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

Patrick Cousot, NYU & ENS
Radhia Cousot, CNRS & ENS & MSR
Francesco Logozzo, MSR

The problem: Array analysis

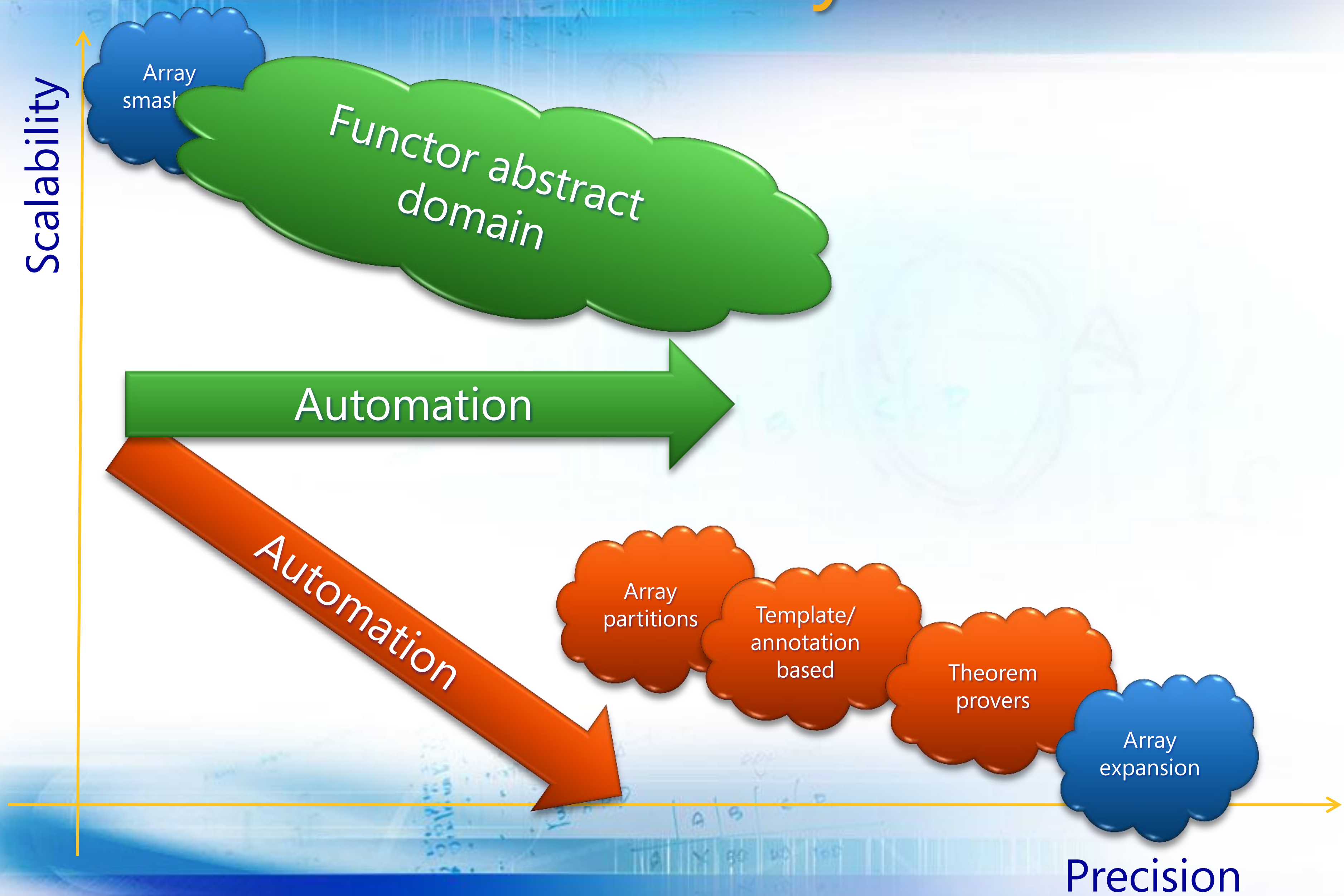
```
public void Init(int[] a)
{
  Contract.Requires(a.Length > 0);

  var j = 0;

  while (j < a.Length)
  {
    a[j] = 11;
    j++;
  }

  // here: ∀k.0≤k<j⇒a[k]=11
}
```

if `j = 0` then
  `a[0]` … not known
else if `j > 0 ∧ j ≤ a.Length`
  `a[0] = … a[j-1] = 11`
else
  impossible

Challenge 1:
All the elements are initialized

Challenge 2:
Handling of disjunction

Haven't we solved it yet?



Functor abstract domain by example

Array Materialization

```
public void Init(int[] a)
{
   Contract.Requires(a.Length > 0);

   var j = 0;

   while (j < a.Length)
   {
      a[j] = 11;

      j++;

   }
}
```

{0}	Top	{a.Length}?

Segment limits

Segment abstraction

Possibly empty segment

'?' Removal

```
public void Init(int[] a)
{
   Contract.Requires(a.Length > 0);

   var j = 0;

   while (j < a.Length)
   {
      a[j] = 11;

      j++;

   }
}
```

{0}	Top	{a.Length}

Remove doubt

Constant Assignment

```
public void Init(int[] a)
{
  Contract.Requires(a.Length > 0);

  var j = 0;

  while (j < a.Length)
  {
    a[j] = 11;

    j++;

  }
}
```

{0,j}	Top	{a.Length}	j:[0,0]

Record j = 0

Scalar variables abstraction (omit a.Length ∈ [1, +∞))

Test

```
public void Init(int[] a)
{
   Contract.Requires(a.Length > 0);

   var j = 0;

   while (j < a.Length)
   {
      a[j] = 11;

      j++;

   }
}
```

{0,j}	Top	{a.Length}	j:[0,0]

Array assignment

```
public void Init(int[] a)
{
  Contract.Requires(a.Length > 0);

  var j = 0;

  while (j < a.Length)
  {
    a[j] = 11;

    j++;

  }
}
```

{0,j}	[11, 11]	{1, j+1}	Top	{a.Length}?	j:[0,0]

Materialize segment

Introduce '?'

Scalar Assignment

```
public void Init(int[] a)
{
  Contract.Requires(a.Length > 0);

  var j = 0;

  while (j < a.Length)
  {
    a[j] = 11;

    j++;

  }
}
```

{0,j-1}	[11, 11]	{1,j}	Top	{a.Length}?	j:[1,1]

Replace j by j-1

Join

```
public void Init(int[] a)
{
  Contract.Requires(a.Length > 0);

  var j = 0;
```

{0,j}	Top	{a.Length}	j:[0,0]

```
  while (j < a.Length)
  {
    a[j] = 11;

    j++;
```

{0,j-1}	[11, 11]	{1, j}	Top	{a.Length}?	j:[1,1]

```
  }
}
```

Segment unification

1. Unify the segments

{0,j}	Top	{a.Length}

{0}	⊥	{j}?	Top	{a.Length}

{0,j-1}	[11, 11]	{1,j}	Top	{a.Length}?

{0}	[11, 11]	{j}	Top	{a.Length}?

2. Point-wise join

{0}	[11, 11]	{j}?	Top	{a.Length}?

- Similar for order, meet and widening

After the first iteration

```
public void Init(int[] a)
{
  Contract.Requires(a.Length > 0);

  var j = 0;

  while (j < a.Length)
  {
    a[j] = 11;

    j++;

  }
}
```

{0}	[11, 11]	{j}?	Top	{a.Length}?	j ∈ [0,1]

Test

```
public void Init(int[] a)
{
   Contract.Requires(a.Length > 0);

   var j = 0;

   while (j < a.Length)
   {
      a[j] = 11;

      j++

   }
}
```

{0}	[11, 11]	{j}?	Top	{a.Length}	j ∈ [0,1]

Remove '?'

Array assignment

```
public void Init(int[] a)
{
  Contract.Requires(a.Length > 0);

  var j = 0;

  while (j < a.Length)
  {
    a[j] = 11;

    j++;

  }
}
```

{0}	[11,11]	{j}?	[11,11]	{j+1}	Top	{a.Length}?	j ∈ [0,1]

Scalar assignement

```
public void Init(int[] a)
{
  Contract.Requires(a.Length > 0);

  var j = 0;

  while (j < a.Length)
  {
    a[j] = 11;

    j++;
  }
}
```

{0}	[11,11]	{j-1}?	[11,11]	{j}?	Top	{a.Length}?	j ∈ [1,2]

Widening

```
public void Init(int[] a)
{
  Contract.Requires(a.Length > 0);

  var j = 0;
```

{0}	[11, 11]	{j}?	Top	{a.Length}?	j ∈ [0,1]

```
  while (j < a.Length)
  {
    a[j] = 11;

    j++;
```

{0}	[11,11]	{j-1}?	[11,11]	{j}?	Top	{a.Length}?	j ∈ [1,2]

```
  }
}
```

Fixpoint

```
public void Init(int[] a)
{
  Contract.Requires(a.Length > 0);

  var j = 0;

  while (j < a.Length)
  {
    a[j] = 11;

    j++;

  }
}
```

{0}	[11, 11]	{j}?	Top	{a.Length}?	j ∈ [0,+∞)

Reduction

```
public void Init(int[] a)
{
  Contract.Requires(a.Length > 0);

  var j = 0;

  while (j < a.Length)
  {
    a[j] = 11;

    j++;

  }
  // here j ≥ a.Length
}
```

{0}	[11, 11]	{j}?	Top	{a.Length}?	j ∈ [0,+∞)

Remove the empty segment

{0}	[11, 11]	{j, a.Length}	j ∈ [1,+∞)

Abstract Semantics

The Functor FunArray

- Given an abstract domain
  - B for bounds
  - S for segments
  - E for scalar variables environment
- Constructs an abstract domain *F*(B, S, E) to analyze programs with arrays
- (Main) Advantages
  - Fine tuning of the precision/cost ratio
  - Easy lifting of existing analyses

Segment bounds

- Sets of symbolic expressions
  - In our examples: Exp := k | x | x + k
- Meaning:

$\{ e_0 \ldots e_n \} \{ e'_1 \ldots e'_m\} \stackrel{\text{def}}{=} e_0 = \ldots = e_n < e'_1 = \ldots = e'_m$

$\{ e_0 \ldots e_n \} \{ e'_1 \ldots e'_m\}? \stackrel{\text{def}}{=} e_0 = \ldots = e_n \leq e'_1 = \ldots = e'_m$

- Possibly empty segments are key for scalability

Disjunction: Partitions & co.

```
public void CopyNonNull(object[] a, object[] b)
{
 Contract.Requires(a.Length <= b.Length);

 var j = 0;
 for (var i = 0; i < a.Length; i++)
 {
  if (a[i] != null)
  {
   b[j] = a[i];
   j++;
  }
 }
}
```

Four partitions:

$j = 0 \lor$

$0 \le j < b.Length-1 \lor$

$j = b.Length-1 \lor$

$j = b.Length$

Disjunction: Our approach

```
public void CopyNonNull(object[] a, object[] b)
{
 Contract.Requires(a.Length <= b.Length);

 var j = 0;
 for (var i = 0; i < a.Length; i++)
 {
  if (a[i] != null)
  {
   b[j] = a[i];
   j++;
  }
 }
}
```

{0}	NotNull	{j}?	Top	{b.Length}?	$j \in [0,+\infty)$

Segmentation discovered by the analysis

Segment Abstraction

- Uniform abstraction for pairs `(i, a[i])`
  - More general than usual McCarthy definition
- Wide choice of abstract domains
  - Fine tuning the cost/precision ratio
- Ex: Cardinal power of constants by parity [CC79]

```
public void EvenOdd(int n)
{
 var a = new int[n];
 var i = 0;
 while (i < n)
 {
  a[i++] = 1;
  a[i++] = -1;
 }
}
```

{0}	even → 1 odd → -1	{i, n, a.Length}?	i ∈ [0,+∞)

Segmentation Unification

- Given two segmentations, find a common segmentation
- Crucial for order/join/meet/widening:
  1. Unify the segments
  2. Apply the operation point-wise
- In the concrete, a lattice of solutions
- In the abstract, a partial order of solutions
- Our algorithm tuned up by examples
  - Details in the paper

Read: x = a[exp]

- Search the bounds for exp

…	…	B_n	A_n	…	A_{m-1}	B_m	…	…	σ

  - The search queries the scalar environment σ
    - More precision
    - A form of abstract domains reduction
- Set $\sigma' = \sigma\,[x \mapsto A_n \sqcup \ldots \sqcup A_{m-1}]$

Write: a[exp] = x

- Search the bounds for exp

…	…	B_n	A_n	…	A_{m-1}	B_m	…	…

- Join the segments

…	…	B_n	$A_n \sqcup .. \sqcup A_{m-1}$	B_m	…	…

- Split the segment

…	…	B_n	$A_n \sqcup .. \sqcup A_{m-1}$	exp	σ(x)	exp+1	$A_n \sqcup .. \sqcup A_{m-1}$	B_m	…	…

- Adjust emptiness
  - May query scalar variables environment

Scalar assignment

- Invertible assignment `x = g(x)`
  - Replace x by $g^{-1}(x)$ in all the segments
- Non-Invertible assignment `x = g()`
  - Remove x in all the segments
  - Remove all the empty segments
  - Add x to all the bounds containing `g()`

Assumptions (and tests)

- `Assume x == y`
  - Search for segments containing `x/y`
  - Add `y/x` to them
- `Assume x < y`
  - Adjust emptiness
- `Assume x ≤ y`
  - Does the state implies `x ≥ y` ?
  - If yes, `Assume x == y`
- Assumptions involving arrays similar

Implementation

- Fully implemented in CCCheck
  - Static checker for CodeContracts
  - Users: Professional programmers
- Array analysis completely transparent to users
  - No parameters to tweak, templates, partitions …
- Instantiated with
  - Expressions = Simple expressions (this talk)
  - Segments = Intervals + NotNull + Weak bounds
  - Environment = CCCheck default

Results

- Main .NET v2.0 Framework libraries
  - Un-annotated code

Assembly	# funcs	base	With functor	Δ	# array invariants
Mscorlib	21 475	4:06	4:15	**0:09**	**2 430**
System	15 489	3:40	3:46	**0:06**	**1 385**
System.Data	12 408	4:49	4:55	**0:06**	**1 325**
System.Drawings	3 123	0:28	0:29	**0:01**	**289**
System.Web	23 647	4:56	5:02	**0:06**	**840**
System.Xml	10 510	3:59	4:16	**0:17**	**807**

- Analyzes itself at each build (**0** warnings)
  - 5297 lines of annotated C#

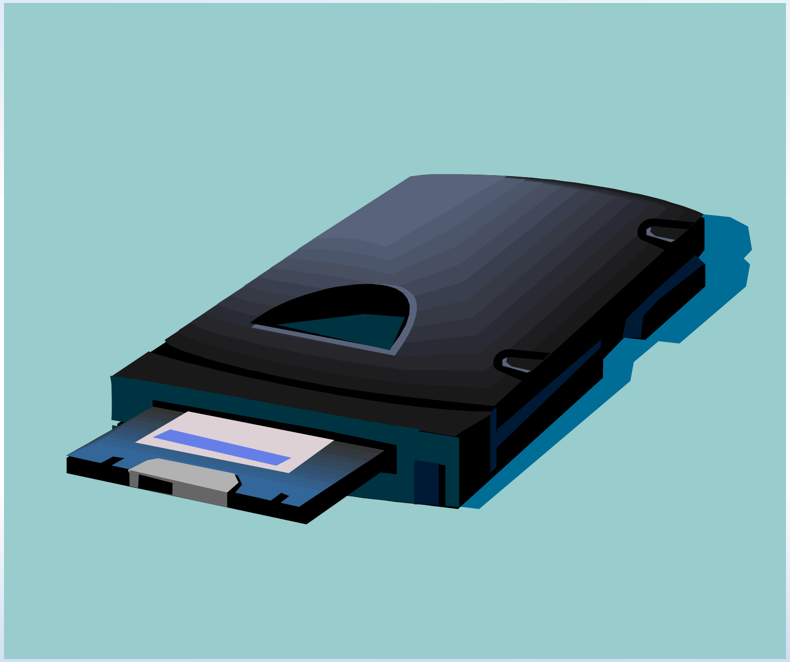
More?

- Inference of quantified preconditions
  - See our VMCAI'11 Paper
- Handling of multi-dimensional matrixes
  - With auto-application
- Inference of existential ∀∃ facts
  - When segments interpreted existentially
- Array purity check
  - The callee does not modify a sub-array
- …

To Sum up…

- Fully Automatic
  - Once the functor is instantiated
  - No hidden hypotheses
- Compact representation for disjunction
  - Enables Scalability
- Precision/Cost ratio tunable
  - Refine the functor parameters
  - Refine the scalar abstract environment
- Used everyday in an industrial analyzer
  - **1%** Overhead on average

Backup slides



Is this as Array Partitions?

- **No**
- [GRS05] and [HP07]
  - They require a pre-determined array partition
    - Main *weakness* of their approach
  - Our segmentation is inferred by the analysis
    - Totally automatic
  - They explicitly handle disjunctions
  - We have possibly empty segments

Calls

- Orthogonal issue
- In the implementation in CCCheck
  - Havoc arrays passed as parameters
  - Assignment of unknown if by ref of one element
  - Assume the postcondition
- Array element passed by ref
  - Ex: f(ref a[x])
  - The same as assignment a[x] = Top
  - Assume the postcondition

Multiple arrays as parameters

- Orthogonal issue
- Depends on the underlying heap analysis
- In CCCheck:
  - Optimistic hypotheses on non-aliasing
- FunArray easily fits in other heap models