Array content static analysis by segmentation

Patrick Cousot & Radhia Cousot
NYU & ENS CNRS & ENS
joint work with Francesco Logozzo (MSR)

Abstract

We present a parametric segmentation abstract functor for fully automatic and scalable inference of array content properties. The main idea is to automatically divide arrays into consecutive non-overlapping possibly empty segments whose content is abstracted uniformly. The array segmentation is automatically and semantically inferred during the static analysis depending on the way array elements are modified and accessed. The segment bounds are represented by symbolic expressions. The analysis of the segment element properties or the relation between the index and array value in segments is a parameter of the functor so as to tune the cost/ratio of the analysis.

A prototype analyzer has been implemented to adjust the algorithms and obtain the appropriate precision/cost ratio before implementing the analysis in a professional static analyzer for object-oriented languages used in an industrial context. This has shown the analysis to scale up with satisfactory precision and cost contrary to previous attempts which did not scale properly or required more user interaction than can be sustained in a typical engineering project.

Motivation

The problem of array content analysis

- Statically and fully automatically determine properties of array elements in finite reasonable time
- Undecidable problem $\rightarrowtail$ abstract interpretation
- Example:

```
                int n = 10;
                int i, A[n];
                i = 0;
/* 1: */
                while /* 2: */ (i < n) {
/* 3: */
                    A[i] = 0;
/* 4: */
                    i = i + 1;
/* 5: */
                }
/* 6: */
```

$\longleftarrow \forall i \in [0\text{-}n): A[i] = 0$

Contribution

- A new simple parametric array segmentation abstract domain functor
- An evaluation prototype for experimentation + an implementation in Clousot [1] by Francesco Logozzo
- Example:

```
                int n = 10;
                int i, A[n];
                i = 0;
/* 1: */
                while /* 2: */ (i < n) {
/* 3: */
                   A[i] = 0;
/* 4: */
                   i = i + 1;
/* 5: */
                }
/* 6: */

p6 = <A: {0} [0,0] {n,10,i}>; [ i: [10,10] n: [10,10] ]
0.000713 s
```

[1] This version of Clousot should be available shortly on DevLabs.

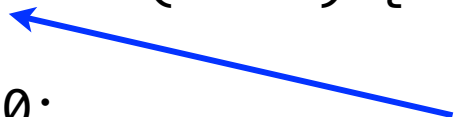
Self-imposed constraints for solving the array content analysis problem

- A basic abstraction usable in compilers and general purpose static analyzers
- A bit like *intervals* for numerical values which
  - is simple to implement
  - has low analysis cost and so does scale up
  - answers 60 to 95% of questions e.g. in compilers
- Parametrizable (to reuse existing abstractions)
- Fully automatic (no hidden hypotheses or dependence on other verification/proof systems)

The array segmentation abstraction

Which kind of invariants do we need?

```
            int n = 10;
            int i, A[n];
            i = 0;
/* 1: */
            while /* 2: */ (i < n) {
/* 3: */
                A[i] = 0;
/* 4: */
                i = i + 1;
/* 5: */
            }
/* 6: */
```



Invariant:

if `i = 0;` then
    array `A` not initialized
else if `i > 0` then
    A[0] = ... = A[`i`-1] = 0
else (* `i < 0` *)
    Impossible

Which kind of invariants do we need?

```
            int n = 10;
            int i, A[n];
            i = 0;
/* 1: */
            while /* 2: */ (i < n) {
/* 3: */
                A[i] = 0;
/* 4: */
                i = i + 1;
/* 5: */
            }
/* 6: */
```

Invariant:

if `i = 0;` then

array `A` not initialized

else if `i > 0` then

A[0] = ... = A[`i`-1] = 0

else (* `i < 0` *)

Impossible

Disjunction (case analysis)

Which kind of invariants do we need?

```
            int n = 10;
            int i, A[n];
            i = 0;
/* 1: */
            while /* 2: */ (i < n) {
/* 3: */
                A[i] = 0;
/* 4: */
                i = i + 1;
/* 5: */
            }
/* 6: */
```

Invariant:

if `i = 0;` then

array `A` not initialized

else if `i > 0` then

A[0] = ... = A[i-1] = 0

else (* `i < 0` *)

Impossible

Disjunction (case analysis)

Array segment

Which kind of invariants do we need?

```
            int n = 10;
            int i, A[n];
            i = 0;
/* 1: */
            while /* 2: */ (i < n) {
/* 3: */
                A[i] = 0;
/* 4: */
                i = i + 1;
/* 5: */
            }
/* 6: */
```

Invariant:

if i = 0; then
  array A not initialized
else if i > 0 then
  A[0] = ... = A[i-1] = 0
else (* i < 0 *)
  Impossible

Disjunction (case analysis)

Array segment

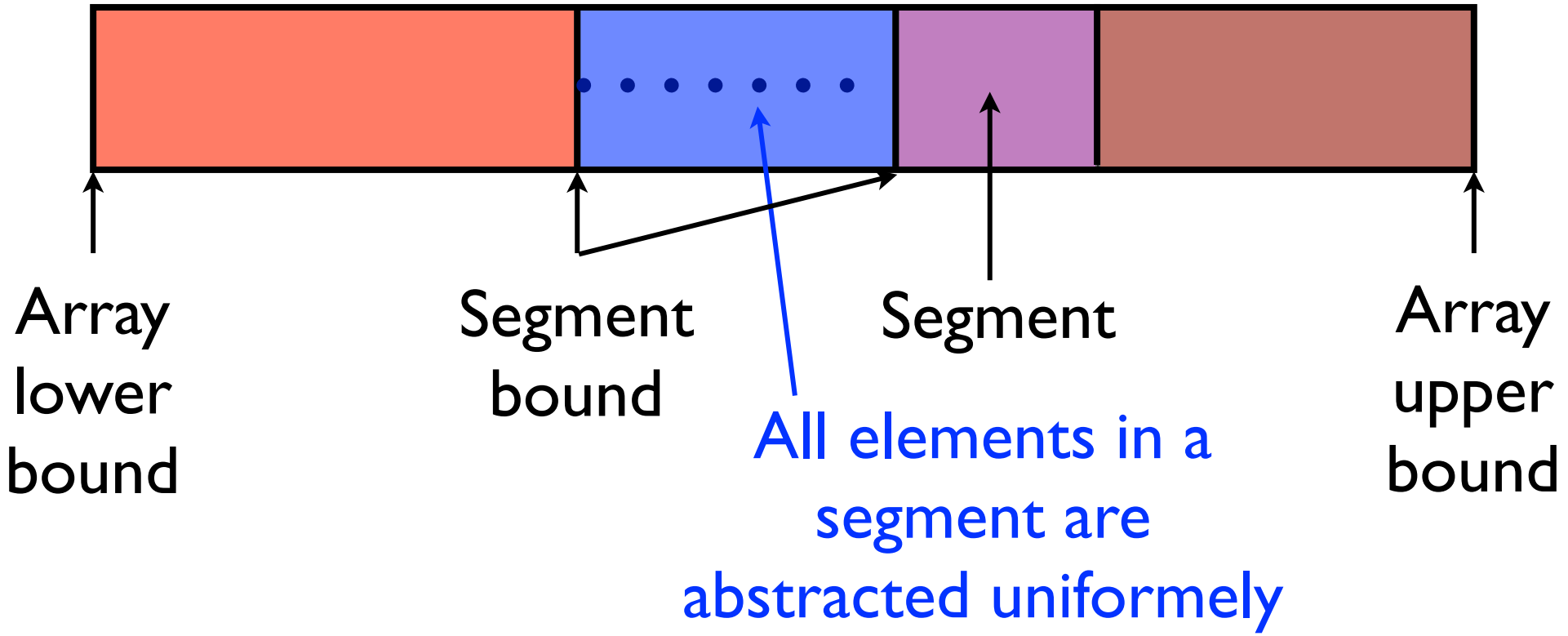
Segment bounds related to variables

The array segmentation abstract domain functor: abstract properties

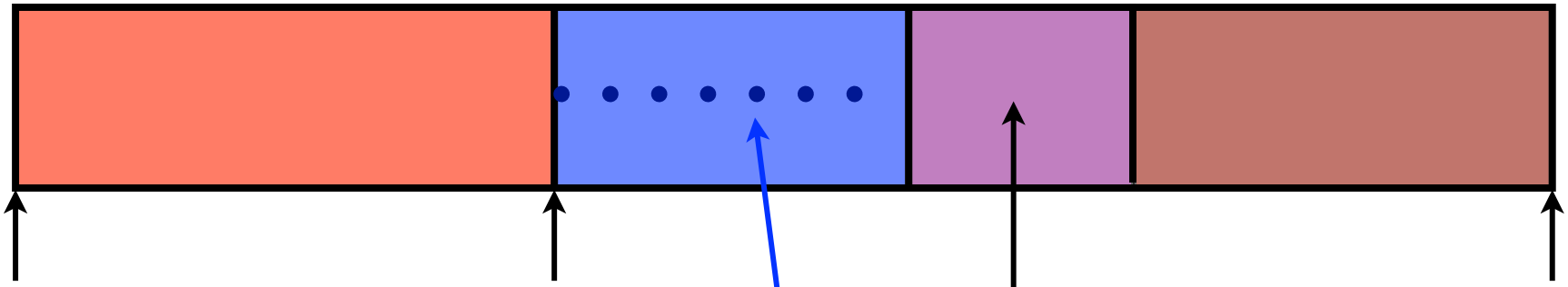
Array segmentation

- Classical array abstractions elementwise or

- Refinement by segments



Array segmentation



Array lower bound

Segment bound

Segment

Array upper bound

All elements in a segment are abstracted uniformely

<{0} [0,0] {i+1} [2,+oo] {10,n}>

-oo is min_int +oo is max_int

Symbolic array segment bounds

- Array segments are
  - in strict increasing order of the array indices
  - delimited by sets of expressions known to have equal values

<{0} [0,1] {i-1} [2,5] {i} [6,+oo] {n,10}>

so 0 < i-1 < i < n = 10

Symbolic array segment bounds

- Refinement of the segmentation: through assignments to array elements
- Coarsening of the segmentation: through widening
- Purely symbolic (variables abstract values are not strictly necessary to handle segment limits so works for all value abstractions!)

```
          int n = 10;
          int i, A[n];
          i = n;
/* 1: */
          while /* 2: */ (0 < i) {
/* 3: */
             i = i - 1;
/* 4: */
             A[i] = 0;
/* 5: */
          }
/* 6: */
Analysis with (arrays: interval domain x variables: top domain):
p6 = [ A: <{0} [-oo,+oo] {n,10}?> ] [ i: T n: T ]
0.000212 s
```

Top abstraction of simple variables and expressions

The explanation of this question mark ? is forthcoming

Symbolic array segment bounds (cont'd)

- symbolic not numerical so handles arrays of unknown size

```
          parameter int n; /* assume n>1 */
          int i  A[n];
          i = n;
/* 1: */
          while /* 2: */ (0 < i) {
/* 3: */
              i = i - 1;
/* 4: */
              A[i] = 0;
/* 5: */
          }
/* 6: */
```

Array of fixed but unknown size

```
Analysis with widening/narrowing and (arrays: interval domain x variables:
interval domain):
p6 = [ A: <{0,i} [0,0] {n}> ] [ i: [0,0] n: [2,+oo] ]
0.001854 s
```

Todo: should work with Javascript arrays (& iterators) with -∞ +∞ bounds and segments with float limits (?).

The semantics of arrays

- The classical operational semantics (John McCarthy):

  Array $\in$ Set of indices $\rightarrow$ Set of values

- Our semantics for segmentation:

  Array $\in$ Values of variables $\rightarrow$ Set of indices $\rightarrow$ Set of values

John McCarthy: Recursive Functions of Symbolic Expressions and Their Computation by Machine Part I. C ACM 3(4): 184-195 (1960)

The semantics of arrays (revisited I)

- The classical operational semantics (John McCarthy):

  Array $\in$ Set of indices $\rightarrow$ Set of values

- Our semantics for segmentation:

  Array $\in$ Values of variables $\rightarrow$ Set of indices $\rightarrow$ Set of values

John McCarthy: Recursive Functions of Symbolic Expressions and Their Computation by Machine Part I. C ACM 3(4): 184-195 (1960)

The semantics of arrays (revisited II)

- The classical operational semantics (J. McCarthy):

  Array $\in$ Set of indices $\rightarrow$ Set of values

- Our semantics for relational segmentation:

  Array $\in$ Values of variables $\rightarrow$ Set of indices
  $\rightarrow$ Set of (index x values)

Segments

Relation between indexes and values per segment

John McCarthy: Recursive Functions of Symbolic Expressions and Their Computation by Machine Part I. C ACM 3(4): 184-195 (1960)

Disjunctions

- Disjunctions are needed (as shown by the array initialization example)
- Disjunctive enumeration of cases leads to combinatorial explosion (e.g. because of conditionals and/or loops)
- Abstract interpretation offers a *standard solution* through over-approximation (preserves soundness but not completeness)
- A simple & cheap join is needed for any efficient array content analysis abstract domain (can over-approximate the lub/disjunction)

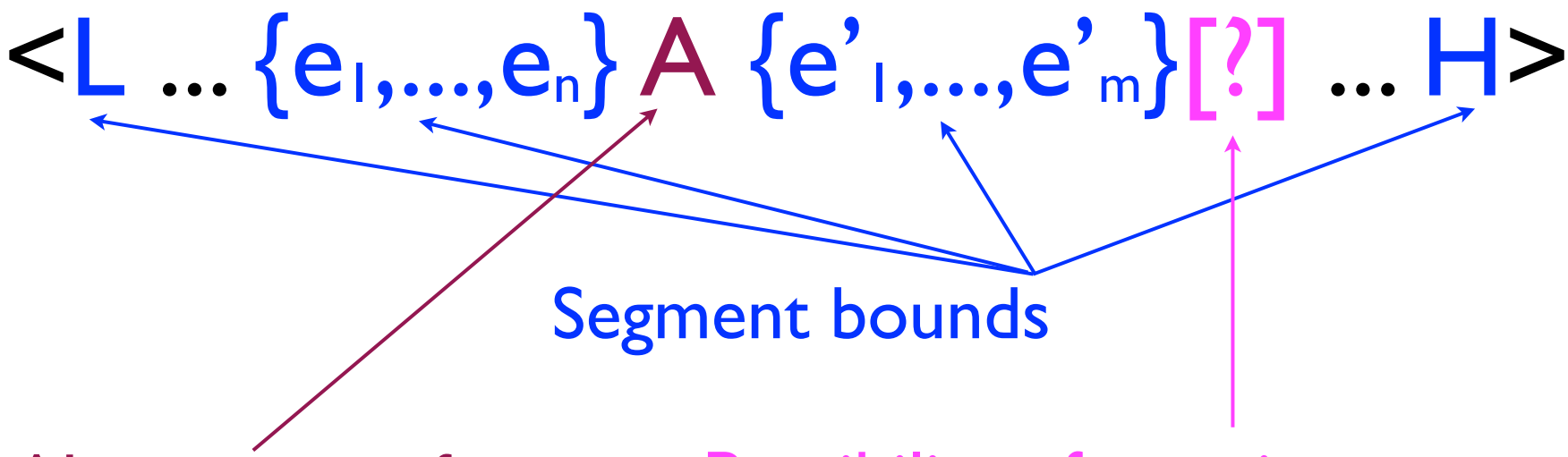
A very simple solution for disjunction: possibly empty segments

- Disjunctions are introduced exclusively through possibly empty segments

<{0} [0,0] {i}? [-oo,+oo] {n,10}?>

if `i = 0;` then
block is empty (so array `A` is not initialized)
else if `i > 0` then
A[0] = ... = A[`i`-1] = 0
else (* `i < 0` *)
Impossible

The array segmentation abstract domain



Abstraction of array element pairs (i v_i) within the segment

Possibility of emptiness:

- $e_1=...=e_n<e'_1=...=e'_m \longrightarrow$ —
- $e_1=...=e_n \leq e'_1=...=e'_m \longrightarrow$?

Parametrization of the array segmentation abstract domain functor

- Which symbolic expressions are used in block bounds?
- Which array abstraction is used to abstract array element values (i, v_i) within a segment?
- Which variables abstraction is used to abstract variables appearing in expressions?
- Which reductions are performed between symbolic block limits and abstractions of variables?
- Which coarseness is chosen for widenings/ narrowings?

The ARRAYAL prototype

- Symbolic expressions :
  - constant
  - variable ± constant

Could be more expressive but very simple solver for

$e = < \leq e'$ *!*

- Array abstraction and variables abstraction choice of
  - top
  - constant
  - parity
  - intervals
  - reduced product[(*)] (parity x intervals)
  - reduced cardinal power[(*)] of intervals by parity

Could be functors!

- 5699 lines of Ocaml (+6481 for unit tests)

Note: ARRAYAL is an abstract domain functor not a static analyzer so the abstract equations for programs of this talk have been established by hand (for lack of time for the equation generator).

(*) Patrick Cousot and Radhia Cousot: Systematic Design of Program Analysis Frameworks. POPL 1979: pp. 269-282.

The importance of parametrization

- The array segmentation abstract domain will work in any analysis context since no other information is necessary on simple variables (but for aliasing) although it can is exploited if available
- The segmentation and ordering information is inferred during the analysis (not given by the user/ or another (pre-)analysis)
- The cost/precision can be balanced by
  - appropriate abstraction of array *element and variable values*
  - degree of precision of *reductions*
- No need for any other external component

Example of reduction of array segments bounds by the variable values abstraction

```
          parameter int n; /* assume n>1 */
          int i, A[n];
          i = n;
/* 1: */
          while /* 2: */ (0 < i) {
/* 3: */
              i = i - 1;
/* 4: */
              A[i] = 0;
/* 5: */
          }
/* 6: */

Analysis with widening/narrowing and (arrays: interval domain x variables:
interval domain):

Segmentation reduction ('?' elimination)? (y/n): no
p6 = [ A: <{0} [-oo,+oo] {i}? [0,0] {n}?> ] [ i: [0,0] n: [2,+oo] ]

Segmentation reduction ('?' elimination)? (y/n): yes
p6 = [ A: <{0,i} [0,0] {n}> ] [ i: [0,0] n: [2,+oo] ]
0.001832 s
```

The fact that i=0 is not taken into account

Here it is!

A segmentation analysis example

A detailed example

```
          int n = 10;
          int i, A[n];
          i = 0;
/* 1: */
          while /* 2: */ (i < n) {
/* 3: */
             A[i] = 0;
/* 4: */
             i = i + 1;
/* 5: */
          }
/* 6: */
```

```
p1 = A[n][n=10;i=0] = <{0,i} [-oo,+oo] {n,10}>; [ i: [0,0] n: [10,10] ]
p2 = ... = p5 = p6  = <>; [ i: _|_ n: _|_ ]
```

A detailed example (cont'd)

```
          int n = 10;
          int i, A[n];
          i = 0;
/* 1: */
          while /* 2: */ (i < n) {
/* 3: */
             A[i] = 0;
/* 4: */
             i = i + 1;
/* 5: */
          }
/* 6: */
```

```
p1 = A[n][n=10 i=0] = <{0,i} [-oo,+oo] {n,10}>; [ i: [0,0] n: [10,10] ]
p2 = ... = p5 = p6  = <>; [ i: _|_ n: _|_ ]
p2 = p2 W (p1 U p5) = <{0,i} [-oo,+oo] {n,10}>; [ i: [0,0] n: [10,10] ]
```

A detailed example (cont'd)

```
          int n = 10;
          int i, A[n];
          i = 0;
/* 1: */
          while /* 2: */ (i < n) {
/* 3: */
             A[i] = 0;
/* 4: */
             i = i + 1;
/* 5: */
          }
/* 6: */
```

```
p1 = A[n][n=10 i=0] = <{0,i} [-oo,+oo] {n,10}>; [ i: [0,0] n: [10,10] ]
p2 = ... = p5 = p6  = <>; [ i: _|_ n: _|_ ]
p2 = p2 W (p1 U p5) = <{0,i} [-oo,+oo] {n,10}>; [ i: [0,0] n: [10,10] ]
p3 = p2[i<n]        = <{0,i} [-oo,+oo] {n,10}>; [ i: [0,0] n: [10,10] ]
```

A detailed example (cont'd)

```
          int n = 10;
          int i, A[n];
          i = 0;
/* 1: */
          while /* 2: */ (i < n) {
/* 3: */
             A[i] = 0;
/* 4: */
             i = i + 1;
/* 5: */
          }
/* 6: */
```

```
p1 = A[n][n=10 i=0] = <{0,i} [-oo,+oo] {n,10}>; [ i: [0,0] n: [10,10] ]
p2 = ... = p5 = p6  = <>; [ i: _|_ n: _|_ ]
p2 = p2 W (p1 U p5) = <{0,i} [-oo,+oo] {n,10}>; [ i: [0,0] n: [10,10] ]
p3 = p2[i<n]        = <{0,i} [-oo,+oo] {n,10}>; [ i: [0,0] n: [10,10] ]
p4 = p3[A[i]=0]     = <{0,i} [0,0] {1,i+1} [-oo,+oo] {n,10}>; [ i: [0,0] n: [10,10] ]
```

A detailed example (cont'd)

```
          int n = 10;
          int i, A[n];
          i = 0;
/* 1: */
          while /* 2: */ (i < n) {
/* 3: */
             A[i] = 0;
/* 4: */
             i = i + 1;
/* 5: */
          }
/* 6: */
```

```
p1 = A[n][n=10 i=0] = <{0,i} [-oo,+oo] {n,10}>; [ i: [0,0] n: [10,10] ]
p2 = ... = p5 = p6  = <>; [ i: _|_ n: _|_ ]
p2 = p2 W (p1 U p5) = <{0,i} [-oo,+oo] {n,10}>; [ i: [0,0] n: [10,10] ]
p3 = p2[i<n]        = <{0,i} [-oo,+oo] {n,10}>; [ i: [0,0] n: [10,10] ]
p4 = p3[A[i]=0]     = <{0,i} [0,0] {1,i+1} [-oo,+oo] {n,10}>; [ i: [0,0] n: [10,10] ]
p5 = p4[i=i+1]      = <{0 i-1} [0,0] {1,i} [-oo,+oo] {n,10}>; [ i: [1,1] n: [10,10] ]
```

A detailed example (cont'd)

```
          int n = 10;
          int i, A[n];
          i = 0;
/* 1: */
          while /* 2: */ (i < n) {
/* 3: */
             A[i] = 0;
/* 4: */
             i = i + 1;
/* 5: */
          }
/* 6: */
```

```
p1 = A[n][n=10 i=0] = <{0,i} [-oo,+oo] {n,10}>; [ i: [0,0] n: [10,10] ]
p2 = ... = p5 = p6  = <>; [ i: _|_ n: _|_ ]
p2 = p2 W (p1 U p5) = <{0,i} [-oo,+oo] {n,10}>; [ i: [0,0] n: [10,10] ]
p3 = p2[i<n]        = <{0,i} [-oo,+oo] {n,10}>; [ i: [0,0] n: [10,10] ]
p4 = p3[A[i]=0]     = <{0,i} [0,0] {1,i+1} [-oo,+oo] {n,10}>; [ i: [0,0] n: [10,10] ]
p5 = p4[i=i+1]      = <{0 i-1} [0,0] {1,i} [-oo,+oo] {n,10}>; [ i: [1,1] n: [10,10] ]
p2 = p2 W (p1 U p5) = <{0} [0,0] {i}? [-oo,+oo] {n,10}>; [ i: [0,+oo] n: [10,10] ]
```

A detailed example (cont'd)

```
          int n = 10;
          int i, A[n];
          i = 0;
/* 1: */
          while /* 2: */ (i < n) {
/* 3: */
             A[i] = 0;
/* 4: */
             i = i + 1;
/* 5: */
          }
/* 6: */
```

```
p1 = A[n][n=10 i=0] = <{0,i} [-oo,+oo] {n,10}>; [ i: [0,0] n: [10,10] ]
p2 = ... = p5 = p6  = <>; [ i: _|_ n: _|_ ]
p2 = p2 W (p1 U p5) = <{0,i} [-oo,+oo] {n,10}>; [ i: [0,0] n: [10,10] ]
p3 = p2[i<n]        = <{0,i} [-oo,+oo] {n,10}>; [ i: [0,0] n: [10,10] ]
p4 = p3[A[i]=0]     = <{0,i} [0,0] {1,i+1} [-oo,+oo] {n,10}>; [ i: [0,0] n: [10,10] ]
p5 = p4[i=i+1]      = <{0 i-1} [0,0] {1,i} [-oo,+oo] {n,10}>; [ i: [1,1] n: [10,10] ]
p2 = p2 W (p1 U p5) = <{0} [0,0] {i}? [-oo,+oo] {n,10}>; [ i: [0,+oo] n: [10,10] ]
p3 = p2[i<n]        = <{0} [0,0] {i}? [-oo,+oo] {n,10}>; [ i: [0,9] n: [10,10] ]
```

A detailed example (cont'd)

```
          int n = 10;
          int i, A[n];
          i = 0;
/* 1: */
          while /* 2: */ (i < n) {
/* 3: */
             A[i] = 0;
/* 4: */
             i = i + 1;
/* 5: */
          }
/* 6: */
```

```
p1 = A[n][n=10 i=0] = <{0,i} [-oo,+oo] {n,10}>; [ i: [0,0] n: [10,10] ]
p2 = ... = p5 = p6  = <>; [ i: _|_ n: _|_ ]
p2 = p2 W (p1 U p5) = <{0,i} [-oo,+oo] {n,10}>; [ i: [0,0] n: [10,10] ]
p3 = p2[i<n]        = <{0,i} [-oo,+oo] {n,10}>; [ i: [0,0] n: [10,10] ]
p4 = p3[A[i]=0]     = <{0,i} [0,0] {1,i+1} [-oo,+oo] {n,10}>; [ i: [0,0] n: [10,10] ]
p5 = p4[i=i+1]      = <{0 i-1} [0,0] {1,i} [-oo,+oo] {n,10}>; [ i: [1,1] n: [10,10] ]
p2 = p2 W (p1 U p5) = <{0} [0,0] {i}? [-oo,+oo] {n,10}>; [ i: [0,+oo] n: [10,10] ]
p3 = p2[i<n]        = <{0} [0,0] {i}? [-oo,+oo] {n,10}>; [ i: [0,9] n: [10,10] ]
p4 = p3[A[i]=0]     = <{0} [0,0] {i}? [0,0] {i+1} [-oo,+oo] {n,10}?>; [ i: [0,9] n: [10,10] ]
```

A detailed example (cont'd)

```
          int n = 10;
          int i, A[n];
          i = 0;
/* 1: */
          while /* 2: */ (i < n) {
/* 3: */
             A[i] = 0;
/* 4: */
             i = i + 1;
/* 5: */
          }
/* 6: */
```

```
p1 = A[n][n=10 i=0] = <{0,i} [-oo,+oo] {n,10}>; [ i: [0,0] n: [10,10] ]
p2 = ... = p5 = p6  = <>; [ i: _|_ n: _|_ ]
p2 = p2 W (p1 U p5) = <{0,i} [-oo,+oo] {n,10}>; [ i: [0,0] n: [10,10] ]
p3 = p2[i<n]        = <{0,i} [-oo,+oo] {n,10}>; [ i: [0,0] n: [10,10] ]
p4 = p3[A[i]=0]     = <{0,i} [0,0] {1,i+1} [-oo,+oo] {n,10}>; [ i: [0,0] n: [10,10] ]
p5 = p4[i=i+1]      = <{0 i-1} [0,0] {1,i} [-oo,+oo] {n,10}>; [ i: [1,1] n: [10,10] ]
p2 = p2 W (p1 U p5) = <{0} [0,0] {i}? [-oo,+oo] {n,10}>; [ i: [0,+oo] n: [10,10] ]
p3 = p2[i<n]        = <{0} [0,0] {i}? [-oo,+oo] {n,10}>; [ i: [0,9] n: [10,10] ]
p4 = p3[A[i]=0]     = <{0} [0,0] {i}? [0,0] {i+1} [-oo,+oo] {n,10}?>; [ i: [0,9] n: [10,10] ]
p5 = p4[i=i+1]      = <{0} [0,0] {i-1}? [0,0] {i} [-oo,+oo] {n,10}?>; [ i: [1,10] n: [10,10] ]
```

A detailed example (cont'd)

```
          int n = 10;
          int i, A[n];
          i = 0;
/* 1: */
          while /* 2: */ (i < n) {
/* 3: */
             A[i] = 0;
/* 4: */
             i = i + 1;
/* 5: */
          }
/* 6: */
```

```
p1 = A[n][n=10 i=0] = <{0,i} [-oo,+oo] {n,10}>; [ i: [0,0] n: [10,10] ]
p2 = ... = p5 = p6  = <>; [ i: _|_ n: _|_ ]
p2 = p2 W (p1 U p5) = <{0,i} [-oo,+oo] {n,10}>; [ i: [0,0] n: [10,10] ]
p3 = p2[i<n]        = <{0,i} [-oo,+oo] {n,10}>; [ i: [0,0] n: [10,10] ]
p4 = p3[A[i]=0]     = <{0,i} [0,0] {1,i+1} [-oo,+oo] {n,10}>; [ i: [0,0] n: [10,10] ]
p5 = p4[i=i+1]      = <{0 i-1} [0,0] {1,i} [-oo,+oo] {n,10}>; [ i: [1,1] n: [10,10] ]
p2 = p2 W (p1 U p5) = <{0} [0,0] {i}? [-oo,+oo] {n,10}>; [ i: [0,+oo] n: [10,10] ]
p3 = p2[i<n]        = <{0} [0,0] {i}? [-oo,+oo] {n,10}>; [ i: [0,9] n: [10,10] ]
p4 = p3[A[i]=0]     = <{0} [0,0] {i}? [0,0] {i+1} [-oo,+oo] {n,10}?>; [ i: [0,9] n: [10,10] ]
p5 = p4[i=i+1]      = <{0} [0,0] {i-1}? [0,0] {i} [-oo,+oo] {n,10}?>; [ i: [1,10] n: [10,10] ]
p2 = p2 W (p1 U p5) = <{0} [0,0] {i}? [-oo,+oo] {n,10}?>; [ i: [0,+oo] n: [10,10] ]
```

A detailed example (cont'd)

```
          int n = 10;
          int i, A[n];
          i = 0;
/* 1: */
          while /* 2: */ (i < n) {
/* 3: */
             A[i] = 0;
/* 4: */
             i = i + 1;
/* 5: */
          }
/* 6: */
```

```
p1 = A[n][n=10 i=0] = <{0,i} [-oo,+oo] {n,10}>; [ i: [0,0] n: [10,10] ]
p2 = ... = p5 = p6  = <>; [ i: _|_ n: _|_ ]
p2 = p2 W (p1 U p5) = <{0,i} [-oo,+oo] {n,10}>; [ i: [0,0] n: [10,10] ]
p3 = p2[i<n]        = <{0,i} [-oo,+oo] {n,10}>; [ i: [0,0] n: [10,10] ]
p4 = p3[A[i]=0]     = <{0,i} [0,0] {1,i+1} [-oo,+oo] {n,10}>; [ i: [0,0] n: [10,10] ]
p5 = p4[i=i+1]      = <{0 i-1} [0,0] {1,i} [-oo,+oo] {n,10}>; [ i: [1,1] n: [10,10] ]
p2 = p2 W (p1 U p5) = <{0} [0,0] {i}? [-oo,+oo] {n,10}>; [ i: [0,+oo] n: [10,10] ]
p3 = p2[i<n]        = <{0} [0,0] {i}? [-oo,+oo] {n,10}>; [ i: [0,9] n: [10,10] ]
p4 = p3[A[i]=0]     = <{0} [0,0] {i}? [0,0] {i+1} [-oo,+oo] {n,10}?>; [ i: [0,9] n: [10,10] ]
p5 = p4[i=i+1]      = <{0} [0,0] {i-1}? [0,0] {i} [-oo,+oo] {n,10}?>; [ i: [1,10] n: [10,10] ]
p2 = p2 W (p1 U p5) = <{0} [0,0] {i}? [-oo,+oo] {n,10}?>; [ i: [0,+oo] n: [10,10] ]

... one more iteration with {n,10}? instead of {n,10} changes nothing

p6 = p2[i>=n]       = <{0} [0,0] {n,10,i}>; [ i: [10 +oo] n: [10,10] ]
```

Concretization (meaning of abstract properties)

Variable environments

- Variable environments $\rho \in \mathcal{R}_v$ map variable names $\mathtt{i} \in \mathbb{X}$ to their values $\rho(\mathtt{i}) \in \mathcal{V}$:

$$\mathcal{R}_v \triangleq \mathbb{X} \mapsto \mathcal{V}$$

Semantics of expressions

- Expressions $\mathsf{e} \in \mathbb{E}$ have concrete semantics $[\![\mathsf{e}]\!]\rho$ so that $[\![\mathsf{e}]\!] \in \mathcal{R}_v \mapsto \mathcal{V}$
- In all examples $\mathcal{V} = \mathbb{Z}$

Array Environments

- Array environments $\theta \in \mathcal{R}_a$ map array names $\mathtt{A} \in \mathbb{A}$ to array values $\theta(\mathtt{A}) \in \mathcal{A}$ so that

$$\mathcal{R}_a \triangleq \mathbb{A} \mapsto \mathcal{A}$$

The semantics of arrays (revisited II)

- The value a of an array $\mathtt{A}$ is

$$a = (\rho, \mathtt{A.low}, \mathtt{A.high}, A) \in \mathcal{A}$$

such that

  - $\rho \in \mathcal{R}_v$ is a variable environment
  - $\mathtt{A.low} \in \mathbb{E}$ is the symbolic lower bound
  - $\mathtt{A.high} \in \mathbb{E}$ is the symbolic upper bound
  - The array value A maps indexes $i \in [[\![\mathtt{A.low}]\!]\rho, [\![\mathtt{A.high}]\!]\rho)$ to value pairs $(i, A(i))$

- so

$$\mathcal{A} \triangleq \mathcal{R}_v \times \mathbb{E} \times \mathbb{E} \times (\mathbb{Z} \mapsto (\mathbb{Z} \times \mathcal{V}))$$

Example

```
            parameter int n; /* assume n>1 */
            int i, A[n];
            i = 0;
/* 1: */    while /* 2: */ (i < n) {
/* 3: */        A[i] = i;
/* 4: */        i = i + 1;
/* 5: */    }
/* 6: */
```

The final value of A is $a_6 = (\rho_6, 0, \mathtt{n}, A_6)$ with $A_6(i) = (i, i)$ for all $i \in [0, n)$. Because ρ_6, 0, and n are easily understood from the context, we write A[i] $= (i, i)$ by abuse of notation where the value i of i is assumed to be within the bounds.

Concretization

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

- Concretization of variables:

$$\gamma_v \in \overline{\mathcal{R}} \mapsto \wp(\mathcal{R}_v)$$

- Concretization of bound expressions:

$$\gamma_e \in \overline{\mathcal{E}} \mapsto \overline{\mathcal{R}} \mapsto \wp(\mathcal{V})$$

- Concretization of segment bounds:

$$\gamma_b \in \overline{\mathcal{B}} \mapsto \overline{\mathcal{R}} \mapsto \wp(\mathcal{R}_v)$$

- Concretization of segment abstract values:

$$\gamma_a \in \overline{\mathcal{A}} \mapsto \wp(\mathbb{Z} \times \mathcal{V})$$

Concretization (cont'd)

$$\{e^1_1, \ldots, e^1_{m^1}\}\ P_1\ \{e^2_1, \ldots, e^2_{m^2}\}[?^2]\ P_2\ \ldots\ P_{n-1}\ \{e^n_1, \ldots, e^n_{m^n}\}[?^n]$$

- Concretization of a segment $B\ P\ B'\ [?]$:

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

($<_\sqcup$ stands for $<$ while $<?$ stands for $\leqslant$)

Concretization (cont'd)

$$\{e^1_1, \ldots, e^1_{m^1}\}\ P_1\ \{e^2_1, \ldots, e^2_{m^2}\}[?^2]\ P_2\ \ldots\ P_{n-1}\ \{e^n_1, \ldots, e^n_{m^n}\}[?^n]$$

- Concretization of an array:

$$\gamma_s(B_1 P_1 B_2[?^2] P_2 \ldots P_{n-1} B_n[?^n])\overline{\rho} \quad \triangleq$$

$$\{(\rho, \ell, h, A) \in \bigcap_{i=1}^{n-1} \gamma'_s(B_i P_i B_{i+1}[?^{i+1}])\overline{\rho} \mid$$

$$\forall \mathsf{e}_1 \in B_1 : [\![\mathsf{e}_1]\!]\rho = [\![\ell]\!]\rho \wedge \forall \mathsf{e}_n \in B_n : [\![\mathsf{e}_n]\!]\rho = [\![h]\!]\rho\}$$

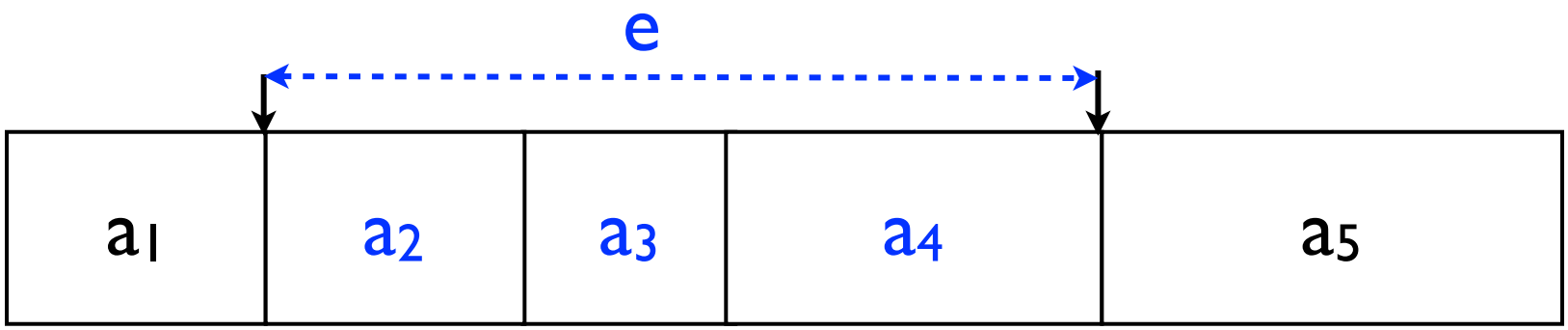
and $\gamma_s(\bot) = \emptyset$.

The array segmentation abstract domain functor: abstract operations

Abstract value of an array element

Value of A[e]:

1. Determine to which segment(s) of A the index e *may* belong
2. If none signal an array overrun
3. Select the corresponding abstract value of array elements (their join if more than one)

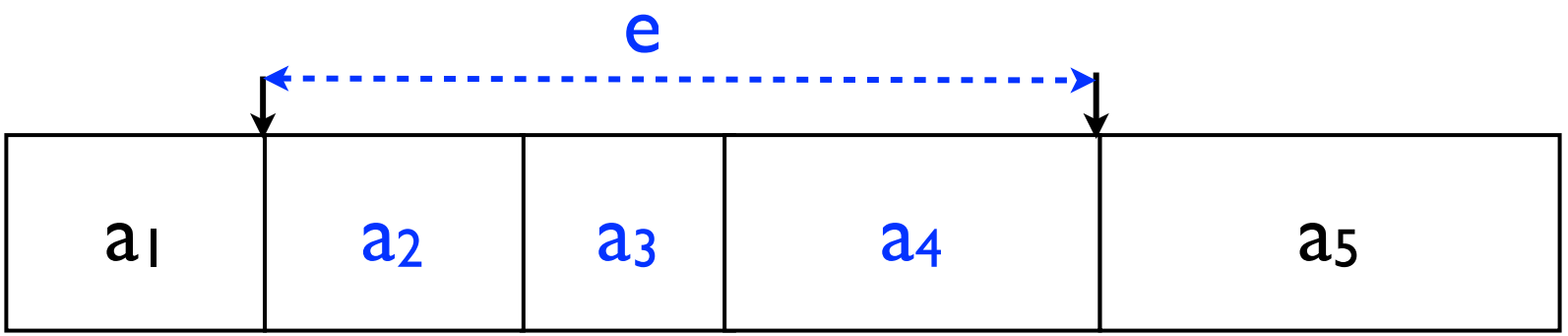


$$A[e] := a_2 \sqcup a_3 \sqcup a_4$$

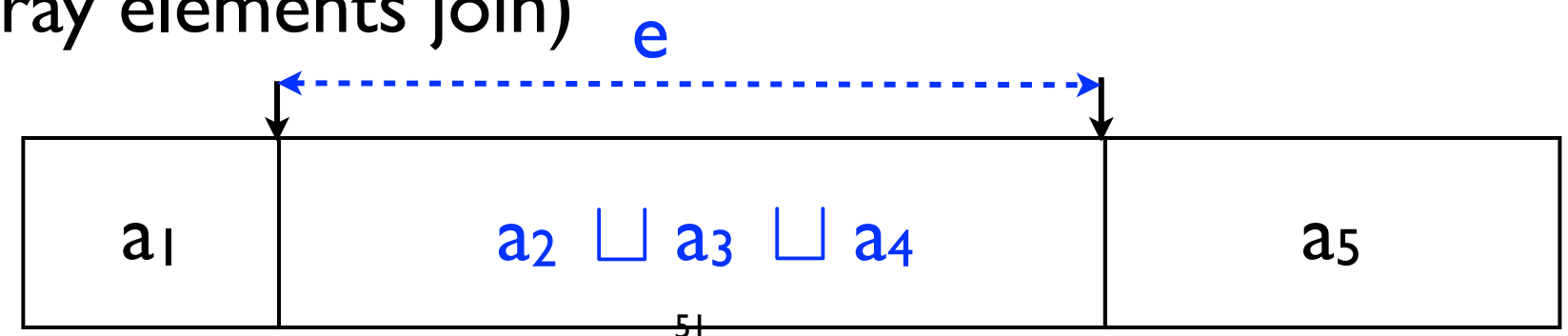
Assignment to an array element

Assignment to A[e] := v

1. Determine to which segment(s) the index e may belong
2. If none signal a array overrun



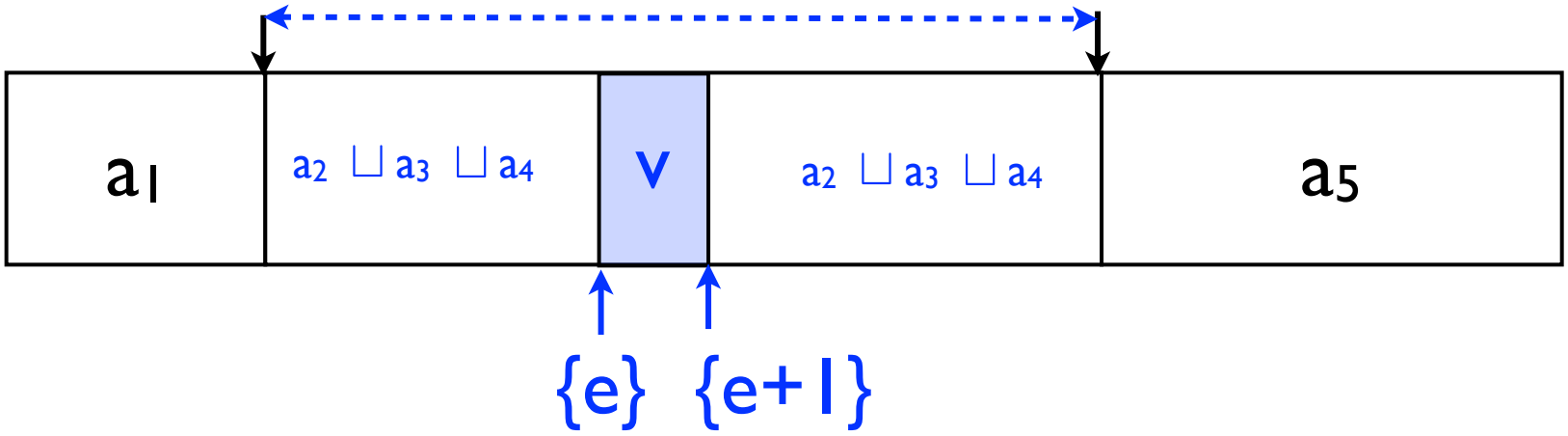
3. If more than one join these segments (using the array elements join)



Assignment to an array element

Assignment to A[e] := v (continued)

4. Split the segment to insert abstract value v of assigned element (with special cases for assignments to segment bounds positions)



5. Adjust emptiness of resulting segments

Assignment to a simple variable

- Invertible assignment $i_{new} = e(i_{old})$ so $i_{old} = e^{-1}(i_{new})$
  - Replace i by $e^{-1}(i_{new})$ in all expressions in array segment bounds where i does appear

```
[ A: <{0} [-oo,+oo] {i} [1 +oo-1] {n}?> ] [ i: [1 +oo] n: [2,+oo] ]
i=i-1;
[ A: <{0} [-oo,+oo] {i+1} [1 +oo-1] {n}?> ] [ i: [0 +oo-1] n: [2,+oo] ]
```

- Non-invertible assignment to $i = e$
  - Eliminate all expressions in array segment bounds where i does appear
  - If a block limit becomes empty, join adjacent blocks
  - Add i to all block limits containing e

Conditionals on simple variables

- Test e = e'
  - Add e/e' in segment bounds with e'/e
- Test e < e'
  - Adjust emptiness (and reduce block bounds)

Conditionals on array elements

- Access + restriction by test + assignment

Segmentwise comparison, join, meet, widening, narrowing

- For identical segmentations binary operations are performed segmentwise
- Example: join

$$
\begin{array}{rl}
 & \langle\{0\}\ [0,0]\ \ \{i\}\ [0,2]\ \ \{n\}\rangle \\
\sqcup & \langle\{0\}\ [1,1]\ \ \{i\}\ [-1,0]\ \ \{n\}\rangle \\
= & \langle\{0\}\ [0,1]\ \ \{i\}\ [-1,2]\ \ \{n\}\rangle
\end{array}
$$

Segmentation unification

- For non-identical segmentations a segment unification must be performed first:
  - By splitting segments when possible

<{0} a {i} b {n}> ⟶ <{0} a {i} b {j} b {n}>

<{0} a' {i} b' {j} c' {n}> ⟶ <{0} a' {i} b' {j} c' {n}>

  - Otherwise by joining adjacent segments

<{0} a {i} b {n}> ⟶ <{0} a ⊔ b {n}>

<{0} a' {j} b' {n}> ⟶ <{0} a' ⊔ b' {n}>

(assuming i and j are incomparable with their variable abstractions and in the other array segmentations)

Example of segmentation unification in a union

$$\begin{array}{ll} & \mathtt{A} : \{0, \mathtt{i}\}\top\{10, \mathtt{n}\},\ \mathtt{i} : [0,0],\ \mathtt{n} : [10,10] \\ \sqcup & \mathtt{A} : \{0, \mathtt{i\text{-}1}\}0\{1, \mathtt{i}\}\top\{10, \mathtt{n}\},\ \mathtt{i} : [1,1],\ \mathtt{n} : [10,10] \\ = & \mathtt{A} : \{0\}\bot\{\mathtt{i}\}?\top\{10, \mathtt{n}\},\ \mathtt{i} : [0,0],\ \mathtt{n} : [10,10] \\ \sqcup & \mathtt{A} : \{0\}0\{\mathtt{i}\}\top\{10, \mathtt{n}\},\ \mathtt{i} : [1,1],\ \mathtt{n} : [10,10] \\ = & \mathtt{A} : \{0\}0\{\mathtt{i}\}?\top\{10, \mathtt{n}\},\ \mathtt{i} : [0,1],\ \mathtt{n} : [10,10] \end{array}$$

Comparison of expressions e =/≤/< e' in segment bounds

- Purely symbolically (Pratt's graphalgorithm with Roy/Warshall-Floyd transitive closure)
  e.g. $x + i < y + j$ since $x = y \,\&\, i < j$

- Using non-relational information on variables
  e.g. $x + 1 < y$ since $x:[-\infty, 3] \,\&\, y:[5, +\infty]$

- Using information on (other) array segment ordering
  e.g. $x+1 < y$ since ...{x}?...{...}...{y+1}...

- Using information provided by a relational abstract domain (e.g. pentagons, DBM, octagons, sub-polyhedra polyhedra ...)

A few more examples

Array partitioning

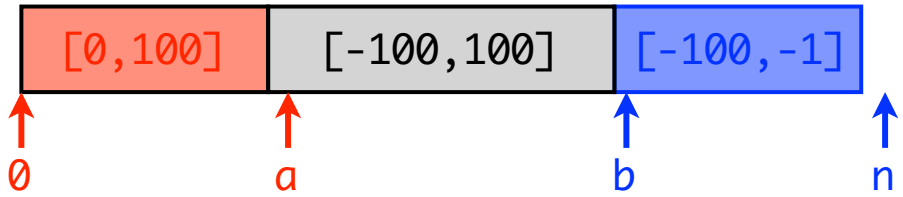
```
            parameter int n  /* assume n>1 */
            var int a  b  c  A[n];
            assume A: {0}[-100 +100]{n}
            a = 0; b = 0; c = 0;
/*  1: */
            while /* 2: */ (a < n) {
/*  3: */
                if A[a] >= 0 then {
/*  4: */
                    B[b] = A[a]; b = b + 1;
/*  5: */
                } else {
/*  6: */
                    C[c] = A[a]; c = c + 1;
/*  7: */
                }
/*  8: */
                a = a + 1;
/*  9: */
            }
/* 10: */
```

```
p10 = [ A: <{0} [-100,100] {n}?> B: <{0} [0,100] {b}? [-oo,+oo] {n}?> C: <{0}
[-100,-1] {c}? [-oo,+oo] {n}?> ] [ a: [2,+oo] b: [0,+oo] c: [0,+oo] n:
[2,+oo] ]
0.003711 s
```

Example from: Laura Kovács, Andrei Voronkov: Finding Loop Invariants for Programs over Arrays Using a Theorem Prover. FASE 2009: 470–485.

In situ array partitioning

```
            parameter int n; /* assume n>1 */
            var int a  b  x  A[n];
            assume A: {0}[-100 +100]{n}
            a = 0; b = n;
/*  1: */
            while /* 2: */ (a < b) {
/*  3: */
                if A[a] >= 0 then {
/*  4: */
                    a = a + 1;
/*  5: */
                } else {
/*  6: */
                    b = b - 1;
/*  7: */
                    x = A[a]; A[a] = A[b]; A[b] := x;
/*  8: */
                }
/*  9: */
            }
/* 10: */
```



Analysis with widening/narrowing and (interval domain x interval domain):

p1 = [A: <{0,a} [-100,100] {n,b}>] [a: [0,0] b: [2,+oo] n: [2,+oo] x: [-oo,+oo]]

p2 = [A: <{0} [0,100] {a}? [-100,100] {b}? [-100,-1] {n}?>] [a: [0,+oo] b: [0,+oo] n: [2,+oo] x: [-oo,+oo]]

p10 = [A: <{0} [0,100] {b,a}? [-100,-1] {n}?>] [a: [0,+oo] b: [0,+oo] n: [2,+oo] x: [-oo,+oo]]

0.015378 s

Partial initialization

```
int a[100], c[100], i, j;

i = 0, j = 0;

while (i < 100) {
    if (P(a[i])) { // For some predicate P
        c[j] = i;
        j = j + 1;
    }
    i = i + 1;
}
```

```
[ c: <{0},[0,99],{j}?, [-oo, +oo] ,{i,100}?> ] [ ... ]
```

I – Non-relational analysis on values (I)

```
          int n = 10;
          int i  A[n];
          i = 0;
/* 1: */
          while /* 2: */ (i < n) {
/* 3: */
             A[i] = 0;
/* 4: */
             i = i + 1:
/* 5: */
             A[i] = -16;
/* 6: */
             i = i + 1:
/* 7: */
          }
/* 8: */
```


Array: reduced product of parity and intervals – i.e. semantics $A[i] := v_i$

Variables: reduced product of parity and intervals

```
p1 = <{0,i} (T  [-oo,+oo]) {n,10}>; [ i: (e, [0,0]) n: (e, [10,10]) ]
p2 = <{0} (e, [-16,0]) {i}? (T  [-oo,+oo]) {n,10}?>; [ i: (e, [0 +oo-1]) n: (e, [10,10]) ]
p8 = <{0} (e, [-16,0]) {n,10,i}>; [ i: (e, [10,+oo-1]) n: (e, [10,10]) ]

0.000832 s
```

II – Non-relational analysis on values (II)

```
          int n = 10;
          int i  A[n];
          i = 0;
/* 1: */
          while /* 2: */ (i < n) {
/* 3: */
             A[i] = 0;
/* 4: */
             i = i + 1:
/* 5: */
             A[i] = -16;
/* 6: */
             i = i + 1:
/* 7: */
          }
/* 8: */
```

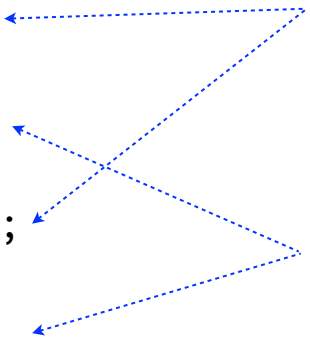

Array: interval power parity on array elements – i.e. semantics A[i] := v_i

Variables: reduced product of parity and intervals

```
p1 = <{0,i} (o -> [-oo,+oo] e -> [-oo,+oo]) {n,10}>; [ i: (e, [0,0]) n: (e, [10,10]) ]
p2 = <{0} (o -> _|_ e -> [-16,0]) {i}? (o -> [-oo,+oo] e -> [-oo,+oo]) {n,10}?>;
                                                  [ i: (e, [0,+oo-1]) n: (e, [10,10]) ]
p8 = <{0} (o -> _|_ e -> [-16,0]) {n,10,i}>; [ i: (e, [10,+oo-1]) n: (e, [10,10]) ]

0.00088 s
```

III – Relational analysis on (indexes x values)

```
          int n = 10;
          int i  A[n];
          i = 0;
/* 1: */
          while /* 2: */ (i < n) {
/* 3: */
              A[i] = 0;
/* 4: */
              i = i + 1:
/* 5: */
              A[i] = -16;
/* 6: */
              i = i + 1:
/* 7: */
          }
/* 8: */
```



Array: interval power parity on array elements – i.e. semantics A[i] := (i v_i)

Variables: reduced product of parity and intervals

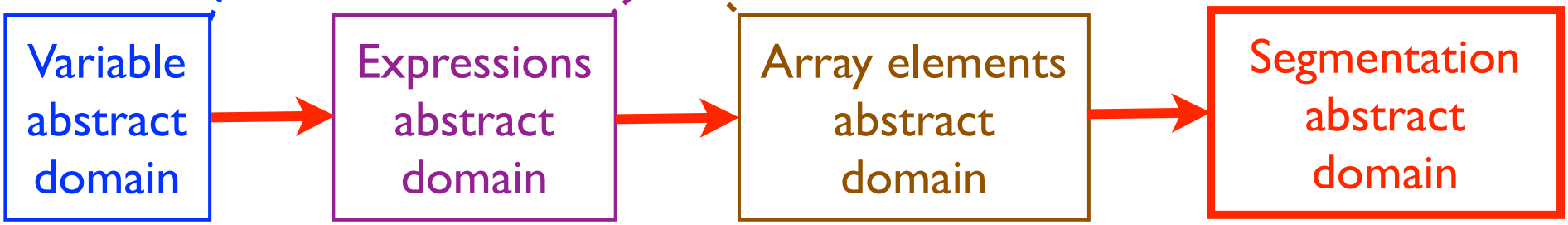
```
p1 = <{0,i} (o -> [-oo,+oo] e -> [-oo,+oo]) {n,10}>; [ i: (e, [0,0]) n: (e, [10,10]) ]
p2 = <{0} (o -> [-16 -16] e -> [0,0]) {i}? (o -> [-oo,+oo] e -> [-oo,+oo]) {n,10}?>; [ i:
(e, [0 +oo-1]) n: (e, [10,10]) ]
p8 = <{0} (o -> [-16 -16] e -> [0,0]) {n,10,i}>; [ i: (e, [10,+oo-1]) n: (e, [10,10]) ]

0.001274 s
```

The segmentation abstract domain functor

- Our semantics for relational segmentation:

  Array $\in$ Values of variables $\rightarrow$ Set of indices
  $\rightarrow$ Set of (index x values)

- The abstraction functor:



Sound automatic terminating but incomplete...

```
          parameter int n; /* assume n>1 */
          int i  A[n];
          i = n;
/* 1: */
          while /* 2: */ (0 < i) {
/* 3: */
             i = i - 1;
/* 4: */
             A[i] = i;
/* 5: */
          }
/* 6: */
```

Analysis with widening/narrowing without thresholds and (interval domain x interval domain):

```
[ -oo +oo ]
```

```
p6 = [ A: <{0,i} [-oo,+oo-1] {n}> ] [ i: [0,0] n: [2,+oo] ]
0.003486 s
```

Sound automatic terminating but incomplete...

```
          parameter int n; /* assume n>1 */
          int i  A[n];
          i = n;
/* 1: */
          while /* 2: */ (0 < i) {
/* 3: */
             i = i - 1;
/* 4: */
             A[i] = i;
/* 5: */
          }
/* 6: */
```

i: [2,+oo] initial
i: [1 +oo-1] decrementation
i: [-oo,+oo] widening
i: [0,+oo] test & narrowing

```
Analysis with widening/narrowing without thresholds and (interval
domain x interval domain):
[ -oo +oo ]

p6 = [ A: <{0,i} [-oo +oo-1] {n}> ] [ i: [0,0] n: [2,+oo] ]
0.003486 s
```

Improvement ... 1st solution

- Widening/narrowing with thresholds

```
          parameter int n; /* assume n>1 */
          int i  A[n];
          i = n;
/* 1: */
          while /* 2: */ (0 < i) {
/* 3: */
             i = i - 1;
/* 4: */
             A[i] = i;
/* 5: */
          }
/* 6: */
```

Analysis with widening/narrowing with following thresholds and (interval domain x interval domain):

[-oo -1 0 1 +oo]

```
p6 = [ A: <{0,i} [0 +oo-1] {n}> ] [ i: [0,0] n: [2,+oo] ]
0.001868 s
```

Improvement ... 2nd solution

- Recurrent reanalysis

```
          parameter int n; /* assume n>1 */
          int i  A[n];
          i = n;
/* 1: */
          while /* 2: */ (0 < i) {
/* 3: */
             i = i - 1;
/* 4: */
             A[i] = i;
/* 5: */
          }
/* 6: */
```

Analysis with widening/narrowing without thresholds but with reiteration for arrays on stabilized simple variables and (interval domain x interval domain):

```
[ -oo +oo ]

p6 = [ A: <{0,i} [0 +oo-1] {n}> ] [ i: [0,0] n: [2,+oo] ]
0.002766 s
```

Principle of recurrent reanalysis

$$A_0, V_0 = \mathrm{lfp}^{\bot, \bot}_{A_0, V_0}\ \lambda x, x'.\ x, x'\ (\triangledown \times \triangledown)\ F(x, x')$$
$$A_1, V_1 = \mathrm{gfp}\ \lambda x, x'.\ x, x'\ (\triangle \times \triangle)\ F(x, x')$$
$$A_2, V_2 = \mathrm{lfp}^{\bot, V_1}\ \lambda x, x'.\ x, x'\ (\triangledown \times \sqcup)\ F(x, x')$$
$$A_3, V_3 = \mathrm{gfp}^{A_2 \sqcap V_2}\ \lambda x, x'.\ x, x'\ (\triangle \times \sqcap)\ F(x, x')$$

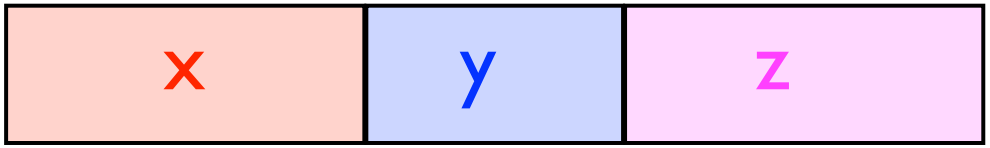
...

arrays × *variables*

Segmentation relational analyzes (not yet implemented)

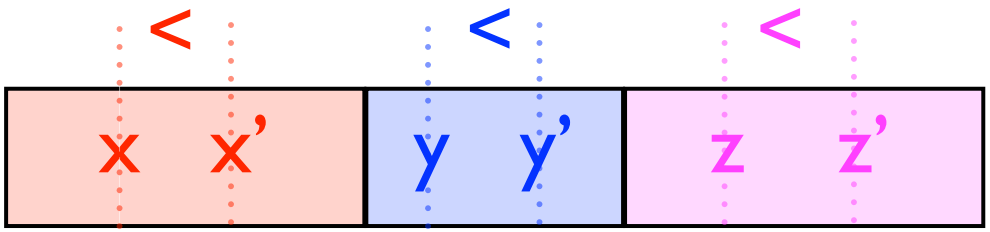
Relational analyses

- Inter-segments



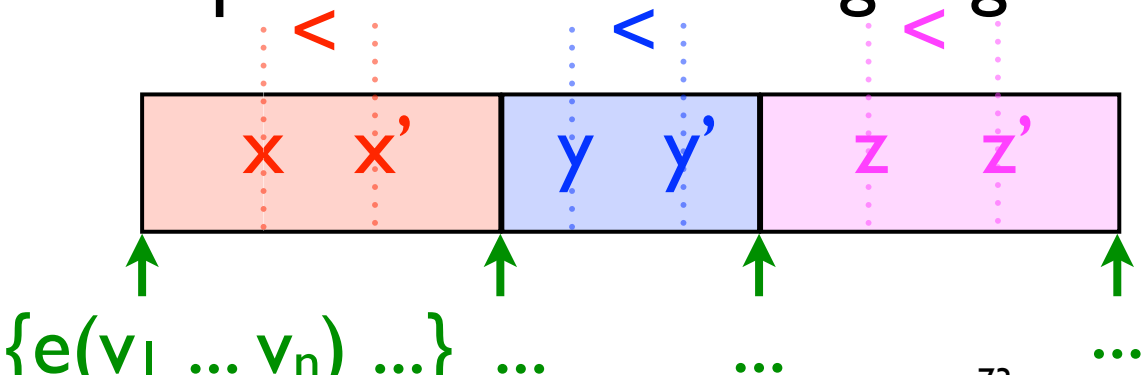
r(x y z)

- Intra/inter-segment



r(x x' y y' z z')

- Can also relate to variables appearing in sets of expressions delimiting segment bounds



r(x x' y y' z z' v_1 ... v_n)

Extensions

Existential instead of universal intra-segment properties

$$A{:}\langle L \ldots \{e_1 \ldots e_n\}\ a\ \{e'_1 \ldots e'_m\}[?] \ldots H\rangle$$

- Universal:

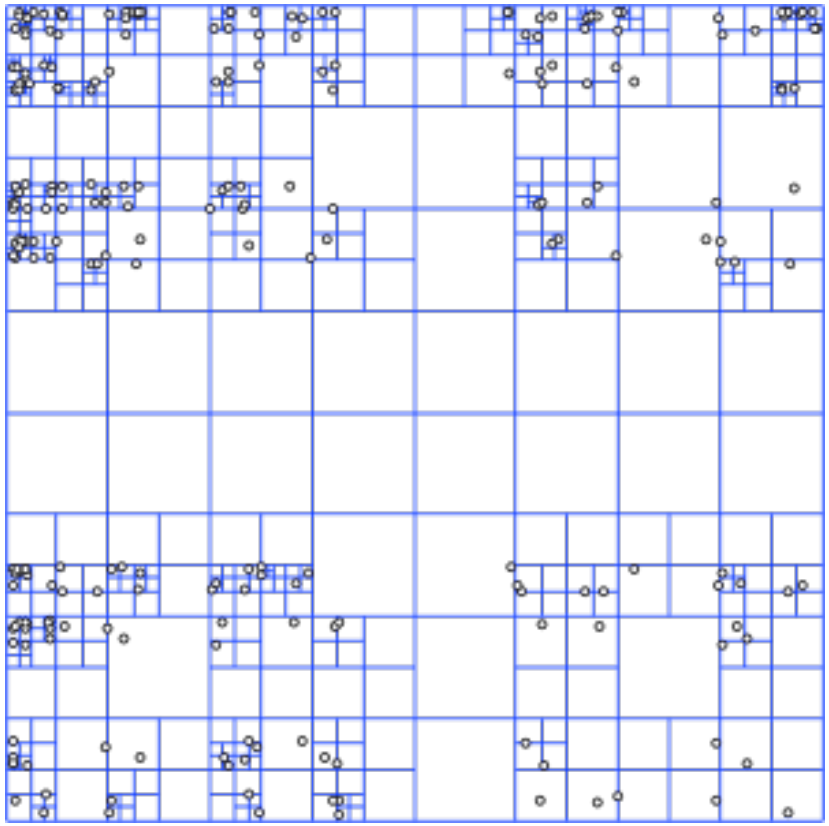
$$[\![e_1]\!] = \ldots = [\![e_n]\!] = l < [\leq] [\![e'_1]\!] = \ldots = [\![e'_m]\!] = h \wedge$$
$$\forall i: (l \leq i \leq h) \Rightarrow (A[i] \in \gamma(a))$$

- Existential:

$$[\![e_1]\!] = \ldots = [\![e_n]\!] = l < [\leq] [\![e'_1]\!] = \ldots = [\![e'_m]\!] = h \wedge$$
$$\exists i: (l \leq i \leq h) \Rightarrow (A[i] \in \gamma(a))$$

Multi-dimentional arrays

- Consider matrices as an array of arrays of elements and instanciate the functor twice;
- More complex tilings (e.g. region quadtrees) are also conceivable



Related work

Related work

- Of course there are many static analyzes related to bounds of array indexes starting from

Patrick Cousot & Radhia Cousot. Static Determination of Dynamic Properties of Programs. IProceedings of the second international symposium on Programming, Paris, 106—130, 1976, Dunod, Paris.

- including for non-uniform alias analysis

Stephen J. Fink, Kathleen Knobe, Vivek Sarkar: Unified Analysis of Array and Object References in Strongly Typed Languages. SAS 2000: 155-174

Arnaud Venet: Nonuniform Alias Analysis of Recursive Data Structures and Arrays. SAS 2002: 36-51

- vectorization parallelization ...

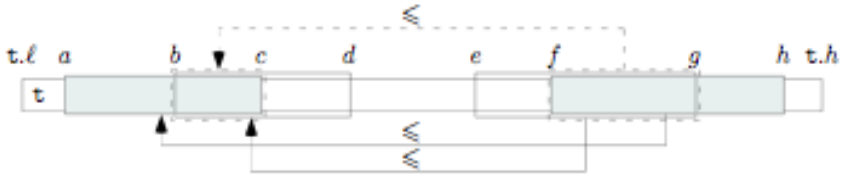
Gerald Roth, Ken Kennedy: Dependence Analysis of Fortran90 Array Syntax. PDPTA 1996: 1225-1235

- etc, etc.

Related work (cont'd)

- Our basic inspiration: parametric predicate abstraction

P. Cousot:, Verification by Abstract Interpretation. Verification: Theory and Practice. LNCS 2772, 2003: 243–26



used in many automatic abstract-interpretation-based array analyzes (often using partitions)

Denis Gopan, Thomas W. Reps, Shmuel Sagiv: A framework for numeric analysis of array operations. POPL 2005: 338–350

Nicolas Halbwachs, Mathias Péron: Discovering properties about arrays in simple programs. PLDI 2008: 339–348

Xavier Allamigeon: Non-disjunctive Numerical Domain for Array Predicate Abstraction. ESOP 2008: 163–177

Related work (cont'd)

- Predicate abstraction with refinement and/or more arbitrary forms of predicates

Cormac Flanagan, Shaz Qadeer: Predicate abstraction for software verification. POPL 2002: 191–202

Shuvendu K. Lahiri, Randal E. Bryant: Indexed Predicate Discovery for Unbounded System Verification. CAV 2004: 135–147

Shuvendu K. Lahiri, Randal E. Bryant: Constructing Quantified Invariants via Predicate Abstraction. VMCAI 2004: 267–281

Shuvendu K. Lahiri, Randal E. Bryant: Predicate abstraction with indexed predicates. ACM Trans. Comput. Log. 9(1): (2007)

Alessandro Armando, Massimo Benerecetti, Jacopo Mantovani: Abstraction Refinement of Linear Programs with Arrays. TACAS 2007: 373–388

Mohamed Nassim Seghir, Andreas Podelski, Thomas Wies: Abstraction Refinement for Quantified Array Assertions. SAS 2009: 3–18

Related work (con'd)

- Theorem prover-based with refinement and/or arbitrary forms of predicates

Ranjit Jhala, Kenneth L. McMillan: Array Abstractions from Proofs. CAV 2007: 193-206

Sumit Gulwani, Bill McCloskey, Ashish Tiwari: Lifting abstract interpreters to quantified logical domains. POPL 2008: 235-246

Laura Kovács, Andrei Voronkov: Finding Loop Invariants for Programs over Arrays Using a Theorem Prover. FASE 2009: 470-485

Evaluation criteria

Important evaluation criteria not always very clear from the array content analysis literature:

- without program restrictions ?,
- fully automatic without user-given specifications and inductive invariants ??,
- scales up ???,
- used/usable in production-quality static analysis tools ????

Conclusion

The array segmentation abstract domain functor

- Fully automatic analysis (no hidden hypotheses)
- Simple
- Efficient (does scale up)
- Autonomous (no required dependencies on index abstractions or other analyzes)
- Parametric (precision can be gained by precise array element/index analyzes)
- The abstract domain functor has been integrated in a production-quality static analyzer (Clousot by Francesco Logozzo at MSR)
- Found useful by end-users to checks contracts.

```
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` terminology (adopted in `.NET` from $v4.0$) to express preconditions, assertions and bounded universal quantifications [4].

Lib	# func.	# instr.	time	time with arr.	Δ	# invariants
mscorlib.dll	21 475	4 550 656	4:06	4:15	0:09	2 430
System.dll	15 489	3 178 496	3:40	3:46	0:06	1 385
System.data.dll	12 408	2 933 248	4:49	4:55	0:06	1 325
System.Drawings.dll	3 123	626 688	0:28	0:29	0:01	289
System.Web.dll	23 647	5 242 880	4:56	5:02	0:06	840
System.Xml.dll	10 510	2 048 000	3:59	4:16	0:17	807

Table 1: The execution time with and without the array analysis, and the number of non-trivial array invariants. Time is in minutes.

(I) This version of Clousot should be available shortly on DevLabs.

The End