to be effective should handle very efficiently the above cases.

At the beginning of this project, our first attempt was to implement the technique of [19] 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 [19, Sect. 5]). More generally, the problem of [17, 19] 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 must then 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 libraries are not annotated with contracts, so we cannot report on it. In the experimental setting `Clousot` is run in its off-the-shelf configuration except for the iterative refinement which is switched off (as it is not needed for the particular experience). 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. More interestingly, we did not encountered 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 design, 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. arr.	Δ	# inv
`mscorlib.dll`	21 475	4:06	4:15	0:09	2 430
`System.dll`	15 489	3:40	3:46	0:06	1 385
`System.data.dll`	12 408	4:49	4:55	0:06	1 325
`System.Drawings.dll`	3 123	0:28	0:29	0:01	289
`System.Web.dll`	23 647	4:56	5:02	0:06	840
`System.Xml.dll`	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%.

13.2 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`. Once again we run `Clousot` in the off-of-the-shelf configuration (and iterative refinement on). 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 slow-down 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` implementa-

tion (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 `Clousot`, 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 negligeable cost.

14. Conclusions

Our main goal was to have a non-intrusive, precise and scalable static analysis for array contents. We achieved it through these core ideas: (i) 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); (ii) to exploit segment unification for partial order, joins, meets, widenings, and narrowings; (iii) to carefully treat disjunction (via possibly empty segments, symbolic bounds with different instances, and relation of array values to indexes). 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 [17, 19] 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 $\mathtt{x}_i, \mathtt{x}_j, \ldots$ to denote the values of elements indexed $i \leqslant j \leqslant \ldots$ within the segment and relating them in the scalar environment. This would handle sorting algorithms [6]. 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 `Clousot`.

Acknowledgments We would like to thank Manuel Fähndrich for the insightful discussions and the support provided with the implementation of the decompilation of `ForAll` in `Clousot`. Work partly supported by the CMACS NSF Expeditions in Computing.

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, 2010.

[4] D. Beyer, T. A. Henzinger, R. Majumdar, and A. Rybalchenko. Path invariants. *PLDI'07*, 300–309. ACM, 2007.

[5] P. Chalin, J. Kinirya, G. Leavens, and E. Poll. Beyond assertions: Advanced specification and verification with JML and ESC/Java2. *FMCO'05*, LNCS 4111, 77–101. Springer, 2006.

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

[7] 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, 1977.

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

[9] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Combination of abstractions in the Astrée static analyzer. *ASIAN*, LNCS 4435, 272–300. Springer, 2006.

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

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

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

[13] I. Dillig, T. Dillig, and A. Aiken. Precise reasoning for programs using containers. *37th POPL*. ACM, 2011.

[14] M. Fähndrich and F. Logozzo. Static contract checking with abstract interpretation. *FoVeOOS'10*, LNCS. Springer, 2010.

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

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

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

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

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

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

[21] M. Karr. Affine relationships among variables of a program. *Acta Inf.*, 6:133–151, 1976.

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

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

[24] V. Laviron and F. Logozzo. Subpolyhedra: A (more) scalable approach to infer linear inequalities. *VMCAI*, LNCS 5403, 229–244. Springer, 2009.

[25] 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.

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

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

[28] F. Logozzo and M. Fähndrich. Pentagons: a weakly relational abstract domain for the efficient validation of array accesses. *SAC*, 184–188. ACM, 2008.

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

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

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

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

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

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

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

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

[37] H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. W. O'Hearn. Scalable shape analysis for systems code. *CAV'98*, LNCS 5123, 385–398. Springer, 2008.