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

Patrick Cousot, NYU & ENS
Radhia Cousot, CNRS & ENS & MSR
Francesco Logozzo, MSR
public void Init(int[] a)
{
    Contract.Requires(a.Length > 0);

    var j = 0;
    while (j < a.Length)
    {
        a[j] = 11;
        j++;
    }

    // here: \forall k.0 \leq k \lt j \Rightarrow a[k]=11
}

if j = 0 then
    a[0] ... not known
else if j > 0 \land j \leq 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?

Scalability

Automated

Function abstract domain

Automation

Array partitions

Template/annotation based

Theorem provers

Array expansion

Precision
Functor abstract domain by example
public void Init(int[] a) {
    Contract.Requires(a.Length > 0);
    var j = 0;
    while (j < a.Length) {
        a[j] = 11;
        j++;
    }
}
public void Init(int[] a) {
    Contract.Requires(a.Length > 0);
    var j = 0;
    while (j < a.Length) {
        a[j] = 11;
        j++;
    }
}
public void Init(int[] a)
{
    Contract.Requires(a.Length > 0);

    var j = 0;
    while (j < a.Length)
    {
        a[j] = 11;
        j++;
    }
}
public void Init(int[] a)
{
    Contract.Requires(a.Length > 0);

    var j = 0;

    while (j < a.Length)
    {
        a[j] = 11;
        j++;
    }
}
```csharp
public void Init(int[] a)
{
    Contract.Requires(a.Length > 0);

    var j = 0;

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

---

**Array assignment**

Materialize segment

Introduce ‘?’
public void Init(int[] a)
{
    Contract.Requires(a.Length > 0);

    var j = 0;
    while (j < a.Length)
    {
        a[j] = 11;
        j++;
        j++;
    }
}
public void Init(int[] a)
{
    Contract.Requires(a.Length > 0);

    var j = 0;
    while (j < a.Length)
    {
        a[j] = 11;
        j++;
    }
}
1. **Unify the segments**

\[
{0, j} \quad \text{Top} \quad \{a.\text{Length}\}
\]

\[
{0} \quad \bot \quad \{j\} \quad \text{Top} \quad \{a.\text{Length}\}
\]

2. **Point-wise join**

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

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

Similar for order, meet and widening
public void Init(int[] a)
{
    Contract.Requires(a.Length > 0);

    var j = 0;
    while (j < a.Length)
    {
        a[j] = 11;
        j++;
    }
}
```csharp
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 '?'
public void Init(int[] a)
{
    Contract.Requires(a.Length > 0);

    var j = 0;

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

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

    var j = 0;

    while (j < a.Length)
    {
        a[j] = 11;
        j++;
    }
}
public void Init(int[] a)
{
    Contract.Requires(a.Length > 0);

    var j = 0;
    while (j < a.Length)
    {
        a[j] = 11;
        j++;
    }
}
public void Init(int[] a)
{
    Contract.Requires(a.Length > 0);

    var j = 0;
    while (j < a.Length)
    {
        a[j] = 11;
        j++;
    }
}
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
}

Remove the empty segment
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
Sets of symbolic expressions

In our examples: \( \text{Exp} := k \mid x \mid x + k \)

Meaning:

\[
\begin{align*}
\{ e_0 \ldots e_n \} \{ e'_1 \ldots e'_m \} & \equiv e_0 = \ldots = e_n < e'_1 = \ldots = e'_m \\
\{ e_0 \ldots e_n \} \{ e'_1 \ldots e'_m \} & \equiv e_0 = \ldots = e_n \leq e'_1 = \ldots = e'_m
\end{align*}
\]

Possibly empty segments are key for scalability
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  V  
    0 ≤ j< b.Length-1  V  
    j = b.Length-1  V  
    j = b.Length
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++;
        }
    }
}

Segmentation discovered by the analysis
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 \([CC79]\)

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

| \(\{0\}\) | even \(\rightarrow\) 1 | odd \(\rightarrow\) -1 | \(\{i, n, a.Length\}\)? | \(i \in [0, +\infty)\) |
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: $x = a[\text{exp}]$

- **Search** the bounds for exp

The search queries the scalar environment $\sigma$

- More precision
- A form of abstract domains reduction

Set $\sigma' = \sigma \left[ x \mapsto A_n \sqcup \ldots \sqcup A_{m-1} \right]$
Write: $a[\text{exp}] = x$

- **Search** the bounds for exp

  | ... | ... | $B_n$ | $A_n$ | ... | $A_{m-1}$ | $B_m$ | ... | ... |

- **Join** the segments

  | ... | ... | $B_n$ | $A_n \cup \ldots \cup A_{m-1}$ | $B_m$ | ... | ... |

- **Split** the segment

  | ... | ... | $B_n$ | $A_n \cup \ldots \cup A_{m-1}$ | exp | $\sigma(x)$ | exp+1 | $A_n \cup \ldots \cup A_{m-1}$ | $B_m$ | ... | ... |

- **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** \( x \leq y \)
  - Does the state **implies** \( x \geq y \)?
  - 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

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

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** $\forall \exists$ 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
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
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