# *iscc* Tutorial

Sven Verdoolaege

INRIA, France and KU Leuven
Sven.Verdoolaege@inria.fr

December 19, 2014

# Outline

# Outline

# Introduction

- What is `iscc`?
  - ⇒ interactive interface to the `barvinok` counting library
  - ⇒ also provides interface to the `pet` polyhedral model extractor and to some operations from the `isl` integer set library, including AST generation
  - ⇒ inspired by Omega Calculator from the Omega Project

# Introduction

- What is `iscc`?
  - ⇒ interactive interface to the `barvinok` counting library
  - ⇒ also provides interface to the `pet` polyhedral model extractor and to some operations from the `isl` integer set library, including AST generation
  - ⇒ inspired by Omega Calculator from the Omega Project
- Where to get `iscc`?
  - ⇒ currently distributed as part of `barvinok` package
  - ⇒ available from `http://barvinok.gforge.inria.fr/`

# Introduction

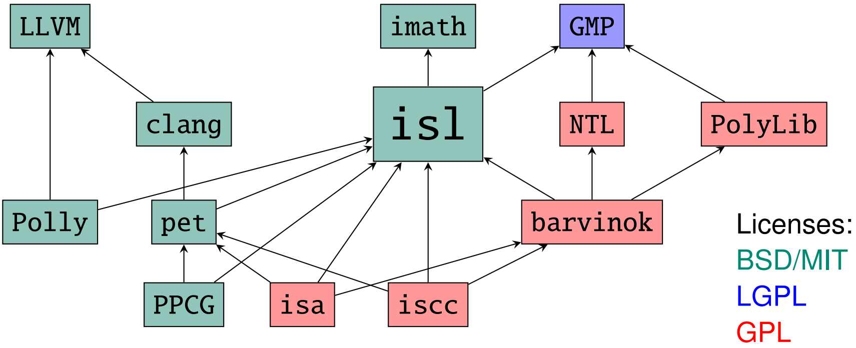
- What is `iscc`?
  - ⇒ interactive interface to the `barvinok` counting library
  - ⇒ also provides interface to the `pet` polyhedral model extractor and to some operations from the `isl` integer set library, including AST generation
  - ⇒ inspired by Omega Calculator from the Omega Project
- Where to get `iscc`?
  - ⇒ currently distributed as part of `barvinok` package
  - ⇒ available from `http://barvinok.gforge.inria.fr/`
- How to run `iscc`?
  - ⇒ compile and install `barvinok` following the instructions in `README`
  - ⇒ run `iscc`

    Note: `iscc` currently does not use readline, so you may want to use a readline front-end: `rlwrap iscc`

# Introduction

- What is `iscc`?
  - ⇒ interactive interface to the `barvinok` counting library
  - ⇒ also provides interface to the `pet` polyhedral model extractor and to some operations from the `isl` integer set library, including AST generation
  - ⇒ inspired by Omega Calculator from the Omega Project
- Where to get `iscc`?
  - ⇒ currently distributed as part of `barvinok` package
  - ⇒ available from `http://barvinok.gforge.inria.fr/`
- How to run `iscc`?
  - ⇒ compile and install `barvinok` following the instructions in `README`
  - ⇒ run `iscc`
    Note: `iscc` currently does not use readline, so you may want to use a readline front-end: `rlwrap iscc`

# Interaction with 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
`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*

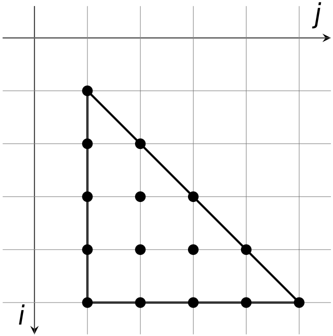
# Outline

# Statement Instance Set

```
for (i = 1; i <= 5; ++i)
    for (j = 1; j <= i; ++j)
        /* S */
```

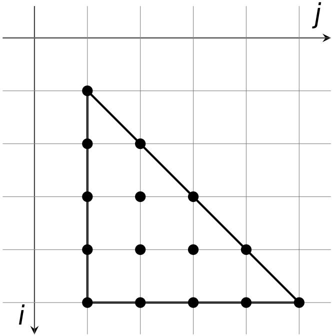
# Statement Instance Set

```
for (i = 1; i <= 5; ++i)
    for (j = 1; j <= i; ++j)
        /* S */
```



# Statement Instance Set

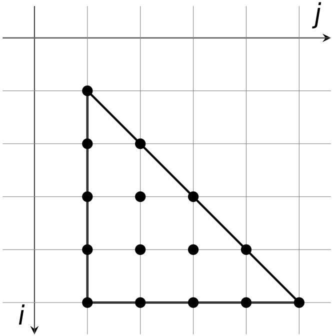
```
for (i = 1; i <= 5; ++i)
    for (j = 1; j <= i; ++j)
        /* S */
```




```
{ S[i,j] : 1 <= i <= 5 and 1 <= j <= i }
```

# Statement Instance Set

```
for (i = 1; i <= 5; ++i)
    for (j = 1; j <= i; ++j)
        /* S */
```

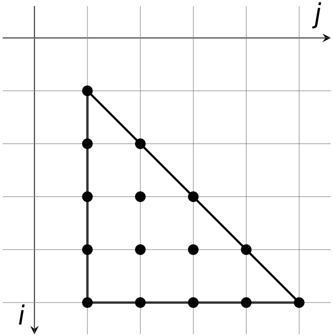



(optional) name of space

```
{ S[i,j] : 1 <= i <= 5 and 1 <= j <= i }
```

# Statement Instance Set

```
for (i = 1; i <= 5; ++i)
    for (j = 1; j <= i; ++j)
        /* S */
```



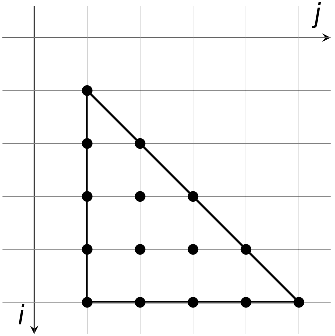

(optional) name of space

{ S[i,j] : 1 <= i <= 5 and 1 <= j <= i }

set variables

# Statement Instance Set

```
for (i = 1; i <= 5; ++i)
    for (j = 1; j <= i; ++j)
        /* S */
```




(optional) name of space

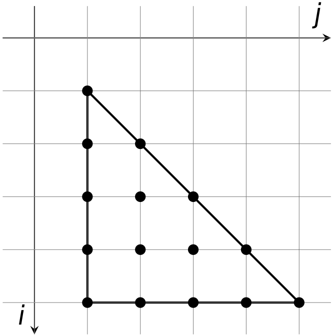
{ S[i,j] : 1 <= i <= 5 and 1 <= j <= i }

set variables

Presburger formula

# Statement Instance Set

```
for (i = 1; i <= n; ++i)
    for (j = 1; j <= i; ++j)
        /* S */
```



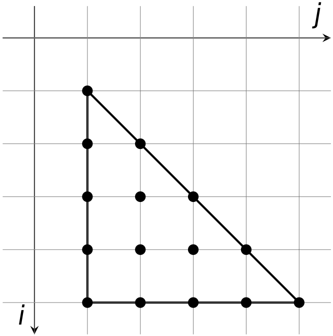

(optional) name of space

```
[n] -> { S[i,j] : 1 <= i <= n and 1 <= j <= i }
```

set variables Presburger formula

# Statement Instance Set

```
for (i = 1; i <= n; ++i)
    for (j = 1; j <= i; ++j)
        /* S */
```




(optional) name of space

```
[n] -> { S[i,j] : 1 <= i <= n and 1 <= j <= i }
```

symbolic constants set variables Presburger formula

# Set Variables and Symbolic Constants

- set variables
  - local to set
  - identified by position
- symbolic constants
  - global
  - identified by name

# Set Variables and Symbolic Constants

- set variables
  - local to set
  - identified by position
- symbolic constants
  - global
  - identified by name

```
[n] -> { [i,j] : 1 <= i <= n and 1 <= j <= i }
```

is equal to

```
[n] -> { [a,b] : 1 <= a <= n and 1 <= b <= a }
```

but not equal to

```
[n] -> { [j,i] : 1 <= i <= n and 1 <= j <= i }
```

or

```
[m] -> { [i,j] : 1 <= i <= m and 1 <= j <= i }
```

# AST Generation, Schedules and Maps

```
for (i = 1; i <= n; ++i)
    for (j = 1; j <= i; ++j)
        /* S */
```

```
codegen [n] -> { S[i,j] : 1 <= i <= n and 1 <= j <= i };
```

⇒ generate AST that visits elements in lexicographic order

# AST Generation, Schedules and Maps

```
for (i = 1; i <= n; ++i)
    for (j = 1; j <= i; ++j)
        /* S */
```

```
codegen [n] -> { S[i,j] : 1 <= i <= n and 1 <= j <= i };
```

⇒ generate AST that visits elements in lexicographic order

What if a different order is needed?
⇒ apply a schedule: maps instance set to multi-dimensional time
⇒ multi-dimensional time is ordered lexicographically

Example: interchange `i` and `j`

```
{S[i,j] -> [t1,t2] : t1 = j and t2 = i}
```

# AST Generation, Schedules and Maps

```
for (i = 1; i <= n; ++i)
    for (j = 1; j <= i; ++j)
        /* S */
```

```
codegen [n] -> { S[i,j] : 1 <= i <= n and 1 <= j <= i };
```

⇒ generate AST that visits elements in lexicographic order

What if a different order is needed?
⇒ apply a schedule: maps instance set to multi-dimensional time
⇒ multi-dimensional time is ordered lexicographically

Example: interchange `i` and `j`
`{S[i,j] -> [t1,t2] : t1 = j and t2 = i}` or `{S[i,j] -> [j,i]}`

# AST Generation, Schedules and Maps

```
for (i = 1; i <= n; ++i)
    for (j = 1; j <= i; ++j)
        /* S */
```

```
codegen [n] -> { S[i,j] : 1 <= i <= n and 1 <= j <= i };
```

⇒ generate AST that visits elements in lexicographic order

What if a different order is needed?
⇒ apply a schedule: maps instance set to multi-dimensional time
⇒ multi-dimensional time is ordered lexicographically

Example: interchange `i` and `j`
`{S[i,j] -> [t1,t2] : t1 = j and t2 = i}` or `{S[i,j] -> [j,i]}`

```
S := [n] -> { S[i,j] : 1 <= i <= n and 1 <= j <= i };
codegen ({S[i,j] -> [j,i]} * S);
```

# AST Generation, Schedules and Maps

```
for (i = 1; i <= n; ++i)
    for (j = 1; j <= i; ++j)
        /* S */
```

```
codegen [n] -> { S[i,j] : 1 <= i <= n and 1 <= j <= i };
```

⇒ generate AST that visits elements in lexicographic order

What if a different order is needed?
⇒ apply a schedule: maps instance set to multi-dimensional time
⇒ multi-dimensional time is ordered lexicographically

Example: interchange `i` and `j`
`{S[i,j] -> [t1,t2] : t1 = j and t2 = i}` or `{S[i,j] -> [j,i]}`

```
S := [n] -> { S[i,j] : 1 <= i <= n and 1 <= j <= i };
codegen ({S[i,j] -> [j,i]} * S);
```

intersect domain of map on the left with set on the right

# AST Generation, Schedules and Maps

Generating AST for more than one space/statement

- $\Rightarrow$ spaces should be named to distinguish them from each other
- $\Rightarrow$ schedule is required because no ordering defined over spaces with different names

# AST Generation, Schedules and Maps

Generating AST for more than one space/statement

- ⇒ spaces should be named to distinguish them from each other
- ⇒ schedule is required because no ordering defined over spaces with different names

Examples:

```
S := [n] -> { A[i] : 0 <= i <= n; B[i] : 0 <= i <= n };
M := { A[i] -> [0,i]; B[i] -> [1,i] };
codegen (M * S);
```

codegen3, codegen4

# AST Generation, Schedules and Maps

Generating AST for more than one space/statement

- ⇒ spaces should be named to distinguish them from each other
- ⇒ schedule is required because no ordering defined over spaces with different names

Examples:

disjunction

```
S := [n] -> { A[i] : 0 <= i <= n; B[i] : 0 <= i <= n };
M := { A[i] -> [0,i]; B[i] -> [1,i] };
codegen (M * S);
```

# AST Generation, Schedules and Maps

Generating AST for more than one space/statement

⇒ spaces should be named to distinguish them from each other

⇒ schedule is required because no ordering defined over spaces with different names

Examples:

disjunction

```
S := [n] -> { A[i] : 0 <= i <= n; B[i] : 0 <= i <= n };
M := { A[i] -> [0,i]; B[i] -> [1,i] };
codegen (M * S);
```

all elements of A before any element of B

# AST Generation, Schedules and Maps

Generating AST for more than one space/statement

- ⇒ spaces should be named to distinguish them from each other
- ⇒ schedule is required because no ordering defined over spaces with different names

disjunction

Examples:

```
S := [n] -> { A[i] : 0 <= i <= n; B[i] : 0 <= i <= n };
M := { A[i] -> [0,i]; B[i] -> [1,i] };
codegen (M * S);
```

all elements of A before any element of B

```
S := [n] -> { A[i] : 0 <= i <= n; B[i] : 0 <= i <= n };
M := { A[i] -> [i,1]; B[i] -> [i,0] };
codegen (M * S);
```

# AST Generation, Schedules and Maps

Generating AST for more than one space/statement

- ⇒ spaces should be named to distinguish them from each other
- ⇒ schedule is required because no ordering defined over spaces with different names

Examples:

disjunction

```
S := [n] -> { A[i] : 0 <= i <= n; B[i] : 0 <= i <= n };
M := { A[i] -> [0,i]; B[i] -> [1,i] };
codegen (M * S);
```

all elements of A before any element of B

```
S := [n] -> { A[i] : 0 <= i <= n; B[i] : 0 <= i <= n };
M := { A[i] -> [i,1]; B[i] -> [i,0] };
codegen (M * S);
```

each element of A after corresponding element of B

# Access Relations and Polyhedral Model

Simple program with temporary array `t`:

```
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]);
```

An access relation maps a statement instance to an array index

For example, the access relation for the read in `S2`:

```
[N] -> { S2[i] -> t[N-i-1] }
```

# Access Relations and Polyhedral Model

Simple program with temporary array `t`:

```
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]);
```

An access relation maps a statement instance to an array index
For example, the access relation for the read in `S2`:

```
[N] -> { S2[i] -> t[N-i-1] }
```

Polyhedral model of a program consists of

- statement instance set
- access relations (must writes, may writes, reads)
- initial schedule

```
M := parse_file("simple.c");
D := M[0]; W := M[1]; R := M[3]; S := M[4];
```

# Lexicographic Optimization

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

- What is the last iteration of the loop?

```
S := [N] -> { [i,j] : 0<=i<N and 0<=j<N-i };
lexmax S;
```

# Lexicographic Optimization

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

- What is the last iteration of the loop?

```
S := [N] -> { [i,j] : 0<=i<N and 0<=j<N-i };
lexmax S;
```

lexicographically last element of set

# Lexicographic Optimization

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

- What is the last iteration of the loop?

```
S := [N] -> { [i,j] : 0<=i<N and 0<=j<N-i };
lexmax S;
```

lexicographically last element of set

- When is a given array element accessed last?

```
A:=[N]->{[i,j]->a[i+j]:0<=i<N and 0<=j<N-i};
lexmax (A^-1);
```

# Lexicographic Optimization

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

- What is the last iteration of the loop?

```
S := [N] -> { [i,j] : 0<=i<N and 0<=j<N-i };
lexmax S;
```

lexicographically last element of set

- When is a given array element accessed last?

```
A:=[N]->{[i,j]->a[i+j]:0<=i<N and 0<=j<N-i};
lexmax (A^-1);
```

inverse map

# Lexicographic Optimization

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

- What is the last iteration of the loop?

```
S := [N] -> { [i,j] : 0<=i<N and 0<=j<N-i };
lexmax S;
```

lexicographically last element of set

- When is a given array element accessed last?

```
A:=[N]->{[i,j]->a[i+j]:0<=i<N and 0<=j<N-i};
lexmax (A^-1);
```

inverse map

lexicographically last image element

# Dataflow Analysis

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

Simple case: array written through a single access

```
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]);
```

# Dataflow Analysis

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

Simple case: array written through a single access

```
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]);
```

# Dataflow Analysis

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

Simple case: array written through a single access

```
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:

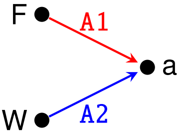
```
A1:=[N]->{F[i,j]->a[i+j]:0<=i<N and 0<=j<N-i};
A2:=[N]->{W[i] -> a[i] : 0 <= i < N };
```

# Dataflow Analysis

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

Simple case: array written through a single access

```
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:

```
A1:=[N]->{F[i,j]->a[i+j]:0<=i<N and 0<=j<N-i};
A2:=[N]->{W[i] -> a[i] : 0 <= i < N };
```

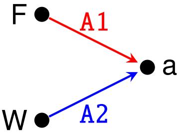
Map to all writes: `R := A2 . (A1^-1);`

# Dataflow Analysis

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

Simple case: array written through a single access

```
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:

```
A1:=[N]->{F[i,j]->a[i+j]:0<=i<N and 0<=j<N-i};
A2:=[N]->{W[i] -> a[i] : 0 <= i < N };
```

Map to all writes: `R := A2 . (A1^-1);`

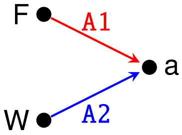
Last write: `lexmax R;`

# Dataflow Analysis

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

Simple case: array written through a single access

```
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:

```
A1:=[N]->{F[i,j]->a[i+j]:0<=i<N and 0<=j<N-i};
A2:=[N]->{W[i] -> a[i] : 0 <= i < N };
```

Map to all writes: `R := A2 . (A1^-1);`

Last write: `lexmax R;`

In general: impose lexicographical order on shared iterators

# Dataflow Analysis

In general:

`last Write before Read under Schedule`

Result: last write + set of reads without corresponding write

# Dataflow Analysis

In general:

```
last Write before Read under Schedule
```

Result: last write + set of reads without corresponding write

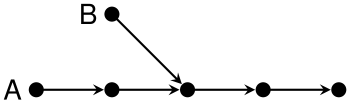
```
for (i = 0; i < n; ++i)
T:  t[i] = a[i];
for (i = 0; i < n; ++i)
    for (j = 0; j < n - i; ++j)
F:      t[j] = f(t[j], t[j+1]);
for (i = 0; i < n; ++i)
B:  b[i] = t[i];
```

```
M := parse_file("dep.c");
Write := M[1]; Read := M[2]; Sched := M[3];
last Write before Read under Sched;
```

# Transitive Closures

Given a graph (represented as an affine map)

```
M := { A[i] -> A[i+1] : 0 <= i <= 3; B[] -> A[2] };
```

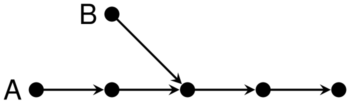



What is the transitive closure?

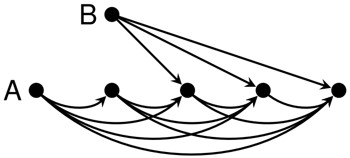
# Transitive Closures

Given a graph (represented as an affine map)

```
M := { A[i] -> A[i+1] : 0 <= i <= 3; B[] -> A[2] };
```



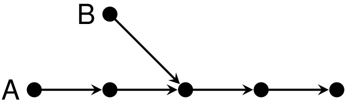

What is the transitive closure? ⇒ `M^+;`



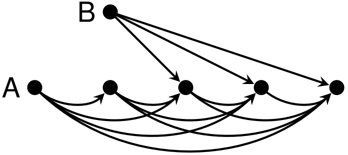
# Transitive Closures

Given a graph (represented as an affine map)

```
M := { A[i] -> A[i+1] : 0 <= i <= 3; B[] -> A[2] };
```




What is the transitive closure? ⇒ `M^+;`



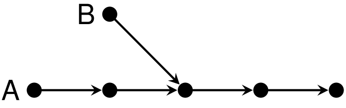

Result:

```
({ B[] -> A[o0] : o0 <= 4 and o0 >= 3; B[] -> A[2];
   A[i] -> A[o0] : i >= 0 and i <= 3 and o0 >= 1 and
   o0 <= 4 and o0 >= 1 + i }, True)
```

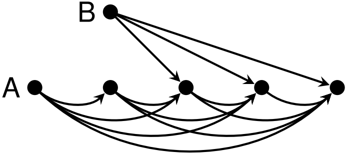
# Transitive Closures

Given a graph (represented as an affine map)

```
M := { A[i] -> A[i+1] : 0 <= i <= 3; B[] -> A[2] };
```




What is the transitive closure? ⇒ `M^+;`




Result:

exact transitive closure

```
({ B[] -> A[o0] : o0 <= 4 and o0 >= 3; B[] -> A[2];
   A[i] -> A[o0] : i >= 0 and i <= 3 and o0 >= 1 and
   o0 <= 4 and o0 >= 1 + i }, True)
```

# Reachability Analysis

```
double x[2][10];
int old = 0, new = 1, i, t;
for (t = 0; t<1000; t++) {
  for (i = 0; i<10;i++)
    x[new][i] = g(x[old][i]);
  new = (new+1) %2; old = (old+1) %2;
}
```

Invariant between `new` and `old`?

# Reachability Analysis

```
double x[2][10];
int old = 0, new = 1, i, t;
for (t = 0; t<1000; t++) {
  for (i = 0; i<10;i++)
    x[new][i] = g(x[old][i]);
  new = (new+1) %2; old = (old+1) %2;
}
```

Invariant between `new` and `old`?

```
T := {[new,old] -> [(new+1)%2,(old+1)%2]};
S0 := {[0,1]};
(T^+)(S0);
```

# Cardinality

```
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?

```
S := [N] -> { [i,j] : 0<=i<N and 0<=j<N-i };
card S;
```

# Cardinality

```
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?

```
S := [N] -> { [i,j] : 0<=i<N and 0<=j<N-i };
card S;
```

number of elements in the set

# Cardinality

```
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?

```
S := [N] -> { [i,j] : 0<=i<N and 0<=j<N-i };
card S;
```

number of elements in the set

- How many times is a given array element written?

```
A:=[N]->{[i,j]->a[i+j]:0<=i<N and 0<=j<N-i};
card (A^-1);
```

# Cardinality

```
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?

```
S := [N] -> { [i,j] : 0<=i<N and 0<=j<N-i };
card S;
```

number of elements in the set

- How many times is a given array element written?

```
A:=[N]->{[i,j]->a[i+j]:0<=i<N and 0<=j<N-i};
card (A^-1);
```

number of image elements

card1,card2,card3

# Cardinality

```
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?

```
S := [N] -> { [i,j] : 0<=i<N and 0<=j<N-i };
card S;
```

number of elements in the set

- How many times is a given array element written?

```
A:=[N]->{[i,j]->a[i+j]:0<=i<N and 0<=j<N-i};
card (A^-1);
```

number of image elements

- How many array elements are written?

```
A:=[N]->{[i,j]->a[i+j]:0<=i<N and 0<=j<N-i};
card (ran A);
```

card1,card2,card3

# Quasipolynomials

```
for (i = 1; i <= n; ++i)
    for (j = 1; j <= n - 2 * i; ++j)
        /* S */
```

How many times is S executed?

```
card [n] -> { [i,j] : 1 <= i <= n and 1 <= j <= n - 2i };
```

# Quasipolynomials

```
for (i = 1; i <= n; ++i)
    for (j = 1; j <= n - 2 * i; ++j)
        /* S */
```

How many times is S executed?

```
card [n] -> { [i,j] : 1 <= i <= n and 1 <= j <= n - 2i };
```

Result:

```
[n] -> { ((-1/4 * n + 1/4 * n^2) - 1/2 * floor((n)/2)) :
n >= 3 }
```

That is,

$$-\frac{n}{4} + \frac{n^2}{4} - \frac{1}{2}\left\lfloor \frac{n}{2} \right\rfloor \quad \text{if } n \geq 3.$$

# Quasipolynomials

```
for (i = 1; i <= n; ++i)
    for (j = 1; j <= n - 2 * i; ++j)
        /* S */
```

How many times is S executed?

```
card [n] -> { [i,j] : 1 <= i <= n and 1 <= j <= n - 2i };
```

Result:

```
[n] -> { ((-1/4 * n + 1/4 * n^2) - 1/2 * floor((n)/2)) :
n >= 3 }
```

That is,

$$-\frac{n}{4} + \frac{n^2}{4} - \frac{1}{2}\left\lfloor \frac{n}{2} \right\rfloor \quad \text{if } n \geq 3.$$

Polynomial approximations

$\Rightarrow$ run `iscc --polynomial-approximation`

# Memory Requirements

```
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?

# Memory Requirements

```
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?

```
ub [N] -> {[i,j] -> i*j+i-N+1: 0 <= i < N and i <= j < N};
```

# Memory Requirements

```
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?

```
ub [N] -> {[i,j] -> i*j+i-N+1: 0 <= i < N and i <= j < N};
```

Result:

```
([N] -> { max((1 - 2 * N + N^2)) : N >= 1 }, True)
```

# Memory Requirements

```
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?

```
ub [N] -> {[i,j] -> i*j+i-N+1: 0 <= i < N and i <= j < N};
```

Result:

```
([N] -> { max((1 - 2 * N + N^2)) : N >= 1 }, True)
```

bound is tight

# Incremental Counting

```
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?

- direct computation

```
card [N] -> { [i,j] : 0<=i<N and 0<=j<N-i };
```

# Incremental Counting

```
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?

- direct computation

```
card [N] -> { [i,j] : 0<=i<N and 0<=j<N-i };
```

- incremental computation

```
card [N] -> { [i] -> [j] : 0<=i<N and 0<=j<N-i };
```

# Incremental Counting

```
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?

- direct computation

  ```
  card [N] -> { [i,j] : 0<=i<N and 0<=j<N-i };
  ```

- incremental computation

  ```
  card [N] -> { [i] -> [j] : 0<=i<N and 0<=j<N-i };
  ```

  Result:

  ```
  [N] -> { [i] -> (N - i) : i <= -1 + N and i >= 0 }
  ```

  ```
  sum [N] -> { [i] -> (N - i) : i <= -1 + N and i >= 0 };
  ```

# Incremental Counting

```
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?

- direct computation

  ```
  card [N] -> { [i,j] : 0<=i<N and 0<=j<N-i };
  ```

- incremental computation

  ```
  card [N] -> { [i] -> [j] : 0<=i<N and 0<=j<N-i };
  ```

  Result:

  ```
  [N] -> { [i] -> (N - i) : i <= -1 + N and i >= 0 }
  ```

  ```
  sum [N] -> { [i] -> (N - i) : i <= -1 + N and i >= 0 };
  ```

  sum over all elements in domain

# Total Memory Allocation

```
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?

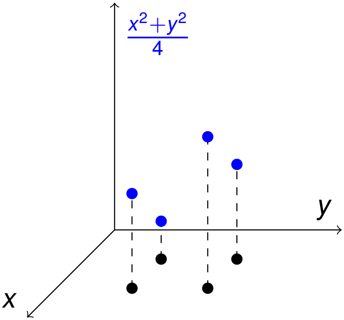
# Total Memory Allocation

```
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?

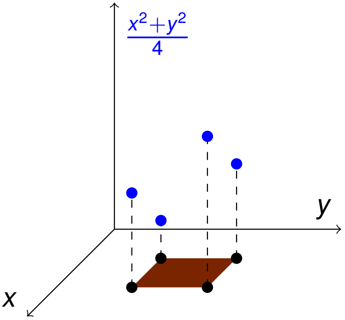
```
sum [N] -> {[i,j] -> i*j+i-N+1: 0 <= i < N and i <= j < N};
```

# Weighted Counting



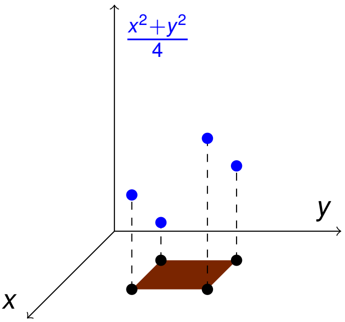

```
F := { [x,y] -> 1/4*x^2+1/4*y^2 : 1 <= x,y <= 2 };
```

# Weighted Counting




```
F := { [x,y] -> 1/4*x^2+1/4*y^2 : 1 <= x,y <= 2 };
D := dom F;
```

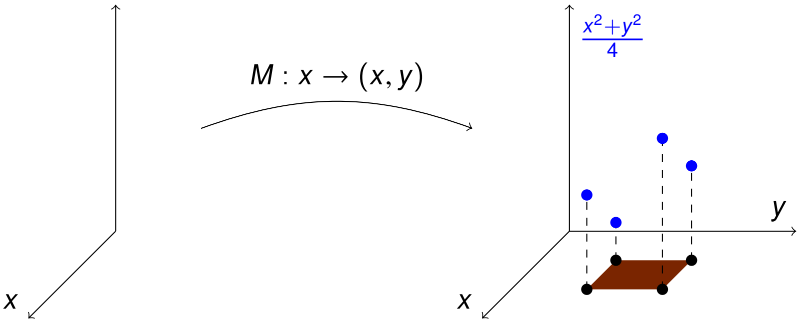
# Weighted Counting




```
F := { [x,y] -> 1/4*x^2+1/4*y^2 : 1 <= x,y <= 2 };
D := dom F;
F(D);
```

⇒ sum of `F` over points in `D`

# Weighted Counting

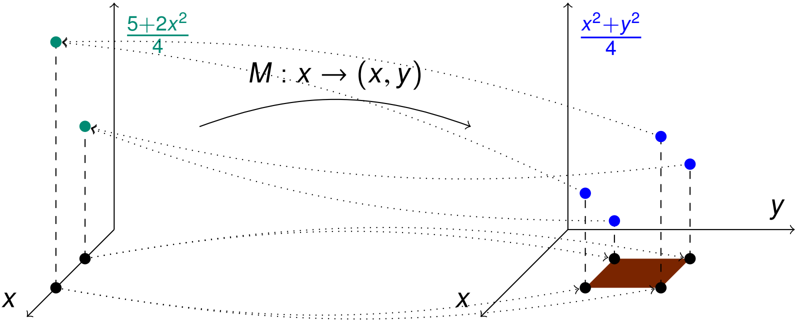



```
F := { [x,y] -> 1/4*x^2+1/4*y^2 : 1 <= x,y <= 2 };
D := dom F;
F(D);
```

⇒ sum of `F` over points in `D`

```
M := { [x] -> [x,y] };
```

# Weighted Counting




```
F := { [x,y] -> 1/4*x^2+1/4*y^2 : 1 <= x,y <= 2 };
D := dom F;
F(D);
```

⇒ sum of `F` over points in `D`

```
M := { [x] -> [x,y] };
F(M);
```

⇒ sum of `F` over image of `M` (alternative notation: `M . F`)

# Compositions with Piecewise (Folds of) Quasipolynomials

`f . g;`

- `f`: $D_1 \to D_2$ is a map
- `g`: $D_2 \to \mathbb{Q}$ may be
  - piecewise quasipolynomial
    (result of counting problems)

    $\Rightarrow$ take sum over intersection of `ran f` and `dom g`
  - piecewise fold of quasipolynomials
    (result of upper bound computation)

    $\Rightarrow$ compute bound over intersection of `ran f` and `dom g`
- `(f . g)`: $D_1 \to \mathbb{Q}$ of same type as `g`

Note: if $f$ is single-valued, then sum/bound is computed over a single point

# Outline

# Pointer Conversion

```
p = a;
for (i = 0; i < N; ++i)
    for (j = i; j < N; ++j) {
        p += 1 + j * ((j-i)/4);
        *p = hard_work(i,j);
    }
```

Can we parallelize this code?

# Pointer Conversion

```
p = a;
for (i = 0; i < N; ++i)
    for (j = i; j < N; ++j) {
        p += 1 + j * ((j-i)/4);
        *p = hard_work(i,j);
    }
```

Can we parallelize this code?

⇒ No, (false) dependency through `p`
⇒ Compute closed formula for `p`

$$p = a + \sum_{\substack{(i',j') \in S \\ (i',j') \preccurlyeq (i,j)}} j' \left\lfloor \frac{j' - i'}{4} \right\rfloor$$

with $S = \{\, (i', j') \in \mathbb{Z}^2 \mid 0 \leq i' < N \wedge i' \leq j' < N \,\}$

# Pointer Conversion

```
p = a;
for (i = 0; i < N; ++i)
    for (j = i; j < N; ++j) {
        p += 1 + j * ((j-i)/4);
        *p = hard_work(i,j);
    }
```

Can we parallelize this code?

⇒ No, (false) dependency through `p`
⇒ Compute closed formula for `p`

$$p = a + \sum_{\substack{(i',j') \in S \\ (i',j') \boxed{\preccurlyeq} (i,j)}} j' \left\lfloor \frac{j' - i'}{4} \right\rfloor$$

lexicographically less than

with $S = \{\, (i', j') \in \mathbb{Z}^2 \mid 0 \leq i' < N \wedge i' \leq j' < N \,\}$

# Pointer Conversion

$$p = a + \sum_{\substack{(i',j') \in S \\ (i',j') \preccurlyeq (i,j)}} j' \left\lfloor \frac{j' - i'}{4} \right\rfloor$$

with $S = \{\, (i', j') \in \mathbb{Z}^2 \mid 0 \leq i' < N \wedge i' \leq j' < N \,\}$

# Pointer Conversion

$$p = a + \sum_{\substack{(i',j')\in S \\ (i',j')\preccurlyeq(i,j)}} j' \left\lfloor \frac{j'-i'}{4} \right\rfloor$$

with $S = \{\, (i', j') \in \mathbb{Z}^2 \mid 0 \le i' < N \wedge i' \le j' < N \,\}$

```
S := [N] -> { [i,j] : 0 <= i < N and i <= j < N };
L := S <<= S;
INC := { [[i,j] -> [i',j']] -> 1 + j' * [(j'-i')/4] };
INC := INC * (wrap (L^-1));
sum INC;
```

# Pointer Conversion

$$p = a + \sum_{\substack{(i',j') \in S \\ (i',j') \preccurlyeq (i,j)}} j' \left\lfloor \frac{j' - i'}{4} \right\rfloor$$

with $S = \{ (i', j') \in \mathbb{Z}^2 \mid 0 \le i' < N \wedge i' \le j' < N \}$

map: (elements of) left set lexicographically smaller than right set

```
S := [N] -> { [i,j] : 0 <= i < N and i <= j < N };
L := S <<= S;
INC := { [[i,j] -> [i',j']] -> 1 + j' * [(j'-i')/4] };
INC := INC * (wrap (L^-1));
sum INC;
```

# Pointer Conversion

$$p = a + \sum_{\substack{(i',j') \in S \\ (i',j') \preccurlyeq (i,j)}} j' \left\lfloor \frac{j' - i'}{4} \right\rfloor$$

with $S = \{ (i',j') \in \mathbb{Z}^2 \mid 0 \le i' < N \wedge i' \le j' < N \}$

map: (elements of) left set lexicographically smaller than right set

```
S := [N] -> { [i,j] : 0 <= i < N and i <= j < N };
L := S <<= S;
INC := { [[i,j] -> [i',j']] -> 1 + j' * [(j'-i')/4] };
INC := INC * (wrap (L^-1));
sum INC;
```

embed map in a set

# Pointer Conversion

$$p = a + \sum_{\substack{(i',j') \in S \\ (i',j') \preccurlyeq (i,j)}} j' \left\lfloor \frac{j' - i'}{4} \right\rfloor$$

with $S = \{ (i', j') \in \mathbb{Z}^2 \mid 0 \le i' < N \wedge i' \le j' < N \}$

map: (elements of) left set lexicographically smaller than right set

```
S := [N] -> { [i,j] : 0 <= i < N and i <= j < N };
L := S <<= S;
INC := { [[i,j] -> [i',j']] -> 1 + j' * [(j'-i')/4] };
INC := INC * (wrap (L^-1));
sum INC;
```

embed map in a set

Note: if domain of argument to `sum` [ub] is an embedded map, then sum [bound] is computed over range of embedded map

# Dynamic Memory Requirement Estimation [CFGV2006]

How much memory is needed to execute the following program?

```
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 [CFGV2006]

How much memory is needed to execute the following program?

```
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;
}
```

```
D := {
m0[m]->S1[c] : 0<=c<m;
m0[m]->S2[c] : 0<=c<m;
m1[k]->S3[i] : 1<=i<=k;
m1[k]->S4[i] : 1<=i<=k;
m2[n]->S5[];
m2[n]->S6[j] : 1<=j<=n
};
```

# Dynamic Memory Requirement Estimation [CFGV2006]

How much (scoped) memory is needed?

⇒ compute for each method

| | |
|---|---|
| $\mathrm{ret}_{\mathtt{m}}$ | size of memory returned by `m` |
| $\mathrm{cap}_{\mathtt{m}}$ | size of memory “captured” (not returned) by `m` |
| $\mathrm{memRq}_{\mathtt{m}}$ | total memory requirements of `m` |

$$\mathrm{ret}_{\mathtt{m}} + \mathrm{cap}_{\mathtt{m}} = \sum_{\text{p called by } \mathtt{m}} \mathrm{ret}_{\mathtt{p}}$$

$$\mathrm{memRq}_{\mathtt{m}} = \mathrm{cap}_{\mathtt{m}} + \max_{\text{p called by } \mathtt{m}} \mathrm{memRq}_{\mathtt{p}}$$

# Dynamic Memory Requirement Estimation [CFGV2006]

How much (scoped) memory is needed?

⇒ compute for each method

- $\mathrm{ret}_{\mathtt{m}}$ size of memory returned by `m`
- $\mathrm{cap}_{\mathtt{m}}$ size of memory "captured" (not returned) by `m`
- $\mathrm{memRq}_{\mathtt{m}}$ total memory requirements of `m`

$$\mathrm{ret}_{\mathtt{m}} + \mathrm{cap}_{\mathtt{m}} = \sum_{\text{p called by } \mathtt{m}} \mathrm{ret}_{\mathtt{p}}$$

$$\mathrm{memRq}_{\mathtt{m}} = \mathrm{cap}_{\mathtt{m}} + \max_{\text{p called by } \mathtt{m}} \mathrm{memRq}_{\mathtt{p}}$$

⇒ summarize over statement instances, i.e., compose with $M = (\overrightarrow{\mathrm{dom}}\, I)^{-1}$

```
D := {
m0[m]->S1[c] : 0<=c<m;  m0[m]->S2[c] : 0<=c<m;
m1[k]->S3[i] : 1<=i<=k; m1[k]->S4[i] : 1<=i<=k;
m2[n]->S5[];            m2[n]->S6[j] : 1<=j<=n };
DM := (domain_map D)^-1;
```

# Dynamic Memory Requirement Estimation [CFGV2006]

How much (scoped) memory is needed?
⇒ compute for each method

| | |
|---|---|
| $\mathrm{ret}_{\mathtt{m}}$ | size of memory returned by `m` |
| $\mathrm{cap}_{\mathtt{m}}$ | size of memory “captured” (not returned) by `m` |
| $\mathrm{memRq}_{\mathtt{m}}$ | total memory requirements of `m` |

$$\mathrm{ret}_{\mathtt{m}} + \mathrm{cap}_{\mathtt{m}} = \sum_{\text{p called by } \mathtt{m}} \mathrm{ret}_{\mathtt{p}}$$

$$\mathrm{memRq}_{\mathtt{m}} = \mathrm{cap}_{\mathtt{m}} + \max_{\text{p called by } \mathtt{m}} \mathrm{memRq}_{\mathtt{p}}$$

# Dynamic Memory Requirement Estimation [CFGV2006]

How much (scoped) memory is needed?
⇒ compute for each method

| | |
|---|---|
| $\mathrm{ret}_{\mathtt{m}}$ | size of memory returned by `m` |
| $\mathrm{cap}_{\mathtt{m}}$ | size of memory “captured” (not returned) by `m` |
| $\mathrm{memRq}_{\mathtt{m}}$ | total memory requirements of `m` |

$$\mathrm{ret}_{\mathtt{m}} + \mathrm{cap}_{\mathtt{m}} = \sum_{\text{p called by } \mathtt{m}} \mathrm{ret}_{\mathtt{p}}$$

$$\mathrm{memRq}_{\mathtt{m}} = \mathrm{cap}_{\mathtt{m}} + \max_{\text{p called by } \mathtt{m}} \mathrm{memRq}_{\mathtt{p}}$$

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

mem2

# Dynamic Memory Requirement Estimation [CFGV2006]

How much (scoped) memory is needed?
⇒ compute for each method

| | |
|---|---|
| $\mathrm{ret}_{\mathtt{m}}$ | size of memory returned by `m` |
| $\mathrm{cap}_{\mathtt{m}}$ | size of memory "captured" (not returned) by `m` |
| $\mathrm{memRq}_{\mathtt{m}}$ | total memory requirements of `m` |

$$\mathrm{ret}_{\mathtt{m}} + \mathrm{cap}_{\mathtt{m}} = \sum_{\text{p called by m}} \mathrm{ret}_{\mathtt{p}}$$

$$\mathrm{memRq}_{\mathtt{m}} = \mathrm{cap}_{\mathtt{m}} + \max_{\text{p called by m}} \mathrm{memRq}_{\mathtt{p}}$$

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

```
ret_m2 := DM .
   { [m2[n] -> S5[]] -> n : n >= 0 };
cap_m2 := DM .
   { [m2[n] -> S6[j]] -> 1 };
req_m2 := cap_m2 +
   { m2[n] -> max(0) };
```

mem2

# Dynamic Memory Requirement Estimation [CFGV2006]

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

$$\text{cap}_{\mathtt{m1}}(k) = \sum_{1 \le i \le k} (1 + \text{ret}_{\mathtt{m2}}(i))$$

`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`

# Dynamic Memory Requirement Estimation [CFGV2006]

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

$$\mathrm{cap}_{\mathtt{m1}}(k) = \sum_{1 \leq i \leq k} (1 + \mathrm{ret}_{\mathtt{m2}}(i))$$

`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

```
CB_m1 := { [m1[k] -> S4[i]] -> m2[i] };
cap_m1 := DM . ({ [m1[k]->S3[i]] -> 1 } + (CB_m1 . ret_m2));
```

# Dynamic Memory Requirement Estimation [CFGV2006]

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

$$\mathrm{memRq_m} = \mathrm{cap_m} + \max_{\text{p called by } \mathtt{m}} \mathrm{memRq_p}$$

```
CB_m1 := { [m1[k] -> S4[i]] -> m2[i] };
ret_m1 := { m1[k] -> 0 };
cap_m1 := DM . ({ [m1[k]->S3[i]] -> 1 } + (CB_m1 . ret_m2));
req_m1 := cap_m1 + (DM . CB_m1 . req_m2);
```

# Dynamic Memory Requirement Estimation [CFGV2006]

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

```
CB_m0 := { [m0[m] -> S1[c]] -> m1[c];
           [m0[m] -> S2[c]] -> m2[2 * m - c] };
ret_m0 := { m0[m] -> 0 };
cap_m0 := DM . CB_m0 . (ret_m1 + ret_m2);
req_m0 := cap_m0 + (DM . CB_m0 . (req_m1 . req_m2));
```

# Dynamic Memory Requirement Estimation [CFGV2006]

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

```
CB_m0  := { [m0[m] -> S1[c]] -> m1[c];
            [m0[m] -> S2[c]] -> m2[2 * m - c] };
ret_m0  := { m0[m] -> 0 };
cap_m0  := DM . CB_m0 . (ret_m1 + ret_m2);
req_m0  := cap_m0 + (DM . CB_m0 . (req_m1 . req_m2));
```

combine reductions

# Reuse Distance Computation

Given an access to a cache line $\ell$, how many distinct cache lines have been accessed since the previous access to $\ell$?
$\Rightarrow$ Is the cache line still in the cache?

# Reuse Distance Computation

Given an access to a cache line $\ell$, how many distinct cache lines have been accessed since the previous access to $\ell$?
$\Rightarrow$ Is the cache line still in the cache?

```
for (i = 0; i <= 7; ++i) {
    A[i];              //reference a
    A[7-i];            //reference b
    if (i <= 3)
        A[2*i];        //reference c
}
```

Assume A[i] in cache line $\lfloor i/3 \rfloor$

# Reuse Distance Computation

Given an access to a cache line $\ell$, how many distinct cache lines have been accessed since the previous access to $\ell$?
$\Rightarrow$ Is the cache line still in the cache?

```
for (i = 0; i <= 7; ++i) {
    A[i];                //reference a
    A[7-i];              //reference b
    if (i <= 3)
        A[2*i];          //reference c
}
```

Assume `A[i]` in cache line $\lfloor i/3 \rfloor$

| $i$ | 0 | | | 1 | | | 2 | | | 3 | | | 4 | | 5 | | 6 | | 7 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $r$ | a | b | c | a | b | c | a | b | c | a | b | c | a | b | a | b | a | b | a | b |
| $r@i$ | 0 | 7 | 0 | 1 | 6 | 2 | 2 | 5 | 4 | 3 | 4 | 6 | 4 | 3 | 5 | 2 | 6 | 1 | 7 | 0 |
| $\lfloor (r@i)/3 \rfloor$ | 0 | 2 | 0 | 0 | 2 | 0 | 0 | 1 | 1 | 1 | 1 | 2 | 1 | 1 | 1 | 0 | 2 | 0 | 2 | 0 |
| distance | 0 | 0 | 2 | 1 | 2 | 2 | 1 | 0 | 1 | 1 | 1 | 3 | 2 | 1 | 1 | 3 | 3 | 2 | 2 | 2 |

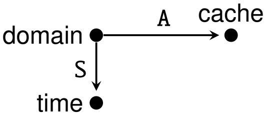
# Reuse Distance Computation

```
for (i = 0; i <= 7; ++i) {
    A[i];           //reference a
    A[7-i];         //reference b
    if (i <= 3)
        A[2*i];     //reference c
}
```

Assume `A[i]` in cache line $\lfloor i/3 \rfloor$

# Reuse Distance Computation

```
for (i = 0; i <= 7; ++i) {
    A[i];             //reference a
    A[7-i];           //reference b
    if (i <= 3)
        A[2*i];       //reference c
}
```

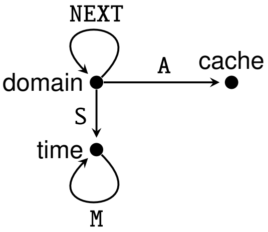



Assume `A[i]` in cache line $\lfloor i/3 \rfloor$

```
D := { a[i] : 0 <= i <= 7; b[i] : 0 <= i <= 7; c[i] : 0 <= i <= 3 };
C := { A[i] -> L[j] : j = floor(i/3) };
A := ({ a[i] -> A[i]; b[i] -> A[7-i]; c[i] -> A[2i] } . C) * D;
S := { a[i] -> [i,0]; b[i] -> [i,1]; c[i] -> [i,2] } * D;
```

# Reuse Distance Computation

```
for (i = 0; i <= 7; ++i) {
    A[i];            //reference a
    A[7-i];          //reference b
    if (i <= 3)
        A[2*i];      //reference c
}
```




Assume `A[i]` in cache line $\lfloor i/3 \rfloor$

```
D := { a[i] : 0 <= i <= 7; b[i] : 0 <= i <= 7; c[i] : 0 <= i <= 3 };
C := { A[i] -> L[j] : j = floor(i/3) };
A := ({ a[i] -> A[i]; b[i] -> A[7-i]; c[i] -> A[2i] } . C) * D;
S := { a[i] -> [i,0]; b[i] -> [i,1]; c[i] -> [i,2] } * D;
TIME := ran S; LT := TIME << TIME; LE := TIME <<= TIME;
T := ((S^-1) . A . (A^-1) . S) * LT;
M := lexmin T;
NEXT := S . M . (S^-1); # map to next access to same cache line
AFTER_PREV := (NEXT^-1) . (S . LE . (S^-1));
BEFORE := S . (LE^-1) . (S^-1);
card ((AFTER_PREV * BEFORE) . A);
```