Polyhedral Compilation without Polyhedra

Sven Verdoolaege

INRIA and KU Leuven
Sven.Verdoolaege@inria.fr

January 20, 2015
Outline

1 Polyhedral Model
   - Introduction
   - Representation

2 Polyhedral Transformation
   - Schedules
   - AST Generation

3 Dependences
   - Schedule Validity
   - Dependences
   - Structures

4 Dataflow
   - Parallelism
   - Dataflow
   - Approximate Dataflow
   - Run-time dependent Dataflow
   - Reductions

5 Aliasing

6 Counting
   - Cardinality
   - Bounds
   - Weighted Counting
   - Dynamic Memory Requirement

7 Transitive Closures

8 Tools
Part I

Polyhedral Compilation
Outline

1 Polyhedral Model
   - Introduction
   - Representation

2 Polyhedral Transformation
   - Schedules
   - AST Generation

3 Dependences
   - Schedule Validity
   - Dependences
   - Structures

4 Dataflow
   - Parallelism
   - Dataflow
   - Approximate Dataflow
   - Run-time dependent Dataflow
   - Reductions

5 Aliasing

6 Counting
   - Cardinality
   - Bounds
   - Weighted Counting
   - Dynamic Memory Requirement

7 Transitive Closures
Polyhedral Compilation

Analyzing and/or transforming loop programs using the *polyhedral model*

Polyhedral Model

Abstract representation of a loop program

- instance based
  - statement *instances*
  - array *elements*
- compact representation based on polyhedra or similar objects
  - integer points in unions of parametric polyhedra
  - Presburger sets and relations
- parametric
  - description may depend on symbolic constants
Polyhedral Compilation

Analyzing and/or transforming loop programs using the *polyhedral model*

Polyhedral Model

Abstract representation of a loop program

- instance based
  - statement *instances*
  - array *elements*

- compact representation based on polyhedra or similar objects
  - integer points in unions of parametric polyhedra
  - Presburger sets and relations

- parametric
  - description may depend on symbolic constants

Note: naming is “historical”

- polyhedral compilation does not require polyhedra (e.g., omega(+) )
- other approaches also use polyhedra (e.g., abstract interpretation)
Polyhedral Model

Main constituents of program representation

- **Instance Set**
  - the set of all statement instances

- **Access Relations**
  - the array elements accessed by a statement instance

- **Dependences**
  - the statement instances that depend on a statement instance

- **Schedule**
  - the relative execution order of statement instances
Polyhedral Model Requirements

Requirements for basic polyhedral model: SANA input

- Static control
  \[ \Rightarrow \text{control does not depend on input data} \]

- Affine
  \[ \Rightarrow \text{all relevant expressions are (quasi-)affine} \]

- No Aliasing
  \[ \Rightarrow \text{essentially no pointer manipulations} \]
Polyhedral Model Requirements

Requirements for basic polyhedral model: SANA input

- Static control
  ⇒ control does not depend on input data

- Affine
  ⇒ all relevant expressions are (quasi-)affine

- No Aliasing
  ⇒ essentially no pointer manipulations

Note:

- polyhedral model may be *approximation* of input that does not strictly satisfy all requirements

- many *extensions* are available
  a small selection of these extensions will be discussed in this tutorial
Illustrative Example

R: \( h(A[2]); \)
   for (int \( i = 0; i < 2; ++i \))
      for (int \( j = 0; j < 2; ++j \))
         \( A[i + j] = f(i, j); \)

S: \( A[i + j] = f(i, j); \)
   for (int \( k = 0; k < 2; ++k \))
      \( g(A[k], A[0]); \)
Illustrative Example

\[ \text{R: } h(A[2]); \]
\[ \text{for (int } i = 0; i < 2; ++i) \]
\[ \text{for (int } j = 0; j < 2; ++j) \]
\[ \text{S: } A[i + j] = f(i, j); \]
\[ \text{for (int } k = 0; k < 2; ++k) \]
\[ \text{T: } g(A[k], A[0]); \]

- **Instance Set** (set of statement instances)
  \[ I = \{ \text{R(); S(0, 0); S(0, 1); S(1, 0); S(1, 1); T(0); T(1)} \} \]
Illustrative Example

R: \( h(A[2]) \);
   for (int i = 0; i < 2; ++i)
      for (int j = 0; j < 2; ++j)
         S: \( A[i + j] = f(i, j) \);
   for (int k = 0; k < 2; ++k)
      T: \( g(A[k], A[0]) \);

Instance Set (set of statement instances)
\[ I = \{ R(); S(0, 0); S(0, 1); S(1, 0); S(1, 1); T(0); T(1) \} \]

Access Relations (accessed array elements; \( W \): write, \( R \): read)
\[ W = \{ S(0, 0) \rightarrow A(0); S(0, 1) \rightarrow A(1); S(1, 0) \rightarrow A(1);
   S(1, 1) \rightarrow A(2) \} \]
\[ R = \{ R() \rightarrow A(2); T(0) \rightarrow A(0); T(1) \rightarrow A(1); T(1) \rightarrow A(0) \} \]
Illustrative Example

R: $h(A[2]);$
  for (int $i = 0; i < 2; ++i)$
    for (int $j = 0; j < 2; ++j)$
      $A[i + j] = f(i, j);$ 
  
S: $A[i + j] = f(i, j);$ 
  for (int $k = 0; k < 2; ++k)$
    $g(A[k], A[0]);$

- Instance Set (set of statement instances)
  $I = \{ R(); S(0, 0); S(0, 1); S(1, 0); S(1, 1); T(0); T(1) \}$

- Access Relations (accessed array elements; $W$: write, $R$: read)
  $W = \{ S(0, 0) \rightarrow A(0); S(0, 1) \rightarrow A(1); S(1, 0) \rightarrow A(1);$
  $S(1, 1) \rightarrow A(2) \}$
  $R = \{ R() \rightarrow A(2); T(0) \rightarrow A(0); T(1) \rightarrow A(1); T(1) \rightarrow A(0) \}$

- Schedule (relative execution order)
  $R(), S(0, 0), S(0, 1), S(1, 0), S(1, 1), T(0), T(1)$
Illustrative Example

\[
R: \quad h(A[2]); \\
\text{for (int } i = 0; i < 2; ++i) \\
\quad \text{for (int } j = 0; j < 2; ++j) \\
S: \quad A[i + j] = f(i, j); \\
\text{for (int } k = 0; k < 2; ++k) \\
T: \quad g(A[k], A[0]);
\]

- **Instance Set** (set of statement instances)
  \[
  I = \{ R(); S(0, 0); S(0, 1); S(1, 0); S(1, 1); T(0); T(1) \}
  \]

- **Access Relations** (accessed array elements; \(W\): write, \(R\): read)
  \[
  W = \{ S(0, 0) \rightarrow A(0); S(0, 1) \rightarrow A(1); S(1, 0) \rightarrow A(1); \\
  \quad S(1, 1) \rightarrow A(2) \}
  \\
  R = \{ R() \rightarrow A(2); T(0) \rightarrow A(0); T(1) \rightarrow A(1); T(1) \rightarrow A(0) \}
  \]

- **Schedule** (relative execution order)
  \[
  R(), S(0, 0), S(0, 1), S(1, 0), S(1, 1), T(0), T(1)
  \]
Illustrative Example

\[ R: \quad h(A[2]); \]
for (int \( i = 0; i < 2; ++i \))
for (int \( j = 0; j < 2; ++j \))
\[ S: \quad A[i + j] = f(i, j); \]
for (int \( k = 0; k < 2; ++k \))
\[ T: \quad g(A[k], A[0]); \]

- **Instance Set** (set of statement instances)
  \[ I = \{ R(); S(0, 0); S(0, 1); S(1, 0); S(1, 1); T(0); T(1) \} \]

- **Access Relations** (accessed array elements; \( W \): write, \( R \): read)
  \[ W = \{ S(0, 0) \rightarrow A(0); S(0, 1) \rightarrow A(1); S(1, 0) \rightarrow A(1); S(1, 1) \rightarrow A(2) \} \]

- **Constraints**
  "read off", or
  obtained through abstract interpretation

- **Schedule** (relative execution order)
  \[ R(), S(0, 0), S(0, 1), S(1, 0), S(1, 1), T(0), T(1) \]
Illustrative Example

R: \( h(A[2]) \);
   \[
   \text{for (int } i = 0; i < 2; ++i) \\
   \text{for (int } j = 0; j < 2; ++j) \\
   \]
S: \( A[i + j] = f(i, j) \);
   \[
   \text{for (int } k = 0; k < 2; ++k) \\
   \]
T: \( g(A[k], A[0]) \);

- Instance Set (set of statement instances)
  \[
  I = \{ R(); S(i,j) : 0 \leq i < 2 \land 0 \leq j < 2; T(k) : 0 \leq k < 2 \}
  \]
- Access Relations (accessed array elements; \( W \): write, \( R \): read)
  \[
  W = \{ S(0,0) \rightarrow A(0); S(0,1) \rightarrow A(1); S(1,0) \rightarrow A(1); S(1,1) \rightarrow A(2) \}
  \]
  \[
  R = \{ R() \rightarrow A(2); T(0) \rightarrow A(0); T(1) \rightarrow A(1); T(1) \rightarrow A(0) \}
  \]
- Schedule (relative execution order)
  \[
  R(), S(0,0), S(0,1), S(1,0), S(1,1), T(0), T(1)
  \]
Illustrative Example

R: \( h(A[2]); \)
   for (int i = 0; i < 2; ++i)
      for (int j = 0; j < 2; ++j)
         A[i + j] = f(i, j);

S: \( g(A[k], A[0]); \)
   for (int k = 0; k < 2; ++k)
      T[k] = g(A[k], A[0]);

- Instance Set (set of statement instances)
  \( I = \{ R(); S(i, j) : 0 \leq i < 2 \land 0 \leq j < 2; T(k) : 0 \leq k < 2 \} \)

- Access Relations (accessed array elements; \( W: \) write, \( R: \) read)
  \( W = \{ S(0, 0) \rightarrow A(0); S(0, 1) \rightarrow A(1); S(1, 0) \rightarrow A(1); S(1, 1) \rightarrow A(2) \} \)
  \( R = \{ S(i, j) \rightarrow A(i + j) : 0 \leq i < 2 \land 0 \leq j < 2 \} \rightarrow A(0) \} \)

- Schedule (relative execution order)
  \( R(), S(0, 0), S(0, 1), S(1, 0), S(1, 1), T(0), T(1) \)
Illustrative Example

R: \[ h(A[2]); \]
for (int i = 0; i < 2; ++i)
for (int j = 0; j < 2; ++j)
S: \[ A[i + j] = f(i, j); \]
for (int k = 0; k < 2; ++k)
T: \[ g(A[k], A[0]); \]

- **Instance Set** (set of statement instances)
  \[ I = \{ R(); S(i, j): 0 \leq i < 2 \land 0 \leq j < 2; T(k): 0 \leq k < 2 \} \]

- **Access Relations** (accessed array elements; \( W: \) write, \( R: \) read)
  \[ W = \{ S(i, j) \rightarrow A(i + j): 0 \leq i < 2 \land 0 \leq j < 2 \} \]

  \[ R = \{ R() \rightarrow A(2); T(0) \rightarrow A(0); T(1) \rightarrow A(1); T(1) \rightarrow A(0) \} \]

- **Schedule** (relative execution order)
  \[ R(), S(0, 0), S(0, 1), S(1, 0), S(1, 1), T(0), T(1) \]
Illustrative Example

R: \( h(A[2]); \)
   for (int i = 0; i < 2; ++i)
      for (int j = 0; j < 2; ++j)
         \( S: A[i + j] = f(i, j); \)
      for (int k = 0; k < 2; ++k)
         \( T: g(A[k], A[0]); \)

- Instance Set (set of statement instances)
  \[ I = \{ R(); S(i, j) : 0 \leq i < 2 \land 0 \leq j < 2; T(k) : 0 \leq k < 2 \} \]

- Access Relations (accessed array elements; \( W: \text{write}, R: \text{read} \))
  \[ W = \{ S(i, j) \rightarrow A(i + j) : 0 \leq i < 2 \land 0 \leq j < 2 \} \]

- Instance based compact representation

\[ R = \{ R() \rightarrow A(2); T(0) \rightarrow A(0); T(1) \rightarrow A(1); T(1) \rightarrow A(0) \} \]

\[ \text{Sche} = \{ R() \rightarrow A(2); T(k) \rightarrow A(0) : 0 \leq k < 2; T(k) \rightarrow A(k) : 0 \leq k < 2 \} \]

R(), S(0, 0), S(0, 1), S(1, 0), S(1, 1), T(0), T(1)
Illustrative Example

R: \[ h(A[2]); \]
   for (int i = 0; i < 2; ++i)
      for (int j = 0; j < 2; ++j)
          A[i + j] = f(i, j);

S: \[ A[i + j] = f(i, j); \]
   for (int k = 0; k < 2; ++k)
      g(A[k], A[0]);

Instance Set (set of statement instances)
\[ I = \{ R(); S(i, j) : 0 \leq i < 2 \land 0 \leq j < 2; T(k) : 0 \leq k < 2 \} \]

Access Relations (accessed array elements; \( W \): write, \( R \): read)
\[ W = \{ S(i, j) \rightarrow A(i + j) : 0 \leq i < 2 \land 0 \leq j < 2 \} \]

\[ R = \{ R() \rightarrow A(2); T(k) \rightarrow A(0) : 0 \leq k < 2; T(k) \rightarrow A(k) : 0 \leq k < 2 \} \]

Schedule (relative execution order)
\[ R(), S(0, 0), S(0, 1), S(1, 0), S(1, 1), T(0), T(1) \]
Parametric Example: Matrix Multiplication

```c
for (int i = 0; i < M; i++)
    for (int j = 0; j < N; j++) {
        S1:  C[i][j] = 0;
            for (int k = 0; k < K; k++)
        S2:  C[i][j] = C[i][j] + A[i][k] * B[k][j];
    }

- **Instance Set** (set of statement instances)

  \[
  \begin{align*}
  \{ & S1(i, j) : 0 \leq i < M \land 0 \leq j < N; \\
  & S2(i, j, k) : 0 \leq i < M \land 0 \leq j < N \land 0 \leq k < K \}
  \end{align*}
  \]

- **Access Relations** (accessed array elements; \(W\): write, \(R\): read)

  \[
  \begin{align*}
  W &= \{ S1(i, j) \rightarrow C(i, j); S2(i, j, k) \rightarrow C(i, j) \} \\
  R &= \{ S2(i, j, k) \rightarrow C(i, j); S2(i, j, k) \rightarrow A(i, k); S2(i, j, k) \rightarrow B(k, j) \}
  \end{align*}
  \]

- **Schedule** (relative execution order)

  \(S1(0, 0), S2(0, 0, 0), S2(0, 0, 1), \ldots, S1(0, 1), S2(0, 1, 0), S2(0, 1, 1), \ldots, \)
Named Presburger Sets and Relations

Examples

\{ R(); S(i,j) : 0 \leq i < 2 \land 0 \leq j < 2; T(k) : 0 \leq k < 2 \} \\
\{ R() \rightarrow A(2); T(k) \rightarrow A(0) : 0 \leq k < 2; T(k) \rightarrow A(k) : 0 \leq k < 2 \} \\

General form

- Sets
  \{ S_1(i) : f_1(i); S_2(i) : f_2(i); \ldots \},
  with \( f_k \) Presburger formulas
  \Rightarrow set of elements of the form \( S_1(i) \), one for each \( i \) satisfying \( f_1(i) \), \ldots

- Binary relations
  \{ S_1(i) \rightarrow T_1(j) : f_1(i,j); S_2(i) \rightarrow T_2(j) : f_2(i,j); \ldots \} \\
  \Rightarrow set of pairs of elements of the form \( S_1(i) \rightarrow T_1(j) \)
Named Presburger Sets and Relations

Examples

\{ R(); S(i, j) : 0 \leq i < 2 \land 0 \leq j < 2; T(k) : 0 \leq k < 2 \} \\
\{ R() \rightarrow A(2); T(k) \rightarrow A(0) : 0 \leq k < 2; T(k) \rightarrow A(k) : 0 \leq k < 2 \} \\

General form

- **Sets**

  \{ S_1(i) : f_1(i); S_2(i) : f_2(i); \ldots \},

  with \( f_k \) Presburger formulas

  \Rightarrow \text{set of elements of the form } S_1(i), \text{ one for each } i \text{ satisfying } f_1(i), \ldots

- **Binary relations**

  \{ S_1(i) \rightarrow T_1(j) : f_1(i, j); S_2(i) \rightarrow T_2(j) : f_2(i, j); \ldots \}

  \Rightarrow \text{set of pairs of elements of the form } S_1(i) \rightarrow T_1(j)

Note: despite “\( \rightarrow \)”, not necessarily (single valued) functions
Named Presburger Sets and Relations

General form

- Sets

\[ \{ S_1(i) : f_1(i); S_2(i) : f_2(i); \ldots \}, \]

where \( f_k(i) \) are Presburger formulas with \( i \) as only free variables

\[ \Rightarrow \text{set of elements of the form } S_1(i), \text{ one for each } i \text{ satisfying } f_1(i), \ldots \]
Named Presburger Sets and Relations

General form

- Sets

\[ \{ S_1(i) : f_1(i); S_2(i) : f_2(i); \ldots \}, \]

where \( f_k(i) \) are Presburger formulas with \( i \) as only free variables

\[ \Rightarrow \] set of elements of the form \( S_1(i) \), one for each \( i \) satisfying \( f_1(i), \ldots \)

Note: may depend on interpretation of symbolic constants

\[ \{ S(i) : 0 \leq i \leq n() \} \]

is equal to

\[
\begin{cases} 
\emptyset & \text{if } n < 0 \\
\{ S(0) \} & \text{if } n = 0 \\
\{ S(0); S(1) \} & \text{if } n = 1 \\
\{ S(0); S(1); S(2) \} & \text{if } n = 2 \\
\ldots & 
\end{cases}
\]
Quasi-Affine Expressions and Presburger Formulae

- **Symbolic Constant**
  - has unknown but fixed value
  - typically used to represent size parameter
Quasi-Affine Expressions and Presburger Formulae

- **Symbolic Constant**
  - has unknown but fixed value
  - typically used to represent size parameter

- **Quasi-Affine Expression**
  - variable
  - symbolic constant
  - integer constant
  - addition (+), subtraction (−)
  - integer division by a constant (⌊·/d⌋)
Quasi-Affine Expressions and Presburger Formulae

- **Symbolic Constant**
  - has unknown but fixed value
  - typically used to represent size parameter

- **Quasi-Affine Expression** $\leadsto$ **Presburger Term**
  - variable
  - symbolic constant
  - integer constant
  - addition ($+$), subtraction ($-$)
  - integer division by a constant ($\lfloor \cdot / d \rfloor$)

- **Presburger Formula**
  - true
  - equality on terms ($=$)
  - less than or equal on terms ($\leq$)
  - logical connectives ($\land, \lor, \neg$)
  - quantification ($\exists, \forall$)
Syntactic Sugar

- \texttt{false} is equal to \texttt{\neg true}
Syntactic Sugar

- `false` is equal to `¬true`
- `a ⇒ b` is equal to `¬a ∨ b`
Syntactic Sugar

- false is equal to \( \neg \text{true} \)
- \( a \implies b \) is equal to \( \neg a \lor b \)
- \( \{ S(i) \} \) is equal to \( \{ S(i) : \text{true} \} \)
Syntactic Sugar

- \texttt{false} is equal to \( \neg \text{true} \)
- \( a \Rightarrow b \) is equal to \( \neg a \lor b \)
- \{ \text{\( S(i) \)} \} is equal to \{ \( S(i) : \text{true} \) \}
- \{ \text{\( S(i_1, \ldots, i_{n-1}, g(i_1, \ldots, i_{n-1}), i_{n+1}, \ldots) : f(i) \)} \} is equal to
  \{ \text{\( S(i_1, \ldots, i_{n-1}, i_n, i_{n+1}, \ldots) : f(i) \land i_n = g(i_1, \ldots, i_{n-1}) \)} \}
Example: \{ \text{\( S(j) \rightarrow S(i+1) \)} \} is equal to \{ \text{\( S(i) \rightarrow S(j) : j = i + 1 \)} \}
Syntactic Sugar

- **false** is equal to \( \neg \text{true} \)
- \( a \Rightarrow b \) is equal to \( \neg a \lor b \)
- \( \{ S(i) \} \) is equal to \( \{ S(i) : \text{true} \} \)
- \( \{ S(i_1, \ldots, i_{n-1}, g(i_1, \ldots, i_{n-1}), i_{n+1}, \ldots) : f(i) \} \) is equal to \( \{ S(i_1, \ldots, i_{n-1}, i_n, i_{n+1}, \ldots) : f(i) \land i_n = g(i_1, \ldots, i_{n-1}) \} \)
  
  Example: \( \{ S(j) \rightarrow S(i + 1) \} \) is equal to \( \{ S(i) \rightarrow S(j) : j = i + 1 \} \)
- \( n \) is equal to \( n() \)
Syntactic Sugar

- false is equal to \( \neg \text{true} \)
- \( a \Rightarrow b \) is equal to \( \neg a \lor b \)
- \( \{ S(i) \} \) is equal to \( \{ S(i) : \text{true} \} \)
- \( \{ S(i_1, \ldots, i_{n-1}, g(i_1, \ldots, i_{n-1}), i_{n+1}, \ldots) : f(i) \} \) is equal to
  \( \{ S(i_1, \ldots, i_{n-1}, i_n, i_{n+1}, \ldots) : f(i) \land i_n = g(i_1, \ldots, i_{n-1}) \} \)
  Example: \( \{ S(j) \rightarrow S(i + 1) \} \) is equal to \( \{ S(i) \rightarrow S(j) : j = i + 1 \} \)
- \( n \) is equal to \( n() \)
- \( a < b \) is equal to \( a \leq b - 1 \)
Syntactic Sugar

- `false` is equal to \( \neg \text{true} \)
- \( a \Rightarrow b \) is equal to \( \neg a \lor b \)
- \( \{ S(i) \} \) is equal to \( \{ S(i) : \text{true} \} \)
- \( \{ S(i_1, \ldots i_{n-1}, g(i_1, \ldots, i_{n-1}), i_{n+1}, \ldots) : f(i) \} \) is equal to \( \{ S(i_1, \ldots i_{n-1}, i_n, i_{n+1}, \ldots) : f(i) \land i_n = g(i_1, \ldots, i_{n-1}) \} \)
  - Example: \( \{ S(i) \rightarrow S(i + 1) \} \) is equal to \( \{ S(i) \rightarrow S(j) : j = i + 1 \} \)
- \( n \) is equal to \( n() \)
- \( a < b \) is equal to \( a \leq b - 1 \)
- \( a \geq b \) is equal to \( b \leq a \)
Syntactic Sugar

- \( \text{false} \) is equal to \( \neg \text{true} \)
- \( a \implies b \) is equal to \( \neg a \lor b \)
- \( \{ S(i) \} \) is equal to \( \{ S(i) : \text{true} \} \)
- \( \{ S(i_1, \ldots i_{n-1}, g(i_1, \ldots, i_{n-1}), i_{n+1}, \ldots) : f(i) \} \) is equal to
  \( \{ S(i_1, \ldots i_{n-1}, i_n, i_{n+1}, \ldots) : f(i) \land i_n = g(i_1, \ldots, i_{n-1}) \} \)
- Example: \( \{ S(j) \rightarrow S(i + 1) \} \) is equal to \( \{ S(i) \rightarrow S(j) : j = i + 1 \} \)
- \( n \) is equal to \( n() \)
- \( a < b \) is equal to \( a \leq b - 1 \)
- \( a \geq b \) is equal to \( b \leq a \)
- \( a > b \) is equal to \( a \geq b + 1 \)
Syntactic Sugar

- false is equal to \( \neg \text{true} \)
- \( a \Rightarrow b \) is equal to \( \neg a \lor b \)
- \( \{ S(i) \} \) is equal to \( \{ S(i) : \text{true} \} \)
- \( \{ S(i_1, \ldots i_{n-1}, g(i_1, \ldots, i_{n-1}), i_n, \ldots) : f(i) \} \) is equal to \( \{ S(i_1, \ldots i_{n-1}, i_n, i_{n+1}, \ldots) : f(i) \land i_n = g(i_1, \ldots, i_{n-1}) \} \)
  Example: \( \{ S(i) \rightarrow S(i+1) \} \) is equal to \( \{ S(i) \rightarrow S(j) : j = i + 1 \} \)
- \( n \) is equal to \( n() \)
- \( a < b \) is equal to \( a \leq b - 1 \)
- \( a \geq b \) is equal to \( b \leq a \)
- \( a > b \) is equal to \( a \geq b + 1 \)
- \( a, b \oplus c \) is equal to \( a \oplus c \land b \oplus c \) with \( \oplus \in \{ \leq, <, \geq, >, = \} \)
  Example: \( \{ S(i, j) : i, j \geq 0 \} \) is equal to \( \{ S(i, j) : i \geq 0 \land j \geq 0 \} \)
Syntactic Sugar

- `false` is equal to \( \neg \text{true} \)
- `a \Rightarrow b` is equal to \( \neg a \lor b \)
- \( \{ S(i) \} \) is equal to \( \{ S(i) : \text{true} \} \)
- \( \{ S(i_1, \ldots, i_{n-1}, g(i_1, \ldots, i_{n-1}), i_{n+1}, \ldots) : f(i) \} \) is equal to \( \{ S(i_1, \ldots, i_{n-1}, i_n, i_{n+1}, \ldots) : f(i) \land i_n = g(i_1, \ldots, i_{n-1}) \} \)
  Example: \( \{ S(i) \rightarrow S(i + 1) \} \) is equal to \( \{ S(i) \rightarrow S(j) : j = i + 1 \} \)
- `n` is equal to \( n() \)
- `a < b` is equal to \( a \leq b - 1 \)
- `a \geq b` is equal to \( b \leq a \)
- `a > b` is equal to \( a \geq b + 1 \)
- `a, b \oplus c` is equal to \( a \oplus c \land b \oplus c \) with \( \oplus \in \{ \leq, <, \geq, >, = \} \)
  Example: \( \{ S(i, j) : i, j \geq 0 \} \) is equal to \( \{ S(i, j) : i \geq 0 \land j \geq 0 \} \)
- `a \oplus_1 b \oplus_2 c` is equal to \( a \oplus_1 b \land b \oplus_2 c \) with \( \{ \oplus_1, \oplus_2 \} \subset \{ \leq, <, \geq, >, = \} \)
  Example: \( \{ S(i) : 0 \leq i \leq 10 \} \) is equal to \( \{ S(i) : 0 \leq i \land i \leq 10 \} \)
Syntactic Sugar (2)

- $-e$ is equal to $0 - e$
Syntactic Sugar (2)

- $-e$ is equal to $0 - e$
- $n \cdot e$ is equal to $\underbrace{e + e + \cdots + e}_{n \text{ times}}$ (with $n$ a non-negative integer constant)

$a_1, \ldots, a_n \preceq b_1, \ldots, b_n$ is equal to $\lor_{i=1}^{n} ((\land_{j=1}^{i-1} a_j = b_j) \land a_i < b_i)$

Example:

$\{ S(i_1, i_2) \rightarrow S(j_1, j_2) : i_1, i_2 \prec j_1, j_2 \}$ is equal to

$\{ S(i_1, i_2) \rightarrow S(j_1, j_2) : i_1 < j_1 \lor (i_1 = j_1 \land i_2 < j_2) \}$

$a_1, \ldots, a_n \succeq b_1, \ldots, b_n$ is equal to

$b_1, \ldots, b_n \succeq a_1, \ldots, a_n$

$a_1, \ldots, a_n \preceq b_1, \ldots, b_n$ is equal to

$b_1, \ldots, b_n \preceq a_1, \ldots, a_n$

$a \bmod n$ is equal to $a - n \lfloor a/n \rfloor$
Syntactic Sugar (2)

- \(-e\) is equal to \(0 - e\)
- \(n \cdot e\) is equal to \(e + e + \cdots + e\) (with \(n\) a non-negative integer constant)
  
  \[n \text{ times}\]

- \(a_1, \ldots, a_n \preceq b_1, \ldots, b_n\) is equal to \(\bigvee_{i=1}^n \left( \left( \bigwedge_{j=1}^{i-1} a_j = b_j \right) \land a_i < b_i \right)\)

Example: \(\{ S(i_1, i_2) \rightarrow S(j_1, j_2) : i_1, i_2 \preceq j_1, j_2 \}\) is equal to \(\{ S(i_1, i_2) \rightarrow S(j_1, j_2) : i_1 < j_1 \lor (i_1 = j_1 \land i_2 < j_2) \}\)
Syntactic Sugar (2)

- $-e$ is equal to $0 - e$
- $n \cdot e$ is equal to $\underbrace{e + e + \cdots + e}_{n \text{ times}}$ (with $n$ a non-negative integer constant)
- $a_1, \ldots, a_n \prec b_1, \ldots, b_n$ is equal to $\bigvee_{i=1}^{n} \left( \left( \bigwedge_{j=1}^{i-1} a_j = b_j \right) \land a_i < b_i \right)$

Example: $\{ S(i_1, i_2) \rightarrow S(j_1, j_2) : i_1, i_2 \prec j_1, j_2 \}$ is equal to $\{ S(i_1, i_2) \rightarrow S(j_1, j_2) : i_1 < j_1 \lor (i_1 = j_1 \land i_2 < j_2) \}$

- $a_1, \ldots, a_n \preceq b_1, \ldots, b_n$ is equal to $a_1, \ldots, a_n \prec b_1, \ldots, b_n \lor a_1, \ldots, a_n = b_1, \ldots, b_n$
Syntactic Sugar (2)

- $-e$ is equal to $0 - e$
- $n \cdot e$ is equal to $\underbrace{e + e + \cdots + e}_{n \text{ times}}$ (with $n$ a non-negative integer constant)
- $a_1, \ldots, a_n \prec b_1, \ldots, b_n$ is equal to $\bigvee_{i=1}^{n} \left( \left( \bigwedge_{j=1}^{i-1} a_j = b_j \right) \land a_i < b_i \right)$
  
  Example: $\{ S(i_1, i_2) \rightarrow S(j_1, j_2) : i_1, i_2 \prec j_1, j_2 \}$ is equal to $\{ S(i_1, i_2) \rightarrow S(j_1, j_2) : i_1 < j_1 \lor (i_1 = j_1 \land i_2 < j_2) \}$
- $a_1, \ldots, a_n \preceq b_1, \ldots, b_n$ is equal to $a_1, \ldots, a_n \prec b_1, \ldots, b_n \lor a_1, \ldots, a_n = b_1, \ldots, b_n$
- $a_1, \ldots, a_n \succeq b_1, \ldots, b_n$ is equal to $b_1, \ldots, b_n < a_1, \ldots, a_n$
Syntactic Sugar (2)

- $-e$ is equal to $0-e$
- $n \cdot e$ is equal to $e + e + \cdots + e$ (with $n$ a non-negative integer constant)

\[ n \text{ times} \]

- $a_1, \ldots, a_n \prec b_1, \ldots, b_n$ is equal to $\bigvee_{i=1}^{n} \left( \left( \bigwedge_{j=1}^{i-1} a_j = b_j \right) \land a_i < b_i \right)$
  
  Example: $\{ S(i_1, i_2) \rightarrow S(j_1, j_2) : i_1, i_2 \prec j_1, j_2 \}$ is equal to $\{ S(i_1, i_2) \rightarrow S(j_1, j_2) : i_1 < j_1 \lor (i_1 = j_1 \land i_2 < j_2) \}$

- $a_1, \ldots, a_n \preceq b_1, \ldots, b_n$ is equal to $a_1, \ldots, a_n \prec b_1, \ldots, b_n \lor a_1, \ldots, a_n = b_1, \ldots, b_n$

- $a_1, \ldots, a_n \succ b_1, \ldots, b_n$ is equal to $b_1, \ldots, b_n \prec a_1, \ldots, a_n$

- $a_1, \ldots, a_n \succeq b_1, \ldots, b_n$ is equal to $b_1, \ldots, b_n \preceq a_1, \ldots, a_n$
Syntactic Sugar (2)

- $-e$ is equal to $0 - e$
- $n \cdot e$ is equal to $e + e + \cdots + e$ (with $n$ a non-negative integer constant)
- $a_1, \ldots, a_n \prec b_1, \ldots, b_n$ is equal to $\bigvee_{i=1}^{n} \left( \left( \bigwedge_{j=1}^{i-1} a_j = b_j \right) \land a_i < b_i \right)$

Example: \{ $S(i_1, i_2) \rightarrow S(j_1, j_2) : i_1, i_2 \prec j_1, j_2$ \} is equal to \{ $S(i_1, i_2) \rightarrow S(j_1, j_2) : i_1 < j_1 \lor (i_1 = j_1 \land i_2 < j_2)$ \}

- $a_1, \ldots, a_n \preceq b_1, \ldots, b_n$ is equal to $a_1, \ldots, a_n \prec b_1, \ldots, b_n \lor a_1, \ldots, a_n = b_1, \ldots, b_n$
- $a_1, \ldots, a_n \succ b_1, \ldots, b_n$ is equal to $b_1, \ldots, b_n \prec a_1, \ldots, a_n$
- $a_1, \ldots, a_n \succeq b_1, \ldots, b_n$ is equal to $b_1, \ldots, b_n \preceq a_1, \ldots, a_n$
- $a \mod n$ is equal to $a - n \lfloor a/n \rfloor$
Spaces

Recall general form

- Sets

\[ \{ S_1(i) : f_1(i); S_2(i) : f_2(i); \ldots \}, \]

- Binary relations

\[ \{ S_1(i) \rightarrow T_1(j) : f_1(i,j); S_2(i) \rightarrow T_2(j) : f_2(i,j); \ldots \} \]

The identifier (e.g., \( S_1 \), \( S_2 \), \( T_1 \), \( T_2 \)), together with the dimension, i.e., number of elements in subsequent tuple (e.g., \( i \), \( j \)), will be called a space

When we say \( S_2(i) = T_1(j) \), we mean

- the identifiers \( S_2 \) and \( T_1 \) are the same
- the dimensions of \( i \) and \( j \) are the same

Examples: \( S() \neq S(i) \), \( S(a) = S(b) \), \( S() \neq T() \)
Space Decomposition

General form can be rewritten

\[
\left\{ \begin{array}{l}
S_1(i) \rightarrow T_1(j) : f_1(i, j); S_2(i) \rightarrow T_2(j) : f_2(i, j); \\
\vdots
\end{array} \right\} = \bigcup_k \left\{ S_k(i) \rightarrow T_k(j) : f_k(i, j) \right\}
\]

- some operations distribute with union
- other operations are defined on a single (pair of) space(s)
  \Rightarrow \text{“space local” operations}
  \Rightarrow \text{replace } \left\{ \begin{array}{l}
S(i) \rightarrow T(j) : f_1(i, j); S(i) \rightarrow T(j) : f_2(i, j)
\end{array} \right\}
  \text{by } \left\{ \begin{array}{l}
S(i) \rightarrow T(j) : f_1(i, j) \lor f_2(i, j)
\end{array} \right\}

In both cases, we define
- unary operator $\oplus$
  \[
  \bigoplus \bigcup R_i := \bigcup_i \bigoplus R_i
  \]
- binary operator $\oplus$
  \[
  \left( \bigcup_i R_i \right) \oplus \left( \bigcup_j S_j \right) := \bigcup_i \bigcup_j \left( R_i \oplus S_j \right)
  \]
Basic Operations

\[ A = \{ S(i_1) \to T(j_1) : f(i_1, j_1) \} \quad B = \{ U(i_2) \to V(j_2) : g(i_2, j_2) \} \]

- **Union**
  \[ A \cup B = \{ S(i) \to T(j) : f(i, j); U(i) \to V(j) : g(i, j) \} \]

- **Intersection**
  \[ A \cap B = \begin{cases} \{ S(i) \to T(j) : f(i, j) \land g(i, j) \} & \text{if } S(i_1) = U(i_2) \text{ and } T(j_1) = V(j_2) \\ \emptyset & \text{otherwise} \end{cases} \]

- **Difference**
  \[ A \setminus B = \begin{cases} \{ S(i) \to T(j) : f(i, j) \land \neg g(i, j) \} & \text{if } S(i_1) = U(i_2) \text{ and } T(j_1) = V(j_2) \\ \{ S(i) \to T(j) : f(i, j) \} & \text{otherwise} \end{cases} \]
Emptiness Check and Comparisons

- Emptiness check
  Does a set or binary relation contain any elements?
  (for any value of the symbolic constants)
Emptiness Check and Comparisons

- Emptiness check
  Does a set or binary relation contain any elements?
  (for any value of the symbolic constants)
  Is
  \[ \{(a, b, c, d) : 3d \geq -21 + 19a - 11b - 6c \land 3d \leq 21 + 17a - b - 6c \land \\
  2b \leq -15 + a \land 3d \leq 2 + a + b \land 3d \geq a + b \} \]
  empty?
Emptiness Check and Comparisons

- Emptiness check
  Does a set or binary relation contain any elements? (for any value of the symbolic constants)
  Is

\[
\{ (a, b, c, d) : 3d \geq -21 + 19a - 11b - 6c \land 3d \leq 21 + 17a - b - 6c \land 2b \leq -15 + a \land 3d \leq 2 + a + b \land 3d \geq a + b \}\]

empty?

⇒ No, contains (13, −1, 38, 4) (and infinitely many other elements)
Emptiness Check and Comparisons

- Emptiness check
  Does a set or binary relation contain any elements? (for any value of the symbolic constants)
  Is

  \[
  \{(a, b, c, d) : 3d \geq -21 + 19a - 11b - 6c \wedge 3d \leq 21 + 17a - b - 6c \wedge 2b \leq -15 + a \wedge 3d \leq 2 + a + b \wedge 3d \geq a + b\}\]

  empty?

  ⇒ No, contains \((13, -1, 38, 4)\) (and infinitely many other elements)

- Comparisons
  - \(A \subseteq B\) is defined as \(A \setminus B = \emptyset\)
  - \(A \supseteq B\) is defined as \(B \subseteq A\)
  - \(A = B\) is defined as \(A \subseteq B \wedge A \supseteq B\)
  - \(A \subset B\) is defined as \(A \subseteq B \wedge \neg(A = B)\)
  - \(A \supset B\) is defined as \(B \subset A\)
Cardinality

- Cardinality of a set
  - ⇒ number of elements in the set
  - ⇒ may depend on symbolic constants

\[ S = \{ S(i) : f(i) \} \]

\[ \text{card} \ S = \{ n : n = \#i : f(i) \} \]

\[ \text{card} \ \{ A(i) : 0 \leq i \leq n ; B() \} = n + 2 \]
Cardinality

- **Cardinality of a set**
  - number of elements in the set
  - may depend on symbolic constants
  
  $$S = \{ S(i) : f(i) \}$$

  $$\text{card } S = \{ n : n = \#i : f(i) \}$$

  $$\text{card } \bigcup_i S_i := \sum_i \text{card } S_i$$

  $$\text{card } \{ A(i) : 0 \leq i \leq n; B() \} = n + 2$$

- **Cardinality of a binary relation**
  - for each domain element, number of corresponding images
  
  $$R = \{ S(i) \rightarrow T(j) : f(i, j) \}$$

  $$\text{card } R = \{ S(i) \rightarrow n : n = \#j : f(i, j) \}$$

  $$\text{card } \bigcup_i R_i := \sum_i \text{card } R_i$$

  $$R = \{ A(i) \rightarrow C(i) : 0 \leq i \leq n; B() \rightarrow C(i) : 0 \leq i \leq n \}$$

  $$\text{card } R = \{ A(i) \rightarrow 1 : 0 \leq i \leq n; B() \rightarrow n + 1 \}$$
Cardinality

- Cardinality of a set
  - number of elements in the set
  - may depend on symbolic constants
  \[ S = \{ S(i) : f(i) \} \]
  \[ \text{card} \ S = \{ n : n = \# i : f(i) \} \]
  \[ \text{card} \left( \bigcup_{i} S_i \right) := \sum_{i} \text{card} \ S_i \]
  \[ \text{card} \left\{ A(i) : 0 \leq i \leq n; B() \right\} = n + 2 \]

- Cardinality of a binary relation
  - for each domain element, number of corresponding images
  \[ R = \{ S(i) \rightarrow T(j) : f(i, j) \} \]
  \[ \text{card} \ R = \{ S(i) \rightarrow n : n = \# j : f(i, j) \} \]
  \[ \text{card} \left( \bigcup_{i} R_i \right) := \sum_{i} \text{card} \ R_i \]
  \[ R = \{ A(i) \rightarrow C(i) : 0 \leq i \leq n; B() \rightarrow C(i) : 0 \leq i \leq n \} \]
  \[ \text{card} \ R = \{ A(i) \rightarrow 1 : 0 \leq i \leq n; B() \rightarrow n + 1 \} \]
  \[ \Rightarrow \text{not a Presburger formula} \]
**isl and Related Libraries and Tools**

isl: manipulates parametric affine sets and relations  
barvinok: counts elements in parametric affine sets and relations  
pet: extracts polyhedral model from clang AST  
PPCG: Polyhedral Parallel Code Generator  
iscc: interactive calculator  
isca: prototype tool set including derivation of process networks and equivalence checker
**isl/iscc syntax**

**Relation description**

| () (tuple) | [] |
| +, − | +, − |
| =, ≤, <, ≥, > | =, <=, <, >=, > |
| true | true |
| false | false |
| ∧ | and |
| ∨ | or |
| ¬ | not |
| ∃v: | exists v: |
| ∀v: | not exists v: not |
| ≼, ≪, ≽, ≻ | (not available yet; write out explicitly) |

**Operations on relations**

| U | + |
| ∩ | * |
| \ | − |
| = | = |
| ⊂, ⊆, ⊃, ⊇ | <, <=, >, >= |
| card | card |

Note: symbolic constants need to be explicitly declared
Parametric Example: Matrix Multiplication

```java
for (int i = 0; i < M; i++)
    for (int j = 0; j < N; j++) {
        S1: C[i][j] = 0;
            for (int k = 0; k < K; k++)
        S2: C[i][j] = C[i][j] + A[i][k] * B[k][j];
    }
```

- Number of statement instances

\[
\text{card}\{ S1(i,j) : 0 \leq i < M \land 0 \leq j < N; \\
S2(i,j,k) : 0 \leq i < M \land 0 \leq j < N \land 0 \leq k < K \}
\]
Parametric Example: Matrix Multiplication

```java
for (int i = 0; i < M; i++)
    for (int j = 0; j < N; j++) {
        S1: C[i][j] = 0;
            for (int k = 0; k < K; k++)
        S2: C[i][j] = C[i][j] + A[i][k] * B[k][j];
    }
```

- Number of statement instances

\[
\text{card}\{ S1(i,j) : 0 \leq i < M \land 0 \leq j < N; \\
S2(i,j,k) : 0 \leq i < M \land 0 \leq j < N \land 0 \leq k < K \}
\]

- Number of array elements accessed by each instance

\[
\text{card}\{ S1(i,j) \rightarrow C(i,j); S2(i,j,k) \rightarrow C(i,j); \\
S2(i,j,k) \rightarrow C(i,j); S2(i,j,k) \rightarrow A(i,k); S2(i,j,k) \rightarrow B(k,j) \}
\]
Exercise

```c
int f1(int m, int n, int A[const static m][n])
{
    int t = 0;
    for (int i = 0; i < m; ++i)
        t += A[i][i];
    return t;
}
void f2(int m, int n, int A[/*.*/*][n][n], int B[/*.*/*][n])
{
    for (int i = 0; i < m; ++i) {
        S:
            B[i][0] = 0;
        for (int j = 0; j < n; ++j) {
            if (j == i)
                continue;
            T:
                B[i][j] = f1(j, n, A[i]);
        }
    }
}
```

How many statement instances are executed by `f2`?

```c
m > n ? m * n - n + m : m * n
```

How many array elements accessed by each instance?

```c
S[i] -> 1; T[i,j] -> 1 + j
```
**Exercise**

```c
int f1(int m, int n, int A[const static m][n])
{
    int t = 0;
    for (int i = 0; i < m; ++i)
        t += A[i][i];
    return t;
}

void f2(int m, int n, int A[/*.*/*/m][n][n], int B[/*.*/*/m][n])
{
    for (int i = 0; i < m; ++i) {
        for (int j = 0; j < n; ++j) {
            if (j == i) continue;
            B[i][j] = f1(j, n, A[i]);
        }
    }
}
```

How many statement instances are executed by `f2`?
How many array elements accessed by each instance?
Exercise

```c
int f1(int m, int n, int A[const static m][n]) {
    int t = 0;
    for (int i = 0; i < m; ++i)
        t += A[i][i];
    return t;
}

void f2(int m, int n, int A[/*.**/m][n][n], int B[/*.**/m][n]) {
    for (int i = 0; i < m; ++i) {
        S:
            B[i][0] = 0;
        for (int j = 0; j < n; ++j) {
            if (j == i)
                continue;
            T:
                B[i][j] = f1(j, n, A[i]);
        }
    }
}
```

How many statement instances are executed by `f2`? `m > n ? m*n-n+m : m*n`

How many array elements accessed by each instance? `S[i]->1; T[i,j]->1+j`
Exercise

```c
int f1(int m, int n, int A[const static m][n])
{
    int t = 0;
    for (int i = 0; i < m; ++i)
        t += A[i][i];
    return t;
}

void f2(int m, int n, int A[/*.*/m][n][n], int B[/*.*/m][n])
{
    for (int i = 0; i < m; ++i) {
        S:
            B[i][0] = 0;
        for (int j = 0; j < n; ++j) {
            if (j == i)
                continue;
            T:
                B[i][j] = f1(j, n, A[i]);
        }
    }
}
```

How many statement instances are executed by f2? m>n ? m*n-n+m : m*n
How many array elements accessed by each instance? S[i]->1; T[i,j]->1+j
Outline

1. Polyhedral Model
   - Introduction
   - Representation

2. Polyhedral Transformation
   - Schedules
   - AST Generation

3. Dependences
   - Schedule Validity
   - Dependences
   - Structures

4. Dataflow
   - Parallelism
   - Dataflow
   - Approximate Dataflow
   - Run-time dependent Dataflow
   - Reductions

5. Aliasing

6. Counting
   - Cardinality
   - Bounds
   - Weighted Counting
   - Dynamic Memory Requirement

7. Transitive Closures
Polyhedral Transformation

Two approaches

1. encode execution order in statement instance indices
   ⇒ transformation performed by manipulating instance set

2. keep track of execution order separately: schedule
   ⇒ transformation performed by manipulating initial/current schedule, or
   ⇒ transformation performed by constructing new schedule from scratch
Polyhedral Transformation

Two approaches

1. encode execution order in statement instance indices
   ⇒ transformation performed by manipulating instance set

2. keep track of execution order separately: schedule
   ⇒ transformation performed by manipulating initial/current schedule, or
   ⇒ transformation performed by constructing new schedule from scratch

Schedule $O$ keeps track of relative execution order of statement instances

⇒ for each pair of statement instances $S(i)$ and $T(j)$, schedule determines
   ▶ $S(i)$ executed before $T(j)$ $O(S(i), T(j)) < 0$
   ▶ $S(i)$ executed after $T(j)$ $O(S(i), T(j)) > 0$
   ▶ $S(i)$ and $T(j)$ may be executed simultaneously $O(S(i), T(j)) = 0$
Schedule Representations

Types of schedule representations

- Combined representations
  - schedule tree
  - named Presburger relation

- Scattered representations
  - Kelly’s abstraction
  - “2d + 1”
Schedule Representations

Types of schedule representations

- Combined representations
  - schedule tree
  - named Presburger relation

- Scattered representations
  - Kelly’s abstraction
  - “2d + 1”

Schedule Trees

- Main node types
  - sequence: children are executed in order
  - band: instance are executed according to associated piecewise quasi-affine partial schedule $P$

- Deriving schedule tree from AST
  - for loop $\Rightarrow$ single-dimensional band
  - compound statement $\Rightarrow$ sequence
Parametric Example: Matrix Multiplication

```c
for (int i = 0; i < M; i++)
    for (int j = 0; j < N; j++) {
        S1:  C[i][j] = 0;
            for (int k = 0; k < K; k++)
        S2:  C[i][j] = C[i][j] + A[i][k] * B[k][j];
    }
```

\[ S1(i,j) \rightarrow (i); S2(i,j,k) \rightarrow (i) \]

\[ S1(i,j) \rightarrow (j); S2(i,j,k) \rightarrow (j) \]

sequence

\[ S1(i,j) \]

\[ S2(i,j,k) \rightarrow (k) \]
Parametric Example: Matrix Multiplication

\[
\begin{array}{l}
\text{for (int } i = 0; i < M; i++) \\
\quad \text{for (int } j = 0; j < N; j++) \\
\quad \text{S1: } C[i][j] = 0; \\
\quad \text{for (int } k = 0; k < K; k++) \\
\quad \text{S2: } C[i][j] = C[i][j] + A[i][k] \times B[k][j];
\end{array}
\]

Sequence:

\[
\begin{array}{l}
S1(i,j) \rightarrow (i); S2(i,j,k) \rightarrow (i) \\
\quad \text{sequence} \\
S1(i,j) \rightarrow (j); S2(i,j,k) \rightarrow (j) \\
S2(i,j,k) \rightarrow (k)
\end{array}
\]
Parametric Example: Matrix Multiplication

```c
for (int i = 0; i < M; i++)
    for (int j = 0; j < N; j++) {
        C[i][j] = 0;
        for (int k = 0; k < K; k++)
            C[i][j] = C[i][j] + A[i][k] * B[k][j];
    }
```

$S_1(i, j) \rightarrow (i)$; $S_2(i, j, k) \rightarrow (i)$

$S_1(i, j) \rightarrow (j)$; $S_2(i, j, k) \rightarrow (j)$

sequence

$S_1(i, j) \rightarrow (j)$;

$S_2(i, j, k) \rightarrow (k)$
Parametric Example: Matrix Multiplication

```c
for (int i = 0; i < M; i++)
    for (int j = 0; j < N; j++) {
        S1:  C[i][j] = 0;
        for (int k = 0; k < K; k++)
            S2:  C[i][j] = C[i][j] + A[i][k] * B[k][j];
    }
```

S1(i, j) → (i); S2(i, j, k) → (i)

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>
S1(i, j) → (j); S2(i, j, k) → (j)

sequence

S1(i, j)               S2(i, j, k)

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
</tbody>
</table>
S2(i, j, k) → (k)
Parametric Example: Matrix Multiplication

for (int i = 0; i < M; i++)
    for (int j = 0; j < N; j++) {
        S1: \( C[i][j] = 0; \)
        for (int k = 0; k < K; k++)
        S2: \( C[i][j] = C[i][j] + A[i][k] \times B[k][j]; \)
    }

\[
\text{S1}(i,j) \rightarrow (i); \text{S2}(i,j,k) \rightarrow (i) \\
\text{S1}(i,j) \rightarrow (j); \text{S2}(i,j,k) \rightarrow (j) \\
\text{sequence}
\]

\[
\text{S1}(i,j) \\
\text{S2}(i,j,k) \rightarrow (k)
\]
Schedule Trees — Execution Order

What is the execution order of statement instances $S(i)$ and $T(j)$ determined by schedule tree $O$?

Start at root of schedule tree

while current node is not a leaf do

  if current node is sequence then

    if $S(i)$ and $T(j)$ appear in same child then
      Move to common child

    else if $S(i)$ appears in earlier child then
      return $O(S(i), T(j)) < 0$

    else
      return $O(S(i), T(j)) > 0$

  else

    if $P(S(i)) = P(T(j))$ then
      Move to single child

    else if $P(S(i)) < P(T(j))$ then
      return $O(S(i), T(j)) < 0$

    else
      return $O(S(i), T(j)) > 0$

return $O(S(i), T(j)) = 0$
Named Presburger Relation Schedules

Schedule tree with single (band) node
Named Presburger Relation Schedules

Schedule tree with single (band) node

Flattening a schedule tree

- two nested band nodes
  ⇒ replace by single band node with concatenated partial schedule

- sequence with as children either leaves or trees consisting of a single band node
  ⇒ treat leaves as zero-dimensional band nodes
  ⇒ pad lower-dimensional bands (e.g., with zero)
  ⇒ construct one-dimensional band assigning increasing value to children
  ⇒ combine one-dimensional band with children
Parametric Example: Matrix Multiplication

for (int i = 0; i < M; i++)
    for (int j = 0; j < N; j++) {
        S1: C[i][j] = 0;
            for (int k = 0; k < K; k++)
        S2: C[i][j] = C[i][j] + A[i][k] * B[k][j];
    }

S1(i, j) → (i); S2(i, j, k) → (i)
    |                  |
S1(i, j) → (j); S2(i, j, k) → (j)
    |                  |
sequence

S1(i, j)                  S2(i, j, k)
    |                  |
S2(i, j, k) → (k)
Parametric Example: Matrix Multiplication

```c
for (int i = 0; i < M; i++)
    for (int j = 0; j < N; j++) {
        S1: C[i][j] = 0;
            for (int k = 0; k < K; k++)
        S2: C[i][j] = C[i][j] + A[i][k] * B[k][j];
    }
```

```
S1(i, j) → (i); S2(i, j, k) → (i)
  |
S1(i, j) → (j); S2(i, j, k) → (j)
  |
sequence
```

```
S1(i, j) → (0)
  |
S2(i, j, k) → (k)
```
Parametric Example: Matrix Multiplication

```java
for (int i = 0; i < M; i++)
    for (int j = 0; j < N; j++) {
        S1: C[i][j] = 0;
            for (int k = 0; k < K; k++)
        S2: C[i][j] = C[i][j] + A[i][k] * B[k][j];
    }

S1(i, j) → (i); S2(i, j, k) → (i)
    | 
S1(i, j) → (j); S2(i, j, k) → (j)
    | 
S1(i, j) → (0, 0); S2(i, j, k) → (1, k)
```
Parametric Example: Matrix Multiplication

```c
for (int i = 0; i < M; i++)
    for (int j = 0; j < N; j++) {
        S1: C[i][j] = 0;
            for (int k = 0; k < K; k++)
        S2: C[i][j] = C[i][j] + A[i][k] * B[k][j];
    }
```

\[ S1(i, j) \rightarrow (i); S2(i, j, k) \rightarrow (i) \]

\[ S1(i, j) \rightarrow (j, 0, 0); S2(i, j, k) \rightarrow (j, 1, k) \]
Parametric Example: Matrix Multiplication

```java
for (int i = 0; i < M; i++)
    for (int j = 0; j < N; j++) {
        S1: C[i][j] = 0;
            for (int k = 0; k < K; k++)
        S2: C[i][j] = C[i][j] + A[i][k] * B[k][j];
    }

S1(i,j) \rightarrow (i,j,0,0); S2(i,j,k) \rightarrow (i,j,1,k)
```
Domain and Range of a Relation

\[ R = \{ S(i) \rightarrow T(j) : f(i, j) \} \]

- **Domain**
  
  \[ \text{dom } R = \{ S(i) : \exists j : f(i, j) \} \]

  \( W = \{ S(i, j) \rightarrow A(i + j) : 0 \leq i < 2 \land 0 \leq j < 2 \} \)

  \[ \text{dom } W = \{ S(i, j) : 0 \leq i < 2 \land 0 \leq j < 2 \} \]

  \Rightarrow \text{statement instances writing any array element}
Domain and Range of a Relation

\[ R = \{ S(i) \rightarrow T(j) : f(i, j) \} \]

- **Domain**
  
  \[
  \text{dom } R = \{ S(i) : \exists j : f(i, j) \} \quad \text{(iscc: dom)}
  \]

  \[ W = \{ S(i, j) \rightarrow A(i + j) : 0 \leq i < 2 \land 0 \leq j < 2 \} \]
  
  \[
  \text{dom } W = \{ S(i, j) : 0 \leq i < 2 \land 0 \leq j < 2 \}
  \]

  \[ \Rightarrow \text{statement instances writing any array element} \]

- **Range**
  
  \[
  \text{ran } R = \{ T(j) : \exists i : f(i, j) \} \quad \text{(iscc: ran)}
  \]

  \[ W = \{ S(i, j) \rightarrow A(i + j) : 0 \leq i < 2 \land 0 \leq j < 2 \} \]
  
  \[
  \text{ran } W = \{ A(a) : 0 \leq a \leq 2 \}
  \]

  \[ \Rightarrow \text{written array elements} \]
Domain/Range Restriction

\[ A = \{ S_1(i_1) : f(i_1) \} \]
\[ B = \{ T_1(j_1) : g(j_1) \} \]
\[ C = \{ S_2(i_2) \rightarrow T_2(j_2) : h(i_2, j_2) \} \]

- **Product relation**
  \[ A \rightarrow B = \{ S_1(i) \rightarrow T_1(j) : f(i) \land g(j) \} \]

(iscc: \(\rightarrow\))
Domain/Range Restriction

\[
A = \{ S_1(i_1) : f(i_1) \} \\
B = \{ T_1(j_1) : g(j_1) \} \\
C = \{ S_2(i_2) \rightarrow T_2(j_2) : h(i_2, j_2) \}
\]

- Product relation
  \[
  A \rightarrow B = \{ S_1(i) \rightarrow T_1(j) : f(i) \land g(j) \}
  \]

- Domain restriction
  \[
  R \cap_{\text{dom}} S = R \cap (S \rightarrow (\text{ran } R))
  \]
  \[
  C \cap_{\text{dom}} A = \begin{cases} 
  \{ S_2(i) \rightarrow T_2(j) : f(i) \land h(i, j) \} & \text{if } S_1(i_1) = S_2(i_2) \\
  \emptyset & \text{otherwise}
  \end{cases}
  \]
Domain/Range Restriction

\[ A = \{ S_1(i_1) : f(i_1) \} \quad B = \{ T_1(j_1) : g(j_1) \} \]
\[ C = \{ S_2(i_2) \rightarrow T_2(j_2) : h(i_2, j_2) \} \]

- **Product relation**
  \[ A \rightarrow B = \{ S_1(i) \rightarrow T_1(j) : f(i) \land g(j) \} \]

- **Domain restriction**
  \[
  R \cap_{dom} S = R \cap (S \rightarrow (\text{ran } R))
  \]
  \[
  C \cap_{dom} A = \begin{cases} 
  \{ S_2(i) \rightarrow T_2(j) : f(i) \land h(i, j) \} & \text{if } S_1(i_1) = S_2(i_2) \\
  \emptyset & \text{otherwise}
  \end{cases}
  \]

- **Range restriction**
  \[
  R \cap_{ran} S = R \cap ((\text{dom } R) \rightarrow S)
  \]
  \[
  C \cap_{ran} A = \begin{cases} 
  \{ S_2(i) \rightarrow T_2(j) : f(j) \land h(i, j) \} & \text{if } S_1(i_1) = T_2(j_2) \\
  \emptyset & \text{otherwise}
  \end{cases}
  \]
AST Generation

Input:
- instance set
- schedule

Output:
- AST that visits each domain element according to the order specified by the schedule

Note: in case of flat schedule, schedule order is lexicographic order of output space

⇒ single output space

iscc codegen operation takes as input flat schedule with instance set encoded in domain

⇒ apply * to “pure” schedule and instance set first
Parametric Example: Matrix Multiplication

```java
for (int i = 0; i < M; i++)
    for (int j = 0; j < N; j++) {
        S1:   C[i][j] = 0;
            for (int k = 0; k < K; k++)
        S2:   C[i][j] = C[i][j] + A[i][k] * B[k][j];
    }
```
Parametric Example: Matrix Multiplication

```c
for (int i = 0; i < M; i++)
    for (int j = 0; j < N; j++) {
        C[i][j] = 0;
        for (int k = 0; k < K; k++)
            C[i][j] = C[i][j] + A[i][k] * B[k][j];
    }
```

\[
I := [M,N,K] \rightarrow \{ S1[i,j] : 0 \leq i < M \text{ and } 0 \leq j < N; \\
    S2[i,j,k] : 0 \leq i < M \text{ and } 0 \leq j < N \text{ and } 0 \leq k < K \};
\]

\[
O := \{ S1[i,j] \rightarrow [i,j,0,0]; S2[i,j,k] \rightarrow [i,j,1,k] \};
\]

codegen (O * I);
```
Parametric Example: Matrix Multiplication

for (int i = 0; i < M; i++)
    for (int j = 0; j < N; j++) {
        S1: C[i][j] = 0;
        for (int k = 0; k < K; k++)
            S2: C[i][j] = C[i][j] + A[i][k] * B[k][j];
    }

I := \{M,N,K\} \rightarrow \{ S1[i,j] : 0 \leq i < M \text{ and } 0 \leq j < N;
    S2[i,j,k] : 0 \leq i < M \text{ and } 0 \leq j < N \text{ and } 0 \leq k < K \};

O := \{ S1[i,j] \rightarrow [i,j,0,0]; S2[i,j,k] \rightarrow [i,j,1,k] \};

codegen (O * I);

for (int c0 = 0; c0 < M; c0 += 1)
    for (int c1 = 0; c1 < N; c1 += 1) {
        S1(c0, c1);
        for (int c3 = 0; c3 < K; c3 += 1)
            S2(c0, c1, c3);
    }
Exercise

```c
int f1(int m, int n, int A[const static m][n])
{
    int t = 0;
    for (int i = 0; i < m; ++i)
        t += A[i][i];
    return t;
}

void f2(int m, int n, int A[/*.**/m][n][n], int B[/*.**/m][n])
{
    for (int i = 0; i < m; ++i) {
        B[i][0] = 0;
        for (int j = 0; j < n; ++j) {
            if (j == i)
                continue;
            B[i][j] = f1(j, n, A[i]);
        }
    }
}
```

Write down schedule and generate AST codegen (O * I)
Outline

1. Polyhedral Model
   - Introduction
   - Representation

2. Polyhedral Transformation
   - Schedules
   - AST Generation

3. Dependences
   - Schedule Validity
   - Dependences
   - Structures

4. Dataflow
   - Parallelism
   - Dataflow
   - Approximate Dataflow
   - Run-time dependent Dataflow
   - Reductions

5. Aliasing

6. Counting
   - Cardinality
   - Bounds
   - Weighted Counting
   - Dynamic Memory Requirement

7. Transitive Closures
Schedule Validity

Not all schedules correspond to a valid execution order
Schedule Validity

Not all schedules correspond to a valid execution order

\[ \text{R(a) \hspace{1cm} W(a) \rightarrow R(a) \hspace{1cm} W(b) \hspace{1cm} W(a) \hspace{1cm} W(a)} \]

Internal restrictions

- A read of a value should not be scheduled after the write of the value
- No other write to same memory location may be scheduled in between

External restrictions (on non-temporaries)

- No write may be scheduled before initial read from a memory location
- No write may be scheduled after last write to a memory location
Schedule Validity

Not all schedules correspond to a valid execution order

\[ R(a) \rightarrow W(a) \rightarrow R(a) \rightarrow W(b) \rightarrow W(a) \rightarrow W(a) \]

Internal restrictions
- A read of a value should not be scheduled after the write of the value
- No other write to same memory location may be scheduled in between

External restrictions (on non-temporaries)
- No write may be scheduled before initial read from a memory location
- No write may be scheduled after last write to a memory location
**Schedule Validity**

Not all schedules correspond to a valid execution order

\[ R(a) \rightarrow W(a) \rightarrow R(a) \rightarrow W(b) \rightarrow W(a) \rightarrow W(a) \]

**Internal restrictions**
- A read of a value should not be scheduled after the write of the value
- No other write to same memory location may be scheduled in between

**External restrictions (on non-temporaries)**
- No write may be scheduled before initial read from a memory location
- No write may be scheduled after last write to a memory location
Schedule Validity

Not all schedules correspond to a valid execution order

```
R(a)  W(a)  →  R(a)  W(b)  W(a)  W(a)
```

Internal restrictions
- A read of a value should not be scheduled after the write of the value
- No other write to same memory location may be scheduled in between

External restrictions (on non-temporaries)
- No write may be scheduled before initial read from a memory location
- No write may be scheduled after last write to a memory location
Schedule Validity

Not all schedules correspond to a valid execution order

R(a) → W(a) → R(a) → W(b) → W(a) → W(a)

Internal restrictions
- A read of a value should not be scheduled after the write of the value
- No other write to same memory location may be scheduled in between

External restrictions (on non-temporaries)
- No write may be scheduled before initial read from a memory location
- No write may be scheduled after last write to a memory location
Schedule Validity

Not all schedules correspond to a valid execution order

Internal restrictions
- A read of a value should not be scheduled after the write of the value
- No other write to same memory location may be scheduled in between

External restrictions (on non-temporaries)
- No write may be scheduled before initial read from a memory location
- No write may be scheduled after last write to a memory location
Schedule Validity

Not all schedules correspond to a valid execution order

Internal restrictions
- A read of a value should not be scheduled after the write of the value
- No other write to same memory location may be scheduled in between

External restrictions (on non-temporaries)
- No write may be scheduled before initial read from a memory location
- No write may be scheduled after last write to a memory location

Sufficient conditions:
- Every read of a memory location is scheduled after every previous write to the same memory location
- Every write to a memory location is scheduled after every previous read or write to the same memory location
Dependences

Sufficient conditions for schedule validity:

- Every read of a memory location is scheduled after every previous write to the same memory location.
- Every write to a memory location is scheduled after every previous read or write to the same memory location.

Dependence relation $D$: pairs of statement instances

- accessing the same memory location
- of which at least one is a write
- with the first executed before the second

Sufficient condition:

$$\forall S(i) \rightarrow T(j) \in D : O(S(i), T(j)) < 0$$
Inverse Relation and Composition

\[ A = \{ S_1(i_1) \rightarrow T_1(j_1) : f(i_1, j_1) \} \quad B = \{ S_2(i_2) \rightarrow T_2(j_2) : g(i_2, j_2) \} \]

- Inverse
  \[ A^{-1} = \{ T_1(j) \rightarrow S_1(i) : f(i, j) \} \]

\[ W = \{ S(i, j) \rightarrow A(i + j) : 0 \leq i < 2 \land 0 \leq j < 2 \} \]
\[ W^{-1} = \{ A(a) \rightarrow S(i, j) : a = i + j \land 0 \leq i < 2 \land 0 \leq j < 2 \} \]

⇒ statement instances writing array element
Inverse Relation and Composition

\[ A = \{ S_1(i_1) \rightarrow T_1(j_1) : f(i_1, j_1) \} \quad B = \{ S_2(i_2) \rightarrow T_2(j_2) : g(i_2, j_2) \} \]

• Inverse
  \[ A^{-1} = \{ T_1(j) \rightarrow S_1(i) : f(i, j) \} \quad (\text{iscc: } ^{-1}) \]
  \[ W = \{ S(i, j) \rightarrow A(i + j) : 0 \leq i < 2 \land 0 \leq j < 2 \} \]
  \[ W^{-1} = \{ A(a) \rightarrow S(i, j) : a = i + j \land 0 \leq i < 2 \land 0 \leq j < 2 \} \]
  \[ \Rightarrow \text{statement instances writing array element} \]

• Composition
  \[ B \circ A = \begin{cases} \{ S_1(i) \rightarrow T_2(j) : \exists k : f(i, k) \land g(k, j) \} & \text{if } T_1(j_1) = S_2(i_2) \\ \emptyset & \text{otherwise} \end{cases} \quad (\text{iscc: after}) \]
  \[ W^{-1} \circ W = \{ S(i, j) \rightarrow S(i', j') : 0 \leq i, i', j, j' < 2 \land i + j = i' + j' \} \]
  \[ \Rightarrow \text{pairs of statement instances that write same array element} \]
Lexicographic Order

- **Sets**
  
  \[
  A = \{ S(i) : f(i) \} 
  \]
  
  \[
  B = \{ T(j) : g(j) \} 
  \]
  
  \[
  A \prec B = \begin{cases} 
  \{ S(i) \rightarrow S(j) : f(i) \land g(j) \land i \prec j \} & \text{if } S(i) = T(j) \\ 
  \emptyset & \text{otherwise} 
  \end{cases}
  \] (iscc: <<)
Lexicographic Order

- **Sets**
  
  \[
  A = \{ \text{S}(i) : f(i) \} \\
  B = \{ \text{T}(j) : g(j) \}
  \]

  \[
  A \prec B = \begin{cases}
  \{ \text{S}(i) \rightarrow \text{S}(j) : f(i) \land g(j) \land i \prec j \} & \text{if } S(i) = T(j) \\
  \emptyset & \text{otherwise}
  \end{cases}
  \]

- **Relations**
  
  \[
  \Rightarrow \quad \text{binary relation on domains reflecting lexicographic order of images}
  \]

  \[
  A = \{ \text{S}_1(i_1) \rightarrow \text{T}_1(j_1) : f(i_1, j_1) \} \\
  B = \{ \text{S}_2(i_2) \rightarrow \text{T}_2(j_2) : g(i_2, j_2) \}
  \]

  \[
  A \prec B = \begin{cases}
  \{ \text{S}_1(i_1) \rightarrow \text{S}_2(i_2) : \exists j_1, j_2 : f(i_1, j_1) \land g(i_2, j_2) \land j_1 \prec j_2 \} & \text{if } T_1(j_1) = T_2(j_2) \\
  \emptyset & \text{otherwise}
  \end{cases}
  \]
Parametric Example: Matrix Multiplication

for (int i = 0; i < M; i++)
    for (int j = 0; j < N; j++) {
        S1:  C[i][j] = 0;
            for (int k = 0; k < K; k++)
        S2:  C[i][j] = C[i][j] + A[i][k] * B[k][j];
    }
Parametric Example: Matrix Multiplication

for (int i = 0; i < M; i++)
    for (int j = 0; j < N; j++) {
        S1:  C[i][j] = 0;
            for (int k = 0; k < K; k++)
        S2:  C[i][j] = C[i][j] + A[i][k] * B[k][j];
    }

0 := { S1[i,j] -> [i,j,0,0]; S2[i,j,k] -> [i,j,1,k] };
0 << 0;
Parametric Example: Matrix Multiplication

```c
for (int i = 0; i < M; i++)
    for (int j = 0; j < N; j++) {
        S1:   C[i][j] = 0;
            for (int k = 0; k < K; k++)
        S2:   C[i][j] = C[i][j] + A[i][k] * B[k][j];
    }
0 := { S1[i,j] -> [i,j,0,0]; S2[i,j,k] -> [i,j,1,k] };
0 <= 0;

{  S2[i, j, k] -> S2[i’, j’, k’] : i’ >= 1 + i;
   S2[i, j, k] -> S2[i, j’, k’] : j’ >= 1 + j;
   S2[i, j, k] -> S2[i, j, k’] : k’ >= 1 + k;
   S1[i, j] -> S2[i’, j’, k] : i’ >= 1 + i;
   S1[i, j] -> S2[i, j’, k] : j’ >= 1 + j;
   S1[i, j] -> S2[i, j, k]; S2[i, j, k] -> S1[i’, j’] : i’ >= 1 + i;
   S2[i, j, k] -> S1[i, j’] : j’ >= 1 + j;
   S1[i, j] -> S1[i’, j’] : i’ >= 1 + i;
   S1[i, j] -> S1[i, j’] : j’ >= 1 + j }
```
Dependence Analysis

Recall: sufficient condition for schedule validity

\[ \forall S(i) \rightarrow T(j) \in D : O(S(i), T(j)) < 0 \]

Dependence relation \( D \): pairs of statement instances

- accessing the same memory location
- of which at least one is a write
- with the first executed before the second
Dependence Analysis

Recall: sufficient condition for schedule validity

\[ \forall S(i) \rightarrow T(j) \in D : O(S(i), T(j)) < 0 \]

Dependence relation \( D \): pairs of statement instances
- accessing the same memory location
- of which at least one is a write
- with the first executed before the second

Computation:

\[ D = \left( (W^{-1} \circ R) \cup (W^{-1} \circ W) \cup (R^{-1} \circ W) \right) \cap (O \prec O) \]

\( W \): write access relation
\( R \): read access relation
\( O \): original schedule
### Parametric Example: Matrix Multiplication

```java
for (int i = 0; i < M; i++)
    for (int j = 0; j < N; j++) {
        S1:  C[i][j] = 0;
            for (int k = 0; k < K; k++)
        S2:  C[i][j] = C[i][j] + A[i][k] * B[k][j];
    }
```

Parametric Example: Matrix Multiplication

```c
for (int i = 0; i < M; i++)
    for (int j = 0; j < N; j++) {
        S1: C[i][j] = 0;
            for (int k = 0; k < K; k++)
        S2: C[i][j] = C[i][j] + A[i][k] * B[k][j];
    }

W := { S1[i,j] -> C[i,j]; S2[i,j,k] -> C[i,j] };
R := { S2[i,j,k] -> C[i,j]; S2[i,j,k] -> A[i,k];
      S2[i,j,k] -> B[k,j] };
I := [M,N,K] -> { S1[i,j] : 0 <= i < M and 0 <= j < N;
                 S2[i,j,k] : 0 <= i < M and 0 <= j < N and 0 <= k < K };
O := { S1[i,j] -> [i,j,0,0]; S2[i,j,k] -> [i,j,1,k] };
((R . W^-1) + (W . W^-1) + (W . R^-1)) * (O << 0);
```
Parametric Example: Matrix Multiplication

```c
for (int i = 0; i < M; i++)
  for (int j = 0; j < N; j++) {
    S1:  C[i][j] = 0;
        for (int k = 0; k < K; k++)
    S2:  C[i][j] = C[i][j] + A[i][k] * B[k][j];
  }

W := { S1[i,j] -> C[i,j]; S2[i,j,k] -> C[i,j] };
R := { S2[i,j,k] -> C[i,j]; S2[i,j,k] -> A[i,k];
        S2[i,j,k] -> B[k,j] };
I := [M,N,K] -> { S1[i,j] : 0 <= i < M and 0 <= j < N;
                S2[i,j,k] : 0 <= i < M and 0 <= j < N and 0 <= k < K };
O := { S1[i,j] -> [i,j,0,0]; S2[i,j,k] -> [i,j,1,k] };
((R . W^-1) + (W . W^-1) + (W . R^-1)) * (O << O);

{ S2[i, j, k] -> S2[i, j, k'] : k' >= 1 + k;
  S1[i, j] -> S2[i, j, k] }
```
Exercise

```c
int f1(int m, int n, int A[const static m][n])
{
    int t = 0;
    for (int i = 0; i < m; ++i)
        t += A[i][i];
    return t;
}
void f2(int m, int n, int A[/*.*/*/m][n][n], int B[/*.*/*/m][n])
{
    for (int i = 0; i < m; ++i) {
        S:
            B[i][0] = 0;
            for (int j = 0; j < n; ++j) {
                if (j == i)
                    continue;
                T:
                    B[i][j] = f1(j, n, A[i]);
            }
    }
}
```

Compute dependence relation
Exercise

int f1(int m, int n, int A[const static m][n])
{
    int t = 0;
    for (int i = 0; i < m; ++i)
        t += A[i][i];
    return t;
}

void f2(int m, int n, int A[/* */m][n][n], int B[/* */m][n])
{
    for (int i = 0; i < m; ++i) {
        B[i][0] = 0;
        for (int j = 0; j < n; ++j) {
            if (j == i)
                continue;
            B[i][j] = f1(j, n, A[i]);
        }
    }
}

Compute dependence relation
Accesses to Structure Fields and Nested Relations

No special treatment is needed for representing accesses to structure fields

⇒ structure field encoded in name

⇒ of target space of access relations
Accesses to Structure Fields and Nested Relations

No special treatment is needed for representing accesses to structure fields
⇒ structure field encoded in name of target space of access relations

```c
struct s {
    int a;
    int b[10];
};

void f(struct s s[const static 10][10])
{
    for (int i = 0; i < 10; ++i)
    S:
        s[i][i].b[9 - i] = 0;
}

{ S(i) → s_b(i, i, 9 - i) }
```
Accesses to Structure Fields and Nested Relations

No special treatment is needed for representing accesses to structure fields

⇒ structure field encoded in name and/or structure of target space of access relations

```c
struct s {
    int a;
    int b[10];
};

void f(struct s s[const static 10][10])
{
    for (int i = 0; i < 10; ++i)
        s[i][i].b[9 - i] = 0;
}
```

In pet, structure encoded in nested relation

```
{ S(i) → s_b(i, i, 9 − i) }
```

In pet, structure encoded in nested relation

```
{ S(i) → s_b(s(i, i) → b(9 − i)) }
```
Accesses to Structures

Dependence analysis needs to take into account that access to structure represents access to all fields of structure

```c
struct c {
    float re;
    float im;
};

void f(struct c A[const static 10])
{
}

Write access relation: \{ S() \rightarrow A(0); T() \rightarrow A.re(A(1) \rightarrow re()) \}
Accesses to Structures

Dependence analysis needs to take into account that access to structure represents access to all fields of structure

```c
struct c {
    float re;
    float im;
};
```

```c
void f(struct c A[const static 10])
{
}
```

Write access relation: \( \{ S() \rightarrow A(0); T() \rightarrow A_{\text{re}}(A(1) \rightarrow \text{re}()) \} \)

Expansion: \( \{ A(a) \rightarrow A_{\text{re}}(A(a) \rightarrow \text{re}()); A(a) \rightarrow A_{\text{im}}(A(a) \rightarrow \text{im}()); \} \)

Expanded write access relation:
\( \{ S() \rightarrow A_{\text{re}}(A(0) \rightarrow \text{re}()); S() \rightarrow A_{\text{im}}(A(0) \rightarrow \text{im}());
    T() \rightarrow A_{\text{re}}(A(1) \rightarrow \text{re}()) \} \)
Schedule Optimization Criteria

Typical optimization criteria

- increase parallelism
- increase locality
- reduce memory requirements
Schedule Optimization Criteria

Typical optimization criteria

- increase parallelism
- increase locality
- reduce memory requirements

Parallelism:
Pairs of statement instances $S(i)$ and $T(j)$ may be executed in parallel if they do not depend on each other:

$$\{ S(i) \rightarrow T(j); T(j) \rightarrow S(i) \} \cap D = \emptyset$$
Local Parallelism

Global parallelism

\[
\{ S(i) \rightarrow T(j); T(j) \rightarrow S(i) \} \cap D = \emptyset
\]

Local parallelism

\[
\{ S(i) \rightarrow T(j); T(j) \rightarrow S(i) \} \cap L = \emptyset
\]

- Root of schedule tree: \( L = D \)
- Child of band node \( b \) with partial schedule \( P \):
  \[
  L = L_b \cap \{ S(i) \rightarrow T(j) : P(S(i)) = P(T(j)) \}
  \]
- Child of sequence node \( s \) with instance set \( F \)
  \[
  L = L_s \cap \{ S(i) \rightarrow T(j) : S(i) \in F \land T(j) \in F \}
  \]
Encoding Parallelism

- Coarse grain parallelism (root)
  - placement maps statement instances to virtual processors
  - schedule interpreted within each virtual processor
Encoding Parallelism

- Coarse grain parallelism (root)
  - *placement* maps statement instances to virtual processors
  - schedule interpreted within each virtual processor
- Fine grain parallelism (leaf)
  - statement instances $S(i)$ and $T(j)$ for which $O(S(i), T(j)) = 0$ may be executed in parallel within leaf that contains both
Encoding Parallelism

- Coarse grain parallelism (root)
  - *placement* maps statement instances to virtual processors
  - schedule interpreted within each virtual processor
- Fine grain parallelism (leaf)
  - statement instances $S(i)$ and $T(j)$ for which $O(S(i), T(j)) = 0$ may be executed in parallel within leaf that contains both
- General case (arbitrary position in schedule tree)
  - schedules defines total order, but,
  - some sequence nodes and/or some band dimensions are explicitly marked parallel
False Dependences

for (int i = 0; i < n; ++i) {
S: t = f1(A[i]);
T: B[i] = f2(t);
}

Dependences

- read after write ("true"): \{ S(i) \rightarrow T(i') : i' \geq i \}
- write after read ("anti"): \{ T(i) \rightarrow S(i') : i' > i \}
- write after write ("output"): \{ S(i) \rightarrow S(i') : i' > i \}

"false" False dependences not from dataflow, but from reuse of memory location t

Possible solution: expansion/privatization

for (int i = 0; i < n; ++i) {
S: t[i] = f1(A[i]);
T: B[i] = f2(t[i]);
}

dataflow (subset of "true" dependences):

\{ S(i) \rightarrow T(i) : i' \geq i \}
False Dependences

for (int i = 0; i < n; ++i) {
    S: t = f1(A[i]);
    T: B[i] = f2(t);
}

Dependences

- read after write ("true"): \{ S(i) \rightarrow T(i') : i' \geq i \}
- write after read ("anti"): \{ T(i) \rightarrow S(i') : i' > i \}
- write after write ("output"): \{ S(i) \rightarrow S(i') : i' > i \}

"false"
False Dependences

```java
for (int i = 0; i < n; ++i) {
    S: t = f1(A[i]);
    T: B[i] = f2(t);
}
```

Dependences

- read after write ("true"): \( \{ S(i) \rightarrow T(i') : i' \geq i \} \)
- write after read ("anti"): \( \{ T(i) \rightarrow S(i') : i' > i \} \)
- write after write ("output"): \( \{ S(i) \rightarrow S(i') : i' > i \} \)

False dependences not from dataflow, but from reuse of memory location \( t \)

Possible solution: expansion/privatization

```java
for (int i = 0; i < n; ++i) {
    S: t[i] = f1(A[i]);
    T: B[i] = f2(t[i]);
}
```

- dataflow (subset of "true" dependences): \( \{ S(i) \rightarrow T(i) \} \)
False Dependences

for (int i = 0; i < n; ++i) {
S: \quad t = f1(A[i]);
T: \quad B[i] = f2(t);
}

Dependences

- read after write ("true"): \{ S(i) \rightarrow T(i') : i' \geq i \}
- write after read ("anti"): \{ T(i) \rightarrow S(i') : i' > i \}
- write after write ("output"): \{ S(i) \rightarrow S(i') : i' > i \}

"false"

False dependences not from dataflow, but from reuse of memory location \( t \)

Possible solution: expansion/privatization

for (int i = 0; i < n; ++i) {
S: \quad \textcolor{red}{t[i]} = f1(A[i]);
T: \quad B[i] = f2(t[i]);
}

- dataflow (subset of "true" dependences): \{ S(i) \rightarrow T(i) \}
Lexicographic Optimization (Space Local)

- Lexicographic Minimum of Sets
  \[
  S = \{ S(i) : f(i) \}
  \]
  \[
  \text{lexmin } S = \{ S(i) : f(i) \land \forall i' : f(i') \Rightarrow i \preceq i' \}
  \]

- Lexicographic Maximum of Relations
  \[
  R = \{ S(i) \rightarrow T(j) : f(i, j) \}
  \]
  \[
  \text{lexmax } R = \{ S(i) \rightarrow T(j) : f(i, j) \land \forall j' : f(i, j') \Rightarrow j \succeq j' \}
  \]

- Lexicographic Maximum of Sets
  \[
  S = \{ S(i) : f(i) \}
  \]
  \[
  \text{lexmax } S = \{ S(i) : f(i) \land \forall i' : f(i') \Rightarrow i \succeq i' \}
  \]
Lexicographic Optimization (Space Local)

- Lexicographic Minimum of Sets
  \[ S = \{ S(i) : f(i) \} \]
  \[ \text{lexmin } S = \{ S(i) : f(i) \land \forall i' : f(i') \Rightarrow i \preceq i' \} \]
  \[ I = \{ R(); S(i, j) : 0 \leq i < 2 \land 0 \leq j < 2; T(k) : 0 \leq k < 2 \} \]
  \[ \text{lexmin } I = \{ R(); S(0, 0); T(0) \} \]

- Lexicographic Maximum of Sets
  \[ S = \{ S(i) : f(i) \} \]
  \[ \text{lexmax } S = \{ S(i) : f(i) \land \forall i' : f(i') \Rightarrow i \succeq i' \} \]
  \[ \text{lexmax } I = \{ R(); S(1, 1); T(1) \} \]
Lexicographic Optimization (Space Local)

- **Lexicographic Minimum of Sets** *(iscc: lexmin)*
  \[ S = \{ S(i) : f(i) \} \]
  \[ \text{lexmin } S = \{ S(i) : f(i) \land \forall i' : f(i') \Rightarrow i \preceq i' \} \]
  \[ I = \{ R(); S(i, j) : 0 \leq i < 2 \land 0 \leq j < 2; T(k) : 0 \leq k < 2 \} \]
  \[ \text{lexmin } I = \{ R(); S(0, 0); T(0) \} \]

- **Lexicographic Maximum of Sets** *(iscc: lexmax)*
  \[ S = \{ S(i) : f(i) \} \]
  \[ \text{lexmax } S = \{ S(i) : f(i) \land \forall i' : f(i') \Rightarrow i \succeq i' \} \]
  \[ \text{lexmax } I = \{ R(); S(1, 1); T(1) \} \]

- **Lexicographic Maximum of Relations** *(iscc: lexmax)*
  \[ R = \{ S(i) \rightarrow T(j) : f(i, j) \} \]
  \[ \text{lexmax } R = \{ S(i) \rightarrow T(j) : f(i, j) \land \forall j' : f(i, j') \Rightarrow j \succeq j' \} \]
  \[ W^{-1} = \{ A(a) \rightarrow S(i, j) : a = i + j \land 0 \leq i < 2 \land 0 \leq j < 2 \} \]
  \[ \text{lexmax}(W^{-1}) = \{ A(a) \rightarrow S(a, 0) : 0 \leq a \leq 1; A(2) \rightarrow S(1, 1) \} \]
  \[ \Rightarrow \text{last statement instance writing array element} \]
Array Dataflow Analysis

Given a read from an array element, what was the last write to the same array element before the read?

```plaintext
for (i = 0; i < N; ++i)
    for (j = 0; j < N - i; ++j)
F: a[i+j] = f(a[i+j]);
for (i = 0; i < N; ++i)
W: Write(a[i]);
```

Array Dataflow Analysis

*Given a read from an array element, what was the last write to the same array element before the read?*

```c
for (i = 0; i < N; ++i)
    for (j = 0; j < N - i; ++j)
F:     a[i+j] = f(a[i+j]);
for (i = 0; i < N; ++i)
W:     Write(a[i]);
```
Array Dataflow Analysis

Given a read from an array element, what was the last write to the same array element before the read?

```c
for (i = 0; i < N; ++i)
    for (j = 0; j < N - i; ++j)
F:     a[i+j] = f(a[i+j]);
for (i = 0; i < N; ++i)
W:     Write(a[i]);
```

Access relations:

\[ A_1 = \{ F(i,j) \rightarrow a(i+j) : 0 \leq i < N \land 0 \leq j < N - i \} \]

\[ A_2 = \{ W(i) \rightarrow a(i) : 0 \leq i < N \} \]
Array Dataflow Analysis

*Given a read from an array element, what was the last write to the same array element before the read?*

```plaintext
for (i = 0; i < N; ++i)
    for (j = 0; j < N - i; ++j)
        F: a[i+j] = f(a[i+j]);
for (i = 0; i < N; ++i)
    W: Write(a[i]);
```

Access relations:

\[ A_1 = \{ F(i, j) \rightarrow a(i + j) : 0 \leq i < N \land 0 \leq j < N - i \} \]

\[ A_2 = \{ W(i) \rightarrow a(i) : 0 \leq i < N \} \]

Map to all writes:

\[ R' = A_1^{-1} \circ A_2 = \{ W(i) \rightarrow F(i', i - i') : 0 \leq i' \leq i < N \} \]
Array Dataflow Analysis

Given a read from an array element, what was the last write to the same array element before the read?

for (i = 0; i < N; ++i)
    for (j = 0; j < N - i; ++j)
        F: \( a[i+j] = f(a[i+j]) \);
for (i = 0; i < N; ++i)
    W: Write(a[i]);

Access relations:
\( A_1 = \{ F(i, j) \rightarrow a(i + j): 0 \leq i < N \land 0 \leq j < N - i \} \)
\( A_2 = \{ W(i) \rightarrow a(i): 0 \leq i < N \} \)

Map to all writes:
\( R' = A_1^{-1} \circ A_2 = \{ W(i) \rightarrow F(i', i - i'): 0 \leq i' \leq i < N \} \)

Last write: \( R = \text{lexmax } R' = \{ W(i) \rightarrow F(i, 0): 0 \leq i < N \} \)
Array Dataflow Analysis

Given a read from an array element, what was the last write to the same array element before the read?

```plaintext
for (i = 0; i < N; ++i)
    for (j = 0; j < N - i; ++j)
        F: a[i+j] = f(a[i+j]);
for (i = 0; i < N; ++i)
    W: Write(a[i]);
```

Access relations:

\[ A_1 = \{ F(i, j) \rightarrow a(i + j) : 0 \leq i < N \land 0 \leq j < N - i \} \]

\[ A_2 = \{ W(i) \rightarrow a(i) : 0 \leq i < N \} \]

Map to all writes:

\[ R' = A_1^{-1} \circ A_2 = \{ W(i) \rightarrow F(i', i - i') : 0 \leq i' \leq i < N \} \]

Last write: \[ R = \text{lexmax } R' = \{ W(i) \rightarrow F(i, 0) : 0 \leq i < N \} \]

In general: impose lexicographical order on shared branch of schedule tree
Expansion

Assume:

- instance sets and access relations are static and exact
  ⇒ each read has exactly one corresponding write
- single read and write per statement
  ⇒ expanded array indexed by statement instance of write
**Expansion**

Assume:

- instance sets and access relations are static and exact
  ⇒ each read has exactly one corresponding write
- single read and write per statement
  ⇒ expanded array indexed by statement instance of write

```java
for (int i = 0; i < n; ++i) {
    S: t = f1(A[i]);
    T: B[i] = f2(t);
}
```

Dataflow: \{ S(i) \rightarrow T(i) \}
Expansion

Assume:

- instance sets and access relations are static and exact
  ⇒ each read has exactly one corresponding write
- single read and write per statement
  ⇒ expanded array indexed by statement instance of write

```java
for (int i = 0; i < n; ++i) {
    S: t = f1(A[i]);
    T: B[i] = f2(t);
}
```

Dataflow: \{ S(i) \rightarrow T(i) \}

```java
for (int i = 0; i < n; ++i) {
    S: S[i] = f1(A[i]);
    T: B[i] = f2(S[i]);
}
```

⇒ only remaining dependences are dataflow induced
Tagged Access Relations

Expansion in case of multiple reads or writes per statement

⇒ statement instance not enough to identify memory access
⇒ use identifier of array reference instead
Tagged Access Relations

Expansion in case of multiple reads or writes per statement

⇒ statement instance not enough to identify memory access
⇒ use identifier of array reference instead
⇒ pet: embed array reference in “tagged” instance set
⇒ “ternary” access relations

\[ I \to R \to A \]

- \( I \): statement instance
- \( R \): reference identifier
- \( O \): array element

⇒ in practice: nested binary relation

\[ (I \to R) \to A \]

For example, \( \{ (S(i) \to R1()) \to A(i) \} \)
Wrapping, Unwrapping, Domain Map and Range Map

\[ R = \{ S(i) \rightarrow T(j) : f(i, j) \} \quad S = \{ (U(i) \rightarrow V(j)) : g(i, j) \} \]

- Wrap
  \[ \mathcal{W}R = \{ (S(i) \rightarrow T(j)) : f(i, j) \} \]  
  (iscc: wrap)
Wrapping, Unwrapping, Domain Map and Range Map

\[ R = \{ S(i) \rightarrow T(j) : f(i, j) \} \quad S = \{ (U(i) \rightarrow V(j)) : g(i, j) \} \]

- Wrap
  \[ WR = \{ (S(i) \rightarrow T(j)) : f(i, j) \} \]
- Unwrap
  \[ W^{-1}S = \{ U(i) \rightarrow V(j) : g(i, j) \} \]
Wrapping, Unwrapping, Domain Map and Range Map

\[ R = \{ S(i) \rightarrow T(j) : f(i,j) \} \quad S = \{ (U(i) \rightarrow V(j)) : g(i,j) \} \]

- **Wrap**
  \[ \mathcal{W}R = \{ (S(i) \rightarrow T(j)) : f(i,j) \} \]  
  (iscc: wrap)

- **Unwrap**
  \[ \mathcal{W}^{-1}S = \{ U(i) \rightarrow V(j) : g(i,j) \} \]  
  (iscc: unwrap)

- **Domain map**
  \[ \text{dom} R = \{ (S(i) \rightarrow T(j)) \rightarrow S(i) : f(i,j) \} \]  
  (iscc: domain_map)

\[ I = \{ (S(i) \rightarrow R1()) : 0 \leq i < n \} \]

\[ \text{dom}(\mathcal{W}^{-1}I) = \{ (S(i) \rightarrow R1()) \rightarrow S(i) : 0 \leq i < n \} \]

⇒ maps tagged instance set to untagged instance set
⇒ precompose with instance set based relations to obtain
  tagged instance set based relations
Wrapping, Unwrapping, Domain Map and Range Map

\[ R = \{ S(i) \rightarrow T(j) : f(i, j) \} \]
\[ S = \{ (U(i) \rightarrow V(j)) : g(i, j) \} \]

- **Wrap**
  \[ \mathcal{W}R = \{ (S(i) \rightarrow T(j)) : f(i, j) \} \]

- **Unwrap**
  \[ \mathcal{W}^{-1}S = \{ U(i) \rightarrow V(j) : g(i, j) \} \]

- **Domain map**
  \[ \text{dom} R = \{ (S(i) \rightarrow T(j)) \rightarrow S(i) : f(i, j) \} \]

\[ I = \{ (S(i) \rightarrow R1()) : 0 \leq i < n \} \]
\[ \text{dom}(\mathcal{W}^{-1}I) = \{ (S(i) \rightarrow R1()) \rightarrow S(i) : 0 \leq i < n \} \]

\[ \Rightarrow \text{maps tagged instance set to untagged instance set} \]
\[ \Rightarrow \text{precompose with instance set based relations to obtain tagged instance set based relations} \]

- **Range map**
  \[ \text{ran} R = \{ (S(i) \rightarrow T(j)) \rightarrow T(j) : f(i, j) \} \]
Parametric Example: Matrix Multiplication

```java
for (int i = 0; i < M; i++)
    for (int j = 0; j < N; j++) {
        S1: C[i][j] = 0;
        for (int k = 0; k < K; k++)
            S2: C[i][j] = C[i][j] + A[i][k] * B[k][j];
    }
```

- **Tagged Access Relations**
  
  $W = \{ (S1(i,j) \rightarrow R0()) \rightarrow C(i,j); (S2(i,j,k) \rightarrow R1()) \rightarrow C(i,j) \}$
  
  $R = \{ (S2(i,j,k) \rightarrow R2()) \rightarrow C(i,j); (S2(i,j,k) \rightarrow R3()) \rightarrow A(i,k);
  (S2(i,j,k) \rightarrow R4()) \rightarrow B(k,j) \}$

- **Tagged Schedule**
  
  \[
  \{ (S1(i,j) \rightarrow R0()) \rightarrow (i,j,0,0); (S2(i,j,k) \rightarrow R1()) \rightarrow (i,j,1,k) \\
  (S2(i,j,k) \rightarrow R2()) \rightarrow (i,j,1,k); (S2(i,j,k) \rightarrow R3()) \rightarrow (i,j,1,k); \\
  (S2(i,j,k) \rightarrow R4()) \rightarrow (i,j,1,k) \}
  \]

- **Tagged Dataflow**
  
  \[
  \{ (S1(i,j) \rightarrow R0()) \rightarrow (S2(i,j,0) \rightarrow R2()) \\
  (S2(i,j,k) \rightarrow R1()) \rightarrow (S2(i,j,k + 1) \rightarrow R2()) \}
  \]
Maximal Static Expansion

```java
for (int i = 0; i < n; ++i) {
    S1:   t = f1(i);
    S2:   A[i] = t;
    S3:   t = f2(i);
    S4:   if (f3(i))
          S5:       t = f4(i);
    S6:   B[i] = t;
}
```

Dataflow cannot be determined independently of run-time information
Maximal Static Expansion

for (int i = 0; i < n; ++i) {
    t = f1(i);
    A[i] = t;
    t = f2(i);
    if (f3(i))
        t = f4(i);
    B[i] = t;
}

Dataflow cannot be determined independently of run-time information

⇒ approximate dataflow

{ S1(i) → S2(i); S3(i) → S6(i); S5(i) → S6(i) }
Maximal Static Expansion

\[
\text{for (int } i = 0; i < n; ++i) \{ \\
S1: & \quad t = f1(i); \\
S2: & \quad A[i] = t; \\
S3: & \quad t = f2(i); \\
S4: & \quad \text{if (f3(i))} \\
S5: & \quad \quad t = f4(i); \\
S6: & \quad B[i] = t; \\
\}
\]

Dataflow cannot be determined independently of run-time information

$\Rightarrow$ approximate dataflow

\[\{ S1(i) \rightarrow S2(i); S3(i) \rightarrow S6(i); S5(i) \rightarrow S6(i) \}\]

$\Rightarrow$ a read may be associated to more than one write

$\Rightarrow$ corresponding equivalence classes should not be expanded apart
Maximal Static Expansion

```
for (int i = 0; i < n; ++i) {
S1: t = f1(i);          t1[i] = f1(i);
S2: A[i] = t;            A[i] = t1[i];
S3: t = f2(i);          t2[i] = f2(i);
S4: if (f3(i))         if (f3(i))
S5:          t = f4(i);  t2[i] = f4(i);
S6: B[i] = t;            B[i] = t2[i];
}
```

Dataflow cannot be determined independently of run-time information

⇒ approximate dataflow

\{ S1(i) \rightarrow S2(i); S3(i) \rightarrow S6(i); S5(i) \rightarrow S6(i) \}

⇒ a read may be associated to more than one write

⇒ corresponding equivalence classes should not be expanded apart
Approximate Dataflow Analysis

How to compute dataflow in presence of data dependent control?

Two approaches
- Direct computation
  - distinguish between may- and must-writes
Approximate Dataflow Analysis

How to compute dataflow in presence of data dependent control?

Two approaches

- Direct computation
  - distinguish between may- and must-writes

- Derived from exact run-time dependent dataflow
  - compute exact dataflow in terms of run-time information
  - exploit properties of run-time information
  - project out run-time information
Approximate Dataflow Analysis

How to compute dataflow in presence of data dependent control?

Two approaches

- **Direct computation**
  - distinguish between may- and must-writes

- **Derived from exact run-time dependent dataflow**
  - compute exact dataflow in terms of run-time information
  - exploit properties of run-time information
  - project out run-time information
May Writes

Keep track of whether write is possible or definite

- Must-writes
  Array elements that are definitely accessed by statement instance

- May-writes
  Array elements that are possibly accessed by statement instance

Must-write access relation is subset of may-write access relation
May Writes

Keep track of whether write is possible or definite

- **Must-writes**
  Array elements that are definitely accessed by statement instance

- **May-writes**
  Array elements that are possibly accessed by statement instance

  - statement instance not necessarily executed
    ```
    for (i = 0; i < n; ++i)
      if (A[i] > 0)
        S: B[i] = A[i];
    May-write: \{ S(i) \rightarrow B(i) \}
    ```

Must-write access relation is subset of may-write access relation
May Writes

Keep track of whether write is possible or definite

- Must-writes
  Array elements that are definitely accessed by statement instance

- May-writes
  Array elements that are possibly accessed by statement instance

  ▶ statement instance not necessarily executed
    ```
    for (i = 0; i < n; ++i)
        if (A[i] > 0)
            S: B[i] = A[i];
    May-write: \{ S(i) \rightarrow B(i) \}
    ```

  ▶ array element not necessarily accessed
    ```
    int A[N];
    /* ... */
    T: A[B[0]] = 5;
    May-write: \{ T() \rightarrow A(a) : 0 \leq a < N \}
    ```

Must-write access relation is subset of may-write access relation
Approximate Dataflow — Direct Computation

- Read after write dependences
  - write and read access same memory location
  - write executed before the read
Approximate Dataflow — Direct Computation

- **Read after write dependences**
  - write and read access same memory location
  - write executed before the read

- **Dataflow dependences**
  - write and read access same memory location
  - write executed before the read
  - no intermediate write to same memory location
  \[ \Rightarrow \text{intermediate write kills dependence} \]
Approximate Dataflow — Direct Computation

- Read after write dependences
  - write and read access same memory location
  - write executed before the read

- Dataflow dependences
  - write and read access same memory location
  - write executed before the read
  - no intermediate write to same memory location
    \(\Rightarrow\) intermediate write kills dependence

- Approximate dataflow dependences
  - may-write and read access same memory location
  - may-write executed before the read
  - no intermediate must-write to same memory location
    \(\Rightarrow\) intermediate must-write kills dependence
Approximate Dataflow — Direct Computation

- **Read after write dependences**
  - write and read access same memory location
  - write executed before the read

  $\Rightarrow$ Approximate dataflow analysis with no must-writes

- **Dataflow dependences**
  - write and read access same memory location
  - write executed before the read
  - no intermediate write to same memory location

    $\Rightarrow$ intermediate write kills dependence

- **Approximate dataflow dependences**
  - may-write and read access same memory location
  - may-write executed before the read
  - no intermediate must-write to same memory location

    $\Rightarrow$ intermediate must-write kills dependence
Performing Dataflow Analysis

Two possibilities

1. dataflow analysis on polyhedral model
   - first extract instance set, access relations and schedule
   - then perform dataflow analysis
   E.g., isl

2. dataflow analysis on AST before/during model extraction
   Proposed by, e.g., Maslov (1994)
Performing Dataflow Analysis

Two possibilities

1. dataflow analysis on polyhedral model
   - first extract instance set, access relations and schedule
   - then perform dataflow analysis
   
   E.g., isl

2. dataflow analysis on AST before/during model extraction
   Proposed by, e.g., Maslov (1994)

Dataflow in Parallel Programs

1. use refined execution order during dataflow analysis
2. remove spurious dependences after dataflow analysis

Note: dataflow from must-writes cannot be removed without caution
   ⇒ must-write may have killed other dependences
   ⇒ other dependences may have to be added back
Explicit Kills

```c
int A[N];

S: A[0] = 1;

for (int i = 0; i < N; ++i)
T: A[perm[i]] = f(i);
U: f(A[0]);
```

- Assume `perm` represents a permutation
  - ⇒ there can be no dataflow from `S` to `U`
Explicit Kills

```c
int A[N];

S: A[0] = 1;

for (int i = 0; i < N; ++i)
T: A[perm[i]] = f(i);
U: f(A[0]);
```

- Assume `perm` represents a permutation
  -⇒ there can be no dataflow from S to U

- Compiler does not know all elements are written by T
  -⇒ may find dataflow from S to U
  -⇒ user can insert explicit kill
  -⇒ explicit kill used to kill dependences, just like must-write
Explicit Kills

```c
int A[N];

S:  A[0] = 1;
     __pencil_kill(A);
for (int i = 0; i < N; ++i)
T:  A[perm[i]] = f(i);
U:  f(A[0]);
```

- Assume `perm` represents a permutation
  - ⇒ there can be no dataflow from `S` to `U`
- Compiler does not know all elements are written by `T`
  - ⇒ may find dataflow from `S` to `U`
  - ⇒ user can insert explicit kill
  - ⇒ explicit kill used to kill dependences, just like must-write
Local Variables and Kills

```c
for (int i = 0; i < N; ++i) {
    int t;
    /* ... */
}
```

⇒ there can be no dataflow on \( t \) across different iterations

⇒ pet automatically inserts kills
  ▶ before declaration and
  ▶ at end of block containing declaration
Killing False Dependences

Dataflow derived from read after write dependences through killing.

Should we do the same for false dependences?
Killing False Dependences

Dataflow derived from read after write dependences through killing.

Should we do the same for false dependences?

⇒ no need for validity schedule constraints
⇒ removed dependences implied by remaining dependences
Killing False Dependences

Dataflow derived from read after write dependences through killing.

Should we do the same for false dependences?

⇒ no need for validity schedule constraints
⇒ removed dependences implied by remaining dependences

Optimizing locality

- typical criterion: minimize maximal dependence distance

dependence distances: \[ \{ P(S(i)) - P(T(j)) : S(i) \to T(j) \in D \} \]

\( D \): (local) dependence relation, \( P \): partial schedule
Killing False Dependences

Dataflow derived from read after write dependences through killing.

Should we do the same for false dependences?

⇒ no need for validity schedule constraints
⇒ removed dependences implied by remaining dependences

Optimizing locality

• typical criterion: minimize maximal dependence distance

  dependence distances: \( \{ P(S(i)) - P(T(j)) : S(i) \rightarrow T(j) \in D \} \)

  \( D \): (local) dependence relation, \( P \): partial schedule

• may also be useful for false dependences if no expansion is performed

  ⇒ locality of reused memory location

• killing false dependences avoids critical path determined by transitively covered dependences

  ⇒ allow must-writes to kill dependences, but not explicit kills
Approximate Dataflow Analysis

How to compute dataflow in presence of data dependent control?

Two approaches

- Direct computation
  - distinguish between may- and must-writes

- Derived from exact run-time dependent dataflow
  - compute exact dataflow in terms of run-time information
  - exploit properties of run-time information
  - project out run-time information
Approximate Dataflow Analysis

How to compute dataflow in presence of data dependent control?

Two approaches
- Direct computation
  - distinguish between may- and must-writes
- Derived from exact run-time dependent dataflow
  - compute exact dataflow in terms of run-time information
  - exploit properties of run-time information
  - project out run-time information
Run-time Dependent Dataflow Analysis

Approaches

- “fuzzy array dataflow analysis”
- “on-demand-parametric array dataflow analysis”
Run-time Dependent Dataflow Analysis

Approaches

- “fuzzy array dataflow analysis”
- “on-demand-parametric array dataflow analysis”
Run-time Dependent Dataflow Analysis

Approaches

- “fuzzy array dataflow analysis”
- “on-demand-parametric array dataflow analysis”

```
for (int i = 0; i < n; ++i) {
    S1: t = f1(i);
    S2: A[i] = t;
    S3: t = f2(i);
    S4: if (f3(i))
    S5:       t = f4(i);
    S6: B[i] = t;
}
```
Run-time Dependent Dataflow Analysis

Approaches

- “fuzzy array dataflow analysis”
- “on-demand-parametric array dataflow analysis”

```java
for (int i = 0; i < n; ++i) {
    S1: t = f1(i);
    S2: A[i] = t;
    S3: t = f2(i);
    S4: if (f3(i))
    S5: t = f4(i);
    S6: B[i] = t;
}
```

- Run-time dependent dataflow

\[
\{ S1(i) \to S2(i); S3(i) \to S6(i) : \beta_{S6}^{S5} = 0; S5(i) \to S6(i) : \beta_{S6}^{S5} = 1 \} \\
\beta_C^P: \text{any potential source instance } P \text{ is executed for sink } C \\
\lambda_C^P: \text{last potential source instance } P \text{ executed for sink } C
\]
Run-time Dependent Dataflow Analysis

Approaches

- “fuzzy array dataflow analysis”
- “on-demand-parametric array dataflow analysis”

```java
for (int i = 0; i < n; ++i) {
    S1: t = f1(i);
    S2: A[i] = t;
    S3: t = f2(i);
    S4: if (f3(i))
        S5: t = f4(i);
    S6: B[i] = t;
}
```

- Run-time dependent dataflow

\[
\{ S1(i) \rightarrow S2(i); S3(i) \rightarrow S6(i) : \beta_{S6}^{S5} = 0; S5(i) \rightarrow S6(i) : \beta_{S6}^{S5} = 1 \}
\]

\(\beta^P_C\): any potential source instance \(P\) is executed for sink \(C\)

\(\lambda^P_C\): last potential source instance \(P\) executed for sink \(C\)

- Approximate dataflow (project out \(\beta\) and \(\lambda\))

\[
\{ S1(i) \rightarrow S2(i); S3(i) \rightarrow S6(i); S5(i) \rightarrow S6(i) \}
\]
Representing Dynamic Conditions

N1: \( n = f() \);
   for (int \( k = 0; k < 100; ++k \) ) {
   M: \( m = g() \);
      for (int \( i = 0; i < m; ++i \) )
         for (int \( j = 0; j < n; ++j \) )
            a[j][i] = g();
   A: \( n = f() \); 
N2: \( n = f() \ );
}

What is instance set (restricted to A statement)?
Representing Dynamic Conditions

N1: \( n = f(); \)
  for (int \( k = 0; \ k < 100; \ ++k \) ) {
M: \( m = g(); \)
  for (int \( i = 0; \ i < m; \ ++i \) )
    for (int \( j = 0; \ j < n; \ ++j \) )
      a[j][i] = g();
A: \( a[j][i] = g(); \)
N2: \( n = f(); \)
}

What is instance set (restricted to A statement)?
\{ A(k,i,j) : 0 \leq k < 100 \land 0 \leq i < m \land 0 \leq j < n \}?
Representing Dynamic Conditions

\[ \text{N1: } n = f(); \]
\[ \quad \text{for (int } k = 0; k < 100; ++k) \{ \]
\[ \quad \text{M: } \]
\[ \quad \quad m = g(); \]
\[ \quad \quad \text{for (int } i = 0; i < m; ++i) \]
\[ \quad \quad \quad \text{for (int } j = 0; j < n; ++j) \]
\[ \quad \quad \quad a[j][i] = g(); \]
\[ \text{A: } \]
\[ \quad \text{N2: } n = f(); \]
\[ \} \]

What is instance set (restricted to A statement)?

\[ \{ A(k, i, j) : 0 \leq k < 100 \land 0 \leq i < m \land 0 \leq j < n \} \]

\[ \Rightarrow \text{ no, } m \text{ and } n \text{ cannot be treated as symbolic constants (they are modified inside } k\text{-loop) } \]
Representing Dynamic Conditions

N1: \( n = f(); \)
    for (int \( k = 0; k < 100; ++k \)) {
    M: \( m = g(); \)
        for (int \( i = 0; i < m; ++i \))
            for (int \( j = 0; j < n; ++j \))
                A: \( a[j][i] = g(); \)
    }
N2: \( n = f(); \)

What is instance set (restricted to A statement)?
{ \( A(k, i, j) : 0 \leq k < 100 \land 0 \leq i < m \land 0 \leq j < n \) }?

\( \Rightarrow \) no, \( m \) and \( n \) cannot be treated as symbolic constants
(they are modified inside \( k \)-loop)

{ \( A(k, i, j) : 0 \leq k < 100 \land 0 \leq i < valueOf m(k) \land 0 \leq j < valueOf n(k) \) }?
Representing Dynamic Conditions

N1: \( n = f(); \)
   \[
   \text{for (int} \ k = 0; \ k < 100; \ ++k) \ \{
   \]
M: \( m = g(); \)
   \[
   \text{for (int} \ i = 0; \ i < m; \ ++i)
   \]
   \[
   \text{for (int} \ j = 0; \ j < n; \ ++j)
   \]
A: \( a[j][i] = g(); \)
N2: \( n = f(); \)
   \[
   \}
\]

What is instance set (restricted to A statement)?

\[ \{ A(k, i, j) : 0 \leq k < 100 \land 0 \leq i < m \land 0 \leq j < n \} ? \]

\( \Rightarrow \) no, \( m \) and \( n \) cannot be treated as symbolic constants
(they are modified inside \( k \)-loop)

\[ \{ A(k, i, j) : 0 \leq k < 100 \land 0 \leq i < \text{valueOf}_m(k) \land 0 \leq j < \text{valueOf}_n(k) \} ? \]

\( \Rightarrow \) requires uninterpreted functions (of arity > 0)
Representing Dynamic Conditions

N1: \( n = f(); \)
   for (int \( k = 0; k < 100; ++k \)) {
M: \( m = g(); \)
   for (int \( i = 0; i < m; ++i \))
      for (int \( j = 0; j < n; ++j \))
A: \( a[j][i] = g(); \)
N2: \( n = f(); \)
}

What is instance set (restricted to \( A \) statement)?
\{ \( A(k, i, j) : 0 \leq k < 100 \land 0 \leq i < m \land 0 \leq j < n \) \}?

\( \Rightarrow \) no, \( m \) and \( n \) cannot be treated as symbolic constants
(they are modified inside \( k \)-loop)

\{ \( A(k, i, j) : 0 \leq k < 100 \land 0 \leq i < \text{valueOf} m(k) \land 0 \leq j < \text{valueOf} n(k) \) \}?
\( \Rightarrow \) requires uninterpreted functions (of arity > 0)

Alternative: use overapproximation of instance set and keep track of which elements are executed
Representing Dynamic Conditions

N1: \( n = f(); \)
    for (int \( k = 0; k < 100; ++k \)) {
M: \( m = g(); \)
    for (int \( i = 0; i < m; ++i \))
      for (int \( j = 0; j < n; ++j \))
        \( a[j][i] = g(); \)
A: \( n = f(); \)
}

Instance set: \( \{ A(k, i, j) : 0 \leq k < 100 \land 0 \leq i \land 0 \leq j \} \)

Filter:
- Filter access relations: reader \( \rightarrow \) (writer \( \rightarrow \) array element)
  - \( F_1^A = \{ A(k, i, j) \rightarrow (M(k) \rightarrow m()) \} \)
  - \( F_2^A = \{ A(0, i, j) \rightarrow (N1() \rightarrow n()); A(k, i, j) \rightarrow (N2(k - 1) \rightarrow n()) : k \geq 1 \} \)
- Filter value relation:
  \( V^A = \{ A(k, i, j) \rightarrow (m, n) : 0 \leq k \leq 99 \land 0 \leq i < m \land 0 \leq j < n \} \)

Statement instance is executed iff values written by corresponding write accesses (through filter access relations) satisfy filter value relation.
Representing Dynamic Conditions

N1: \( n = f(); \)
    \[
    \text{for (int } k = 0; k < 100; ++k) \{
    \]
M: \( m = g(); \)
    \[
    \text{for (int } i = 0; i < m; ++i) \{
    \]
A: \( a[j][i] = g(); \)
    \[
    \}
\]
N2: \( n = f(); \)
}

- **Instance set:** \( \{ A(k, i, j) : 0 \leq k < 100 \land 0 \leq i \land 0 \leq j \} \)
- **Filter:**
  - Filter access relations: \( \text{reader } \rightarrow (\text{writer } \rightarrow \text{array element}) \)
    - \( F^A_1 = \{ A(k, i, j) \rightarrow (M(k) \rightarrow m()) \} \)
    - \( F^A_2 = \{ A(0, i, j) \rightarrow (N1() \rightarrow n()); A(k, i, j) \rightarrow (N2(k-1) \rightarrow n()) : k \geq 1 \} \)
  - Filter value relation:
    \( V^A = \{ A(k, i, j) \rightarrow (m, n) : 0 \leq k \leq 99 \land 0 \leq i < m \land 0 \leq j < n \} \)

Statement instance is executed iff values written by corresponding write accesses (through filter access relations) satisfy filter value relation.
Parametric Array Dataflow Analysis

while (1) {
   potential source
   N:   n = f();
       a = g();
       if (n < 100)
         H:   a = h();
         if (n > 200)
           T:   t(a);
   }
   sink

Is there any dataflow between potential source and sink at inner level?
Parametric Array Dataflow Analysis

```plaintext
while (1) {
    potential source
    N: n = f();
        a = g();
        if (n < 100)
            H: a = h();
                if (n > 200)
                    T: t(a);
    }
}
```

Is there any dataflow between potential source and sink at inner level?
Parametric Array Dataflow Analysis

while (1) {
    potential source
    N:  n = f();
        a = g();
            if (n < 100)
                H:  a = h();
                    if (n > 200)
                        T:  t(a);
                I = {H(i) : i ≥ 0; T(i) : i ≥ 0}
                F^H = {H(i) → (N(i) → n())}
                V^H = {H(i) → (n) : i ≥ 0 ∧ n < 100}
                F^T = {T(i) → (N(i) → n())}
                V^T = {T(i) → (n) : i ≥ 0 ∧ n > 200}
}

sink

Is there any dataflow between potential source and sink at inner level?

- M = { T(i) → H(i) }
Parametric Array Dataflow Analysis

```c
while (1) {
    N:    n = f();
        a = g();
        if (n < 100)
            H:    a = h();
                  if (n > 200)
        T:    t(a);
}
```

Is there any dataflow between potential source and sink at inner level?

- \( M = \{ T(i) \rightarrow H(i) \} \)
- \( F^H \circ M \subseteq F^T \)

\( \Rightarrow \) filter elements accessed by any potential source instance associated to sink instance forms subset of filter elements accessed by sink instance
Parametric Array Dataflow Analysis

while (1) {
    potential source
    N:  n = f();
        a = g();
        if (n < 100)
            H: a = h();
                if (n > 200)
                    T: t(a);
        }
    sink

Is there any dataflow between potential source and sink at inner level?

- $M = \{ T(i) \rightarrow H(i) \}$
- $F^H \circ M \subseteq F^T$

$\Rightarrow$ filter elements accessed by any potential source instance associated to sink instance forms subset of filter elements accessed by sink instance

$\Rightarrow$ constraints on filter values at sink also apply at corresponding potential source: $V^T \circ M^{-1} = \{ H(i) \rightarrow (n) : i \geq 0 \wedge n > 200 \}$
Parametric Array Dataflow Analysis

while (1) { potential source
  N: n = f();
  a = g();
  if (n < 100)
    H: a = h();
    if (n > 200)
      T: t(a);
  I = {H(i) : i ≥ 0; T(i) : i ≥ 0}

  F^H = {H(i) → (N(i) → n())}
  V^H = {H(i) → (n) : i ≥ 0 ∧ n < 100}
  F^T = {T(i) → (N(i) → n())}
  V^T = {T(i) → (n) : i ≥ 0 ∧ n > 200}
}

Is there any dataflow between potential source and sink at inner level?

- $M = \{ T(i) → H(i) \}$
- $F^H \circ M \subseteq F^T$
  - ⇒ filter elements accessed by any potential source instance associated to sink instance forms subset of filter elements accessed by sink instance
  - ⇒ constraints on filter values at sink also apply at corresponding potential source: $V^T \circ M^{-1} = \{ H(i) → (n) : i ≥ 0 ∧ n > 200 \}$
- $(V^T \circ M^{-1}) \cap V^H = \emptyset$
  - ⇒ there can be no dataflow at inner level
Polyhedral Process Networks

- Main purpose: extract task level parallelism from dataflow graph
  
  statement $\rightarrow$ process
  flow dependence $\rightarrow$ communication channel

  $\Rightarrow$ requires dataflow analysis

- Processes are mapped to parallel hardware (e.g., FPGA)
Polyhedral Process Networks

- Main purpose: extract task level parallelism from dataflow graph
  
  \[ \text{statement} \rightarrow \text{process} \]
  \[ \text{flow dependence} \rightarrow \text{communication channel} \]

  \( \Rightarrow \) requires dataflow analysis

- Processes are mapped to parallel hardware (e.g., FPGA)

Example:

```c
for (int i = 0; i < n; ++i) {
    S: t = f1(A[i]);
    T: B[i] = f2(t);
}
```
Polyhedral Process Networks

- Main purpose: extract task level parallelism from dataflow graph
  
  statement $\rightarrow$ process
  flow dependence $\rightarrow$ communication channel

  $\Rightarrow$ requires dataflow analysis

- Processes are mapped to parallel hardware (e.g., FPGA)

Example:

```c
for (int i = 0; i < n; ++i) {
    S: t = f1(A[i]);
    T: B[i] = f2(t);
}
```

```c
for (int i = 0; i < n; ++i) {
    write(fifo, f1(A[i]));
    B[i] = f2(read(fifo));
}
```
Process Networks with Dynamic Control

```java
for (int i = 0; i < n; ++i) {
    S1: t = f1(i);
    S2: A[i] = t;
    S3: t = f2(i);
    S4: if (f3(i))
        S5: t = f4(i);
    S6: B[i] = t;
}
```

Run-time dependent dataflow:

```
{ S1(i) \rightarrow S2(i); S3(i) \rightarrow S6(i) : \beta_{S6}^{S5} = 0;
   S5(i) \rightarrow S6(i) : \beta_{S6}^{S5} = 1; S4(i) \rightarrow S5(i) }
```
Process Networks with Dynamic Control

```java
for (int i = 0; i < n; ++i) {
    S1: t = f1(i);
    S2: A[i] = t;
    S3: t = f2(i);
    S4: if (f3(i))
        S5: t = f4(i);
    S6: B[i] = t;
}
```

Run-time dependent dataflow:

\{ S1(i) \rightarrow S2(i); S3(i) \rightarrow S6(i) : \beta_{S6}^{S5} = 0; \\
S5(i) \rightarrow S6(i) : \beta_{S6}^{S5} = 1; S4(i) \rightarrow S5(i) \}
Reductions

A: \[ s = 0; \]
   \[ \text{for (int } i = 0; i < n; ++i) \]
B: \[ s += A[i]; \]
C: \[ B[0] = s; \]

Dataflow:
{ \( A() \rightarrow B(0); B(i) \rightarrow B(i + 1) : 0 \leq i < n - 1; B(n - 1) \rightarrow C() \) } \Rightarrow \text{fixes order of reduction}
Reductions

A: \( s = 0; \)
   \[
   \text{for (int } i = 0; i < n; ++i)\]
B: \( s += A[i]; \)
C: \( B[0] = s; \)

Dataflow:
\[
\{ A() \rightarrow B(0); B(i) \rightarrow B(i + 1) : 0 \leq i < n - 1; B(n - 1) \rightarrow C() \}
\]  
⇒ fixes order of reduction

Allow reordering of updates:
- read in C should depend on all updates in B
  ⇒ updates should not kill dependences
  ⇒ remove updates from must-writes
- updates should not depend on each other
  ⇒ remove false dependences between updates that flow to the same read, provided the read does not also depend on intermediate writes
Reductions

A: \( s = 0; \)
   for (int i = 0; i < n; ++i)
B: \( s += A[i]; \)
C: \( B[0] = s; \)

Dataflow:
\{ A() \rightarrow B(0); B(i) \rightarrow B(i+1) : 0 \leq i < n-1; B(n-1) \rightarrow C() \}
⇒ fixes order of reduction

Allow reordering of updates:
- read in C should depend on all updates in B
  ⇒ updates should not kill dependences
  ⇒ remove updates from must-writes
- updates should not depend on each other
  ⇒ remove false dependences between updates that flow to the same read, provided the read does not also depend on intermediate writes

Dataflow: \{ A() \rightarrow B(i) : 0 \leq i < n; B(i) \rightarrow C() : 0 \leq i < n \}
Outline

1. Polyhedral Model
   - Introduction
   - Representation

2. Polyhedral Transformation
   - Schedules
   - AST Generation

3. Dependences
   - Schedule Validity
   - Dependences
   - Structures

4. Dataflow
   - Parallelism
   - Dataflow
   - Approximate Dataflow
   - Run-time dependent Dataflow
   - Reductions

5. Aliasing

6. Counting
   - Cardinality
   - Bounds
   - Weighted Counting
   - Dynamic Memory Requirement

7. Transitive Closures
Some possible ways of handling aliasing:

- use an input language that does not permit aliasing
- pretend the problem does not exist
- require user to ensure absence of aliasing  
  - e.g., use `restrict` keyword
- handle as may-write  
  - may lead to too many dependences
- check aliasing at run-time  
  - use original code in case of aliasing
Outline

1. Polyhedral Model
   - Introduction
   - Representation

2. Polyhedral Transformation
   - Schedules
   - AST Generation

3. Dependences
   - Schedule Validity
   - Dependences
   - Structures

4. Dataflow
   - Parallelism
   - Dataflow
   - Approximate Dataflow
   - Run-time dependent Dataflow
   - Reductions

5. Aliasing

6. Counting
   - Cardinality
   - Bounds
   - Weighted Counting
   - Dynamic Memory Requirement

7. Transitive Closures
Cardinality

- Cardinality of a set
  - number of elements in the set
  - may depend on symbolic constants
  \[ S = \{ S(i) : f(i) \} \]
  \[ \text{card} \ S = \{ n : n = \#i : f(i) \} \]
  \[ \text{card} \left( \bigcup \ S_i \right) := \sum_i \text{card} \ S_i \]
  \[ \text{card} \{ A(i) : 0 \leq i \leq n; B() \} = n + 2 \]

- Cardinality of a binary relation
  - for each domain element, number of corresponding images
  \[ R = \{ S(i) \rightarrow T(j) : f(i,j) \} \]
  \[ \text{card} \ R = \{ S(i) \rightarrow n : n = \#j : f(i,j) \} \]
  \[ \text{card} \left( \bigcup \ R_i \right) := \sum_i \text{card} \ R_i \]
  \[ R = \{ A(i) \rightarrow C(i) : 0 \leq i \leq n; B() \rightarrow C(i) : 0 \leq i \leq n \} \]
  \[ \text{card} \ R = \{ A(i) \rightarrow 1 : 0 \leq i \leq n; B() \rightarrow n + 1 \} \]

\[ \Rightarrow \text{not a Presburger formula} \]
Cardinality Examples

\[
\text{for (i = 0; i < N; ++i)}
    \text{for (j = 0; j < N - i; ++j)}
        \text{a[i+j] = f(a[i+j]);}
\]

- How many times is the statement executed?
  \[
  \text{card}\{(i, j) : 0 \leq i < N \land 0 \leq j < N - i\}
  \Rightarrow \{ \frac{N+N^2}{2} : N \geq 1 \}\]
Cardinality Examples

```c
for (i = 0; i < N; ++i)
    for (j = 0; j < N - i; ++j)
        a[i+j] = f(a[i+j]);
```

- How many times is the statement executed?
  \[ \text{card}\{ (i,j) : 0 \leq i < N \land 0 \leq j < N - i \} \]
  \[ \Rightarrow \{ \frac{N+N^2}{2} : N \geq 1 \} \]

- How many times is a given array element written?
  \[ \text{card}(\{ (i,j) \rightarrow a(i + j) : 0 \leq i < N \land 0 \leq j < N - i \})^{-1} \]
  \[ \Rightarrow \{ a(a) \rightarrow 1 + a : 0 \leq a < N \} \]
Cardinality Examples

```c
for (i = 0; i < N; ++i)
    for (j = 0; j < N - i; ++j)
        a[i+j] = f(a[i+j]);
```

- How many times is the statement executed?
  
  \[
  \text{card}\{(i, j) : 0 \leq i < N \land 0 \leq j < N - i\} \Rightarrow \{ \frac{N+N^2}{2} : N \geq 1 \} \]

- How many times is a given array element written?
  
  \[
  \text{card} (\{(i, j) \rightarrow a(i + j) : 0 \leq i < N \land 0 \leq j < N - i\})^{-1} \Rightarrow \{ a(a) \rightarrow 1 + a : 0 \leq a < N \} \]

- How many array elements are written?
  
  \[
  \text{card} (\text{ran} \{(i, j) \rightarrow a(i + j) : 0 \leq i < N \land 0 \leq j < N - i\}) \Rightarrow \{ N : N \geq 1 \} \]
Cardinality Examples (2)

How many times is $S_1$ executed?

```c
for (i = max(0,N-M); i <= N-M+3; i++)
    for (j = 0; j <= N-2*i; j++)
        S1;

card\{ (i,j) : 0, N − M ≤ i ≤ N − M + 3 ∧ 0 ≤ j ≤ N − 2i \}
```

\[
\begin{cases}
-4N + 8M - 8 & \text{if } M \leq N \leq 2M - 6 \\
MN - 2N - M^2 + 6M - 8 & \text{if } N \leq M \leq N + 3 ∧ N \leq 2M \\
\frac{N^2}{4} + \frac{3}{4}N + \frac{1}{2} \left\lfloor \frac{N}{2} \right\rfloor + 1 & \text{if } 0 \leq N \leq M ∧ 2M \leq N + 6 \\
\frac{N^2}{4} - MN - \frac{5}{4}N + M^2 + 2M + \frac{1}{2} \left\lfloor \frac{N}{2} \right\rfloor + 1 & \text{if } M \leq N \leq 2M \leq N + 6
\end{cases}
\]
Cardinality Examples (2)

How many times is $S1$ executed?

\[
\text{for } (i = \max(0, N-M); \ i <= N-M+3; \ i++) \\
\quad \text{for } (j = 0; \ j <= N-2*i; \ j++) \\
\quad S1; \\
\text{card}\{ (i, j) : 0, N - M \leq i \leq N - M + 3 \land 0 \leq j \leq N - 2i \} \\
\begin{cases} 
-4N + 8M - 8 & \text{if } M \leq N \leq 2M - 6 \\
MN - 2N - M^2 + 6M - 8 & \text{if } N \leq M \leq N + 3 \land N \leq 2M \\
\frac{N^2}{4} + \frac{3}{4}N + \frac{1}{2} \left\lfloor \frac{N}{2} \right\rfloor + 1 & \text{if } 0 \leq N \leq M \land 2M \leq N + 6 \\
\frac{N^2}{4} - MN - \frac{5}{4}N + M^2 + 2M + \frac{1}{2} \left\lfloor \frac{N}{2} \right\rfloor + 1 & \text{if } M \leq N \leq 2M \leq N + 6 
\end{cases}
\]
Cardinality Representation

- Integer quasi affine expression
  \[ \left\lfloor \frac{x}{2} \right\rfloor + 3N \]
  \[ \Rightarrow \text{ Presburger term} \]
  That is, a term constructed from variables, symbolic constants, integer constants, addition (+), subtraction (−) and integer division by a constant (\(\left\lfloor \cdot /d \right\rfloor\)).
Cardinality Representation

- **Integer quasi affine expression**
  - $\lfloor x/2 \rfloor + 3N$
  - Presburger term
  - That is, a term constructed from variables, symbolic constants, integer constants, addition (+), subtraction (−) and integer division by a constant ($\lfloor \cdot/d \rfloor$)

- **Rational polynomial expression**
  - $x^2 - N/2$
  - a term constructed from variables, symbolic constants, rational constants, addition (+), subtraction (−) and multiplication (·)
Cardinality Representation

- Integer quasi affine expression
  \[ \lfloor x/2 \rfloor + 3N \]
  ⇒ Presburger term
  That is, a term constructed from variables, symbolic constants, integer constants, addition (+), subtraction (−) and integer division by a constant (\( \lfloor \cdot / d \rfloor \))

- Rational polynomial expression
  \[ x^2 - N/2 \]
  ⇒ a term constructed from variables, symbolic constants, rational constants, addition (+), subtraction (−) and multiplication (\( \cdot \))

- Quasi polynomial expression
  \[ (\lfloor x/2 \rfloor + 3N)^2 - N/2 \]
  ⇒ a rational polynomial expression with variables replaced by integer quasi affine expressions

Note: in practice, cardinality result does not contain nested integer divisions
Cardinality Representation

- **Integer quasi affine expression**
  
  \[ [x/2] + 3N \]

  \( \Rightarrow \) Presburger term

  That is, a term constructed from variables, symbolic constants, integer constants, addition (+), subtraction (−) and integer division by a constant (\([\cdot]/d\])

- **Rational polynomial expression**

  \[ x^2 - N/2 \]

  \( \Rightarrow \) a term constructed from variables, symbolic constants, rational constants, addition (+), subtraction (−) and multiplication (\(\cdot\))

- **Quasi polynomial expression**

  \[ ([x/2] + 3N)^2 - N/2 \]

  \( \Rightarrow \) a rational polynomial expression with variables replaced by integer quasi affine expressions

- **Piecewise quasi affine/polynomial expression**

  \( \Rightarrow \) a list of pairs of pairs of Presburger sets and quasi affine/polynomial expressions \( E = (S_i, e_i)_i \), with \( S_i \) disjoint

  \[ E(j) = \begin{cases} 
  e_i(j) & \text{if } j \in S_i \\
  \bot / 0 & \text{otherwise} 
  \end{cases} \]
Cardinality Representation

- **Integer quasi affine expression**
  \[\left\lfloor \frac{x}{2} \right\rfloor + 3N\]
  ⇒ Presburger term
  That is, a term constructed from variables, symbolic constants, integer constants, addition (+), subtraction (−) and integer division by a constant (\(\lfloor \cdot / d \rfloor\))

- **Rational polynomial expression**
  \[x^2 - \frac{N}{2}\]
  ⇒ a term constructed from variables, symbolic constants, rational constants, addition (+), subtraction (−) and multiplication (\(\cdot\))

- **Quasi polynomial expression**
  \[(\left\lfloor \frac{x}{2} \right\rfloor + 3N)^2 - \frac{N}{2}\]
  ⇒ a rational polynomial expression with variables replaced by integer quasi affine expressions

- **Piecewise quasi affine/polynomial expression**
  ⇒ a list of pairs of pairs of Presburger sets and quasi affine/polynomial expressions \(E = (S_i, e_i)_i\), with \(S_i\) disjoint

\[E(j) = \begin{cases} 
eq j(S_i) & \text{if } j \in S_i \\ \bot/0 & \text{otherwise} \end{cases}\]

Note: in practice, cardinality result does not contain nested integer divisions
Basic Operations on Piecewise Expressions

- Piecewise (rational) quasi affine expressions
  - addition (+)
  - subtraction (−)
  - negation (−)
  - minimum (min), maximum (max)
  - multiplication by constant (⋯d)
  - division by constant (⋯d)
  - remainder on integer division by constant (mod d)
  - floor (⋯)
  - ceiling (⋯)
Basic Operations on Piecewise Expressions

- Piecewise (rational) quasi affine expressions
  - addition (+)
  - subtraction (−)
  - negation (−)
  - minimum (min), maximum (max)
  - multiplication by constant (·\(d\))
  - division by constant (\(/d\))
  - remainder on integer division by constant (mod\(d\))
  - floor (\(\lfloor \cdot \rfloor\))
  - ceiling (\(\lceil \cdot \rceil\))

- Piecewise quasi polynomial expressions
  - addition (+)
  - subtraction (−)
  - negation (−)
  - multiplication (·)
  - exponentiation by positive integer constant (\(\cdot^d\))
Bounds on Piecewise Quasi Polynomials

\[ m(N) = \max_{(x,y):x,y \geq 0 \land x+y \leq N} 4+x+y-(x-2)^2 \]

**Question**
Can exact maximum be computed in general?
Bounds on Piecewise Quasi Polynomials

\[ m(N) = \max_{(x, y): x, y \geq 0 \land x + y \leq N} 4 + x + y - (x - 2)^2 \]

**Question**

Can exact maximum be computed in general?

Upper bound \( u(N) \geq m(N) \) can be computed.
Bounds on Piecewise Quasi Polynomials

\[ m(N) = \max_{(x,y):x,y \geq 0 \land x+y \leq N} 4 + x + y - (x-2)^2 \]

**Question**

Can exact maximum be computed in general?

Upper bound \( u(N) \geq m(N) \) can be computed
Bounds on Piecewise Quasi Polynomials

\[ 4 + x + y - (x - 2)^2 \]

\[ m(N) = \max_{(x,y): x, y \geq 0 \land x + y \leq N} 4 + x + y - (x - 2)^2 \]

**Question**

Can exact maximum be computed in general?

Upper bound \( u(N) \geq m(N) \) can be computed
Bounds on Piecewise Quasi Polynomials

\[ m(N) = \max_{(x,y): x,y \geq 0 \land x+y \leq N} 4+x+y-(x-2)^2 \]

**Question**

Can exact maximum be computed in general?

Upper bound \( u(N) \geq m(N) \) can be computed
Bounds on Piecewise Quasi Polynomials

\[ m(N) = \max_{(x,y):x,y \geq 0 \land x+y \leq N} 4 + x + y - (x-2)^2 \]

Question

Can exact maximum be computed in general?

Upper bound \( u(N) \geq m(N) \) can be computed
Bounds on Piecewise Quasi Polynomials

\[ m(N) = \max_{(x,y): x,y \geq 0 \land x+y \leq N} (4+x+y-(x-2)^2) \leq u(N) = \max(3N, 5N - N^2) \]

**Question**

Can exact maximum be computed in general?

Upper bound \( u(N) \geq m(N) \) can be computed.
Bounds on Piecewise Quasi Polynomials — Example

```c
for (i = 0; i < N; ++i)
    for (j = i; j < N; ++j) {
        p = malloc(i * j + i - N + 1);
        /* ... */
        free(p);
    }
```

How much memory is needed?
Bounds on Piecewise Quasi Polynomials — Example

for (i = 0; i < N; ++i)
    for (j = i; j < N; ++j) {
        p = malloc(i * j + i - N + 1);
        /* ... */
        free(p);
    }

How much memory is needed?

\[
\text{ub} \begin{cases}
    ij + i - N + 1 & \text{if } 0 \leq i < N \land i \leq j < N
\end{cases}
\]
Bounds on Piecewise Quasi Polynomials — Example

for (i = 0; i < N; ++i)
    for (j = i; j < N; ++j) {
        p = malloc(i * j + i - N + 1);
        /* ... */
        free(p);
    }

How much memory is needed?

$$\text{ub} \begin{cases} i j + i - N + 1 & \text{if } 0 \leq i < N \land i \leq j < N \\ \end{cases}$$

Result:

$$\begin{cases} \max(1 - 2N + N^2) & \text{if } N \geq 1 \\ \end{cases}$$

(exact maximum)
Maximal Number of Live Memory elements

- Assume each statement instance writes to at most one array element
  - Each live element can be identified by write instance
- Compute dataflow relation $D$
- For each write instance compute last read
  $L = O^{-1} \circ \text{lexmax}(O \circ D)$
- For each statement instance $i$, count write instances that precede $i$ such that corresponding last read follows $i$
  - Number of live elements at $i$
  $N = \text{card} \left( ((O \triangleright O) \cap_{\text{ran}} (\text{dom } L)) \cap (L^{-1} \circ (O \preceq O)) \right)$
- Compute upper bound
  $U = \text{ub } N$
Maximal Number of Live Memory elements — Example

for (i = 0; i < N; ++i)
S1: \( t[i] = f(a[i]) \);
for (i = 0; i < N; ++i)
S2: \( b[i] = g(t[N-i-1]) \);

\[ I = \{ S1(i) : 0 \leq i < N; S2(i) : 0 \leq i < N \} \]

\[ O = \{ S1(i) \rightarrow (0, i); S2(i) \rightarrow (1, i) \} \]

\[ D = \{ S1(i) \rightarrow S2(N - 1 - i) : 0 \leq i < N \} \]
Maximal Number of Live Memory elements — Example

```
for (i = 0; i < N; ++i)
S1:   t[i] = f(a[i]);
for (i = 0; i < N; ++i)
S2:   b[i] = g(t[N-i-1]);
```

\[
I = \{ S1(i) : 0 \leq i < N; S2(i) : 0 \leq i < N \}\\
O = \{ S1(i) \rightarrow (0, i); S2(i) \rightarrow (1, i) \}\\
D = \{ S1(i) \rightarrow S2(N - 1 - i) : 0 \leq i < N \}\\
L = O^{-1} \circ \text{lexmax}(O \circ D)\\
    = \{ S1(i) \rightarrow S2(N - 1 - i) : 0 \leq i < N \}\\
N = \text{card} \left( ((O \succ O) \cap_{\text{ran}} (\text{dom} \ L)) \cap (L^{-1} \circ (O \preceq O)) \right)\\
    = \{ S1(i) \rightarrow i : 1 \leq i < N; S2(i) \rightarrow N - i : 0 \leq i < N \}\\
U = \text{ub} \ N\\
    = \{ \max(N) : N \geq 1 \}
```
Weighted Counting

\[ G = F \circ R \]

with \( F \) a piecewise quasi polynomial and \( R \) a Presburger relation is a piecewise quasi polynomial \( G \) such that

\[ G(i) = \sum_{j: R(i,j)} F(j) \]
Weighted Counting

\[ G = F \circ R = \left\{ (x, y) \rightarrow \frac{x^2 + y^2}{4} : 1 \leq x, y \leq 2 \right\} \circ \left\{ (x) \rightarrow (x, y) \right\} \]

with \( F \) a piecewise quasi polynomial and \( R \) a Presburger relation is a piecewise quasi polynomial \( G \) such that

\[ G(i) = \sum_{j: R(i,j)} F(j) \]
Weighted Counting

\[ G = F \circ R = \left\{ (x, y) \rightarrow \frac{x^2 + y^2}{4} : 1 \leq x, y \leq 2 \right\} \circ \left\{ (x) \rightarrow (x, y) \right\} \]

with \( F \) a piecewise quasi polynomial and \( R \) a Presburger relation is a piecewise quasi polynomial \( G \) such that

\[ G(i) = \sum_{j : R(i,j)} F(j) \]
Weighted Counting

\[ G = F \circ R = \left\{ (x, y) \mapsto \frac{x^2 + y^2}{4} : 1 \leq x, y \leq 2 \right\} \circ \left\{ (x) \mapsto (x, y) \right\} \]

\[ = \left\{ (x) \mapsto \frac{5 + 2x^2}{2} : 1 \leq x \leq 2 \right\} \]

with \( F \) a piecewise quasi polynomial and \( R \) a Presburger relation is a piecewise quasi polynomial \( G \) such that

\[ G(i) = \sum_{j : R(i, j)} F(j) \]
Compositions with Piecewise (Folds of) Quasi polynomials

\[ F \circ R \]

- \( R: D_1 \rightarrow D_2 \) is a Presburger relation

- \( F: D_2 \rightarrow \mathbb{Q} \) may be
  - piecewise quasi polynomial
    (result of counting problem)
    \[ \Rightarrow \text{take sum over} \ (\text{ran} \ R) \cap (\text{dom} \ F) \]
  - piecewise fold of quasi polynomials
    (result of upper bound computation)
    \[ \Rightarrow \text{compute bound over} \ (\text{ran} \ R) \cap (\text{dom} \ F) \]

- \( (F \circ R): D_1 \rightarrow \mathbb{Q} \) of same type as \( F \)

if \( R \) is single-valued, then sum/bound is computed over a single point
Compositions with Piecewise (Folds of) Quasi polynomials

\[ F \circ R \quad \text{or} \quad F(S) \]

- \( R: D_1 \to D_2 \) is a Presburger relation
- \( S \subseteq D_2 \) is a Presburger set
- \( F: D_2 \to \mathbb{Q} \) may be
  - piecewise quasi polynomial
    (result of counting problem)
    \[ \Rightarrow \text{take sum over } (\text{ran } R) \cap (\text{dom } F) \quad \text{or} \quad S \cap (\text{dom } F) \]
  - piecewise fold of quasi polynomials
    (result of upper bound computation)
    \[ \Rightarrow \text{compute bound over } (\text{ran } R) \cap (\text{dom } F) \quad \text{or} \quad S \cap (\text{dom } F) \]
- \((F \circ R): D_1 \to \mathbb{Q}\) of same type as \( F \)
- \( F(S): \mathbb{Q} \) of same type as \( F \)

if \( R \) is single-valued, then sum/bound is computed over a single point
Example: Total Memory Allocation

```c
for (i = 0; i < N; ++i)
    for (j = i; j < N; ++j)
        p[i][j] = malloc(i * j + i - N + 1);
/* ... */
for (i = 0; i < N; ++i)
    for (j = i; j < N; ++j)
        free(p[i][j]);
```

How much memory allocated in total?
Example: Total Memory Allocation

```c
for (i = 0; i < N; ++i)
    for (j = i; j < N; ++j)
        p[i][j] = malloc(i * j + i - N + 1);
/* ... */
for (i = 0; i < N; ++i)
    for (j = i; j < N; ++j)
        free(p[i][j]);
```

How much memory allocated in total?

\[
F = \{ (i,j) \rightarrow ij + i - N + 1 \} \\
I = \{ (i,j) : 0 \leq i < N \land i \leq j < N \} \\
F(I) = \left\{ \frac{5}{12}N - \frac{1}{8}N^2 - \frac{5}{12}N^3 + \frac{1}{8}N^4 : N \geq 1 \right\}
\]
Dynamic Memory Requirement Estimation

How much memory is needed to execute the following program?

```java
void m0(int m) {
    for (c = 0; c < m; c++) {
        m1(c);         /* S1*/
        B[] m2Arr = m2(2*m-c); /* S2*/
    }
}

void m1(int k) {
    for (i = 1; i <= k; i++) {
        A a = new A();    /* S3*/
        B[] dummyArr = m2(i); /* S4*/
    }
}

B[] m2(int n) {
    B[] arrB = new B[n];    /* S5*/
    for (j = 1; j <= n; j++)
        B b = new B();    /* S6*/
    return arrB;
}
```
Dynamic Memory Requirement Estimation

How much memory is needed to execute the following program?

```c
void m0(int m) {
    for (c = 0; c < m; c++) {
        m1(c); /* S1 */
        B[] m2Arr = m2(2*m-c); /* S2 */
    }
}

void m1(int k) {
    for (i = 1; i <= k; i++) {
        A a = new A(); /* S3 */
        B[] dummyArr = m2(i); /* S4 */
    }
}

B[] m2(int n) {
    B[] arrB = new B[n]; /* S5 */
    for (j = 1; j <= n; j++)
        B b = new B(); /* S6 */
    return arrB;
}
```

I = \{ m0(m) \rightarrow S1(c) : 0 \leq c < m; \\
        m0(m) \rightarrow S2(c) : 0 \leq c < m; \\
        m1(k) \rightarrow S3(i) : 1 \leq i \leq k; \\
        m1(k) \rightarrow S4(i) : 1 \leq i \leq k; \\
        m2(n) \rightarrow S5(); \\
        m2(n) \rightarrow S6(j) : 1 \leq j \leq n \}
Dynamic Memory Requirement Estimation

How much (scoped) memory is needed?

⇒ compute for each method

\[ \text{ret}_m \] size of memory returned by \( m \)
\[ \text{cap}_m \] size of memory “captured” (not returned) by \( m \)
\[ \text{memRq}_m \] total memory requirements of \( m \)

\[
\text{ret}_m + \text{cap}_m = \sum_{p \text{ called by } m} \text{ret}_p
\]

\[
\text{memRq}_m = \text{cap}_m + \max_{p \text{ called by } m} \text{memRq}_p
\]
Dynamic Memory Requirement Estimation

How much (scoped) memory is needed?

⇒ compute for each method

\[ \text{ret}_m \] size of memory returned by \( m \)

\[ \text{cap}_m \] size of memory “captured” (not returned) by \( m \)

\[ \text{memRq}_m \] total memory requirements of \( m \)

\[
\text{ret}_m + \text{cap}_m = \sum_{p \text{ called by } m} \text{ret}_p
\]

\[
\text{memRq}_m = \text{cap}_m + \max_{p \text{ called by } m} \text{memRq}_p
\]

⇒ summarize over iteration domain, i.e., compose with \( M = (\text{dom } l)^{-1} \)

\[
M = \{ m0(m) \rightarrow (m0(m) \rightarrow S1(c)) : 0 \leq c < m; \]

\[
m0(m) \rightarrow (m0(m) \rightarrow S2(c)) : 0 \leq c < m; \]

\[
m1(k) \rightarrow (m1(k) \rightarrow S3(i)) : 1 \leq i \leq k; \]

\[
m1(k) \rightarrow (m1(k) \rightarrow S4(i)) : 1 \leq i \leq k; \]

\[
m2(n) \rightarrow (m2(n) \rightarrow S5()); m2(n) \rightarrow (m2(n) \rightarrow S6(j)) : 1 \leq j \leq n \} \]
Dynamic Memory Requirement Estimation

\[ ret_m + cap_m = \sum_{p \text{ called by } m} ret_p \]

\[ memRq_m = cap_m + \max_{p \text{ called by } m} memRq_p \]
Dynamic Memory Requirement Estimation

\[ \text{ret}_m + \text{cap}_m = \sum_{p \text{ called by } m} \text{ret}_p \]

\[ \text{memRq}_m = \text{cap}_m + \max_{p \text{ called by } m} \text{memRq}_p \]

```java
B[] m2(int n) {
    B[] arrB = new B[n];    // S5
    for (j = 1; j <= n; j++)
        B b = new B();        // S6
    return arrB;
}
```
Dynamic Memory Requirement Estimation

\[ \text{ret}_m + \text{cap}_m = \sum_{p \text{ called by } m} \text{ret}_p \]

\[ \text{memRq}_m = \text{cap}_m + \max_{p \text{ called by } m} \text{memRq}_p \]

```c
B[] m2(int n) {
    B[] arrB = new B[n]; /* S5 */
    for (j = 1; j <= n; j++)
        B b = new B(); /* S6 */
    return arrB;
}
```

\[ \text{ret}_{m2} = \{ (m2(n) \rightarrow S5()) \rightarrow n : n \geq 0 \} \circ M \]
\[ \text{cap}_{m2} = \{ (m2(n) \rightarrow S6(j)) \rightarrow 1 \} \circ M \]
\[ \text{memRq}_{m2} = \text{cap}_{m2} + \{ m2(n) \rightarrow \max(0) \} \]
Dynamic Memory Requirement Estimation

\[ \text{ret}_m + \text{cap}_m = \sum_{p \text{ called by } m} \text{ret}_p \]

\[ \text{memRq}_m = \text{cap}_m + \max_{p \text{ called by } m} \text{memRq}_p \]

```c
B[ ] m2(int n) {
    B[ ] arrB = new B[n]; /*S5*/
    for (j = 1; j <= n; j++)
        B b = new B(); /*S6*/
    return arrB;
}
```

\[ \text{ret}_{m2} = \{ (m2(n) \rightarrow S5()) \rightarrow n : n \geq 0 \} \circ M = \{ m2(n) \rightarrow n : n \geq 0 \} \]

\[ \text{cap}_{m2} = \{ (m2(n) \rightarrow S6(j)) \rightarrow 1 \} \circ M = \{ m2(n) \rightarrow n : n \geq 1 \} \]

\[ \text{memRq}_{m2} = \text{cap}_{m2} + \{ m2(n) \rightarrow \max(0) \} = \{ m2(n) \rightarrow \max(n) : n \geq 1 \} \]
Dynamic Memory Requirement Estimation

```c
void m1(int k) {
    for (i = 1; i <= k; i++) {
        A a = new A();  /* S3 */
        B[] dummyArr = m2(i);  /* S4 */
    }
}
```

\[ \text{cap}_{m1}(k) = \sum_{1 \leq i \leq k} (1 + \text{ret}_{m2}(i)) \]

\text{ret}_{m2} is a function of the arguments of \text{m2}

We want to use it as a function of the arguments and local variables of \text{m1}
Dynamic Memory Requirement Estimation

```c
void m1(int k) {
    for (i = 1; i <= k; i++) {
        A a = new A(); /* S3 */
        B[] dummyArr = m2(i); /* S4 */
    }
}
```

cap m1(k) = \[ \sum_{1\leq i\leq k} (1 + \text{ret}_{m2}(i)) \]

ret\_m2 is a function of the arguments of m2
We want to use it as a function of the arguments and local variables of m1
⇒ define parameter binding
Dynamic Memory Requirement Estimation

How much memory is needed to execute the following program?

```java
void m0(int m) {
    for (c = 0; c < m; c++) {
        m1(c); /* S1 */
        B[] m2Arr = m2(2*m-c); /* S2 */
    }
}

void m1(int k) {
    for (i = 1; i <= k; i++) {
        A a = new A(); /* S3 */
        B[] dummyArr = m2(i); /* S4 */
    }
}

B[] m2(int n) {
    B[] arrB = new B[n]; /* S5 */
    for (j = 1; j <= n; j++)
        B b = new B(); /* S6 */
    return arrB;
}
```

\[
I = \{m0(m) \rightarrow S1(c) : 0 \leq c < m; \\
m0(m) \rightarrow S2(c) : 0 \leq c < m; \\
m1(k) \rightarrow S3(i) : 1 \leq i \leq k; \\
m1(k) \rightarrow S4(i) : 1 \leq i \leq k; \\
m2(n) \rightarrow S5(); \\
m2(n) \rightarrow S6(j) : 1 \leq j \leq n \}
\]
How much memory is needed to execute the following program?

```java
void m0(int m) {
    for (c = 0; c < m; c++) {
        m1(c); /* S1 */
        B[] m2Arr = m2(2*m-c); /* S2 */
    }
}

void m1(int k) {
    for (i = 1; i <= k; i++) {
        A a = new A(); /* S3 */
        B[] dummyArr = m2(i); /* S4 */
    }
}

B[] m2(int n) {
    B[] arrB = new B[n]; /* S5 */
    for (j = 1; j <= n; j++)
        B b = new B(); /* S6 */
    return arrB;
}
```

$I = \{ m0(m) \rightarrow S1(c) : 0 \leq c < m; \}
\{ m0(m) \rightarrow S2(c) : 0 \leq c < m; \}
\{ m1(k) \rightarrow S3(i) : 1 \leq i \leq k; \}
\{ m1(k) \rightarrow S4(i) : 1 \leq i \leq k; \}
\{ m2(n) \rightarrow S5(); \}
\{ m2(n) \rightarrow S6(j) : 1 \leq j \leq n \}$

$B^{m0} = \{ (m0(m) \rightarrow S1(c)) \rightarrow m1(c); \}
(m0(m) \rightarrow S2(c)) \rightarrow m2(2m - c) \}$

$B^{m1} = \{ (m1(k) \rightarrow S4(i)) \rightarrow m2(i) \}$
Dynamic Memory Requirement Estimation

```c
void m1(int k) {
    for (i = 1; i <= k; i++) {
        A a = new A(); /* S3 */
        B[] dummyArr = m2(i); /* S4 */
    }
}
```

\[
\text{cap}_{m1}(k) = \sum_{1 \leq i \leq k} (1 + \text{ret}_{m2}(i))
\]

\text{ret}_{m2} \text{ is a function of the arguments of } m2
\text{We want to use it as a function of the arguments and local variables of } m1
\Rightarrow \text{ define parameter binding}
Dynamic Memory Requirement Estimation

```java
void m1(int k) {
    for (i = 1; i <= k; i++) {
        A a = new A(); /* S3 */
        B[] dummyArr = m2(i); /* S4 */
    }
}
```

\[
cap_{m1}(k) = \sum_{1 \leq i \leq k} (1 + \text{ret}_{m2}(i))
\]

\[
\text{ret}_{m1} = \{ m1(k) \rightarrow 0 \}
\]

\[
cap_{m1} = (\{ (m1(k) \rightarrow S3(i)) \rightarrow 1 \} + \text{ret}_{m2} \circ B^{m1}) \circ M
\]

\[
\text{memRq}_{m1} = cap_{m1} + (\text{memRq}_{m2} \circ B^{m1} \circ M)
\]
Dynamic Memory Requirement Estimation

```java
void m1(int k) {
    for (i = 1; i <= k; i++) {
        A a = new A(); /* S3 */
        B[] dummyArr = m2(i); /* S4 */
    }
}
```

\[
cap_{m1}(k) = \sum_{1\leq i \leq k} (1 + \text{ret}_{m2}(i))
\]

\[
\text{ret}_{m1} = \{ \text{m1}(k) \rightarrow 0 \}
\]

\[
cap_{m1} = (\{ (\text{m1}(k) \rightarrow \text{S3}(i)) \rightarrow 1 \} + \text{ret}_{m2} \circ B^{m1} \circ M)
\]

\[
= \left\{ \text{m1}(k) \rightarrow \frac{3}{2} k + \frac{1}{2} k^2 : k \geq 1 \right\}
\]

\[
\text{memRq}_{m1} = \text{cap}_{m1} + (\text{memRq}_{m2} \circ B^{m1} \circ M)
\]

\[
= \left\{ \text{m1}(k) \rightarrow \max \left( \frac{5}{2} k + \frac{1}{2} k^2 \right) : k \geq 1 \right\}
\]
Dynamic Memory Requirement Estimation

```c
void m0(int m) {
    for (c = 0; c < m; c++) {
        m1(c);
        /* S1 */
        B[ ] m2Arr = m2(2 * m - c); /* S2 */
    }
}
```

\[ B^{m0} = \{ (m0(k) \rightarrow S1(c)) \rightarrow m1(c); (m0(k) \rightarrow S2(c)) \rightarrow m2(2m - c) \} \]

\[ \text{ret}_{m0} = \{ m0(m) \rightarrow 0 \} \]

\[ \text{cap}_{m0} = (\text{ret}_{m1} + \text{ret}_{m2}) \circ B^{m0} \circ M \]

\[ \text{memRq}_{m0} = \text{cap}_{m0} + ((\text{memRq}_{m1} + \text{memRq}_{m2}) \circ B^{m0} \circ M) \]
Dynamic Memory Requirement Estimation

```c
void m0(int m) {
    for (c = 0; c < m; c++) {
        m1(c); /* S1 */
        B[] m2Arr = m2(2 * m - c); /* S2 */
    }
}
```

\[ B^{m_0} = \{(m0(k) \to S1(c)) \to m1(c); (m0(k) \to S2(c)) \to m2(2m - c)\} \]

\[ \text{ret}_{m_0} = \{m0(m) \to 0\} \]

\[ \text{cap}_{m_0} = (\text{ret}_{m1} + \text{ret}_{m2}) \circ B^{m_0} \circ M \]

\[ = \left\{ \begin{array}{l}
        m0(m) \to \frac{1}{2} m + \frac{3}{2} m^2 : m \geq 1
    \end{array} \right\} \]

\[ \text{memRq}_{m_0} = \text{cap}_{m_0} + ((\text{memRq}_{m1} + \text{memRq}_{m2}) \circ B^{m_0} \circ M) \]

\[ = \left\{ \begin{array}{l}
        m0(m) \to \max \left( -2 + 2m + 2m^2, \frac{5}{2} m + \frac{3}{2} m^2 \right) : m \geq 1
    \end{array} \right\} \]
Outline

1. Polyhedral Model
   • Introduction
   • Representation

2. Polyhedral Transformation
   • Schedules
   • AST Generation

3. Dependences
   • Schedule Validity
   • Dependences
   • Structures

4. Dataflow
   • Parallelism
   • Dataflow
   • Approximate Dataflow
   • Run-time dependent Dataflow
   • Reductions

5. Aliasing

6. Counting
   • Cardinality
   • Bounds
   • Weighted Counting
   • Dynamic Memory Requirement

7. Transitive Closures
Positive Powers

Definition (Power of a Relation)

Let $R$ be a Presburger relation and $k$ a positive integer, then power $k$ of relation $R$ is defined as

$$R^k := \begin{cases} 
R & \text{if } k = 1 \\
R \circ R^{k-1} & \text{if } k \geq 2.
\end{cases}$$
Positive Powers

Definition (Power of a Relation)
Let $R$ be a Presburger relation and $k$ a positive integer, then power $k$ of relation $R$ is defined as

$$R^k := \begin{cases} R & \text{if } k = 1 \\ R \circ R^{k-1} & \text{if } k \geq 2. \end{cases}$$

Example

$$R = \{ (x) \rightarrow (x + 1) \}$$
$$R^k = \{ (x) \rightarrow (x + k) : k \geq 1 \}$$
Definition (Transitive Closure of a Relation)

Let $R$ be a Presburger relation, then the transitive closure $R^+$ of $R$ is the union of all positive powers of $R$,

$$R^+ := \bigcup_{k \geq 1} R^k.$$
Transitive Closures

**Definition (Transitive Closure of a Relation)**

Let $R$ be a Presburger relation, then the transitive closure $R^+$ of $R$ is the union of all positive powers of $R$,

$$R^+ := \bigcup_{k \geq 1} R^k.$$

**Example**

$$R = \{ (x) \rightarrow (x + 1) \}$$

$$R^k = \{ (x) \rightarrow (x + k) : k \geq 1 \}$$

$$R^+ = \{ (x) \rightarrow (y) : \exists k \geq 1 : y = x + k \} = \{ (x) \rightarrow (y) : y \geq x + 1 \}$$
Transitive Closures

Definition (Transitive Closure of a Relation)

Let $R$ be a Presburger relation, then the transitive closure $R^+$ of $R$ is the union of all positive powers of $R$,

$$R^+ := \bigcup_{k \geq 1} R^k.$$

Example

$$R = \{ (x) \rightarrow (x + 1) \}$$

$$R^k = \{ (x) \rightarrow (x + k) : k \geq 1 \}$$

$$R^+ = \{ (x) \rightarrow (y) : \exists k \geq 1 : y = x + k \} = \{ (x) \rightarrow (y) : y \geq x + 1 \}$$

Definition (Transitive Closure of a Relation, Alternative)

Inductive definition:

$$R^+ := R \cup (R \circ R^+)$$
Transitive Closures — Approximation

Fact

Given a Presburger relation $R$, the power $R^k$ (with $k$ a parameter) and the transitive closure $R^+$ may not be Presburger relations.

Example

$$R = \{ (x) \to (2x) \}$$

$$R^k = \{ (x) \to (2^k x) \}$$
Transitive Closures — Approximation

Fact

Given a Presburger relation $R$, the power $R^k$ (with $k$ a parameter) and the transitive closure $R^+$ may not be Presburger relations.

Example

\[
R = \{(x) \rightarrow (2x)\}
\]
\[
R^k = \{(x) \rightarrow (2^k x)\}
\]

⇒ need for approximation

▶ overapproximation $R^+$
▶ underapproximation $R^-$
Transitive Closures — Approximation

Fact

Given a Presburger relation $R$, the power $R^k$ (with $k$ a parameter) and the transitive closure $R^+$ may not be Presburger relations.

Example

$$R = \{ (x) \rightarrow (2x) \}$$
$$R^k = \{ (x) \rightarrow (2^k x) \}$$

$\Rightarrow$ need for approximation
- overapproximation $R^{\overline{+}}$
- underapproximation $R^{\underline{+}}$

Note

Do not use transitive closures if there is an alternative.
Part II

Tools
Outline

8 Tools
Availability — Representation

\[ \{ A(i) : 0 \leq i \leq n; B(i, j) : \exists \alpha : i = 2\alpha \} \]

- Named (and nested) spaces: isl

  \[ [n] \rightarrow \{ A[i] : 0 \leq i \leq n; B[i,j] : \text{exists } a : i = 2a \} \]

In omega(+):
Availability — Representation

\{ A(i) \colon 0 \leq i \leq n; B(i, j) \colon \exists \alpha \colon i = 2\alpha \} \\

- Named (and nested) spaces: isl

\[[n] \rightarrow \{ A[i] \colon 0 \leq i \leq n; B[i,j] \colon \text{exists } a \colon i = 2 a \}\]

In \(\omega(+)\):

- symbolic \(n\);
- \{ [0, i, 0] \colon 0 \leq i \leq n \} \cup \{ [1, i,j] \colon \text{exists } a \colon i = 2 a \}
Availability — Representation

\{ A(i) : 0 \leq i \leq n; B(i,j) : \exists \alpha : i = 2\alpha \}\n
- Named (and nested) spaces: isl
  \[ n \rightarrow \{ A[i] : 0 \leq i \leq n; B[i,j] : \exists \alpha : i = 2\alpha \} \]

In omega(+):

\[
\text{symbolic } n;
\{ [0, i, 0] : 0 \leq i \leq n \} \cup \{ [1, i, j] : \exists \alpha : i = 2\alpha \}
\]

- Presburger sets and relations: isl, omega(+)

\[ \text{padding} \]
Availability — Representation

\{ A(i) : 0 \leq i \leq n; B(i, j) : \exists \alpha : i = 2\alpha \} 

- Named (and nested) spaces: isl
  
  \[ [n] \to \{ A[i] : 0 \leq i \leq n; B[i,j] : \exists a : i = 2a \} \]

  In omega(+):
  \[ \{ [0, i, 0] : 0 \leq i \leq n \} \cup \{ [1, i,j] : \exists a : i = 2a \} \]

- Presburger sets and relations: isl, omega(+)

  In PolyLib:
  
  \[
  \begin{array}{cccccccc}
  2 & 4 & 6 & 0 & -1 & 0 & 0 & 0 & 0 \\
  0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\
  1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
  1 & 0 & -1 & 0 & 1 & 0 & 0 & 0 & 0 \\
  2 & 7 & 0 & -1 & 0 & 0 & 0 & 0 & -1 \\
  0 & 0 & -1 & 0 & 2 & 0 & 0 & 0 & 0 \\
  \end{array}
  \]

  Moreover: PolyLib deals with rational sets (polyhedra)
Availability — Representation (2)

- Uninterpreted functions: \texttt{omega(+)}
  - \(\Rightarrow\) arity can be greater than 0
  - \(\Rightarrow\) not available in \texttt{isl} (yet)

Note: support in \texttt{omega(+)} for uninterpreted functions is very restrictive
- arguments need to be prefix of input/output dimensions
  - \(\Rightarrow\) essentially symbolic constants
## Availability — Operations

<table>
<thead>
<tr>
<th></th>
<th>union</th>
<th>intersection</th>
<th>difference</th>
<th>emptiness</th>
<th>application</th>
</tr>
</thead>
<tbody>
<tr>
<td>PolyLib</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>in $\mathbb{Z}$</td>
</tr>
<tr>
<td>PPL</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>PIP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>lexmin</td>
</tr>
<tr>
<td>omega(+)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>isl</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>barvinok</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>card</td>
</tr>
<tr>
<td>bernstein</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LattE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>card</td>
</tr>
</tbody>
</table>
## Availability — Operations (2)

<table>
<thead>
<tr>
<th></th>
<th>lexmin</th>
<th>cardinality</th>
<th>bounds on polynomials</th>
<th>weighted counting</th>
<th>transitive closure</th>
</tr>
</thead>
<tbody>
<tr>
<td>PolyLib</td>
<td>✔</td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PPL</td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PIP</td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>omega(+)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>“$R^\pm$”</td>
</tr>
<tr>
<td>Presburger</td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>isl</td>
<td>✔</td>
<td></td>
<td></td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>partial</td>
<td>✔</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bernstein</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LattE</td>
<td>✔</td>
<td></td>
<td></td>
<td>✔</td>
<td></td>
</tr>
</tbody>
</table>
isl: manipulates parametric affine sets and relations
barvinok: counts elements in parametric affine sets and relations
pet: extracts polyhedral model from clang AST
PPCG: Polyhedral Parallel Code Generator
iscc: interactive calculator
isa: prototype tool set including derivation of process networks and equivalence checker
Overview of isl

isl is a thread-safe C library for manipulating integer sets and relations

- bounded by affine constraints
- involving symbolic constants and
- existentially quantified variables

and quasi-affine and quasi-polynomial functions on such domains

Supported operations by core library include

- intersection
- union
- set difference
- integer projection
- coalescing
- closed convex hull
- sampling, scanning
- integer affine hull
- lexicographic optimization
- transitive closure (approx.)
- parametric vertex enumeration
- bounds on quasipolynomials

Polyhedral compilation library

- schedule trees
- dataflow analysis
- scheduling
- AST generation
PPCG

PPCG (http://ppcg.gforge.inria.fr/)

- Input: C code
- Output: CUDA or OpenCL code for GPGPUs
PPCG

PPCG (http://ppcg.gforge.inria.fr/)

- Input: C code
- Output: CUDA or OpenCL code for GPGPUs

Steps:
- extract polyhedral model from C code (pet)
- dependence analysis (isl)
- scheduling
  - expose parallelism and tiling opportunities (isl)
  - perform tiling (isl)
  - separate into parts mapped to host, GPU blocks and GPU threads
- memory management
  - add transfers of data to/from GPU
  - detect array reference groups
  - allocate groups to registers and shared memory
- generate AST (isl)
PPCG Example — Input

```c
void matmul(int M, int N, int K,
    float A[static const restrict M][K],
    float B[static const restrict K][N],
    float C[static const restrict M][N])
{
    for (int i = 0; i < M; i++)
        for (int j = 0; j < N; j++) {
            S1: C[i][j] = 0;
            for (int k = 0; k < K; k++)
                S2: C[i][j] = C[i][j] + A[i][k] * B[k][j];
        }
}

Options:
--ctx="[M,N,K] -> { : M = N = K = 256 }
--sizes="{ kernel[i] -> tile[16,16,16];
                kernel[i] -> block[8,16] }"
--pet-autodetect
```
PPCG Example — Output

```c
long b0 = blockIdx.y, b1 = blockIdx.x;
long t0 = threadIdx.y, t1 = threadIdx.x;
__shared__ float s_A[16][16];
float p_C[2][1];
__shared__ float s_B[16][16];

for (long g9 = 0; g9 <= 15; g9 += 1) {
    for (long c0 = t0; c0 <= 15; c0 += 8)
        s_B[c0][t1] = B[(16 * g9 + c0) * (256) + 16 * b1 + t1];
    for (long c0 = t0; c0 <= 15; c0 += 8)
        s_A[c0][t1] = A[(16 * b0 + c0) * (256) + t1 + 16 * g9];
    __syncthreads();
    if (g9 == 0) {
        p_C[0][0] = (0);
        p_C[1][0] = (0);
    }
    for (long c3 = 0; c3 <= 15; c3 += 1) {
        p_C[0][0] = (p_C[0][0] + (s_A[t0][c3] * s_B[c3][t1]));
        p_C[1][0] = (p_C[1][0] + (s_A[t0 + 8][c3] * s_B[c3][t1]));
    }
    __syncthreads();
}
C[(16 * b0 + t0) * (256) + 16 * b1 + t1] = p_C[0][0];
C[(16 * b0 + t0 + 8) * (256) + 16 * b1 + t1] = p_C[1][0];
```
CARP Project

Design tools and techniques to aid Correct and Efficient Accelerator Programming

Key areas:
- High level programming models
- Advanced compilation techniques
- Formal verification

Partners:
- Imperial College London (UK)
- ENS (FR)
- ARM (UK)
- Realeyes (ES)
- RWTH Aachen University (DE)
- Monoidics (UK)
- University of Twente (NL)
- Rightware (FI)
CARP Approach

Domain Specific Languages

DSL -> PENCIL compilers

PENCIL – Platform Neutral Compute Intermediate Language

Performance metadata

Polyhedral compilation

Direct OpenCL programming

OpenCL

Auto tuning

Widely supported industry standard

NVIDIA GPUs

AMD GPUs

ARM GPUs

Other accelerators

Formal verification
References I


References II


References III


References IV

The polyhedron model.

Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies.

Polly - performing polyhedral optimizations on a low-level intermediate representation.
Parallel Processing Letters, 22(04), 2012.
References V


References VI

Parameterized polyhedra and their vertices.

Lazy array data-flow dependence analysis.

isl: An integer set library for the polyhedral model.
References VII

Polyhedral process networks. 

Counting affine calculator and applications. 
In First International Workshop on Polyhedral Compilation Techniques (IMPACT’11), Chamonix, France, Apr. 2011.

Polyhedral parallel code generation for CUDA. 
References VIII


References IX

