Array Region Analyses and Applications

Béatrice Creusillet

SILKAN

Polyhedral School
Saint-Germain-au-Mont-d’Or
May 13-17, 2013
SILKAN

- HPC Project: french company founded in 2007
  - Ronan Keryell was among the founders
- SILKAN: merge of HPC Project and Arion, summer 2012
- Subsidiaries in Canada and USA
- Develops and integrates simulation-based solution
- R&D branch around automatic HPC code generation
- Wild Cruncher appliance
  - Hardware part: Wild Node
  - COLD: Scilab/Matlab and Simulink/XCOS to C
  - Par4All: C to OpenMP/Cuda/OpenCL/...
Par4All 1.x

- Open source initiative funded by SILKAN
- Provides source-to-source compilers for various targets → OpenMP, Cuda, OpenCL, ...
- simple to use: `p4a --openmp sample.c`
- Powered by PIPS
Interprocedural Parallelizer of Scientific Programs

Open source project at Mines ParisTech, 25 years old, still vivid

Tool box: ≈ 300 tunable phases (parsers, semantical analyses, transformations, optimizations, parallelization, prettyprinters, ...)

Powerful semantical analyses

- Abstract interpretation
- Convex polyhedra
- Track modifications of variable values (transformers)
- Relations between variable values (preconditions)
- Array elements sets (array regions)

Relies on C³ linear library
Outline

Foundations
- Context/History
- Nature of array element sets?
- Approximations
- Read and Write Regions
- IN and OUT regions
- Extension to the C Language

APPLICATIONS
- Loop Transformations
- Memory Management
- Static Analysis

CONCLUSION
Array Regions : a Little Bit of History

Birth :

- 1984 : *Analyses de régions de tableaux*, PhD (Rémi Triolet) [Tri84]
- 1986 : *Direct Parallelization of Call Statements* (Triolet, Irigoin, Feautrier) [TFI86]
Array Regions : a Little Bit of History

Birth :
- 1984 : *Analyses de régions de tableaux*, PhD (Rémi Triolet) [Tri84]
- 1986 : *Direct Parallelization of Call Statements* (Triolet, Irigoin, Feautrier) [TFI86]

Formalization :
- 1996 : *Array Region Analyses and Applications* (Béatrice Creusillet) [CI96, Cre96]
Array Regions: a Little Bit of History

Birth:
- 1984: *Analyses de régions de tableaux*, PhD (Rémi Triolet) [Tri84]
- 1986: *Direct Parallelization of Call Statements* (Triolet, Irigoin, Feautrier) [TFI86]

Formalization:
- 1996: *Array Region Analyses and Applications* (Béatrice Creusillet) [CI96, Cre96]

Applications:
- 1996: HPF Compiler (Fabien Coelho) [Coe96]
- 2002: Software Verification (Nga Nguyen) [Ngu02]
- 2007-: Conversion from OpenMP to MPI, STEP (Telecom Sud Paris) [MMPSC09]
- 2009-: Extension to C, Par4All
  - Statement Isolation, redundant load/store optimization (Serge Guelton, 2011) [Gue11]
  - CUDA/OpenCL code generation (Mehdi Amini, 2012) [Ami12]
- 2012-: Task parallelism (Dounia Khaldi)
Source-to-source program transformations
- try to preserve original program characteristics
- keep high-level information

Better results if some coding rules are met \([\text{AAC}^{+}12]\)
Practical Context

- **Source-to-source** program transformations
  - try to preserve original program characteristics
  - keep high-level information
- **Coarse grain** program transformations
  - data and task parallelism
  - using *convex polyhedra*

- Better results if some coding rules are met [AAC+12]
Practical Context

- **Source-to-source** program transformations
  - try to preserve original program characteristics
  - keep high-level information

- **Coarse grain** program transformations
  - data and task parallelism
  - using **convex polyhedra**
  - but still enabling fine grain transformations!

- Better results if some coding rules are met [AAC+12]
Practical Context

- **Source-to-source** program transformations
  - try to preserve original program characteristics
  - keep high-level information

- **Coarse grain** program transformations
  - data and task parallelism
  - using **convex polyhedra**
  - but still enabling fine grain transformations!

- **Interprocedural** analyses ... and transformations
  - full inlining is costly
  - selective inlining, outlining, and cloning

- Better results if some coding rules are met [AAC+12]
Practical Context

- **Source-to-source** program transformations
  - try to preserve original program characteristics
  - keep high-level information
- **Coarse grain** program transformations
  - data and task parallelism
  - using **convex polyhedra**
  - but still enabling fine grain transformations!
- **Interprocedural** analyses ... and transformations
  - full inlining is costly
  - selective inlining, outlining, and cloning
- Input code it correct
- No other restriction on input code (well, almost!)
  - gotos, non-affine array indices, test conditions...
  - function calls (but no recursivity)

- Better results if some coding rules are met [AAC+12]
Practical Context

- **Source-to-source** program transformations
  - try to preserve original program characteristics
  - keep high-level information
- **Coarse grain** program transformations
  - data and task parallelism
  - using *convex polyhedra*
  - but still enabling fine grain transformations!
- **Interprocedural** analyses ... and transformations
  - full inlining is costly
  - selective inlining, outlining, and cloning
- Input code it correct
- No other restriction on input code (well, almost !)
  - gotos, non-affine array indices, test conditions...
  - function calls (but no recursivity)
  - \( \sim \) Approximations

- Better results if some coding rules are met [AAC+12]
Practical Context

- **Source-to-source** program transformations
  - try to preserve original program characteristics
  - keep high-level information
- **Coarse grain** program transformations
  - data and task parallelism
  - using *convex polyhedra*
  - but still enabling fine grain transformations!
- **Interprocedural** analyses ... and transformations
  - full inlining is costly
  - selective inlining, outlining, and cloning
- Input code it correct
- No other restriction on input code (well, almost!)
  - gotos, non-affine array indices, test conditions...
  - function calls (but no recursivity)
- *Approximations*
- Historically for *arrays*, extended to C data structures
- Better results if some coding rules are met [AAC\textsuperscript{+}12]
What are Array Regions?

- Sets of array elements
- Identified by their coordinates in \( \mathbb{Z}^d \):

\[
R(a[\phi_1][\phi_2]) = \{ \phi_1, \phi_2 \mid \phi_1 = 0 \land 0 \leq \phi_2 \leq 3 \}
\]
What are Array Regions?

- Sets of array elements
- Identified by their coordinates in $\mathbb{Z}^d$:

$$\mathcal{R}(a[\phi_1][\phi_2]) = \{\phi_1, \phi_2 | \phi_1 = 0 \land 0 \leq \phi_2 \leq 3\}$$

- Propagated along the program representation (HCFG)
- Depending on values of program variables in current memory state

// $\sigma_1 : \mathcal{W}(a[\phi_1]) = \{\phi_1 | \phi_1 = \sigma_1(k)\}$

\[
a[k] = \ldots;
\]

// $\sigma_2$

\[
k = k + 1;
\]

// $\sigma_3 : \mathcal{W}(a[\phi_1]) = \{\phi_1 | \phi_1 = \sigma_3(k)\}$

\[
a[k] = \ldots;
\]
What are Array Regions?

- Sets of array elements
- Identified by their coordinates in $\mathbb{Z}^d$:

$$\mathcal{R}(a[\phi_1][\phi_2]) = \{\phi_1, \phi_2 \mid \phi_1 = 0 \land 0 \leq \phi_2 \leq 3\}$$

- Propagated along the program representation (HCFG)
- Depending on values of program variables in current memory state:

```cpp
// σ_1 : W(a[φ_1]) = {φ_1 \mid φ_1 = σ_1(k)}
a[k] = ...;
// σ_2
k = k + 1;
// σ_3 : W(a[φ_1]) = {φ_1 \mid φ_1 = σ_3(k)}
a[k] = ...;
```

- $\mathcal{R} : P \rightarrow (\Sigma \rightarrow \wp(\mathbb{Z}^d))$ where:
  - $P$ : program (or a subpart)
  - $\Sigma$ : $\text{Id} \rightarrow \text{Val}$ : memory states
Exact solutions are not always computable

```c
srand(time(NULL));
for (int i = rand(); i < 10; i++)
{
    a[i] = i;
}
```
Computability

- Exact solutions are not always computable

```c
srand(time(NULL));
for (int i = rand(); i < 10; i++)
{
    a[i] = i;
}
```

- or representable with a reasonable complexity

```c
```
Computability

- Exact solutions are not always computable

```c
srand(time(NULL));
for (int i = rand(); i < 10; i++)
{
    a[i] = i;
}
```

- or representable with a reasonable complexity

```c
```

- Solutions:
  1. Restrict the input language
  2. Use approximations
      ⇒ to fit a restricted abstract domain $\varphi(\mathbb{Z}^n)$
Choosing an Abstract Domain

- Trade-off between precision/complexity
- Class of input programs (restrictions?)
- Interactions with prior/client analyses
  - reuse results of prior analyses?
  - which precision for client analyses?
  - delay part of the work to them, or not?
- Memory footprint of representation
- Complexity of operators (copy, feasibility (emptiness), union, intersection, projection, difference, translation across memory spaces)
- Several domains used in related tools (RSDs, Gated RSDs, DADs, lists of RSDs, of polyhedra, . . .)
Convex Polyhedra over $\mathbb{Z}$: Pros and Cons

- compact representation $\{\phi_1 \mid 1 \leq \phi_1 \leq 10000\}$
- symbolic values allowed $\{\phi_1 \mid 1 \leq \phi_1 \leq N\}$
- embeds some context information $\{\phi_1 \mid \phi_1 = i, 1 \leq i \leq N, k = M\}$
- easy to combine with other polyhedral analyses results
- easy to use for dependence analysis and other applications
- complexity
  - intersection: cheap
  - convex hull, feasibility, projection: can be expensive
  - depends on implementation
  - depends on the amount of available information
- limited to affine expressions
- some frequent access patterns cannot be represented
We Need Over-Approximations

Example: coarse grain parallelization

```c
void foo(int i, int n, int src[n], int dest[n]) {
    for(int k = 0; k < n; k += 2)
        dest[k] = src[i] * k;
}

void bar(int a[10], int b[10][10]) {
    for (int i = 0; i < 10; i++)
        {
            a[i] = i;
            foo(i, 10, a, b[i]);
        }
}
```
We Need Over-Approximations

Example: coarse grain parallelization

```c
void foo(int i, int n, int src[n], int dest[n]) {
    for(int k = 0; k < n; k += 2)
        dest[k] = src[i] * k;
}

void bar(int a[10], int b[10][10]) {
    for (int i = 0; i < 10; i++)
        for(int k = 0; k < n; k += 2)
            dest[k] = src[i] * k;
}
```
We Need Over-Approximations

Example: coarse grain parallelization

```c
// R(src[\phi_1]) = \{ \phi_1 \mid \phi_1 = i, n = 10, 0 \leq i \leq 9 \} >
// W(dest[\phi_1]) = \{ \phi_1 \mid 0 \leq \phi_1 \leq 8, n = 10, 0 \leq i \leq 9 \} >
void foo(int i, int n, int src[n], int dest[n]) {
    for(int k = 0; k < n; k += 2)
        dest[k] = src[i] * k;
}

void bar(int a[10], int b[10][10]) {
    #pragma omp parallel for
    for (int i = 0; i < 10; i++)
        // R(a[\phi_1]) = \{ \phi_1 \mid \phi_1 = i, 0 \leq i \leq 9 \} >
        // W(a[\phi_1]) = \{ \phi_1 \mid \phi_1 = i, 0 \leq i \leq 9 \} >
        // W(b[\phi_1]) = \{ \phi_1 \mid 0 \leq \phi_1 \leq 8, 0 \leq i \leq 9 \} >
        {a[i] = i;
        // R(a[\phi_1]) = \{ \phi_1 \mid \phi_1 = i, 0 \leq i \leq 9 \} >
        // W(b[\phi_1][\phi_2]) = \{ \phi_1, \phi_2 \mid \phi_1 = i, 0 \leq \phi_2 \leq 8, 0 \leq i \leq 9 \}
        foo(i, 10, a, b[i]);
    }
}
We also Need Under-Approximations

Example: studying flow of array elements

```c
void foo(int a[10], int b[10][10]) {

    for (int i = 0; i < 10; i++)
    {

        for (int j = 0; j < 10; j++)
            if (cond(j))
                a[j] = f(j);

        ...

    for (int j = 0; j < 10; j++)
        b[i][j] = g(a[j]);

    }
}
```
We also Need Under-Approximations

Example: studying flow of array elements

```c
void foo(int a[10], int b[10][10]) {

    for (int i = 0; i < 10; i++) {
        // \( \overline{W}(a[\phi_1]) = \{ \phi_1 \mid 0 \leq \phi_1 \leq 9, 0 \leq i \leq 9 \} \)
        for (int j = 0; j < 10; j++)
            if (cond(j))
                a[j] = f(j);

        // \( \mathcal{R}(a[\phi_1]) = \{ \phi_1 \mid 0 \leq \phi_1 \leq 9, 0 \leq i \leq 9 \} \)
        for (int j = 0; j < 10; j++)
            b[i][j] = g(a[j]);
    }
}
```
We also Need Under-Approximations

Example: studying flow of array elements

```c
void foo(int a[10], int b[10][10]) {
    for (int i = 0; i < 10; i++) {
        // \( \overline{W}(a[\phi_1]) = \{ \phi_1 \mid 0 \leq \phi_1 \leq 9, 0 \leq i \leq 9 \} \)
        for (int j = 0; j < 10; j++)
            if (cond(j))
                a[j] = f(j);
        ...  
        // \( \mathcal{R}(a[\phi_1]) = \{ \phi_1 \mid 0 \leq \phi_1 \leq 9, 0 \leq i \leq 9 \} \)
        for (int j = 0; j < 10; j++)
            b[i][j] = g(a[j]);
    }
}
```

Computing set of elements which might be read before being written:

\[ \mathcal{R}(a[\phi_1]) - \overline{W}(a[\phi_1]) = \emptyset \]

\[ \rightarrow \text{not intuitively expected result!} \]
$\mathcal{R}_1$
Approximations and Differences

$\mathcal{R}_1$

$\mathcal{R}_2$
Approximations and Differences

$\mathcal{R}_1$ vs $\mathcal{R}_2$
Approximations and Differences

$\mathcal{R}_1$  

$\mathcal{R}_2$  

$\mathcal{R}_1 - \mathcal{R}_2$
Approximations and Differences

$\mathcal{R}_1 \setminus \mathcal{R}_2$
Approximations and Differences

\[ \mathcal{R}_1 - \mathcal{R}_2 \]

\[ \overline{\mathcal{R}_1} - \overline{\mathcal{R}_2} \]
Approximations and Differences

$R_1 - R_2$

$R_1 - \overline{R}_2$
Approximations and Differences

\[ \mathcal{R}_1 - \mathcal{R}_2 \]

\[ \overline{\mathcal{R}}_1 - \overline{\mathcal{R}}_2 \]
Approximations and Differences

\[ \mathcal{R}_1 \]
\[ \mathcal{R}_2 \]
\[ \mathcal{R}_1 - \mathcal{R}_2 \]
\[ \overline{\mathcal{R}_1} - \overline{\mathcal{R}_2} \]
Issues with Under-Approximations

- Exact set
- Under-approximations
- Convex polyhedra
- Lists of polyhedra
- RSDs

There is no better under-approximation, \([\text{is not associative}]\)

Influence of the computation order!

Recursive functions are not continuous!

\[ \text{least fixed points not defined} \]
Issues with Under-Approximations

- convex polyhedra
- lists of polyhedra
- RSDs

There is no better under-approximation, \( \cup \) is not associative

- Influence of the computation order
  - \( \rightarrow \) unpredictable results

- Recursive functions are not continuous
  - \( \rightarrow \) least fixed points not defined
Existing solutions:

- Choose a new representation → may end with the whole program!
- Restraine the input language
- Use narrowing operators to under-approximate fix points → safe but provides imprecise solutions
- Dynamic partitioning (F. Bourdoncle) → chose the lattice on the fly: too costly
- (U)TVPI sub-polyhedral under-approximations (R. Upadrasta & A. Cohen)
Existing solutions:

- Choose a new representation
  → may end with the whole program!
- Restrain the input language
- Use narrowing operators to under-approximate fix points
  → safe but provides imprecise solutions
- Dynamic partitioning (F. Bourdoncle)
  → chose the lattice on the fly: too costly
- (U)TVPI sub-polyhedral under-approximations (R. Upadrasta & A. Cohen)

Practical solution implemented in PIPS:

\[ \mathcal{R} = \begin{cases} \overline{\mathcal{R}} & \text{if } \overline{\mathcal{R}} \equiv \mathcal{R} \\ \emptyset & \text{otherwise} \end{cases} \]
Solutions

- Existing solutions:
  - Choose a new representation → may end with the whole program!
  - Restrain the input language
  - Use narrowing operators to under-approximate fix points → safe but provides imprecise solutions
  - Dynamic partitioning (F. Bourdoncle) → chose the lattice on the fly: too costly
  - (U)TVPI sub-polyhedral under-approximations (R. Upadrasta & A. Cohen)

- Practical solution implemented in PIPS:
  \[
  \mathcal{R} = \begin{cases} 
  \overline{\mathcal{R}} & \text{if } \overline{\mathcal{R}} \equiv \mathcal{R} \\
  \emptyset & \text{otherwise}
  \end{cases}
  \]
  but $\mathcal{R}$ is a priori unknown!
Solutions

- **Existing solutions**:
  - Choose a new representation
    \(\rightarrow\) may end with the whole program!
  - Restrain the input language
  - Use narrowing operators to under-approximate fix points
    \(\rightarrow\) safe but provides imprecise solutions
  - Dynamic partitioning (F. Bourdoncle)
    \(\rightarrow\) chose the lattice on the fly: too costly
  - (U)TVPI sub-polyhedral under-approximations (R. Upadrasta & A. Cohen)

- **Practical solution implemented in PIPS**:

  \[ \mathcal{R} = \begin{cases} \overline{\mathcal{R}} & \text{if } \overline{\mathcal{R}} \equiv \mathcal{R} \\ \emptyset & \text{otherwise} \end{cases} \]

- Clear separation between exact semantics and approximations
Exactness of Array Regions: References and Intersection

- **Boostrrapping**: with array references
  - criterion: all indices are affine expressions of program variables

\[
\begin{align*}
a[2*i+3] & \quad \overline{R} = \{\phi_1 \mid \phi_1 = 2 \times \sigma(i) + 3\} = R \\
a[2*i*n][j] & \quad \overline{R} = \{\phi_1, \phi_2 \mid \phi_2 = \sigma(j)\} \quad R = \emptyset \\
a[b[i]] & \quad \overline{R} = \{\phi_1 \mid \} \quad R = \emptyset
\end{align*}
\]
Exactness of Array Regions: References and Intersection

- **Boostrapping**: with array references
  - criterion: all indices are affine expressions of program variables
    
    \[
    \begin{align*}
    a[2*i+3] & : \overline{R} = \{\phi_1 \mid \phi_1 = 2 \times \sigma(i) + 3\} = R \\
    a[2*i*n][j] & : \overline{R} = \{\phi_1, \phi_2 \mid \phi_2 = \sigma(j)\} \quad R = \emptyset \\
    a[b[i]] & : \overline{R} = \{\phi_1 \mid \} \quad R = \emptyset
    \end{align*}
    \]

- **Intersection**: exact operation!
  
  \[
  \overline{R}_1 = \overline{R}_1 \land \overline{R}_2 = \overline{R}_2 \implies \overline{R}_1 \cap \overline{R}_2 = \overline{R}_1 \cap \overline{R}_2
  \]
or Fourier-Motzkin variable elimination
If
\[ S' = \begin{cases} 
  bA + aB \leq 0 
\end{cases} \]

is the system resulting from the elimination of the variable \( v \) from
\[ S = \begin{cases} 
  av + A \leq 0 \\
  -bv + B \leq 0 
\end{cases} \]

by Fourier-Motzkin algorithm. Then \( S' \) has the same solutions as the exact projection of \( S \) along \( v \) if and only if the equation:
\[ aB + bA + ab - a - b + 1 \leq 0 \]
is redundant with \( S \).

\[ \rightarrow \text{amounts to a feasibility test.} \]
Disjunctive Normal Forms and Conjuncto-Negative Forms

Presburger formula

- a first order predicate over \( \mathbb{Z} \).
- two equivalent representations:
  - **DNF** \( F = \{ x \in \mathbb{Z}^m : \bigvee_i P_i(x) \} \)
  - **CNF** \( F = \{ x \in \mathbb{Z}^m : P_o(x) \land (\land_i \neg P_i(x)) \} \)

where \( P_i(x) \) are convex polyhedra over \( \mathbb{Z}^m \)
Disjunctive Normal Forms and Conjuncto-Negative Forms

Presburger formula

- a first order predicate over $\mathbb{Z}$.
- two equivalent representations:
  - DNF: $F = \{ x \in \mathbb{Z}^m : \bigvee_i P_i(x) \}$
  - CNF: $F = \{ x \in \mathbb{Z}^m : P_o(x) \land (\land_i \neg P_i(x)) \}$

where $P_i(x)$ are convex polyhedra over $\mathbb{Z}^m$

- Leservot: algorithm to find shortest DNF from a CNF
Disjunctive Normal Forms and Conjuncto-Negative Forms

Presburger formula

- a first order predicate over $\mathbb{Z}$.
- two equivalent representations:
  - **DNF** $F = \{x \in \mathbb{Z}^m : \vee_i P_i(x)\}$
  - **CNF** $F = \{x \in \mathbb{Z}^m : P_o(x) \land (\land_i \neg P_i(x))\}$
  where $P_i(x)$ are convex polyhedra over $\mathbb{Z}^m$
- Leservot : algorithm to find shortest DNF from a CNF

→ feasibility test of a CNF : conversion to a DNF
Exactness of Array Regions: Union

$I \cup R$ is over-approximated by $R = \text{convex hull}(R_1, R_2) = R_1 \cup R_2$.

$I \cup (R_1 \cup R_2) \cap R_1^\complement \cap R_2^\complement$ is not feasible.

Feasibility test of CNF (conversion to equivalent DNF).
Exactness of Array Regions : Union

- Over-approximated by \( \overline{R} = \text{convex}_\text{hull}(R_1, R_2) = R_1 \bigcup \overline{R_2} \)
Exactness of Array Regions: Union

- Over-approximated by $\overline{R} = \text{convex}_\text{hull}(R_1, R_2) = R_1 \bigcup R_2$
- $\overline{R} \equiv R_1 \bigcup R_2 \iff \overline{R} \land \neg R_1 \land \neg R_2$ is not feasible
Exactness of Array Regions : Union

- Over-approximated by $\overline{R} = \text{convex}_\text{hull}(R_1, R_2) = R_1 \bigcup R_2$
- $\overline{R} \equiv R_1 \bigcup R_2 \iff \overline{R} \land \neg R_1 \land \neg R_2$ is not feasible
Exactness of Array Regions: Union

- Over-approximated by $\overline{R} = \text{convex}_\text{hull}(R_1, R_2) = R_1 \bigcup R_2$
- $\overline{R} \equiv R_1 \bigcup R_2 \iff \overline{R} \land \neg R_1 \land \neg R_2$ is not feasible
- feasibility test of CNF (conversion to equivalent DNF)
Exactness of Array Regions: Difference

- Equivalent to a CNF: $\mathcal{R}_1 - \mathcal{R}_2 = \mathcal{R}_1 \land \neg \mathcal{R}_2$
- Conversion to shortest DNF: $\mathcal{R}_1 - \mathcal{R}_2 = \bigvee_{k=1}^n P_i$

  if $n \equiv 1$ then $\overline{\mathcal{R}_1 - \mathcal{R}_2} = P_1$ is exact

  else $\overline{\mathcal{R}_1 - \mathcal{R}_2} = \mathcal{R}_1 \mathcal{R}_2 = \text{convex_hull}(P_1, \ldots, P_n)$
Read and Write Regions

- Summarize read and write memory effects of expressions and statements

- Functions of memory state

\[
\mathcal{R} : S \oplus E \rightarrow (\Sigma \rightarrow \prod_{i=1,n} \varphi(\mathbb{Z}^{d_i}))
\]

\[
\mathcal{W} : S \oplus E \rightarrow (\Sigma \rightarrow \prod_{i=1,n} \varphi(\mathbb{Z}^{d_i}))
\]

\[
\overline{\mathcal{R}} : S \oplus E \rightarrow (\Sigma \rightarrow \prod_{i=1,n} \tilde{\varphi}(\mathbb{Z}^{d_i}))
\]

\[
\overline{\mathcal{W}} : S \oplus E \rightarrow (\Sigma \rightarrow \prod_{i=1,n} \tilde{\varphi}(\mathbb{Z}^{d_i}))
\]

- Notation for a given element \( l \) of the language : \( \mathcal{R}[l] \)

- Bootstrap with individual memory cells references

- Simple expressions, sequences of instructions, tests, Fortran DO-loops
Expressions with no Side-effets: Prerequisites

Evaluation of Expressions: $\mathcal{E}$

$$\mathcal{E} : E \rightarrow (\Sigma \rightarrow V)$$

Characteristic function of $E_B : \mathcal{E}_c$

$$\mathcal{E}_c : E_B \rightarrow (\Sigma \rightarrow \Sigma)$$

$$\text{exp} \rightarrow \mathcal{E}_c[\text{exp}] = \lambda \sigma. \text{if } \mathcal{E}[\text{exp}]\sigma \text{ then } \sigma \text{ else } \bot$$

We suppose its approximations to be conveniently defined:

$$\overline{\mathcal{E}}_c : E_B \rightarrow (\Sigma \rightarrow \Sigma)$$

$$\underline{\mathcal{E}}_c : E_B \rightarrow (\Sigma \rightarrow \Sigma)$$

and:

$$\mathcal{E}_c[\text{exp}]\sigma = \sigma \implies \overline{\mathcal{E}}_c[\text{exp}]\sigma = \sigma$$

$$\underline{\mathcal{E}}_c[\text{exp}]\sigma = \sigma \implies \mathcal{E}_c[\text{exp}]\sigma = \sigma$$
Expressions with no Side-effets: Read Regions

Exact Regions

\[ R[c] = \lambda \sigma. \emptyset \]
\[ R[var] = \lambda \sigma. \{ \{ \text{var} \} \} \]
\[ R[var[exp_1] \ldots [exp_k]] = \lambda \sigma. \{ \{ \text{var} [E[exp_1] \sigma] \ldots [E[exp_k] \sigma] \} \} \]

\[ R[op_1 exp] = R[exp] \]
\[ R[exp_1 op_2 exp_2] = R[exp_1] \cup R[exp_2] \]
\[ R[exp_1 && exp_2] = R[exp_1] \cup (R[exp_2] \circ E[exp_1]) \]
\[ R[exp_1 || exp_2] = R[exp_1] \cup (R[exp_2] \circ \neg E[exp_1]) \]
\[ R[exp_1, \ldots, exp_k] = \bigcup_{i=1}^{k} R[exp_i] \]
Expressions with no Side-effets : Read Regions

Over-approximations

\[
\overline{R}[\text{op}_1 \text{exp}] = \overline{R}[\text{exp}]
\]
\[
\overline{R}[\text{exp}_1 \text{op}_2 \text{exp}_2] = \overline{R}[\text{exp}_1] \cup \overline{R}[\text{exp}_2]
\]
\[
\overline{R}[\text{exp}_1 \&\& \text{exp}_2] = \overline{R}[\text{exp}_1] \cup (\overline{R}[\text{exp}_2] \overline{\circ} \overline{E}[\text{exp}_1])
\]
\[
\overline{R}[\text{exp}_1 \parallel \text{exp}_2] = \overline{R}[\text{exp}_1] \cup (\overline{R}[\text{exp}_2] \overline{\circ} \neg E[\text{exp}_1])
\]
\[
\overline{R}[\text{exp}_1, \ldots, \text{exp}_k] = \bigcup_{i=1}^{k} \overline{R}[\text{exp}_i]
\]

Under-approximations are symmetrically defined.
Expressions with no Side-effets: Read Regions

Over-approximations

\[ \overline{R}[\text{op}_1 \ \text{exp}] = \overline{R}[^{}\text{exp}^{}] \]
\[ \overline{R}[\text{exp}_1 \ \text{op}_2 \ \text{exp}_2] = \overline{R}[\text{exp}_1] \cup \overline{R}[\text{exp}_2] \]
\[ \overline{R}[\text{exp}_1 && \text{exp}_2] = \overline{R}[\text{exp}_1] \cup (\overline{R}[\text{exp}_2] \overline{\overline{\epsilon}} \overline{[\text{exp}_1]}) \]
\[ \overline{R}[\text{exp}_1 || \text{exp}_2] = \overline{R}[\text{exp}_1] \cup (\overline{R}[\text{exp}_2] \overline{\overline{\epsilon}} \neg \epsilon [\text{exp}_1]) \]
\[ \overline{R}[\text{exp}_1, \ldots, \text{exp}_k] = \bigcup_{i=1}^{k} \overline{R}[\text{exp}_i] \]

Under-approximations are symmetrically defined.

```c
// \( R(a[\phi_1]) = \{ \phi_1 \mid \phi_1 = i \} \) >
// \( \overline{R}(b[\phi_1]) = \{ \phi_1 \mid \phi_1 = i \} \) >
res = foo(a[i]) && bar(b[i]);
```
Assignment of an Array Element

\[
\mathcal{R}[\text{var}[\text{exp}_1] \ldots [\text{exp}_k] = \text{exp}] = \left( \bigcup_{i=1}^{k} \mathcal{R}[\text{exp}_i] \right) \cup \mathcal{R}[\text{exp}]
\]

\[
\mathcal{W}[\text{var}[\text{exp}_1] \ldots [\text{exp}_k] = \text{exp}] = \mathcal{W}[\text{var}[\text{exp}_1] \ldots [\text{exp}_k]]
\]

Approximations

\[
\overline{\mathcal{R}}[\text{var}[\text{exp}_1] \ldots [\text{exp}_k] = \text{exp}] = \bigcup_{i=1}^{k} \overline{\mathcal{R}}[\text{exp}_i] \cup \overline{\mathcal{R}}[\text{exp}]
\]

\[
\overline{\mathcal{W}}[\text{var}[\text{exp}_1] \ldots [\text{exp}_k] = \text{exp}] = \overline{\mathcal{W}}[\text{var}[\text{exp}_1] \ldots [\text{exp}_k]]
\]

\[
\mathcal{R}[\text{var}[\text{exp}_1] \ldots [\text{Exp}_k] = \text{exp}] = \bigcup_{i=1}^{k} \mathcal{R}[\text{exp}_i] \cup \mathcal{R}[\text{exp}]
\]

\[
\mathcal{W}[\text{var}[\text{exp}_1] \ldots [\text{Exp}_k] = \text{exp}] = \mathcal{W}[\text{var}[\text{exp}_1] \ldots [\text{Exp}_k]]
\]
// \( \sigma_1 \)
{

    k = k + 1;

    // \( \sigma_2 \)
    a[k] = ...;

}
Sequence of Statements: Need for Transformers

// σ₁
{
    k = k + 1;
    // σ₂: \( W(a[\phi_1]) = \{ \phi_1 \mid \phi_1 = \sigma_2(k) \} \)
    a[k] = ...;
}

---

Béatrice Creusillet (SILKAN)
Sequence of Statements : Need for Transformers

// σ₁
{
// Τ : σ₂(k) = σ₁(k) + 1
    k = k + 1;
// σ₂ : \mathcal{W}(a[φ₁]) = \{φ₁ | φ₁ = σ₂(k)\}
    a[k] = \ldots;
}

- Need to model memory state modifications : Transformers

\[ \mathcal{T} : P \rightarrow (\Sigma \rightarrow \Sigma) \]
Need to model memory state modifications: Transformers

\[ T : P \rightarrow (\Sigma \rightarrow \Sigma) \]

Regions of a sequence of instructions:

\[ R[S_1; S_2;] = R[S_1] \cup (R[S_2] \circ T[S_1]) \]
Sequence of Statements : Need for Transformers

```c
// \sigma_1 : W(a[\phi_1]) = \{ \phi_1 \mid \phi_1 = \sigma_1(k) + 1 \}
{
// \mathcal{T} : \sigma_2(k) = \sigma_1(k) + 1
    k = k + 1;
// \sigma_2 : W(a[\phi_1]) = \{ \phi_1 \mid \phi_1 = \sigma_2(k) \}
    a[k] = \ldots ;
}

- Need to model memory state modifications : Transformers

\mathcal{T} : P \rightarrow (\Sigma \rightarrow \Sigma)

- Regions of a sequence of instructions :

\mathcal{R}[S_1; S_2] = \mathcal{R}[S_1] \cup (\mathcal{R}[S_2] \circ \mathcal{T}[S_1])

- Approximations :

\overline{\mathcal{R}}[S_1; S_2] = \overline{\mathcal{R}}[S_1] \cup (\overline{\mathcal{R}}[S_2] \cdot \overline{\mathcal{T}}[S_1])
\underline{\mathcal{R}}[S_1; S_2] = \underline{\mathcal{R}}[S_1] \cup (\underline{\mathcal{R}}[S_2] \cdot \underline{\mathcal{T}}[S_1])
Transformers : Over-Approximations

- Transformers are not always computable
- Transformers are not always computable
- Transformers as relations:

$$k = k + 1; \quad t(\sigma_1, \sigma_2) = \{\sigma_1, \sigma_2 \mid \forall \nu \neq k, \sigma_2(\nu) = \sigma_1(\nu) \land \sigma_2(k) = \sigma_1(k) + 1\}$$

$$k = \text{rand}(); \quad \bar{t}(\sigma_1, \sigma_2) = \{\sigma_1, \sigma_2 \mid \forall \nu \neq k, \sigma_2(\nu) = \sigma_1(\nu)\}$$
Transformers are not always computable

Transformers as relations:

\[ k = k + 1; \quad t(\sigma_1, \sigma_2) = \{ \sigma_1, \sigma_2 \mid \forall v \neq k, \sigma_2(v) = \sigma_1(v), \sigma_2(k) = \sigma_1(k) + 1 \} \]

\[ T(k) \{ k = k # \text{init} + 1 \} \]

\[ k = \text{rand}(); \quad \bar{t}(\sigma_1, \sigma_2) = \{ \sigma_1, \sigma_2 \mid \forall v \neq k, \sigma_2(v) = \sigma_1(v) \} \]

\[ T(k) \{ \} \]
Transformers: Over-Approximations

- Transformers are not always computable
- Transformers as relations:
  \[ t(\sigma_1, \sigma_2) = \{ \sigma_1, \sigma_2 \mid \forall \nu \neq k, \sigma_2(\nu) = \sigma_1(\nu) \wedge \sigma_2(k) = \sigma_1(k) + 1 \} \]

- Two over-approximations:
  \[ T : P \rightarrow (\Sigma \rightarrow \varnothing(\Sigma)) \]
  \[ p \rightarrow \lambda \sigma_2.\{\sigma_1 \mid t(\sigma_1, \sigma_2)\} \]
  \[ T^{-1} : P \rightarrow (\Sigma \rightarrow \varnothing(\Sigma)) \]
  \[ p \rightarrow \lambda \sigma_1.\{\sigma_2 \mid \overline{t}(\sigma_1, \sigma_2)\} \]
Transformers : Over-Approximations

- Transformers are not always computable
- Transformers as relations:
  \[ k = k + 1; \quad t(\sigma_1, \sigma_2) = \{ \sigma_1, \sigma_2 \mid \forall \nu \neq k, \sigma_2(\nu) = \sigma_1(\nu) \land \sigma_2(k) = \sigma_1(k) + 1 \} \]

  \[ k = \text{rand}(); \quad \tilde{t}(\sigma_1, \sigma_2) = \{ \sigma_1, \sigma_2 \mid \forall \nu \neq k, \sigma_2(\nu) = \sigma_1(\nu) \} \]

- Two over-approximations:
  \[ \overline{T} : P \longrightarrow (\Sigma \longrightarrow \wp(\Sigma)) \]
  \[ p \longrightarrow \lambda \sigma_2.\{ \sigma_1 \mid \tilde{t}_l(\sigma_1, \sigma_2) \} \]
  \[ \overline{T}^{-1} : P \longrightarrow (\Sigma \longrightarrow \wp(\Sigma)) \]
  \[ p \longrightarrow \lambda \sigma_1.\{ \sigma_2 \mid \tilde{t}_l(\sigma_1, \sigma_2) \} \]

- Issue: target set is \( \wp(\Sigma) \) but regions are defined on \( \Sigma \)
  \[ \longrightarrow \text{need to define the composition laws } \bullet \text{ and } \circ \]
Composing Regions with Transformers

Over-approximation

\[ \overline{R \bullet T} = \lambda \sigma. \bigcup_{\sigma' \in \overline{T}(\sigma)} \overline{R}(\sigma') \]
Composing Regions with Transformers

Over-approximation

\[
\overline{\mathcal{R}} \circ \overline{T} = \lambda \sigma. \bigcup_{\sigma' \in \overline{T}(\sigma)} \overline{\mathcal{R}}(\sigma') = \lambda \sigma. \bigcup_{\sigma' \in \overline{T}(\sigma)} \{ \Phi \mid \overline{r}(\Phi, \sigma') \}
\]
Composing Regions with Transformers

Over-approximation

\[
\overline{R} \circ \overline{T} = \lambda \sigma. \bigcup_{\sigma' \in \overline{T}(\sigma)} \overline{R}(\sigma') = \lambda \sigma. \bigcup_{\sigma' \in \overline{T}(\sigma)} \{ \Phi \mid \overline{r}(\Phi, \sigma') \}
\]

\[
= \lambda \sigma. \text{proj}_{\sigma'} \left( \{ \Phi \mid \overline{t}(\sigma, \sigma') \land \overline{r}(\Phi, \sigma') \} \right)
\]
Composing Regions with Transformers

Over-approximation

\[
\bar{R} \circ \bar{T} = \lambda \sigma. \bigcup_{\sigma' \in \bar{T}(\sigma)} \bar{R}(\sigma') = \lambda \sigma. \bigcup_{\sigma' \in \bar{T}(\sigma)} \{\Phi \mid \bar{r}(\Phi, \sigma')\}
\]

\[
= \lambda \sigma. \text{proj}_{\sigma'}(\{\Phi \mid \bar{t}(\sigma, \sigma') \land \bar{r}(\Phi, \sigma')\})
\]

\[
= \lambda \sigma. \text{proj}_{\sigma'}(\{\Phi \mid \bar{t}(\sigma, \sigma')\} \cap \{\Phi : \bar{r}(\Phi, \sigma')\})
\]

\[\rightarrow \text{intersection followed by a projection}\]
Composing Regions with Transformers

Over-approximation

$$\overline{\mathcal{R}} \circ \overline{T} = \lambda \sigma. \bigcup_{\sigma' \in \overline{T}(\sigma)} \mathcal{R}(\sigma') = \lambda \sigma. \bigcup_{\sigma' \in \overline{T}(\sigma)} \{ \Phi | \overline{t}(\Phi, \sigma') \}$$

$$= \lambda \sigma. \text{proj}_{\sigma'}(\{ \Phi | \overline{t}(\sigma, \sigma') \land \overline{r}(\Phi, \sigma') \})$$

$$= \lambda \sigma. \text{proj}_{\sigma'}(\{ \Phi | \overline{t}(\sigma, \sigma') \} \cap \{ \Phi : \overline{r}(\Phi, \sigma') \})$$

$$\rightarrow$$ intersection followed by a projection

Under-approximation

- $\mathcal{R}$ is composed with $\overline{T}$ and not $T$
  $$\rightarrow \overline{\mathcal{R}} \circ \overline{T}$$ cannot be defined by exactness criterion of $\overline{\mathcal{R}} \circ \overline{T}$
Composing Regions with Transformers

Over-approximation

\[
\overline{R} \bullet \overline{T} = \lambda \sigma. \bigcup_{\sigma' \in \overline{T}(\sigma)} \overline{R}(\sigma') = \lambda \sigma. \bigcup_{\sigma' \in \overline{T}(\sigma)} \{ \Phi \mid \overline{r}(\Phi, \sigma') \}
\]

\[
= \lambda \sigma. \text{proj}_{\sigma'} \left( \{ \Phi \mid \overline{t}(\sigma, \sigma') \land \overline{r}(\Phi, \sigma') \} \right)
\]

\[
= \lambda \sigma. \text{proj}_{\sigma'} \left( \{ \Phi \mid \overline{t}(\sigma, \sigma') \} \cap \{ \Phi : \overline{r}(\Phi, \sigma') \} \right)
\]

\[
\rightarrow \text{intersection followed by a projection}
\]

Under-approximation

- \( \overline{R} \) is composed with \( \overline{T} \) and not \( \overline{T} \)
  \[
  \rightarrow \overline{R} \bullet \overline{T} \text{ cannot be defined by exactness criterion of } \overline{R} \bullet \overline{T}
  \]

- Define \( \overline{R} \bullet \overline{T} \) and derive an exactness test
  \[
  \overline{R} \bullet \overline{T} = \lambda \sigma. \bigcap_{\sigma' \in \overline{T}(\sigma)} \overline{R}(\sigma')
  \]
Composing Regions with Transformers

Over-approximation

\[ \overline{R} \cdot \overline{T} = \lambda \sigma. \bigcup_{\sigma' \in \overline{T}(\sigma)} \overline{R}(\sigma') = \lambda \sigma. \bigcup_{\sigma' \in \overline{T}(\sigma)} \{ \Phi \mid \overline{r}(\Phi, \sigma') \} \]

\[ = \lambda \sigma. \text{proj}_{\sigma'} \left( \{ \Phi \mid \overline{t}(\sigma, \sigma') \land \overline{r}(\Phi, \sigma') \} \right) \]

\[ = \lambda \sigma. \text{proj}_{\sigma'} \left( \{ \Phi \mid \overline{t}(\sigma, \sigma') \} \cap \{ \Phi : \overline{r}(\Phi, \sigma') \} \right) \]

\[ \rightarrow \text{intersection followed by a projection} \]

Under-approximation

- \( \overline{R} \) is composed with \( \overline{T} \) and not \( \overline{T} \)
  \[ \rightarrow \overline{R} \cdot \overline{T} \] cannot be defined by exactness criterion of \( \overline{R} \cdot \overline{T} \)

- Define \( \overline{R} \cdot \overline{T} \) and derive an exactness test
  \[ \overline{R} \cdot \overline{T} = \lambda \sigma. \bigcap_{\sigma' \in \overline{T}(\sigma)} \overline{R}(\sigma') \]

- Intersection over a possibly infinite set
  \[ \rightarrow \text{under-approximate further!} \]
Composing Regions With transformers: Exactness Test

General idea

- Elimination of a variable not strongly connected to $\phi$ variables:
  $\rightarrow$ exact
Composing Regions With transformers: Exactness Test

General idea

- Elimination of a variable not strongly connected to $\phi$ variables: $\rightarrow$ exact
- For a variable $\nu$ strongly connected to $\phi$ variables:
  - if there exists an implicit or explicit equation linking the value of $\nu$ to a constant or another variable $\rightarrow$ exact
  - otherwise $\rightarrow$ considered as not exact
Composing Regions With transformers:
Exactness Test

General idea

- Elimination of a variable not strongly connected to $\phi$ variables:
  $\rightarrow$ exact

- For a variable $v$ strongly connected to $\phi$ variables:
  - if there exists an implicit or explicit equation linking the value of $v$ to a constant or another variable $\rightarrow$ exact
  - otherwise $\rightarrow$ considered as not exact

<table>
<thead>
<tr>
<th>$R$ and $\overline{R}$</th>
<th>$\overline{T}$ (k modified)</th>
<th>$R \cdot \overline{T}$</th>
<th>$C_{R \cdot \overline{T}=R \cdot \overline{T}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\phi_1 = \sigma'(i)$</td>
<td>$\sigma'(k) \geq \sigma(k)$</td>
<td>$\phi_1 = \sigma(i)$</td>
<td>true</td>
</tr>
<tr>
<td>$\phi_1 = \sigma'(k)$</td>
<td>$\sigma'(k) = \sigma(k) + 1$</td>
<td>$\phi_1 = \sigma(k) + 1$</td>
<td>true</td>
</tr>
<tr>
<td>$\phi_1 = \sigma'(k)$</td>
<td>$\sigma'(k) \geq \sigma(k)$</td>
<td>$\Phi \geq \sigma(k)$</td>
<td>false</td>
</tr>
<tr>
<td>$\phi_1 = \sigma'(k)$</td>
<td>$\sigma'(k)$</td>
<td>$\phi_1 = \sigma(k)$</td>
<td>false</td>
</tr>
<tr>
<td>$2\phi_1 = 3\sigma'(k)$</td>
<td>$4\sigma'(k) = 5\sigma(j)$</td>
<td>$8\phi_1 = 15\sigma(j)$</td>
<td>true</td>
</tr>
</tbody>
</table>
Regions of do-like Loops

- Regions of general loops defined using a lfp
- do loops (or fortran-like for loops) have good properties

\[ \mathcal{R}[\text{for}(i=1; i<=n; i++) B;] \]
\[ k=\sigma(n) \]
\[ = \lambda \sigma.\mathcal{R}[n] \sigma \bigcup_{k=1}^{n} \mathcal{R}[B] \circ \mathcal{T}[\text{for}(i=1; i<=k-1; i++) B;] \sigma \]

- \( \mathcal{T}[\text{for}(i=1; i<=k-1; i++) B;] \) is available in PIPS 😊
- Intersections and projections
Invariant Regions of a \texttt{do} Loop Body

- An interesting computation by-product
- A kind of hybrid object wrt memory state
- Function of the values of variables in the state preceding the loop execution but for the value of the loop index.
- Obtained by eliminating all modified variables but the loop index when composing with the transformer:

\[
\overline{R}^{inv}[B](\sigma, i) = \overline{R}[B] \bullet_{\sigma(i)} \overline{T}[\text{for}(i=1; \ i<=k-1; \ i++) \ B;]
\]

- Over-approximate invariant regions of a \texttt{do}-like loop:

\[
\overline{R}[\text{for}(i=1; \ i<=n; \ i++) \ B;] = \lambda \sigma. \overline{R}[n] \sigma \bigcup \left( \bigcup_{\sigma(i)=1} \overline{R}^{inv}[B;] \sigma \right)
\]
j = 1;

// \sigma_0
for (i = 0; i <= n; i++)
{
  j = j + 1;
  a[j] = i;
}
j = 1;

// σ₀
for (i = 0; i <= n; i++)

// $\overline{R}(a(φ₁)) = \lambda σ.\{φ₁ | φ₁ = σ(j) + 1, σ(i) = σ(j) - 1, 0 ≤ σ(i) ≤ σ(n) - 1\}$
{
    j = j + 1;
    a[j] = i;
}
Invariant Regions of a do Loop Body : Example

j = 1;

// σ₀
for (i=0; i<=n; i++)

// \overline{R}^{inv}(a(\phi₁)) = \lambda(σ₀, σ(i))\{\phi₁ | \phi₁ = σ(i) + 2, σ₀(j) = 1, 0 ≤ σ(i) ≤ σ₀(n) − 1\}
// \overline{R}(a(\phi₁)) = \lambdaσ.\{\phi₁ | \phi₁ = σ(j) + 1, σ(i) = σ(j) − 1, 0 ≤ σ(i) ≤ σ(n) − 1\}
{
    j = j + 1;
    a[j] = i;
}
Backward propagation on the call graph

Summary regions of functions:
1. remove regions concerning local variables
2. eliminate local variables from polyhedra

Translation at call site:
1. translate the polyhedron from the name space of the callee to the name space of the caller
2. translate the $\phi$ variables of the formal parameter into the $\phi$ variables of the actual parameter

2nd step may depend on the memory-layout (row-major or column-major)
```c
int main() {
  int a[10], n = 5, d;
  ...
  
  d = foo(&a[1], n);
  ...
}

// $R(b[\phi_1]) = \{ \phi_1 | \phi_1 = \sigma(\text{index}) \}$
int foo(int b[9], int index) {
  return b[index];
}
```
```c
int main() {
    int a[10], n = 5, d;
    ...

    d = foo(&a[1], n);
    ...
}
```

// $R(b[\phi_1]) = \{ \phi_1 \mid \phi_1 = \sigma(n) \}$
// $R(b[\phi_1]) = \{ \phi_1 \mid \phi_1 = \sigma(index) \land \sigma(index) = \sigma(n) \}$

```c
int foo(int b[9], int index) {
    return b[index];
}
```
int main() {  
  int a[10], d;  
  ...  

  // \( R(a_{\phi_1}) = \{ \phi_1 | \psi_1 = s(n) \land \phi_1 = \psi_1 + 1 \} \)  
  d = foo(&a[1], n);  
  ...  
}

// \( R(b_{\psi_1}) = \{ \phi_1 | \psi_1 = s(n) \} \)  
// \( R(b_{\phi_1}) = \{ \phi_1 | \phi_1 = s(index) \} \)  
int foo(int b[9], int index) {  
  return b[index];  
}
```c
int main() {
    int a[10], d;
    ...
    // \( \mathcal{R}(a[\phi_1]) = \{ \phi_1 \mid \phi_1 = \sigma(n) + 1 \} \)
    // \( \mathcal{R}(a[\phi_1]) = \{ \phi_1 \mid \psi_1 = \sigma(n) \land \phi_1 = \psi_1 + 1 \} \)
    d = foo(&a[1], n);
    ...
}

// \( \mathcal{R}(b[\phi_1]) = \{ \phi_1 \mid \phi_1 = \sigma(n) \} \)
// \( \mathcal{R}(b[\phi_1]) = \{ \phi_1 \mid \phi_1 = \sigma(\text{index}) \} \)
int foo(int b[9], int idx) {
    return b[index];
}
```
IN Regions of a Statement $S$

- Array elements read in $S$ before being over-written in $S = \text{imported}$
IN Regions of a Statement $S$

- Array elements read in $S$ before being over-written in $S = imported$

```c
{
    a[i] = b[i];
    tmp = foo(a[i]);
    b[i] = ... ;
}
```
IN Regions of a Statement $S$

- Array elements read in $S$ before being over-written in $S = \text{imported}$

```
// Read Regions:
// $R(a[\phi_1]) = \{ \phi_1 \mid \phi_1 = \sigma(i) \}$
// $R(b[\phi_1]) = \{ \phi_1 \mid \phi_1 = \sigma(i) \}$
{
    a[i] = b[i];
    tmp = foo(a[i]);
    b[i] = ... ;
}
```
IN Regions of a Statement $S$

- Array elements read in $S$ before being over-written in $S = \text{imported}$

  ```
  // In Regions:
  // $IN(b[\phi_1]) = \{ \phi_1 | \phi_1 = \sigma(i) \}$

  // Read Regions:
  // $R(a[\phi_1]) = \{ \phi_1 | \phi_1 = \sigma(i) \}$
  // $R(b[\phi_1]) = \{ \phi_1 | \phi_1 = \sigma(i) \}$

  {  
    a[i] = b[i];
    tmp = foo(a[i]);
    b[i] = ... ;
  }
IN Regions of a Statement $S$

- Array elements read in $S$ before being over-written in $S = \text{imported}$

```c
// In Regions:
// \text{IN}(b[\phi_1]) = \{ \phi_1 \mid \phi_1 = \sigma(i) \}

// Read Regions:
// \mathcal{R}(a[\phi_1]) = \{ \phi_1 \mid \phi_1 = \sigma(i) \}
// \mathcal{R}(b[\phi_1]) = \{ \phi_1 \mid \phi_1 = \sigma(i) \}
{
    a[i] = b[i];
    tmp = \text{foo}(a[i]);
    b[i] = ... ;
}
```

- Propagated backward and outward on the HCFG and the call graph

\[
\text{IN}[S_1; S_2] = \text{IN}[S_1] \cup (\text{IN}[S_2] \circ \mathcal{T}[S_1]) \square \mathcal{W}[S_1]
\]
\[
\overline{\text{IN}}[S_1; S_2] = \overline{\text{IN}}[S_1] \cup (\overline{\text{IN}}[S_2] \bullet \overline{\mathcal{T}}[S_1]) \square \overline{\mathcal{W}}[S_1]
\]
OUT Regions of a Statement $S$

- Array elements written by $S$ and re-used in the future = exported
OUT Regions of a Statement $S$

- Array elements written by $S$ and re-used in the future = exported
- Interprocedurally live array elements of $S$ also written by $S$
OUT Regions of a Statement $S$

- Array elements written by $S$ and re-used in the future = exported
- Interprocedurally live array elements of $S$ also written by $S$
- Asymmetry with IN regions (do not depend on what happens before $S$)
OUT Regions of a Statement $S$

- Array elements written by $S$ and re-used in the future = exported
- Interprocedurally live array elements of $S$ also written by $S$
- Asymmetry with IN regions (do not depend on what happens before $S$)

```c
{
    a[i] = ...;
    b[i] = foo(a[i]);
} // a[i] not re-used afterwards
```
OUT Regions of a Statement $S$

- Array elements written by $S$ and re-used in the future = exported
- Interprocedurally live array elements of $S$ also written by $S$
- Asymmetry with IN regions (do not depend on what happens before $S$)

```c
// Write Regions:
// \( \mathcal{W}(a[\phi_1]) = \{ \phi_1 \mid \phi_1 = \sigma(i) \} \)
// \( \mathcal{W}(b[\phi_1]) = \{ \phi_1 \mid \phi_1 = \sigma(i) \} \)
{
    a[i] = ...;
    b[i] = foo(a[i]);
} // a[i] not re-used afterwards
```
OUT Regions of a Statement $S$

- Array elements written by $S$ and re-used in the future = exported
- Interprocedurally live array elements of $S$ also written by $S$
- Asymmetry with IN regions (do not depend on what happens before $S$)

```
// Out Regions:
// `OUT(b[φ_1]) = {φ_1 | φ_1 = σ(i)}`

// Write Regions:
// `W(a[φ_1]) = {φ_1 | φ_1 = σ(i)}`
// `W(b[φ_1]) = {φ_1 | φ_1 = σ(i)}`
{
    a[i] = ...
    b[i] = foo(a[i]);
} // a[i] not re-used afterwards
```

c = bar(b[i]);
OUT Regions of a Statement $S$

- Array elements written by $S$ and re-used in the future = exported
- Interprocedurally live array elements of $S$ also written by $S$
- Asymmetry with IN regions (do not depend on what happens before $S$)

// Out Regions:
// $OUT(b[\phi_1]) = \{ \phi_1 \mid \phi_1 = \sigma(i) \}$

// Write Regions:
// $W(a[\phi_1]) = \{ \phi_1 \mid \phi_1 = \sigma(i) \}$
// $W(b[\phi_1]) = \{ \phi_1 \mid \phi_1 = \sigma(i) \}$
{  
a[i] = ...;
  b[i] = foo(a[i]);
} // a[i] not re-used afterwards

c = bar(b[i]);

- Propagated backward in sequences and inward on the HCFG and the call graph
\[\begin{align*}
\text{OUT}[S_2] \circ T[S_1] &= \text{OUT}[S_1; S_2] \cap \mathcal{W}[S_2] \circ T[S_1] \\
\text{OUT}[S_1] &= \mathcal{W}[S_1] \cap (\text{OUT}[S_1; S_2] \Box \text{OUT}[S_2] \circ T[S_1]) \\
&\quad \cup \text{IN}[S_2] \circ T[S_1]
\end{align*}\]
OUT Regions of $S_1; S_2$;

\[
\begin{align*}
\text{OUT}[S_2] \circ T[S_1] &= \text{OUT}[S_1; S_2;] \cap \mathcal{W}[S_2] \circ T[S_1] \\
\text{OUT}[S_1] &= \mathcal{W}[S_1] \cap (\text{OUT}[S_1; S_2;] \sqcap \text{OUT}[S_2] \circ T[S_1]) \\
&\quad \cup \text{IN}[S_2] \circ T[S_1])
\end{align*}
\]

// \text{OUT}[S_1; S_2;](b[\phi_1]) = \{ \phi_1 \mid \phi_1 = \sigma(i) \}

\{
\begin{align*}
a[i] &= \ldots; \\
\text{OUT}[S_2] &= \text{OUT}[S_1; S_2;] \cap \mathcal{W}[S_2] \\
\text{OUT}[S_2](b[\phi_1]) &= \{ \phi_1 \mid \phi_1 = \sigma(i) \}
\end{align*}
\}

b[i] = \text{foo}(a[i]);
OUT Regions of $S_1; S_2$;

\[
\begin{align*}
\text{OUT}[S_2] \circ T[S_1] &= \text{OUT}[S_1; S_2;] \cap \mathcal{W}[S_2] \circ T[S_1] \\
\text{OUT}[S_1] &= \mathcal{W}[S_1] \cap (\text{OUT}[S_1; S_2;] \boxminus \text{OUT}[S_2] \circ T[S_1]) \\
&\quad \cup \text{IN}[S_2] \circ T[S_1]
\end{align*}
\]

// \text{OUT}[^{\text{OUT}[S_1; S_2;]}(b[\phi_1])] = \{\phi_1 | \phi_1 = \sigma(i)\}

\{
\begin{align*}
a[i] &= \ldots; \\
// \text{OUT}[S_2] &= \text{OUT}[S_1; S_2;] \cap \mathcal{W}[S_2] \\
// \mathcal{W}[S_2](b[\phi_1]) &= \{\phi_1 | \phi_1 = \sigma(i)\} \\
// \text{OUT}[S_2](b[\phi_1]) &= \{\phi_1 | \phi_1 = \sigma(i)\} \\
b[i] &= \text{foo}(a[i]);
\end{align*}
\}
OUT Regions of \( S_1; S_2; \)

\[
\begin{align*}
\text{OUT}[S_2] \circ T[S_1] &= \text{OUT}[S_1; S_2; ] \cap \mathcal{W}[S_2] \circ T[S_1] \\
\text{OUT}[S_1] &= \mathcal{W}[S_1] \cap (\text{OUT}[S_1; S_2; ] \boxdot \text{OUT}[S_2] \circ T[S_1]) \\
&\quad \cup \text{IN}[S_2] \circ T[S_1])
\end{align*}
\]

// \( \text{OUT}[S_1; S_2; ](b[\phi_1]) = \{ \phi_1 \mid \phi_1 = \sigma(i) \} \)

{ 
  // \( \text{OUT}[S_1] = \mathcal{W}[S_1] \cap ((\text{OUT}[S_1; S_2; ] \boxdot \text{OUT}[S_2]) \cup \text{IN}[S_2]) \)
  // \( \mathcal{W}[S_1](a[\phi_1]) = \{ \phi_1 \mid \phi_1 = \sigma(i) \} \)

  a[i] = \ldots;

  // \( \text{IN}[S_2](a[\phi_1]) = \{ \phi_1 \mid \phi_1 = \sigma(i) \} \)

  // \( \text{OUT}[S_2](b[\phi_1]) = \{ \phi_1 \mid \phi_1 = \sigma(i) \} \)
  b[i] = \text{foo}(a[i]);
}

Béatrice Creusillet (SILKAN)  Array Region Analyses and Applications  Polyhedral School 2013  37 / 84
OUT Regions of $S_1; S_2$;

\[
\begin{align*}
\text{OUT}[S_2] \circ T[S_1] &= \text{OUT}[S_1; S_2;] \cap \mathcal{W}[S_2] \circ T[S_1] \\
\text{OUT}[S_1] &= \mathcal{W}[S_1] \cap (\text{OUT}[S_1; S_2;] \square \text{OUT}[S_2] \circ T[S_1]) \\
&\quad \cup \text{IN}[S_2] \circ T[S_1])
\end{align*}
\]

\[
\begin{align*}
\text{OUT}[S_1; S_2;](b[\phi_1]) &= \{ \phi_1 | \phi_1 = \sigma(i) \} \\
\{ \\
\text{OUT}[S_1] &= \mathcal{W}[S_1] \cap ((\text{OUT}[S_1; S_2;] \square \text{OUT}[S_2]) \cup \text{IN}[S_2]) \\
\mathcal{W}[S_1](a[\phi_1]) &= \{ \phi_1 | \phi_1 = \sigma(i) \} \\
\text{OUT}[S_1](a[\phi_1]) &= \{ \phi_1 | \phi_1 = \sigma(i) \} \\
a[i] &= \ldots ;
\end{align*}
\]

\[
\begin{align*}
\text{IN}[S_2](a[\phi_1]) &= \{ \phi_1 | \phi_1 = \sigma(i) \} \\
\text{OUT}[S_2](b[\phi_1]) &= \{ \phi_1 | \phi_1 = \sigma(i) \} \\
b[i] &= \text{foo}(a[i]);
\end{align*}
\]
struct {
    int size;
    struct {
        int inner_size;
        float inner_array[10];
    } array[10];
} my_2D_array;

// \mathcal{W}(\text{my}_2\text{D}_2\text{array.array}[\phi_2].\text{inner}_\text{array}[\phi_4]) = \{\phi_2, \phi_4 \mid \phi_2 = 2 \land \phi_4 = 5\}
Extension to C Data Types : Unions

- **Tricky → memory aliasing**

<table>
<thead>
<tr>
<th>Semantically related</th>
<th>Semantically unrelated</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>union {</code></td>
<td><code>union {</code></td>
</tr>
<tr>
<td><code>    struct{uint8 high, low;} b[10];</code></td>
<td><code>    float f_1[10];</code></td>
</tr>
<tr>
<td><code>    uint16 w[5];</code></td>
<td><code>    int f_2[5];</code></td>
</tr>
<tr>
<td><code>} my_words;</code></td>
<td><code>} my_array;</code></td>
</tr>
<tr>
<td>my_words.b[0].high</td>
<td>my_array.f_1[0]</td>
</tr>
<tr>
<td>conflicts with</td>
<td>should not conflict with</td>
</tr>
<tr>
<td>my_words.w[0]</td>
<td>my_array.f_2[0]</td>
</tr>
</tbody>
</table>

- Very coarse representations are frequent
- Byte-level representations are safe, and adapted to 1\(^{st}\) case (Cf. Antoine Miné)
  - specific $\phi$ variables could be used for byte regions
  - but throws away the high-level semantics
- 2\(^{nd}\) case requires fine data-flow analysis and data structure transformation or assumption on input language
Extension to C Data Types: Pointers

\[
\begin{align*}
*p & \leadsto \mathcal{R}(p[\phi_1]) = \{\phi_1 \mid \phi_1 = 0\} \\
p[3] & \leadsto \mathcal{R}(p[\phi_1]) = \{\phi_1 \mid \phi_1 = 3\} \\
p->array[5] & \leadsto \mathcal{R}(p[\phi_1].array[\phi_3]) = \{\phi_1, \phi_3 \mid \phi_1 = 0, \phi_3 = 5\}
\end{align*}
\]
Extension to C Data Types: Pointers

\*p  \sim \mathcal{R}(p[\phi_1]) = \{\phi_1 | \phi_1 = 0\}

p[3]  \sim \mathcal{R}(p[\phi_1]) = \{\phi_1 | \phi_1 = 3\}

p->array[5]  \sim \mathcal{R}(p[\phi_1].array[\phi_3]) = \{\phi_1, \phi_3 | \phi_1 = 0, \phi_3 = 5\}

From Fortran...

- variable names always refer to the same memory location

... to C
Extension to C Data Types: Pointers

\[ *p \quad \leadsto \quad R(p[\phi_1]) = \{ \phi_1 : \phi_1 = 0 \} \]
\[ p[3] \quad \leadsto \quad R(p[\phi_1]) = \{ \phi_1 : \phi_1 = 3 \} \]
\[ p->array[5] \quad \leadsto \quad R(p[\phi_1].array[\phi_3]) = \{ \phi_1, \phi_3 : \phi_1 = 0, \phi_3 = 5 \} \]

From Fortran...

- variable names always refer to the same memory location

... to C

- a single variable name may refer to several memory locations
Extension to C Data Types: Pointers

\[ *p \rightarrow R(p[\phi_1]) = \{ \phi_1 \mid \phi_1 = 0 \} \]
\[ p[3] \rightarrow R(p[\phi_1]) = \{ \phi_1 \mid \phi_1 = 3 \} \]
\[ p->array[5] \rightarrow R(p[\phi_1].array[\phi_3]) = \{ \phi_1, \phi_3 \mid \phi_1 = 0, \phi_3 = 5 \} \]

From Fortran...

- variable names always refer to the same memory location
- variable parts of a symbolic reference are restricted to array indices

... to C

- a single variable name may refer to several memory locations
Extension to C Data Types: Pointers

```
*p  \sim \mathcal{R}(p[\phi_1]) = \{\phi_1 | \phi_1 = 0\}
p[3] \sim \mathcal{R}(p[\phi_1]) = \{\phi_1 | \phi_1 = 3\}
p->array[5] \sim \mathcal{R}(p[\phi_1].array[\phi_3]) = \{\phi_1, \phi_3 | \phi_1 = 0, \phi_3 = 5\}
```

From Fortran...

- variable names always refer to the same memory location
- variable parts of a symbolic reference are restricted to array indices
- an effect on $p(i,k)$ is an effect on $p[\sigma(i)][\sigma(k)]$

... to C

- a single variable name may refer to several memory locations
Extension to C Data Types: Pointers

\[ *p \quad \sim \quad R(p[\phi_1]) = \{ \phi_1 \mid \phi_1 = 0 \} \]
\[ p[3] \quad \sim \quad R(p[\phi_1]) = \{ \phi_1 \mid \phi_1 = 3 \} \]
\[ p->array[5] \quad \sim \quad R(p[\phi_1].array[\phi_3]) = \{ \phi_1, \phi_3 \mid \phi_1 = 0, \phi_3 = 5 \} \]

From Fortran...

- variable names always refer to the same memory location
- variable parts of a symbolic reference are restricted to array indices
- an effect on \( p(i,k) \) is an effect on \( p[\sigma(i)][\sigma(k)] \)

... to C

- a single variable name may refer to several memory locations
- every element of a symbolic reference may be a variable part
Extension to C Data Types: Pointers

\*p \leadsto \mathcal{R}(p[\phi_1]) = \{\phi_1 \mid \phi_1 = 0\}

p[3] \leadsto \mathcal{R}(p[\phi_1]) = \{\phi_1 \mid \phi_1 = 3\}

p->array[5] \leadsto \mathcal{R}(p[\phi_1].array[\phi_3]) = \{\phi_1, \phi_3 \mid \phi_1 = 0, \phi_3 = 5\}

From Fortran...

- variable names always refer to the same memory location
- variable parts of a symbolic reference are restricted to array indices
- an effect on \( p(i,k) \) is an effect on \( p[\sigma(i)][\sigma(k)] \)

... to C

- a single variable name may refer to several memory locations
- every element of a symbolic reference may be a variable part
- an effect on \( p[i][k] \) is an effect on \( \sigma(\sigma(p)[\sigma(i)])[\sigma(k)] \)
int **p, **q, *a, *b;

a = (int *) malloc(10*sizeof(int)); // malloc_1
b = (int *) malloc(10*sizeof(int)); // malloc_2

p = &a;
p[0][2] = ...;

p = &b;
p[0][2] = ...;

q = p;
q[0][2] = ...;
**Proposition of Array Regions in the Presence of Pointers: What We Would Like**

```c
int **p, **q, *a, *b;

a = (int *) malloc(10*sizeof(int)); // malloc_1
b = (int *) malloc(10*sizeof(int)); // malloc_2

p = &a;
p[0][2] = ...; // \mathcal{W}(p[\phi_1][\phi_2]) = \{\phi_1, \phi_2 \mid \phi_1 = 0 \land \phi_2 = 2\}

p = &b;
p[0][2] = ...; // \mathcal{W}(p[\phi_1][\phi_2]) = \{\phi_1, \phi_2 \mid \phi_1 = 0 \land \phi_2 = 2\}

q = p;
q[0][2] = ...; // \mathcal{W}(q[\phi_1][\phi_2]) = \{\phi_1, \phi_2 \mid \phi_1 = 0 \land \phi_2 = 2\}
```
Propagation of Array Regions in the Presence of Pointers: What We Would Like

```c
int **p, **q, *a, *b;

a = (int *) malloc(10*sizeof(int)); // malloc_1
b = (int *) malloc(10*sizeof(int)); // malloc_2

p = &a;
p[0][2] = ...; // \mathcal{W}(p[\phi_1][\phi_2]) = \{\phi_1, \phi_2 \mid \phi_1 = 0 \wedge \phi_2 = 2\}

p = &b;
p[0][2] = ...; // \mathcal{W}(p[\phi_1][\phi_2]) = \{\phi_1, \phi_2 \mid \phi_1 = 0 \wedge \phi_2 = 2\}

q = p;
q[0][2] = ...; // \mathcal{W}(q[\phi_1][\phi_2]) = \{\phi_1, \phi_2 \mid \phi_1 = 0 \wedge \phi_2 = 2\}
```

Béatrice Creusillet (SILKAN)
Propagation of Array Regions in the Presence of Pointers: What We Would Like

```c
int **p, **q, *a, *b;

a = (int *) malloc(10*sizeof(int)); // malloc_1
b = (int *) malloc(10*sizeof(int)); // malloc_2

p = &a;
p[0][2] = ...; // W(p[\phi_1][\phi_2]) = \{\phi_1, \phi_2 \mid \phi_1 = 0 \land \phi_2 = 2\}

p = &b;

// W(p[\phi_1][\phi_2]) = \{\phi_1, \phi_2 \mid \phi_1 = 0 \land \phi_2 = 2\}
p[0][2] = ...; // W(p[\phi_1][\phi_2]) = \{\phi_1, \phi_2 \mid \phi_1 = 0 \land \phi_2 = 2\}

q = p;
q[0][2] = ...; // W(q[\phi_1][\phi_2]) = \{\phi_1, \phi_2 \mid \phi_1 = 0 \land \phi_2 = 2\}
```
Propagation of Array Regions in the Presence of Pointers: What We Would Like

```c
int **p, **q, *a, *b;

a = (int *) malloc(10*sizeof(int)); // malloc_1
b = (int *) malloc(10*sizeof(int)); // malloc_2

p = &a;
p[0][2] = ...; // \mathcal{W}(p[\phi_1][\phi_2]) = \mathcal{W}(b[\phi_1]) = \{\phi_1 \mid \phi_1 = 2\}
p = &b;

// \mathcal{W}(p[\phi_1][\phi_2]) = \{\phi_1, \phi_2 \mid \phi_1 = 0 \land \phi_2 = 2\}
p[0][2] = ...; // \mathcal{W}(p[\phi_1][\phi_2]) = \mathcal{W}(p[\phi_1][\phi_2]) = \{\phi_1, \phi_2 \mid \phi_1 = 0 \land \phi_2 = 2\}
q = p;
q[0][2] = ...; // \mathcal{W}(q[\phi_1][\phi_2]) = \mathcal{W}(p[\phi_1][\phi_2]) = \{\phi_1, \phi_2 \mid \phi_1 = 0 \land \phi_2 = 2\}
```

Béatrice Creusillet (SILKAN)
int **p, **q, *a, *b;

a = (int *) malloc(10*sizeof(int)); // malloc_1
b = (int *) malloc(10*sizeof(int)); // malloc_2

// ℋ(b[φ₁]) = {φ₁ | φ₁ = 2}
// ℋ(a[φ₁]) = {φ₁ | φ₁ = 2}
p = &a;

p[0][2] = ...; // ℋ(p[φ₁][φ₂]) = {φ₁, φ₂ | φ₁ = 0 ∧ φ₂ = 2}

// ℋ(b[φ₁]) = {φ₁ | φ₁ = 2}
p = &b;

// ℋ(p[φ₁][φ₂]) = {φ₁, φ₂ | φ₁ = 0 ∧ φ₂ = 2}
p[0][2] = ...; // ℋ(p[φ₁][φ₂]) = {φ₁, φ₂ | φ₁ = 0 ∧ φ₂ = 2}

// ℋ(p[φ₁][φ₂]) = {φ₁, φ₂ | φ₁ = 0 ∧ φ₂ = 2}
q = p;

q[0][2] = ...; // ℋ(q[φ₁][φ₂]) = {φ₁, φ₂ | φ₁ = 0 ∧ φ₂ = 2}
int **p, **q, *a, *b;

 // \( \mathcal{W}(\text{malloc}_2[\phi_1]) = \{ \phi_1 | \phi_1 = 2 \} \)
 // \( \mathcal{W}(\text{malloc}_1[\phi_1]) = \{ \phi_1 | \phi_1 = 2 \} \)

 a = (int *) malloc(10*sizeof(int)); // \text{malloc}_1
 b = (int *) malloc(10*sizeof(int)); // \text{malloc}_2

 // \( \mathcal{W}(b[\phi_1]) = \{ \phi_1 | \phi_1 = 2 \} \)
 // \( \mathcal{W}(a[\phi_1]) = \{ \phi_1 | \phi_1 = 2 \} \)

 p = &a;

 p[0][2] = ...; // \( \mathcal{W}(p[\phi_1][\phi_2]) = \{ \phi_1, \phi_2 | \phi_1 = 0 \land \phi_2 = 2 \} \)

 // \( \mathcal{W}(b[\phi_1]) = \{ \phi_1 | \phi_1 = 2 \} \)

 p = &b;

 // \( \mathcal{W}(p[\phi_1][\phi_2]) = \{ \phi_1, \phi_2 | \phi_1 = 0 \land \phi_2 = 2 \} \)

 p[0][2] = ...; // \( \mathcal{W}(p[\phi_1][\phi_2]) = \{ \phi_1, \phi_2 | \phi_1 = 0 \land \phi_2 = 2 \} \)

 // \( \mathcal{W}(p[\phi_1][\phi_2]) = \{ \phi_1, \phi_2 | \phi_1 = 0 \land \phi_2 = 2 \} \)

 q = p;

 q[0][2] = ...; // \( \mathcal{W}(q[\phi_1][\phi_2]) = \{ \phi_1, \phi_2 | \phi_1 = 0 \land \phi_2 = 2 \} \)
Propogation of Array Regions in the Presence of Pointers

\[ \mathcal{R}[S_1; S_2;] = \mathcal{R}[S_1] \cup (\mathcal{R}[S_2] \circ \mathcal{T}[S_1]) \]
Propagation of Array Regions in the Presence of Pointers

\[ \mathcal{R}[S_1; S_2;] = \mathcal{R}[S_1] \cup (\mathcal{R}[S_2] \circ \mathcal{T}[S_1]) \]

Using pointer values transformers:

- Currently not available in PIPS 😞
- Difficult to compute (need for several passes)
Propagation of Array Regions in the Presence of Pointers

\[ \mathcal{R}[S_1; S_2; ] = \mathcal{R}[S_1] \cup (\mathcal{R}[S_2] \circ \mathcal{T}[S_1]) \]

Using pointer values transformers:

- Currently not available in PIPS 😞
- Difficult to compute (need for several passes)

Propagation of Constant Path Regions:

- Memory paths involving no pointers
- Requires a pointer analysis (such as points-to)
- Propagation as usual
- Compatible with previously existing client analyses
```c
int **p, **q, *a, *b;

a = (int *) malloc(10*sizeof(int)); // malloc_1
b = (int *) malloc(10*sizeof(int)); // malloc_2

p = &a;

p[0][2] = ...; // W(p[\phi_1][\phi_2]) = \{\phi_1, \phi_2 | \phi_1 = 0 \land \phi_2 = 2\}
p = &b;

p[0][2] = ...; // W(p[\phi_1][\phi_2]) = \{\phi_1, \phi_2 | \phi_1 = 0 \land \phi_2 = 2\}
q = p;

q[0][2] = ...; // W(q[\phi_1][\phi_2]) = \{\phi_1, \phi_2 | \phi_1 = 0 \land \phi_2 = 2\}
```
int **p, **q, *a, *b;

a = (int *) malloc(10*sizeof(int)); // malloc_1
b = (int *) malloc(10*sizeof(int)); // malloc_2

p = &a;

// p -> a, a -> malloc_1[0]
p[0][2] = ...; // \mathcal{W}(p[\phi_1][\phi_2]) = \{\phi_1, \phi_2 \mid \phi_1 = 0 \land \phi_2 = 2\}

p = &b;

// p -> b, b -> malloc_2[0]
p[0][2] = ...; // \mathcal{W}(p[\phi_1][\phi_2]) = \{\phi_1, \phi_2 \mid \phi_1 = 0 \land \phi_2 = 2\}

q = p;

// q -> b, b -> malloc_2[0]
q[0][2] = ...; // \mathcal{W}(q[\phi_1][\phi_2]) = \{\phi_1, \phi_2 \mid \phi_1 = 0 \land \phi_2 = 2\}
Propagation of Constant Path Regions

```c
int **p, **q, *a, *b;

a = (int *) malloc(10*sizeof(int)); // malloc_1
b = (int *) malloc(10*sizeof(int)); // malloc_2

p = &a;

// \mathcal{W}(\text{malloc}_1[\phi_1]) = \{\phi_1 | \phi_1 = 2\}
// p -> a, a -> \text{malloc}_1[0]
p[0][2] = ...; // \mathcal{W}(p[\phi_1][\phi_2]) = \{\phi_1, \phi_2 | \phi_1 = 0 \land \phi_2 = 2\}

p = &b;

// \mathcal{W}(\text{malloc}_2[\phi_1]) = \{\phi_1 | \phi_1 = 2\}
// p -> b, b -> \text{malloc}_2[0]
p[0][2] = ...; // \mathcal{W}(p[\phi_1][\phi_2]) = \{\phi_1, \phi_2 | \phi_1 = 0 \land \phi_2 = 2\}

q = p;

// \mathcal{W}(\text{malloc}_2[\phi_1]) = \{\phi_1 | \phi_1 = 2\}
// q -> b, b -> \text{malloc}_2[0]
q[0][2] = ...; // \mathcal{W}(q[\phi_1][\phi_2]) = \{\phi_1, \phi_2 | \phi_1 = 0 \land \phi_2 = 2\}
```

Béatrice Creusillet (SILKAN) Array Region Analyses and Applications Polyhedral School 2013 43 / 84
Translation to Constant Path Regions

Using points to:
Translation of a region $\mathcal{R}(a[\phi_1][\phi_2][\phi_3])$ knowing that $a[\phi_1] = \& b[\psi_1][\psi_2] \land p(\phi_1, \psi_1, \psi_2, \sigma)$

Using pointer values
Translation of a region $\mathcal{R}(a[\phi_1][\phi_2][\phi_3])$ knowing that $a[\phi_1] = b[\psi_1][\psi_2] \land p(\phi_1, \psi_1, \psi_2, \sigma)$

- Numerous sources of inexactitude
  - exactness of input information
  - whether $\text{proj}_\psi(p) \subseteq \mathcal{R}(a[\phi_1][\phi_2][\phi_3])$
  - or $\text{proj}_\psi(p) \cap \mathcal{R}(a[\phi_1][\phi_2][\phi_3]) \neq \emptyset$
  - involves projections and feasibility tests

- Interprocedural translation is a particular case.
What Pointer Analysis?

- Pointer analyses are ... global ... hence costly/unprecise
- Precise enough for client analyses
  \[\rightarrow \text{exact points-to for exact regions}\]
- Trade-off between time/memory consumption and precision

Landi and Ryder (20 years of PLDI, 2003)

*We predict that the future will not see a best Pointer May-alias algorithm whose results are suitable for any application, but rather algorithms designed to optimize the tradeoffs to best meet the requirements of some particular application.*
Without any context information:
\[ a[i] + a[j] \sim \{ \phi_1 \mid \} \]

Information provided in PIPS by Preconditions Systematically added to array regions predicates sometimes enables exact convex hulls at the expense of complexity trade-off between precision, speed, memory usage open question: selectively add information about values strongly connected to variables?
Using some Context Information

- Without any context information:
  \[ a[i] + a[j] \sim \{ \phi_1 \} \]

- Knowing that \( \sigma(j) = \sigma(i) + 1 \):
  \[ a[i] + a[j] \sim \{ \phi_1 \mid \sigma(i) \leq \phi_1 \leq \sigma(j) \land \sigma(j) = \sigma(i) + 1 \} \]
Using some Context Information

- Without any context information:
  \[ a[i] + a[j] \sim \{ \phi_1 | \} \]

- Knowing that \( \sigma(j) = \sigma(i) + 1 \):
  \[ a[i] + a[j] \sim \{ \phi_1 | \sigma(i) \leq \phi_1 \leq \sigma(j) \land \sigma(j) = \sigma(i) + 1 \} \]

- Information provided in PIPS by Preconditions
Using some Context Information

- Without any context information:
  \[ a[i] + a[j] \sim \{ \phi_1 \} \]

- Knowing that \( \sigma(j) = \sigma(i) + 1 \):
  \[ a[i] + a[j] \sim \{ \phi_1 \mid \sigma(i) \leq \phi_1 \leq \sigma(j) \land \sigma(j) = \sigma(i) + 1 \} \]

- Information provided in PIPS by Preconditions

- Systematically added to array regions predicates
  
  - Sometimes enables exact convex hulls
  - At the expense of complexity
  - Trade-off between precision, speed, memory usage
  - Open question: Selectively add information about values strongly connected to \( \phi \) variables?
Complexity

A difficult subject

- Depends on numerous characteristics of input applications
  - number of variables \(\sim\) number of regions
  - dimensionality of memory paths \(\sim\) size of polyhedra
  - number of scalar variables for which the value is known or constrained \(\sim\) size of polyhedra
  - the level of connection between variable values, and/or \(\phi\) variables \(\sim\) complexity of operators
  - The more information we have, the more complex it is!
Complexity

A difficult subject

- Depends on numerous characteristics of input applications
  - number of variables $\sim$ number of regions
  - dimensionality of memory paths $\sim$ size of polyhedra
  - number of scalar variables for which the value is known or constrained $\sim$ size of polyhedra
  - the level of connection between variable values, and/or $\phi$ variables $\sim$ complexity of operators
  - The more information we have, the more complex it is!

- Depends on the implementation
  - complexity of propagation and of linear algebra operators
  - interactions between them
  - normalization of the polyhedra
  - complexity of basic operations such as copy
  - the devil is in the details!
Complexity

A difficult subject

- Depends on numerous characteristics of input applications
  - number of variables $\sim$ number of regions
  - dimensionality of memory paths $\sim$ size of polyhedra
  - number of scalar variables for which the value is known or constrained $\sim$ size of polyhedra
  - the level of connection between variable values, and/or $\phi$ variables $\sim$ complexity of operators
  - The more information we have, the more complex it is!

- Depends on the implementation
  - complexity of propagation and of linear algebra operators
  - interactions between them
  - normalization of the polyhedra
  - complexity of basic operations such as copy
  - the devil is in the details!

- Choice of convex polyhedra: trade-off between complexity/memory usage
Outline

Foundations
  Context/History
  Nature of array element sets?
  Approximations
  Read and Write Regions
  IN and OUT regions
  Extension to the C Language

APPLICATIONS
  Loop Transformations
  Memory Management
  Static Analysis

CONCLUSION
Program Transformations

Extend existing fine-grain algorithms

- use regions as summaries for function calls
- fine-grain loop parallelization

Design new coarse-grain algorithms

- use regions as summaries for sub-parts of programs (loop bodies)
- relax usual conditions on internal control-flow
- coarse-grain loop parallelization
- loop fusion
- whole array privatization
- array scalarization
- statement isolation
Allen & Kennedy algorithm [AK02]

- Distribution of perfect loop nests
  - Fortran-like do loops
  - no embedded tests, while loops, branching constructs,...
  - no inner declarations
- Reduces size of loop bodies

- With array regions:
  - relax usual conditions on function calls
  - less costly than inlining

- Requires to build a dependence graph
  - nodes are statements
  - edges are dependences between $\overline{R}$ and $\overline{W}$ regions of statements
  - dependence does not hold if intersection is empty
  - but beware of state modifications!
subroutine extr(ni,nc)
dimension T(52,21,60)
common/cT/T
common/ci/i1,i2
common/cj/j1,j2
common/ck/k1,k2
common/cni/l
read(nxyz) i1,i2,j1,ja,k1,k2
if(j1.ge.1) then
  j1=j1+1
  j2=j2+1
  ja=ja+1
  l=ni
  k=k1
  do j=j1,ja
     s1=d(j,k,j+1)
     s2=d(j,k+1,j,k+2)+s1
     s3=d(j,k+2,j,k+3)+s2
     T(j,1,nc+3)=s2*s3/((s1-s2)*(s1-s3))
     T(j,1,nc+4)=s3*s1/((s2-s3)*(s2-s1))
     T(j,1,nc+5)=s1*s2/((s3-s1)*(s3-s2))
     cjh=j1+j2-j
     T(j1+j2-j,1,nc+3)=T(j,1,nc+3)
     T(j1+j2-j,1,nc+4)=T(j,1,nc+4)
     T(j1+j2-j,1,nc+5)=T(j,1,nc+5)
   enddo
endif
return
end

real function d(j,k,jp,kp)
dimension T(52,21,60)
common/cT/T
common/cni/l
d=sqrt((T(j,k,l )-T(jp,kp,l ))**2
  + (T(j,k,l+1)-T(jp,kp,l+1))**2
  + (T(j,k,l+2)-T(jp,kp,l+2))**2)
return
end
Fine-Grain Parallelization: Example

\[
\begin{align*}
\text{do } & \text{j} = \text{j}_1, \text{ja} \\
\mathcal{R}(T(\phi_1, \phi_2, \phi_3)) & = \begin{cases} 
\phi_1 = \text{j}, 2 \leq \phi_1 \leq 52, \\
1 \leq \phi_2 \leq 21, k \leq \phi_2 \leq k + 1, \\
1 \leq \phi_3 \leq 60, 1 \leq \phi_3 \leq 1 + 2,
\end{cases} \\
& \begin{cases} 
\phi_1 = \text{j}_p, 2 \leq \phi_1 \leq \phi_1, \\
k_{p - 1} \leq \phi_2 \leq k_p, \\
1 \leq \phi_3 \leq 1 + 2,
\end{cases} \\
\mathcal{W}(T(\phi_1, \phi_2, \phi_3)) & = \begin{cases} 
\phi_1 = \text{j}, 2 \leq \phi_1 \leq 52, \\
1 \leq \phi_2 \leq 21, k \leq \phi_2 \leq k + 1, \\
1 \leq \phi_3 \leq 60, 1 \leq \phi_3 \leq 1 + 2,
\end{cases} \\
& \begin{cases} 
\phi_1 = \text{j}_p, 2 \leq \phi_1 \leq \phi_1, \\
k_{p - 1} \leq \phi_2 \leq k_p, \\
1 \leq \phi_3 \leq 1 + 2,
\end{cases} \\
\end{align*}
\]

\[
\begin{align*}
s_1 & = d(\text{j}, k, \text{j}, k + 1) \\
s_2 & = d(\text{j}, k + 1, \text{j}, k + 2) + s_1 \\
s_3 & = d(\text{j}, k + 2, \text{j}, k + 3) + s_2
\end{align*}
\]

\[
\begin{align*}
s & = d\sqrt{(T(\phi_1, k, l) - T(\phi_1, k, l))^{2}} \\
& + (T(\phi_1, k + 1, l) - T(\phi_1, k, l + 1))^{2} \\
& + (T(\phi_1, k + 2, l) - T(\phi_1, k, l + 2))^{2}
\end{align*}
\]

\[
\begin{align*}
t & = s_2 * s_3 / (s_1 - s_2) * (s_1 - s_3) \\
t & = s_3 * s_1 / (s_2 - s_3) * (s_2 - s_1) \\
t & = s_1 * s_2 / (s_3 - s_1) * (s_3 - s_2)
\end{align*}
\]

\[
\begin{align*}
T(\text{j}, 1, \text{nc} + 3) & = s_2 * s_3 / (s_1 - s_2) * (s_1 - s_3) \\
T(\text{j}, 1, \text{nc} + 4) & = s_3 * s_1 / (s_2 - s_3) * (s_2 - s_1) \\
T(\text{j}, 1, \text{nc} + 5) & = s_1 * s_2 / (s_3 - s_1) * (s_3 - s_2)
\end{align*}
\]

\[
\begin{align*}
T(\text{j} + \text{j} - 1, \text{nc} + 3) & = T(\text{j}, 1, \text{nc} + 3) \\
T(\text{j} + \text{j} - 1, \text{nc} + 4) & = T(\text{j}, 1, \text{nc} + 4) \\
T(\text{j} + \text{j} - 1, \text{nc} + 5) & = T(\text{j}, 1, \text{nc} + 5)
\end{align*}
\]

\[
\begin{align*}
\text{enddo}
\end{align*}
\]
$omp \text{ PARALLEL DO PRIVATE}(s1, s2, s3)$
\begin{verbatim}
do j = j1, ja
    s1 = d(j, k, j, k+1)
    s2 = d(j, k+1, j, k+2) + s1
    s3 = d(j, k+2, j, k+3) + s2
    T(j,1,nc+3) = s2*s3/((s1-s2)*(s1-s3))
    T(j,1,nc+4) = s3*s1/((s2-s3)*(s2-s1))
    T(j,1,nc+5) = s1*s2/((s3-s1)*(s3-s2))
enddo
\end{verbatim}

$omp \text{ PARALLEL DO}$
\begin{verbatim}
do j = j1, ja
    T(j1+j2-j,1,nc+5) = T(j,1,nc+5)
enddo
\end{verbatim}

$omp \text{ PARALLEL DO}$
\begin{verbatim}
do j = j1, ja
    T(j1+j2-j,1,nc+4) = T(j,1,nc+4)
enddo
\end{verbatim}

$omp \text{ PARALLEL DO}$
\begin{verbatim}
do j = j1, ja
    T(j1+j2-j,1,nc+3) = T(j,1,nc+3)
enddo
\end{verbatim}
Coarse-Grain Parallelization

- Consider loop bodies as black boxes
- Relax most restrictions:
  - no more conditions on internal control flow
  - exception: no outward going edges!
  - inner declarations are allowed.
- No need for a dependence graph
- Main idea: test Bernstein’s conditions with loop body regions
Coarse-Grain Parallelization : 
Exact Bernstein’s Conditions

Let us consider a normalized Fortran-like do loop of body $B$ and upper bound $N$:

$$
\sigma_1 \xrightarrow{B_1} \sigma_1 \ldots \sigma_k \xrightarrow{B_k} \ldots \sigma_{k'} \xrightarrow{B_{k'}} \ldots \sigma_{\mathcal{E}[N](\sigma_1)} \xrightarrow{B_{\mathcal{E}[N](\sigma_1)}}
$$

- Let $\mathcal{P}_{B_k}$ be the loop body precondition before the $k$ – th iteration
- Let $\mathcal{T}_{B_k \rightarrow B_{k'}}$ be the transformer between $\sigma_k$ and $\sigma_{k'}$
- Let $\mathcal{R}_{B_k}$ and $\mathcal{W}_{B_k}$ be the loop body read and write regions for the $k$ – th iteration
- Then the Bernstein’s conditions for this loop are :

$$
\forall k, k' \mid 1 \leq k < k' \leq \mathcal{E}[N](\sigma_1) \quad \forall \sigma \in \mathcal{P}_{B_k}
\left\{
\begin{array}{ll}
\mathcal{R}_{B_k}(\sigma) \cap (\mathcal{W}_{B_{k'}} \circ \mathcal{T}_{B_k \rightarrow B_{k'}})(\sigma) = \emptyset \\
\mathcal{W}_{B_k}(\sigma) \cap (\mathcal{R}_{B_{k'}} \circ \mathcal{T}_{B_k \rightarrow B_{k'}})(\sigma) = \emptyset \\
\mathcal{W}_{B_k}(\sigma) \cap (\mathcal{W}_{B_{k'}} \circ \mathcal{T}_{B_k \rightarrow B_{k'}})(\sigma) = \emptyset
\end{array}
\right.
$$
Coarse-Grain Parallelization: Over-approximated Bernstein’s Conditions

- Let $\overline{P}$ be an over-approximation of the loop preconditions.
- Let $\overline{R}^\text{inv}_B$ and $\overline{W}^\text{inv}_B$ be the over-approximate invariant loop body regions:
  \[
  \forall k \mid 1 \leq k \leq \mathcal{E}[\mathcal{N}](\sigma_1), R_{B_k} \subseteq \overline{R}^\text{inv}_B
  \]
- Since $\overline{R}^\text{inv} \cap \overline{R}^\text{inv} = \emptyset \implies R \cap R = \emptyset$, then
  \[
  \forall \sigma, \sigma' \in \overline{P} / i, \sigma(i) < \sigma'(i)
  \]
  are sufficient conditions for the parallelization of the loop.
- IN and OUT regions could be used:
  - parallelizing and privatizing at the same time
  - beware of private array sections -> need to generate declarations
program NS
parameter (nvar=3,nxm=2000,nym=2000)
real phi(nvar,nxm,nym),phi1(nvar,nxm,nym)
real xcoef(nxm,nym)

  do j=2,ny-1
    do i=2,nx-1
      xco=xcoef(i,j)
      px=(phi1(3,i+1,j)-phi1(3,i-1,j))*h1p2
      py=(phi1(3,i,j+1)-phi1(3,i,j-1))*h2p2
      phi1(1,i,j)=phi1(1,i,j)-dt*px*xco
      phi1(2,i,j)=phi1(2,i,j)-dt*py*xco
    enddo
  enddo
end

program NS
parameter (nvar=3,nxm=2000,nym=2000)
real phi(nvar, nxm, nym), phi1(nvar, nxm, nym)
real xcoef(nxm, nym)

do j=2,ny-1

\[ c \mathcal{R}(\phi_1(\phi_1, \phi_2, \phi_3)) = \begin{cases} 
\phi_1 \leq 3, 2j + 5 \leq \phi_1 + 2\phi_2 + 2\phi_3, 2\phi_3 + 5 \leq \phi_1 + 2\phi_2 + 2j, \\
2\phi_2 + 2j + 3 \leq \phi_1 + 2\phi_3 + 2nx, 2\phi_2 + 2\phi_3 + 3 \leq \phi_1 + 2j + 2nx, \\
2j + 1 \leq \phi_1 + 2\phi_3, 2\phi_3 + 1 \leq \phi_1 + 2j, \\
2 \leq j \leq ny - 1, 3 \leq nx 
\end{cases} \]

\[ c \mathcal{W}(\phi_1(\phi_1, \phi_2, \phi_3)) = \{\phi_3 = j, 1 \leq \phi_1 \leq 2, 2 \leq \phi_2 \leq nx - 1, 2 \leq j \leq ny - 1\} \]

\[ c \mathcal{R}(xcoef(\phi_1, \phi_2)) = \{\phi_2 = j, 2 \leq \phi_1 \leq nx - 1, 2 \leq j \leq ny - 1\} \]

do i=2,nx-1

\[ c \mathcal{R}(\phi_1(\phi_1, \phi_2, \phi_3)) = \begin{cases} 
\phi_1 \leq 3, 2i + 2j + 1 \leq \phi_1 + 2\phi_2 + 2\phi_3, 2\phi_3 + 2i + 1 \leq \phi_1 + 2\phi_2 + 2j, \\
2\phi_2 + 2j + 1 \leq \phi_1 + 2\phi_3 + 2i, 2\phi_2 + 2\phi_3 + 1 \leq \phi_1 + 2i + 2j, \\
2 \leq i \leq nx - 1, 2 \leq j \leq ny - 1 
\end{cases} \]

\[ c \mathcal{W}(\phi_1(\phi_1, \phi_2, \phi_3)) = \{\phi_2 = i, \phi_3 = j, 1 \leq \phi_1 \leq 2, 2 \leq i \leq nx - 1, 2 \leq j \leq ny - 1\} \]

\[ c \mathcal{R}(xcoef(\phi_1, \phi_2)) = \{\phi_1 = i, \phi_2 = j, 2 \leq i \leq nx - 1, 2 \leq j \leq ny - 1\} \]

xco=xcoef(i,j)
px=(phi1(3,i+1,j)-phi1(3,i-1,j))*h1p2
py=(phi1(3,i,j+1)-phi1(3,i,j-1))*h2p2
phi1(1,i,j)=phi1(1,i,j)-dt*px*xco
phi1(2,i,j)=phi1(2,i,j)-dt*py*xco
enddo
enddo
end
program ns
parameter (nvar=3,nxm=2000,nym=2000)
real phi(nvar,nxm,nym),phi1(nvar,nxm,nym)
real xcoef(nxm,nym)

!$omp PARALLEL DO PRIVATE(i)
do j = 2, ny-1

!$omp PARALLEL DO PRIVATE(px,py,xco)
do i = 2, nx-1
  xco = xcoef(i,j)
  px = (phi1(3,i+1,j)-phi1(3,i-1,j))*h1p2
  py = (phi1(3,i,j+1)-phi1(3,i,j-1))*h2p2
  phi1(1,i,j) = phi1(1,i,j)-dt*px*xco
  phi1(2,i,j) = phi1(2,i,j)-dt*py*xco
enddo
enddo
end
Parallelization: Coarse-Grain vs Fine-Grain?

- What is your target?
- Coarse grain parallelization better for OpenMP and GPU for instance
- But, it may fail:
  - the loop as a whole is not parallel
  - array regions of loop bodies are not convex
- Use fine-grain parallelization afterwards to exhibit more parallelism
  - Try to fuse parallel loops
  - Use a cost model to avoid too small parallel loops
    - e.g. PIPS’ phase limit_parallelism_using_complexity
**Initial Schedule:** \( \text{for}(i_1 \ldots) S_1; \text{for}(i_2 \ldots) S_2; \)

\[
\begin{align*}
L_1 : & \quad S_1^1 & \rightarrow & S_1^2 & \rightarrow & \ldots & \rightarrow & S_1^{n_1} \\
L_2 : & \quad S_2^2 & \rightarrow & S_2^2 & \rightarrow & \ldots & \rightarrow & S_2^{n_2}
\end{align*}
\]

**Target Schedule:** \( \text{for}( \ldots ) S_1; \text{for}( \ldots ) S_1; \text{S}_2 \text{ for}( \ldots ) S_2; \)

\[
\begin{align*}
S_1^1 & \rightarrow \ldots \rightarrow S_1^d \\
\downarrow \\
S_1^{d+1} & \rightarrow S_2^1 & \rightarrow & S_1^{d+2} & \rightarrow & S_2^2 & \rightarrow & \ldots & \rightarrow & S_1^{n_1} & \rightarrow & S_2^{n_1-d} \\
\downarrow \\
S_2^{n_1-d+1} & \rightarrow \ldots & \rightarrow & S_2^{n_2}
\end{align*}
\]
Loop Fusion with Array Regions: Sufficient Conditions

- Let $R_1$ and $W_1$ (resp. $R_2$ and $W_2$) be the loop body regions for loop $L_1$ (resp. $L_2$)
- Permutation of the last iterations of $L_1$ with the first iterations of $L_2$ with delay $d$

$$\forall \sigma_1, \sigma_2, \overline{P}_1(\sigma_1) \land \sigma_2 = T_{1,2}(\sigma_1) \land \sigma_2(i_2) = \sigma_1(i_1) + d$$

\[
\left\{ \begin{array}{l}
R_1(\sigma_1) \cap W_2 = \emptyset \\
W_1(\sigma_1) \cap R_2(\sigma_2) = \emptyset \\
W_1(\sigma_1) \cap W_2(\sigma_2) = \emptyset
\end{array} \right.
\]

- Still sufficient conditions with over-approximations of invariant regions
- No restrictions on the loop bodies, nor on loop index identifiers or ranges
- Current implementation in Pips: $d = 0$ and ranges are the same
- More conditions can be enforced to drive the fusion
- Ideally followed by array scalarization
Memory Management

Array Privatization

- Find out whole arrays or array regions which are privatizable
- Goal: Remove dependences due to memory reuse
- 😞 Increases memory usage

Array Scalarization

- Replace references to array elements by local scalar variables
- Goal: Reduce memory usage, sometimes remove dependences

Statement Isolation

- Goal: Export a statement execution to a new memory space
- Generalization of array region privatization for any statement
Whole Array Privatization

- Array $A$ is privatizable in loop body $B$ if:

$$
\forall \sigma \in \overline{P}_B, \quad \overline{IN}_B(A)\sigma \neq \emptyset \land \overline{IN}_B(A)\sigma = \overline{OUT}_B(A)\sigma = \emptyset
$$

- If we also want that it *certainly* removes inter-iteration $W/W$ dependences:

$$
\exists \sigma, \sigma' \in \overline{P}_P, \quad \overline{W}(A) \cap \overline{W}(A) \neq \emptyset
$$

- If we solely want that it *may* remove inter-iteration dependences:

$$
\exists \sigma, \sigma' \in \overline{P}_P, \quad \overline{W}(A) \cap \overline{W}(A) \neq \emptyset
$$

- Note: see [Cre96] for array section privatization
void compute(int p, int n, float a[n][n]) {
    int t[n];
    int i, j;
    for(i=0;i<n;i++) {
        for (j=0; j< n; j++)
        {
            t[j] = a[i][j] + j;
        }
    }
    for(j=0;j<n;j++)
    {
        if(p>0)
            a[i][j] = t[j];
    }
}

int main() {
    float a[100][100];
    int n = read_input(100, a);
    int p = init_parameter(n, a); 
    compute(p, n, a);
    write_output(n, a);
    return 0;
}
Whole Array Privatization : Example

```c
for (i=0; i<n; i++)

// \( \mathcal{W}(t[\phi_1]) = \{ \phi_1 \mid 0 \leq \phi_1 \leq n - 1 \} \)

// \( \text{IN}(t[\phi_1]) = \emptyset \)
// \( \text{OUT}(t[\phi_1]) = \emptyset \)
{
    // \( \text{OUT}(t[\phi_1]) = \{ \phi_1 \mid 0 \leq \phi_1 \leq n - 1 \} \)
    for (j=0; j<n; j++)
    {
        t[j] = a[i][j] + j;
    }

// \( \overline{\text{IN}}(t[\phi_1]) = \{ \phi_1 \mid 0 \leq \phi_1 \leq n - 1 \} \)
for(j=0; j<n; j++)
{
    if (p>0)
        a[i][j] = t[j];
}
```
void compute(int p, int n, float a[n][n])
{
    int t[n];
    int i, j;
    #pragma omp parallel for private(j,t[n])
    for(i = 0; i <= n-1; i += 1) {
        #pragma omp parallel for
        for(j = 0; j <= n-1; j += 1)
            t[j] = a[i][j]+j;

        #pragma omp parallel for
        for(j = 0; j <= n-1; j += 1)
            if (p>0)
                a[i][j] = t[j];
    }
}
Motivation: after loop fusion, to reduce memory footprint

```
for (i=0; i<10; i++)
c[i] = a[i] + b[i];
```

```
for (i=0; i<10; i++)
e[i] = c[i] * d[i];
```

Very useful when chaining independent source-to-source tools!

Algorithm based on use-defs, using R/W regions

Coarse-grain algorithm based on IN/OUT regions

- Much costly than previous one
  - use it afterwards, or use a pre-analysis to check usefulness
- Less limited by irregular loop bodies
  (even if IN and OUT regions may be imprecise)
- Also valid for global variables thanks to interprocedural OUT regions
  - But may require interprocedural transformations
Coarse Grain Array Scalarization

- Let $L$ be a loop of body $B$ and index $i$
- Let $\mathcal{W}_B$, $\mathcal{R}_B$, $IN_B$ and $OUT_B$ be the body’s regions for array $A$
- Sufficient condition: $B$ reads or write only one element of $A$
  - Let $f : Val \rightarrow \wp(\Phi)$
  - $v \mapsto f(v) = \{ \Phi \mid \exists \sigma, \sigma(i) = v \land \Phi \in \mathcal{W}_B(\sigma) \cap \mathcal{R}_B(\sigma) \}$
  - If $f$ is a mapping, array $A$ can be replaced by a scalar
- Copy-in and copy-out according to $IN_B$ and $OUT_B$
- Profitability criteria necessary
  - at least $n$ references
- Not limited to loops
Goal: replace a statement $S$ in a program $P$ executing in a memory $M$ by a statement $\text{Isol}(S)$ executing in a different memory space $M'$.

$\sigma_1 \xrightarrow{\text{Load}(S, \sigma_1)} S \xrightarrow{\text{Isol}(S)} \sigma_2$

$\sigma_1' \xrightarrow{\text{Isol}(S)} \sigma_2'$

- Related to outlining, privatization, localization...
- Need for load/store optimization for more efficiency
- Algorithm implemented in PIPS based on R/W regions
- Could be based on IN/OUT regions
  - implicitly privatizes variables
  - changes the generation of new variables
Statement Isolation: Algorithm with R/W Regions

1. Compute the memory transfers from/to $M$:
   \[
   \text{Store}(S, \sigma_1) = \bar{W}_S(\sigma_1) \\
   \text{Load}(S, \sigma_1) = \bar{R}_S(\sigma_1) \cup (\text{Store}(S, \sigma_1) \bigcap \bar{W}_S(\sigma_1))
   \]
   Note: Rectangular hulls ([[]]) are necessary because of memory transfer functions.

2. Declare a new variable $\text{New}(\nu)$ for each variable $\nu$ which is loaded or stored.
   for \text{short int } \nu[\text{dim1}][\text{dim2}], if
   \[
   \text{Load}(S, \sigma_1) = \{\nu[\phi_1][\phi_2] | \alpha_1 \leq \phi_1 \leq \beta_1, \alpha_2 \leq \phi_2 \leq \beta_2\}
   \]
   declare \text{short int } \text{new}_\nu[\beta_1 - \alpha_1][\beta_2 - \alpha_2]

3. Assuming the availability of:
   \[
   \text{size_t memcpy2d}(\text{void* } \text{dest}, \text{void* } \text{src}, \text{size_t } \text{dim1}, \text{size_t } \text{offset1}, \text{size_t } \text{count1}, \text{size_t } \text{dim2}, \text{size_t } \text{offset2}, \text{size_t } \text{count2});
   \]
   Generate:
   \[
   \text{memcpy2d}(\text{new}_\nu, \nu, \text{dim1}, \alpha_1, \beta_1 - \alpha_1, \text{dim}_2, \alpha_2, \beta_2 - \alpha_2);
   \]

4. Substitute all references to $\nu$ by references to $\text{new}_\nu$ in $S$:
   \[
   \nu[i][j] \sim \text{new}_\nu[i+\alpha_1][j+\alpha_2]
   \]
Statement Isolation : Example

```c
void foo(int N, int a[N], int randv[N]) {
    int x=N/4, y=0;
    // \mathcal{R} = \{x, y, N\} \ W = \{x, y\}
    // \overline{\mathcal{R}}(\text{randv}) = \{\text{randv}[\phi_1] | \frac{N-3}{4} \leq \phi_1 \leq \frac{N}{3}\} \ \overline{W}(a) = \{a[\phi_1] | \frac{N-3}{4} \leq \phi_1 \leq \frac{5*N+9}{12}\}
    while (x<=N/3) {
        a[x+y] = x+y;
        if (randv[x-y]) x = x+2; else x++, y++;
    }
}
```
Statement Isolation : Example

```c
void foo(int N, int a[N], int randv[N]) {
    int x=N/4, y=0;
    // R = {x,y,N} \ W = {x,y}
    // R(randv) = {randv[\phi_1]| \frac{N-3}{4} \leq \phi_1 \leq \frac{N}{3}} \ W(a) = {a[\phi_1]| \frac{N-3}{4} \leq \phi_1 \leq \frac{5N+9}{12}}
    while (x<=N/3) {
        a[x+y] = x+y;
        if (randv[x-y]) x = x+2; else x++, y++;
    }
}
```

```c
void foo(int N, int a[N], int randv[N]) {
    int x=0, y=0;

    int new_a[N/6], new_randv[(N-9)/12], X, Y;

    memcpy(new_a, a+(N-3)/4, N/6*sizeof(int));
    memcpy(new_randv, randv+(N-3)/4, (N-9)/12*sizeof(int));
    memcpy(&new_x, &x, sizeof(x));
    memcpy(&new_y, &y, sizeof(y));
    memcpy(&new_N, &N, sizeof(x));

    while (new_x<=new_N/3) {
        A[new_x+new_y-(N-3)/4] = new_x+new_y;
        if (new_randv[new_x+new_y-(new_N-3)/4]) new_x = new_x+2; else new_x++, new_y++;
    }

    memcpy(a+(N-3)/4, new_a, N/6*sizeof(int));
    memcpy(&x, &new_x, sizeof(x));
    memcpy(&y, &new_y, sizeof(y));
}
```
// \( \mathcal{R}(\text{src}) = \{ \text{src}[\phi_1] | i \leq \phi_1 \leq i + k - 1 \} \)
// \( \mathcal{W}(\text{dst}) = \{ \text{dst}[\phi_1] | \phi_1 = i \} \)
// \( \mathcal{R}(\text{m}) = \{ \text{m}[\phi_1] | 0 \leq \phi_1 \leq k - 1 \} \)
int kernel(int i, int n, int k,
    int src[n], int dst[n-k],
    int m[k]) {
    int v=0;
    for( int j = 0; j < k; ++j )
        v += src[ i + j ] * m[ j ];
    dst[i]=v;
}

void fir(int n, int k,
    int src[n], int dst[n-k], int m[k])
{
    for( int i = 0; i < n - k + 1; ++i )
        kernel(i, n, k, src, dst, m);
}

// \( \mathcal{R}(\text{src}) = \left\{ \text{src}[\phi_1] \big| \begin{array}{l}
    i \leq \phi_1 \leq i + k - 1, \\
    0 \leq i \leq n - k
\end{array} \right. \)
// \( \mathcal{R}(\text{m}) = \{ \text{m}[\phi_1] | 0 \leq \phi_1 \leq k - 1 \} \)
// \( \mathcal{W}(\text{dst}) = \{ \text{dst}[\phi_1] | \phi_1 = i \} \)
Statement Isolation: Interprocedural Example

\[
\mathcal{R}(\text{src}) = \{ \text{src}[i] | i \leq \phi_1 \leq i + k - 1 \}
\]
\[
\mathcal{W}(\text{dst}) = \{ \text{dst}[\phi_1] | \phi_1 = i \}
\]
\[
\mathcal{R}(m) = \{ m[\phi_1] | 0 \leq \phi_1 \leq k - 1 \}
\]
\[
\mathcal{W}(\text{dst}) = \{ \text{dst}[\phi_1] | \phi_1 = i \}
\]

```c
int kernel(int i, int n, int k,
           int src[n], int dst[n-k],
           int m[k]) {
    
    int v=0;
    for( int j = 0; j < k; ++j )
        v += src[ i + j ] * m[ j ];
    dst[i]=v;
}
```

```c
void fir(int n, int k,
         int src[n], int dst[n-k], int m[k])
{
    for( int i = 0; i < n - k + 1; ++i )
    // \( \mathcal{R}(\text{src}) = \{ \text{src}[i] | 0 \leq i \leq n - k \} \)
    // \( \mathcal{R}(m) = \{ m[0] | 0 \leq i \leq k - 1 \} \)
    // \( \mathcal{W}(\text{dst}) = \{ \text{dst}[\phi_1] | \phi_1 = i \} \)
        kernel(i, n, k, src, dst, m);
}
```

```c
void KERNEL(int i, int n, int k,
            int SRC[k], int DST[1],
            int M[k],
            int SRC_offset, int DST_offset,
            int M_offset) {
    int v=0;
    for( int j = 0; j < k; ++j )
        v += SRC[i+j-SRC_offset]*M[j-M_offset];
    DST[i-SRC_offset]=v;
}
```

```c
void fir(int n, int k,
         int src[n], int dst[n-k], int m[k])
{
    for( int i = 0; i < n - k + 1; ++i )
    {
        int DST[1],SRC[k],M[k];
        memcpy(SRC, src+i, k*sizeof(int));
        memcpy(M, m+0, k*sizeof(int));
        KERNEL(i, n, k, SRC, DST, M,
               i/*SRC*/, i/*DST*/, 0/*M*/);
        memcpy(dst, DST+0, 1*sizeof(int));
    }
}
Generation of communications for TERAPIX, an image-processing accelerator.

Generation of inter-tasks communications for an asymmetric MP-SoC with cores dedicated to task scheduling in the SCALOPES project.

Generation of data transfers between different fields of structures in the SMECY project (tool chain for embedded multi-core architectures)

Code generation for GPUs in Par4All.
Nga Nguyen’s thesis [Ngu02]

- Array declaration checking and fixing
- Array bound checking
- Alias analysis
- Uninitialized array element sets
- Putting it all together
Static Analysis Using Array Regions

Nga Nguyen’s thesis [Ngu02]

- Array declaration checking and fixing
- Array bound checking
- Alias analysis
- Uninitialized array element sets
- Putting it all together
Array Bound Checking

Dynamic Analysis

- Guard each array reference with two checks (upper and lower bound)
- Very costly!

Static Analysis

- Statically check bounds using semantical information
- Lots of false positives due to over-approximations

Statically Optimized Dynamic Analysis

- Elimination of redundant tests using preconditions
- Insertion of unavoidable tests using regions
  - Usual assumption for program transformations:
    THE CODE IS CORRECT $\rightarrow$ no more true!
  - So array regions may not be safe!
Insertion of Unavoidable Tests using Regions

Algorithm Foundations

- Top-down from upper statement $S$
- Prove violation or not, or insert tests at the uppermost possible level
- But regions maybe erroneous :
  - A region at point $p_1$ originating from a subsequent program point $p_2$ is safe if any write to an array on any path from $p_1$ to $p_2$ does not violate array bounds.

Properties on Regions wrt Declarations

For a statement $S$ and an array $A$,

1. if $\forall \sigma \in \overline{P}_S, \overline{R}_A[S](\sigma) \subseteq \text{DECL}_A(\sigma)$
   then there is no bound violation for array $A$ in $S$.

2. if $\forall \sigma \in \overline{P}_S, \overline{R}_A[S](\sigma) \cap (\neg \text{DECL}_A(\sigma)) \neq \emptyset$
   then there is certainly a bound violation for array $A$ in $S$. 
procedure insertion_of_unavoidable_tests_at_statement(S)
    while pick_next_written_array(S, A) is computable
        for each region R of A in ℝ[S] or ℝ[S]
            p_1 = R ∩ Decl(A)
            dnf_2 = R ∩ ¬ Decl(A)
            if infeasible(p_1) then
                no bound violation for A at S
            else if R is exact and infeasible(dnf_2) then
                bound violation for A at S
            else if R is exact
                and p=eliminate_phi_variables(p_1) is exact then
                insert unavoidable test
            else
                for each substatement SS of S
                    insertion_of_unavoidable_tests_at_statement(SS)
        if there are unchecked arrays then
            for each substatement SS of S
                insertion_of_unavoidable_tests_at_statement(SS)
program violation
common ITAB(10), J
real A(10)

read *, M

j = 11

do i = 1, M
   ITAB(i) = 1
enddo

A(j) = 0
end
program violation
common ITAB(10), J
real A(10)

// $\overline{W}(\text{ITAB}(\phi_1)) = \{ \phi_1 | 1 \leq \phi_1 \}$
// $W(A(\phi_1)) = \{ \phi_1 | \phi_1 = 11 \}$
read *, M

j=11

do i = 1, M
   ITAB(i) = 1
enddo

A(j) = 0
end
program violation
common ITAB(10), J
real A(10)

read *, M
// $\mathcal{W}(\text{ITAB}(\phi_1)) = \{\phi_1 | 1 \leq \phi_1 \leq M\}$
// $\mathcal{W}(\text{A}(\phi_1)) = \{\phi_1 | \phi_1 = 11\}$

j = 11

do i = 1, M
    ITAB(i) = 1
endo

A(j) = 0
end
program violation
common ITAB(10), J
real A(10)

read *, M
if (11.1e.M) STOP "Bound violation: array ITAB, 1st dimension"
// \( \mathcal{W}(A(\phi_1)) = \{ \phi_1 | \phi_1 = 11 \} \)
j=11

do i = 1, M
   ITAB(i) = 1
enddo

A(j) = 0
end
program violation
common ITAB(10), J
real A(10)

read *, M
if (11.1e.M) STOP "Bound violation: array ITAB, 1st dimension"
STOP "Bound violation: array A, 1st dimension"
j=11

do i = 1, M
    ITAB(i) = 1
endo

A(j) = 0
end
Outline

Foundations
- Context/History
- Nature of array element sets?
- Approximations
- Read and Write Regions
- IN and OUT regions
- Extension to the C Language

APPLICATIONS
- Loop Transformations
- Memory Management
- Static Analysis

CONCLUSION
Conclusion

The expression “Array regions” is used to designate:

- the theoretical framework for array element sets analyses
- R/W and IN/OUT sets
- the implementation based convex polyhedra
  - easy to combine with other analyses based on polyhedra

Array regions are useful for many applications

- loop transformations (parallelization, fusion)
- memory management
  - localization of variables
  - communications

And thus to generate code for many targets

- Data parallelism with shared memory (OpenMP)
- Heterogeneous distributed memory architectures (GPUs)
- Task parallelism . . .
If You Want to Go Further...

Read some PhD theses, papers,...
- www.cri.ensmp.fr/list_publications.html

The simplest first: try Par4All
- www.par4all.org
- p4a --openmp sample.c
- but tunable/extendable via its options

Want to have full control over the situation: try PIPS
- www.pips4u.org
- write your own tpips or pyps scripts!
- see pipsmake-rc.pdf for options/properties
- write your own phase using regions

Want to get involved in Par4All 2.x

Contact us at SILKAN!
References


